#### 1781. Sum of Beauty of All Substrings

frequency of 1. So, the beauty of the string "abaacc" is 3 - 1 = 2.

substrings of the input string. Let's delve into the specifics:

Medium Hash Table String Counting

#### Problem Description

of occurrences of the most frequent character and the least frequent character in that string. We are to find the beauty of every possible substring of the given string and then sum up these beauties to get a final answer.

Substrings of a string are the sequences of characters that can be derived from the string by deleting some characters (possibly

The problem asks us to calculate the "beauty" of a string. The beauty of a string is defined as the difference between the number

Substrings of a string are the sequences of characters that can be derived from the string by deleting some characters (possibly, none) from the beginning or end of the string. For instance, "ab", "b", "baa", etc. are all substrings of the string "abaacc".

To further clarify the problem, let's take the example given in the problem description: For the string "abaacc", the beauty of this string is calculated by looking at the frequency of each character. The character 'a' appears 3 times, 'b' once, and 'c' twice. The most frequent character is 'a', with a frequency of 3, and the least frequent (excluding characters not present at all) is 'b', with a

What the problem ultimately requires us to do is calculate such beauties for all substrings of the input string 's' and return their sum.

Intuition

#### The solution adopts a brute-force approach to finding all possible substrings and calculating the beauty of each. Breaking down our approach intuitively:

We will iterate over every possible starting point for a substring within the string 's'. This is done by a loop indexed by 'i' going from the start to the end of 's'.
 For each starting point 'i', we will then iterate over all possible ending points. These are determined by the loop indexed by 'j',

- which goes from 'i' to the end of 's'.
- 3. As we expand our substring from 'i' to 'j', we will keep track of the frequencies of the characters appearing in the substring using a Counter (which is essentially a specialized dictionary or hash map provided by Python's collections library).
- substring by finding the frequency of the most and least frequent characters in our Counter. This is the difference between max(cnt.values()) and min(cnt.values()).

  5. We add this beauty to a running total 'ans', which, at the end of the process, will contain the sum of beauties of all substrings.

At each iteration, as we increase the size of the current substring by extending 'j', we calculate the beauty of the new

- Solution Approach

  The implementation of the provided solution follows a brute-force approach and iteratively calculates the beauty of all possible
- 1. **Two Nested Loops**: The algorithm uses two nested loops, which enables it to consider every possible substring of the input

#### end.

2. **Counter:** A Counter from Python's collections library is used to track character frequencies within the current substring. This data structure allows us to efficiently keep a tally as we extend our substring by adding characters one by one.

**Updating Counter**: Within the inner loop, the counter is updated every time a new character is added to the substring:

cnt[s[j]] += 1. This line increments the count of the character at the current end position j.

answer. This occurs in the expression ans += max(cnt.values()) - min(cnt.values()).

accumulated beauty of all substrings—is returned as the result of the function.

string. The outer loop (indexed by i) represents the start of the substring, while the inner loop (indexed by j) represents the

and minimum values of cnt. This is executed by max(cnt.values()) - min(cnt.values()). It represents the frequency of the most common character minus the frequency of the least common character in the current substring.

5. Summing Up the Beauties: The beauty found at each step is then added to a running total ans, which eventually becomes the

Calculating Beauty: After each update to the counter, the beauty of the new substring is determined by finding the maximum

This algorithm is straightforward but not particularly efficient, as it has a time complexity of O(n^3) considering that there are n\* (n+1)/2 possible substrings and for each substring we are computing the beauty in O(n) time. This is an exhaustive method that guarantees the correct summation of the beauties for all substrings but might not scale well for large strings due to its polynomial

Returning the Result: After all iterations, the outer loop concludes, and the final value of ans—which, at this point, holds the

Example Walkthrough

For this string "abc", the possible substrings along with their beauties (difference between most frequent and least frequent

character counts) will be:

1. "a" → Beauty: 0 (since there is only one character)

2. "b" → Beauty: 0 (since there is only one character)

3. "c" → Beauty: 0 (since there is only one character)

### Now, following the solution approach steps: 1. **Two Nested Loops**: We start with the outer loop where i goes from 0 to 2 (the length of the string - 1) to take each character

of the current character.

to the running total ans.

