2067. Number of Equal Count Substrings String ) Counting Medium Prefix Sum

# Leetcode Link

# Problem Description

substring where each unique letter appears exactly count times within it. Essentially, the task is to find all the substrings where the frequency of each distinct letter within the substring equals the given

Intuition

condition (y).

1. We need to find substrings that contain each unique character count times. Since we are limited to lowercase English letters, there is a maximum of 26 unique characters that can appear in substrings.

- there are 26 lowercase letters). The variable x represents this count of unique characters.
- 8. We increment ans each time the number of characters meeting the criterion x is the same as y, which implies we found an equal count substring.
- By the end of this process, we will have iterated over all possible substrings that can have an equal count of all its unique characters. Each time the window slides and the conditions match, we would increment ans, and finally, we return the value of ans as the result.
- Solution Approach
- Here's how the implementation works: A Python Counter from the collections module is utilized to maintain the frequency of each character within the current sliding

The solution begins by iterating over all potential counts of unique characters a substring might have (x), ranging from 1 to 26.

This corresponds to the possible number of characters (considering the alphabet has 26 letters) that can fulfill the equal count

The solution employs a combination of the sliding window technique and a frequency counter (Counter) to efficiently identify all

## • m is calculated as count \* x and represents the minimum size the window needs to have for there to be x unique characters each appearing count times.

window.

condition.

the string.

class Solution:

10

11

12

13

15

16

20

In code, this is represented as:

ans = 0

return ans

• The sliding window is iterated across the string s using a for-loop. For each character c encountered, the Counter cnt is updated to include the new character.

y tracks how many characters within the current window meet the condition of appearing exactly count times.

substring, and ans is incremented.

characters count times each, we start to shrink the window from the left.

def equalCountSubstrings(self, s: str, count: int) -> int:

for i, c in enumerate(s):

y += cnt[c] == count

y -= cnt[c] == count + 1

cnt[s[j]] -= 1

condition matches (where x = y), we increment ans.

y += cnt[s[j]] == count

y -= cnt[s[j]] == count - 1

cnt[c] += 1

if j >= 0:

equal count substrings within the given string 5.

 As we shrink the window, we decrease the frequency of the character that goes out of the window and update y to reflect whether the character going out still meets the count condition or not.

If the current window exactly contains x characters that meet the count condition, then this window is a valid equal count

When the end of the window goes beyond index i - m, indicating that the window is larger than the size needed to fit x unique

The Python Counter data structure speedily updates and keeps track of character frequencies, and the sliding window technique ensures that the algorithm considers each substring without redundant recalculations.

• The above steps are repeated for each x until all potential substring lengths have been checked or until m exceeds the length of

- for x in range(1, 27): m = count \* xif m > len(s): break cnt = Counter()
- Each iteration adaptive to the character presence updates the answer ans when a valid window is found. The algorithm's complexity is primarily driven by the length of the input string s and the interaction between the count of unique characters and the sliding window.

Let's walk through a small example to illustrate the solution approach with a string s = "aabbcc" and count = 2.

3. We initialize a counter cnt and a variable y to keep track of characters meeting the condition.

We must find substrings where each distinct character appears exactly twice. Here's how the sliding window technique and

2. Let's start with x = 1. The minimum length m for a substring to have one unique character appearing twice is m = count \* x = 2

4. As we slide through the string, we will encounter "aa", "bb", and "cc" as substrings of length m = 2. Each time we slide and the

6. Next, take x = 2. Here, m = count \* x = 2 \* 2 = 4. We use the window size of 4, slide through the string, and find "aabb",

1. We iterate from x = 1 to x = 26, where x represents the number of unique characters that must each appear count times within

a substring.

\*1 = 2.

"bbcc".

times.

6

8

9

10

11

12

13

14

15

16

17

18

19

20

22

23

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46 47

48

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32 33

34

35

36

37

38

39

40

41

42

43

44

45

46

};

1 /\*\*

\*/

6

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

35

36

37

38

39

40

41

42

43 };

Typescript Solution

\* @param {string} s - The input string.

const strLength: number = s.length;

let validCount: number = 0;

for (let i = 0; i < strLength; ++i) {</pre>

++charCount[currentCharIndex];

// it is a valid substring.

The variables ans, y, x, m, i, j, and c use constant space.

const startIndex = i - windowSize;

for (let numOfUniqueChars = 1;

++numOfUniqueChars) {

let answer: number = 0;

\* @return {number} The count of valid substrings.

var equalCountSubstrings = function(s: string, count: number): number {

const windowSize: number = numOfUniqueChars \* count;

const charCount: number[] = new Array(26).fill(0);

from collections import Counter

def equalCountSubstrings(self, s: str, count: int) -> int:

required\_length = count \* unique\_chars

# Iterate through characters of the string

start\_index = index - required\_length

char\_count[s[start\_index]] -= 1

current\_count\_match += 1

current\_count\_match -= 1

if char\_count[s[start\_index]] == count:

elif char\_count[s[start\_index]] == count - 1:

num\_substrings += unique\_chars == current\_count\_match

int answer = 0; // This will contain the final count of valid substrings.

