1525. Number of Good Ways to Split a String

```
Medium
          Bit Manipulation
                                   Dynamic Programming
                           String
```

In this problem, we are given a string s. Our task is to determine the number of ways we can split this string into two non-empty substrings s_left and s_right such that their concatenation adds back to the original string s (i.e., s_left + s_right = s) and the number of unique characters in s_left is the same as the number of unique characters in s_right. A split that satisfies these conditions is called a *good split*. We need to return the total count of such good splits.

Intuition

Problem Description

To arrive at the solution, we can use a two-pointer technique that counts the number of unique characters in the left and right parts of the string incrementally. We can start by counting the distinct letters in the entire string s and create a set to keep track of the distinct letters we have seen so far as we iterate through the string from left to right.

• We decrement the count of c in the total character count, which essentially represents the right part of the split (s_right). • If the count of c after decrementing becomes zero, it means that there are no more occurrences of c in the right part (s_right), and we can

For each character c in the string s, we perform the following steps:

remove c from the character count for the right part. • After each character is processed, we check if the size of the visited set (number of unique characters in s_left) is the same as the number of

• We add the character c to the set of visited (or seen) characters, which represents the left part of the split (s_left).

characters remaining in s_right. If they are equal, we have found a good split, and we increment our answer (ans) by one. By the end of this process, ans will hold the total number of good splits that can be made in string s.

character in the string, which we'll use to keep track of characters in the right part of the split (s_right).

- **Solution Approach**
- The implementation of the solution follows these steps: Initialize a Counter object from Python's collections module for string s. This Counter object will hold the count of each

Create an empty set named vis to track the distinct characters we have encountered so far, which represents the left part of the split (s_left).

- Set an answer variable ans to zero. This variable will count the number of good splits. Iterate through each character c in the string s:
- Add the current character c to the vis set, indicating that the character is part of the current s_left. • Decrement the count of character c in the Counter object, reflecting that one less of the character c is left for s_right.
- by the size of the Counter. If they are the same, it means we have an equal number of distinct characters in s_left and s_right, and thus,
- increment ans by one.

approach O(n), where n is the length of the string.

def numSplits(self, s: str) -> int:

cnt.pop(c)

return ans

Example Walkthrough

character in s.

ans to 0.

to 2.

are not equal, so ans remains 2.

Solution Implementation

from collections import Counter

def numSplits(self, s: str) -> int:

visited_chars.add(char)

char_count[char] -= 1

if char_count[char] == 0:

del char_count[char]

char_count = Counter(s)

visited_chars = set()

Python

class Solution:

ans += len(vis) == len(cnt)

from collections import Counter

class Solution:

Here is the implementation encapsulated in the class Solution:

Looping through every character in the string

- If the updated count of character c in the Counter becomes zero (meaning c no longer exists in s_right), remove c from the Counter to keep the counts and distinct elements accurate for remaining s_right. Evaluate if there is a good split by comparing the size of the vis set with the number of remaining distinct characters in s_right as denoted
- After the for loop completes, return the value of ans. Throughout this process, we are using a set to keep track of the unique characters we've seen which is an efficient way to ensure

characters from right to left by iterating through the string, effectively keeping a live count of what remains on each side of the

we only count distinct letters. Utilizing a Counter allows us to accurately track the frequency of characters as we 'move'

- split. Comparing the lengths of the set and the Counter keys at each step allows us to check if a good split has been achieved without needing to recount characters each time. The solution is efficient because it only requires a single pass through the string s, which makes the time complexity of this
- cnt = Counter(s) # Initial count of all characters in `s` # Set to keep track of unique characters seen in `s_left` # Counter for number of good splits ans = 0

for c in s: vis.add(c) cnt[c] -= 1 **if** cnt[c] == 0:

Imagine the string s is "aacaba". We need to calculate the number of good splits for this string.