Solution Implementation

from collections import Counter

for i in range(string\_length):

char\_counter = Counter()

for j in range(i, string\_length):

total\_beauty += current\_beauty

// This function calculates the beauty sum of a string.

int n = s.size(); // Get the size of the string.

for (int end = start; end < n; ++end) {</pre>

int minFreq = 1000, maxFreq = 0;

for (int count : charCounts) {

// const result: number = beautySum("yourStringHere");

# Initialize the sum to store the total beauty of all substrings

# Increment the count of the current character

# Iterate through each character in the string as the starting point

for (int start = 0; start < n; ++start) {</pre>

++charCounts[s[end] - 'a'];

int sum = 0; // Initialize sum to store the beauty sum result.

// Increment the count of the current character.

minFreq = min(minFreq, count);

maxFreq = max(maxFreq, count);

int charCounts[26]; // Array to count occurrences of each character (a-z).

// Explore all substrings starting at 'start' and ending at 'end'.

// Initialize max and min occurrences of characters found so far.

// Iterate over the counts to find the max and min frequencies.

// Iterate over the string starting with substrings of length 1 to n.

# Return the total beauty of all substrings

char\_counter[s[j]] += 1

**Python** 

time complexity.

as the starting point. For each value of i, we enter the inner loop, where j also ranges from i to 2.

6. "abc" → Beauty: 0 (all characters 'a', 'b', and 'c' occur exactly once)

4. "ab" → Beauty: 0 (both 'a' and 'b' occur exactly once)

5. "bc" → Beauty: 0 (both 'b' and 'c' occur exactly once)

Let's take a small string "abc" to illustrate the solution approach.

new substring.

3. **Updating Counter:** For each pair (i, j), we increment the count of s[j] in our Counter by 1, thereby updating the frequency

Counter: We initialize an empty Counter object cnt at the start of each iteration of the outer loop because we are starting a

**Calculating Beauty**: We then calculate the beauty of this particular substring as max(cnt.values()) - min(cnt.values()).

Summing Up the Beauties: For each new substring, the calculated beauty (which is always 0 for our example "abc") is added

Since all characters in our substrings occur at most once, this beauty is always 0 in the case of the example string "abc".

6. **Returning the Result**: After all iterations of both loops, we conclude that the sum of the beauties is 0 since all our substrings have characters appearing only once.

Therefore, for the input string "abc", our function would return the sum of the beauties of all substrings, which is 0 in this case.

- Remember, this method does indeed scale poorly for larger strings, as it must compute the beauty of each substring individually, which becomes exponentially greater as the length of the string increases.
- class Solution:
   def beautySum(self, s: str) -> int:
   # Initialize the sum to store the total beauty of all substrings
   total\_beauty = 0
   # Get the length of the string
   string\_length = len(s)

# Create a counter to keep track of the frequency of each character in the current substring

# Iterate from the current character to the end of the string to form substrings

current\_beauty = max(char\_counter.values()) - min(char\_counter.values())

# Calculate the beauty of the current substring by finding the

# difference between the max and min frequency of characters

# Add the beauty of the current substring to the total beauty

# Iterate through each character in the string as the starting point

# Increment the count of the current character

## Java

class Solution {

class Solution {

int beautySum(string s) {

public:

return total\_beauty

```
public int beautySum(String s) {
    int totalBeauty = 0; // This will hold the cumulative beauty sum
    int stringLength = s.length(); // Store the length of string s
   // Outer loop to go through the substring starting points
    for (int i = 0; i < stringLength; ++i) {</pre>
        int[] frequencyCount = new int[26]; // Frequency array to count letters in the substring
        // Inner loop to go through the substrings ending with character at position j
        for (int j = i; j < stringLength; ++j) {</pre>
            // Increment the frequency count of current character
            ++frequencyCount[s.charAt(j) - 'a'];
            // Set initial max and min frequency of characters. 1000 is assumed to be greater
            // than any possible frequency thus a decent starting value for the minimum.
            int minFrequency = 1000, maxFrequency = 0;
            // Loop through the frequency count array to find the highest and lowest frequency
            // that is greater than zero (character is present)
            for (int freq : frequencyCount) {
                if (freq > 0) {
                    minFrequency = Math.min(minFrequency, freq);
                    maxFrequency = Math.max(maxFrequency, freq);
            // Beauty is calculated as the difference between max and min frequency of
            // characters in the substring. Add this to totalBeauty.
            totalBeauty += maxFrequency - minFrequency;
   // Return the cumulative beauty of all substrings
   return totalBeauty;
```

```
// Add the beauty (difference between max and min frequency) of this substring to the sum.
                sum += maxFreq - minFreq;
        // Return the total beauty sum of all substrings.
        return sum;
TypeScript
/**
* Calculates the sum of beauty of all of its substrings.
 * @param {string} s - The string to process.
* @return {number} - The sum of beauty of all substrings.
*/
const beautySum = (s: string): number => {
    let beautySumResult: number = 0;
    // Iterate through the string
    for (let i = 0; i < s.length; ++i) {</pre>
       // Keep track of the frequency of each character
        const frequencyCounter: Map<string, number> = new Map();
        // Consider all substrings starting with the character at index 'i'
        for (let j = i; j < s.length; ++j) {</pre>
            // Increment the frequency of the current character
            frequencyCounter.set(s[j], (frequencyCounter.get(s[j]) || 0) + 1);
            // Extract frequency values from the map to determine beauty
            const frequencies: number[] = Array.from(frequencyCounter.values());
            // The beauty of the substring is defined by the difference
            // between the max and min frequency chars
            beautySumResult += Math.max(...frequencies) - Math.min(...frequencies);
    // Return the total beauty sum of all substrings
    return beautySumResult;
};
// The function can be used as follows:
```

# Create a counter to keep track of the frequency of each character in the current substring

# Iterate from the current character to the end of the string to form substrings

current\_beauty = max(char\_counter.values()) - min(char\_counter.values())

# Calculate the beauty of the current substring by finding the

# difference between the max and min frequency of characters

# Add the beauty of the current substring to the total beauty

memset(charCounts, 0, sizeof charCounts); // Reset character counts for each new starting point.

if (count > 0) { // Only consider characters that appear in the substring.

# Time and Space Complexity

return total\_beauty

from collections import Counter

total\_beauty = 0

 $string_length = len(s)$ 

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

# Get the length of the string

for i in range(string\_length):

char counter = Counter()

for j in range(i, string\_length):

total\_beauty += current\_beauty

# Return the total beauty of all substrings

char\_counter[s[j]] += 1

class Solution:

## inner loop (indexed by j) iterating over the endpoints extending from the current starting point. For each inner iteration, it updates the Counter and calculates the beauty of the current substring.

**Time Complexity** 

The outer loop runs n times (where n is the length of s). For each iteration of i, the inner loop runs up to n-i times. In the worst case, where i is 0, the inner loop runs n times, and in the best case, where i is n-1, it runs once.

The provided code has two nested loops: the outer loop (indexed by i) iterating over the starting points in the string s, and the

The update and calculation within the inner loop take O(k) time in the worst case, where k is the number of distinct characters in since the Counter needs to iterate over all the keys to find the max and min values.

Therefore, the total time complexity is  $0(n^2 * k)$  where n is the length of the string and k is the number of unique characters.

# Space Complexity The space complex

The space complexity is dictated by the Counter which stores the frequency of each character in the current substring.

In the worst case, the substring could contain all unique characters of the string. Hence, the space complexity is 0(k) where k is the number of unique characters in the string s.