// Update validCharCount for the current character count.

int startCharIndex = s.charAt(startWindowIndex) - 'a';

// Update validCharCount after decrementing.

if (charCount[startCharIndex] == count - 1) {

return totalSubstrings; // Return the total count of valid substrings

\* Counts the number of substrings where each unique character occurs exactly `count` times.

// Iterate through possible substrings with unique character counts from 1 to 26

// Iterate through the string to count the occurrences of each character

validCount -= charCount[currentCharIndex] === count + 1 ? 1 : 0;

// Decrement the count for the character going out of the window

// If the number of valid characters equals the number of unique characters,

validCount += charCount[currentCharIndex] === count ? 1 : 0;

answer += numOfUniqueChars === validCount ? 1 : 0;

const currentCharIndex: number = s.charCodeAt(i) - 'a'.charCodeAt(0);

numOfUniqueChars <= 26 && numOfUniqueChars \* count <= strLength;</pre>

\* @param {number} count - The exact number of times each character should repeat in a substring.

if (charCount[startCharIndex] == count) {

// Iterate over possible distinct character counts from 1 to 26 (inclusive),

// The length of the substring we're looking for based on 'numDistinctChars'.

int[] charCount = new int[26]; // Count of each character in the current window.

int currentCharIndex = s.charAt(i) - 'a'; // Convert char to index (0-25).

// Remove the character from the start of the window if it's outside the window.

--charCount[startCharIndex]; // Decrement the count for this character.

++charCount[currentCharIndex]; // Increment the count for this character.

int stringLength = s.length(); // The length of the input string.

int substringLength = count \* numDistinctChars;

// Iterate over the characters of the string.

if (charCount[currentCharIndex] == count) {

int startWindowIndex = i - substringLength;

if (charCount[currentCharIndex] == count + 1) {

for (int i = 0; i < stringLength; ++i) {</pre>

++validCharCount;

---validCharCount;

if (startWindowIndex >= 0) {

++validCharCount;

--validCharCount;

// but only if 'count \* x' does not exceed the length of the string.

for index, char in enumerate(s):

char\_count[char] += 1

if start\_index >= 0:

# increment the result

return num\_substrings

# Return the total number of valid substrings

public int equalCountSubstrings(String s, int count) {

for unique\_chars in range(1, 27):

if required\_length > len(s):

char\_count = Counter()

current\_count\_match = 0

break

num\_substrings = 0 # Initialize the number of valid substrings

# Initialize a counter for the occurrence of characters

class Solution:

frequency counting would be used:

Example Walkthrough

```
7. For x = 2, our window "aabb" and "bbcc" are valid, so ans = ans + 2 = 3 + 2 = 5.
8. We proceed with this process, increasing x until m exceeds the string length or until x = 26.
```

5. For our example, "aa", "bb", and "cc" are valid substrings when x=1. So, ans = 3 after processing for x=1.

substring, ans = ans + 1 = 5 + 1 = 6. 10. Since we've reached a window length equal to the entire string, we stop our iteration here as no longer substrings can be formed.

Through this process for each possible value of x, we have found the total number of equal count substrings to be 6. This method

aggregates counts for all possible window sizes that match the condition of having each unique character appear exactly count

9. In this example, for x = 3, m = count \* x = 2 \* 3 = 6, which is equal to the length of the string. So "aabbcc" is also a valid

**Python Solution** 

# Iterate through the possible numbers of unique characters (1 to 26 for the alphabet)

# Calculate the length of the substring that must be considered given the count

# If the required length exceeds the string length, no further consideration is needed

# If a character reaches the exact 'count', increment the current count match

# If start index is valid, update the character count and current count match

# If count of a character becomes 'count', increment the current count match

# If the current count match is equal to the number of unique characters needed,

# If count of a character falls below 'count', decrement the current count match

for (int numDistinctChars = 1; numDistinctChars <= 26 && count \* numDistinctChars <= stringLength; ++numDistinctChars) {</pre>

int validCharCount = 0; // Number of characters with exactly 'count' occurrences in the current window.

24 if char\_count[char] == count: 25 current\_count\_match += 1 26 # If a character exceeds the 'count', decrement the current count match elif char\_count[char] == count + 1: current count\_match -= 1 28 29 30 # Calculate the start index of the current window