'b': 1}). The lengths of vis and cnt are now equal (2 each), so we increment ans to 1.

```
As we iterate through the string:
   For the first character 'a', we add it to vis (now vis is {'a'}) and decrement its count in cnt (now cnt is {'a': 3, 'c':
   1, 'b': 1). The lengths of vis and cnt are not equal, so ans remains 0.
   Moving to the second character 'a', we add it to vis (which remains {'a'} since 'a' is already included) and decrement its
   count in cnt (now cnt is {'a': 2, 'c': 1, 'b': 1}). The lengths of vis and cnt are still not equal, so ans remains 0.
```

Now we come to the third character, 'c'. We add 'c' to vis (now vis is {'a', 'c'}) and decrement its count in cnt (now

cnt is {'a': 2, 'c': 0, 'b': 1}), and since the count of 'c' has reached 0, we remove 'c' from cnt (now cnt is {'a': 2,

Then we process the fourth character 'a'. After adding 'a' to vis (which remains {'a', 'c'}) and decrementing its count

in cnt (now cnt is {'a': 1, 'b': 1}), we find that the lengths of vis and cnt are still equal (2 each), so we increment ans

The fifth character is 'b'. We add 'b' to vis (now vis is {'a', 'c', 'b'}) and decrement its count in cnt (now cnt is {'a':

Finally, we process the last character 'a'. We add 'a' to vis (which remains {'a', 'c', 'b'}) and decrement its count in

cnt (now cnt is {'a': 0}), and then remove 'a' from cnt since its count is 0 (now cnt is empty). The lengths of vis and cnt

First, we create a counter from the whole string s which will give us {'a': 4, 'c': 1, 'b': 1} showing the counts of each

We then initialize the set vis to keep track of the unique characters seen in s_left (initially empty) and set our answer count

- 1, 'b': 0}), and remove 'b' from cnt since its count is now 0 (now cnt is {'a': 1}). Lengths of vis and cnt are not equal, so ans remains 2.
- same, the value of ans is 2. Therefore, the total number of good splits for the string "aacaba" is 2.

After the loop finishes, since there were two points where the count of unique characters in s_left and s_right were the

Initialize the count for valid splits to 0 good_splits = 0 # Iterate over each character in the string for char in s:

in the visited characters and remaining characters are the same good_splits += len(visited_chars) == len(char_count) # Return the total number of good splits return good_splits

// Set to keep track of unique characters encountered so far

// Add the current character to the set, indicating it's been seen

// A good split is found when the size of the set (unique characters in the left part)

// is equal to the size of the remaining map (unique characters in the right part)

if (frequencyMap.merge(character, -1, Integer::sum) == 0) {

if (uniqueCharsSeen.size() == frequencyMap.size()) {

Set<Character> uniqueCharsSeen = new HashSet<>();

// Iterate through the characters of the string

frequencyMap.remove(character);

// Initialize the count of good splits to 0

for (char character : s.toCharArray()) {

uniqueCharsSeen.add(character);

goodSplitsCount++;

// Return the count of good splits

// Import relevant classes from TypeScript's collection libraries

const charFrequency: HashMap<string, number> = new HashMap();

charFrequency.set(c, (charFrequency.get(c) || 0) + 1);

const uniqueCharsSeen: HashSet<string> = new HashSet();

// This set will store the unique characters we've encountered so far

// Add the current character to the set of seen unique characters

charFrequency.set(c, currentFrequency - 1); // Update with the decremented count

let goodSplits: number = 0; // Initialize the count of good splits

// Create a frequency map to count the occurrences of each character in the string

// Function to count the number of good splits in a string

return goodSplits;

function numSplits(s: string): number {

// Iterate through the string

uniqueCharsSeen.add(c);

charFrequency.delete(c);

for (const c of s) {

for (const c of s) {

Count the frequency of each character in the string

Initialize a set to keep track of unique characters visited so far

Add the character to the set of visited characters

Decrement the frequency count of the current character

Remove the character from the counter if its frequency becomes 0

Increment the count of valid splits if the number of unique characters

```
// Map to store the frequency of each character in the input string
Map<Character, Integer> frequencyMap = new HashMap<>();
// Populate the frequency map with the count of each character
for (char character : s.toCharArray()) {
    frequencyMap.merge(character, 1, Integer::sum);
```

public int numSplits(String s) {

int goodSplitsCount = 0;