# Initialize a variable to track the number of unique characters with exactly 'count' occurrences

```
Java Solution
   class Solution {
```

```
41
 42
                     // If the number of valid characters matches the distinct character count,
                     // we've found a valid substring, so increment the answer.
 43
                     if (numDistinctChars == validCharCount) {
 44
 45
                         ++answer;
 46
 47
 48
 49
             return answer; // Return the total count of valid substrings.
 50
 51 }
 52
C++ Solution
  1 class Solution {
  2 public:
         int equalCountSubstrings(string s, int count) {
             int totalSubstrings = 0; // This will hold the total count of substrings found
             int strLength = s.size(); // The length of the input string
             int charCount[26]; // Array to keep count of each character within a window
  6
             // Iterate over possible lengths of substrings which are multiples of count
  8
             for (int substringFactor = 1; substringFactor * count <= strLength; ++substringFactor) {</pre>
  9
                 int windowSize = count * substringFactor; // Calculate window size
 10
                 memset(charCount, 0, sizeof charCount); // Initialize character counts to 0
 11
 12
                 int currentFactorCount = 0; // Tracks how many characters have reached the 'count' within the window
 13
                 // Iterate over each character in the input string
 14
 15
                 for (int i = 0; i < strLength; ++i) {</pre>
                     int charIndex = s[i] - 'a'; // Convert character to index (0-25)
 16
 17
                     ++charCount[charIndex]; // Increment the count for this character
 18
 19
                     // Increment currentFactorCount if count for this character has reached 'count'
                     if (charCount[charIndex] == count) {
 20
                         ++currentFactorCount;
 21
                     } else if (charCount[charIndex] == count + 1) {
 22
 23
                         --currentFactorCount; // Decrement if the count exceeds 'count'
 24
 25
 26
                     // If we've moved past the first window, start removing characters that fall out
                     if (i >= windowSize) {
 27
 28
                         int removeIndex = s[i - windowSize] - 'a'; // Character to remove from window
 29
                         --charCount[removeIndex]; // Decrement the count for removed character
 30
 31
                         // Adjust currentFactorCount based on the new count of the removed character
 32
                         if (charCount[removeIndex] == count) {
 33
                             ++currentFactorCount;
 34
                         } else if (charCount[removeIndex] == count - 1) {
 35
                             --currentFactorCount;
 36
 37
 38
                     // If currentFactorCount equals substringFactor, it means we have a valid substring
 39
                     totalSubstrings += (substringFactor == currentFactorCount);
 40
```

### if (startIndex >= 0) { 29 30 const oldCharIndex: number = s.charCodeAt(startIndex) - 'a'.charCodeAt(0); 31 --charCount[oldCharIndex]; 32 validCount += charCount[oldCharIndex] === count ? 1 : 0; 33 validCount -= charCount[oldCharIndex] === count - 1 ? 1 : 0; 34

return answer;

Time and Space Complexity

## alphabet, and it breaks early if the count multiplied by the current index exceeds the length of the string s. The outer loop runs at most 26 times (for each possible unique character count x).

**Time Complexity** 

The time complexity, therefore, would be O(26 \* n) which simplifies to O(n), where n is the length of string s. However, if we delve deeper, we may notice that the computations depend not only on n but also on m which is count \* x. The inner

loop performs work proportional to both n and the window size determined by m. So, even though we iterate over the string once, we

• Within the inner loop, operations such as incrementing Counter values, conditional checks, and +/- operations are constant time.

The provided Python code involves nested loops where the outer loop iterates over a range determined by the input string length

and the constant count. Specifically, the outer loop can run up to 26 times representing the number of unique letters in the English

must consider the cost of maintaining the sliding window. Therefore, the time complexity is more accurately represented as O(n \* x), where x can be up to a maximum of n / count.

The inner loop runs linearly with the length of the string s, processing each character exactly once.

- **Space Complexity**
- The Counter object holds at most 26 keys at any time (for each letter of the English alphabet), but the number of keys will actually correspond to the number of unique characters in each considered substring of s, bound by m.

Counter is capped by the alphabet size, which is a constant. In conclusion, the code has a linear time complexity with respect to the size of the string 's' and a constant space complexity.

Thus, the space complexity is primarily determined by the Counter object, which is O(1) because the number of unique characters in

character we remove and modify the y accordingly to reflect the current state of our window.

- letter. Here's a breakdown of how we can approach the problem:
- 2. We initialize a counter called ans that will keep track of the total number of equal count substrings found.

- The intuition behind the solution is to leverage the sliding window technique combined with frequency counting for every unique
- count. We then need to return the total count of such substrings. Given the nature of strings and substrings, the problem might appear complex because there can be a lot of substrings to consider, especially as the length of the string increases. We are asked to calculate the number of substrings that meet the criteria, not to list them, which slightly simplifies the task.
- criteria. 5. We use a sliding window method in combination with a counter cont to keep track of the occurrences of each character as we slide through the string.

6. By moving the right edge of the window, we add characters to the cot and keep track of how many characters meet the count

7. Once the window reaches the size m, we start moving the left edge to remove characters from the cnt. Adjust the counts for the

- exceeds the string length, we can break out of the loop because it is no longer possible for any further substrings to meet the
- 3. Begin by iterating over a range from 1 to 27, which corresponds to the possible number of unique characters in a substring (as
- 4. For each x, we calculate m, the minimum length a substring needs to be to contain x unique characters count times each. If m

- In the given LeetCode problem, we need to deal with a string s that is composed of lowercase English letters, and we are provided with an integer count. The problem's central concept is to identify "equal count substrings." An equal count substring is defined as a