Java

class Solution {

```
// Return the total number of good splits found
       return goodSplitsCount;
C++
#include <unordered_map>
#include <unordered_set>
#include <string>
class Solution {
public:
    int numSplits(string s) {
       // Count the frequency of each character in the string
        std::unordered_map<char, int> charFrequency;
        for (char& c : s) {
            ++charFrequency[c];
       // This set will store unique characters we've seen so far as we iterate
       std::unordered_set<char> uniqueCharsSeen;
        int goodSplits = 0; // This will hold the count of good splits
       // Iterate through the string once
        for (char& c : s) {
            // Insert the current character into the set of seen characters
            uniqueCharsSeen.insert(c);
           // If the frequency of the character reaches zero after decrementing, erase it
            if (--charFrequency[c] == 0) {
                charFrequency.erase(c);
            // Increase the count of good splits if the number of unique characters
            // seen so far is equal to the number of unique characters remaining
            goodSplits += uniqueCharsSeen.size() == charFrequency.size();
```

import { HashMap, HashSet } from './collections'; // This line assumes there is a 'collections' module available to import these

// Decrease the frequency count of the current character and remove it from the map if the count reaches zero

```
// Decrement the frequency of the character. If it reaches zero, remove it from the map
const currentFrequency = charFrequency.get(c) || 0;
if (currentFrequency - 1 === 0) {
} else {
```

};

TypeScript

```
// Increases the goodSplits counter if the number of unique characters seen is
          // equal to the number of unique characters that remain in the frequency map
          if (uniqueCharsSeen.size() === charFrequency.size()) {
              goodSplits++;
      // Return the total number of good splits found in the string
      return goodSplits;
from collections import Counter
class Solution:
   def numSplits(self, s: str) -> int:
       # Count the frequency of each character in the string
        char_count = Counter(s)
       # Initialize a set to keep track of unique characters visited so far
       visited_chars = set()
       # Initialize the count for valid splits to 0
        good_splits = 0
       # Iterate over each character in the string
        for char in s:
           # Add the character to the set of visited characters
            visited chars.add(char)
            # Decrement the frequency count of the current character
            char_count[char] -= 1
           # Remove the character from the counter if its frequency becomes 0
            if char_count[char] == 0:
               del char_count[char]
           # Increment the count of valid splits if the number of unique characters
           # in the visited characters and remaining characters are the same
            good_splits += len(visited_chars) == len(char_count)
       # Return the total number of good splits
        return good_splits
```

The given code snippet involves iterating over each character of the string s precisely once. Within this single iteration, the operations performed involve adding elements to a set, updating a counter (a dictionary under the hood), checking for equality of lengths, and incrementing an answer counter.

Adding elements to the vis set has an average case time complexity of 0(1) per operation.

Time and Space Complexity

Time Complexity

value pairs.

have an average case time complexity of 0(1). • The equality check len(vis) == len(cnt) is 0(1) because the lengths can be compared directly without traversing the structures. Thus, we have an average case time complexity of O(n), where n is the length of the string s.

• Updating the counts in the Counter and checking if a count is zero is also 0(1) on average for each character because dictionary operations

Space Complexity The space complexity is determined by the additional data structures used:

• A Counter object to store the frequency of each character in s. In the worst case, if all characters in s are unique, the counter would hold n key-

• A set object to keep track of the characters that we have seen as we iterate. This could also hold up to n unique characters in the worst case. Both the Counter and the set will have a space complexity of O(n) in the worst case. Therefore, the overall space complexity is 0(n).