1781. Sum of Beauty of All Substrings

Medium Hash Table String Counting

Leetcode Link

Problem Description

The problem asks us to calculate the "beauty" of a string. The beauty of a string is defined as the difference between the number 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, 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 frequency of 1. So, the beauty of the string "abaacc" is `3 - 1 = 2`.

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:

1. 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'.
2. 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).
4. At each iteration, as we increase the size of the current substring by extending 'j', we calculate the beauty of the new 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.

The brute-force solution may not be the most efficient, especially for very long strings, because the number of substrings grows rapidly. However, it is a straightforward method that guarantees a correct solution by exhaustively checking every option.

Solution Approach

The implementation of the provided solution follows a brute-force approach and iteratively calculates the beauty of all possible substrings of the input string. Let's delve into the specifics:

1. **Two Nested Loops**: The algorithm uses two nested loops, which enables it to consider every possible substring of the input string. The outer loop (indexed by `i`) represents the start of the substring, while the inner loop (indexed by `j`) represents the 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.
3. **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`.
4. **Calculating Beauty**: After each update to the counter, the beauty of the new substring is determined by finding the maximum 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 answer. This occurs in the expression `ans += max(cnt.values()) - min(cnt.values())`.
6. **Returning the Result**: After all iterations, the outer loop concludes, and the final value of `ans`—which, at this point, holds the accumulated beauty of all substrings—is returned as the result of the function.

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 time complexity.

Example Walkthrough

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

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)
4. "ab" -> Beauty: 0 (both 'a' and 'b' occur exactly once)
5. "bc" -> Beauty: 0 (both 'b' and 'c' occur exactly once)
6. "abc" -> Beauty: 0 (all characters 'a', 'b', and 'c' occur exactly once)

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 as the starting point. For each value of `i`, we enter the inner loop, where `j` also ranges from `i` to 2.
2. **Counter**: We initialize an empty Counter object `cnt` at the start of each iteration of the outer loop because we are starting a 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 of the current character.
4. **Calculating Beauty**: We then calculate the beauty of this particular substring as `max(cnt.values()) - min(cnt.values())`. Since all characters in our substrings occur at most once, this beauty is always 0 in the case of the example string "abc".
5. **Summing Up the Beauties**: For each new substring, the calculated beauty (which is always 0 for our example "abc") is added to the running total `ans`.
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.

Python Solution

```python
from collections import Counter

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)

        # Iterate through each character in the string as the starting point
        for i in range(string_length):
            # Create a counter to keep track of the frequency of each character in the current substring
            char_counter = Counter()
            # Iterate from the current character to the end of the string to form substrings
            for j in range(i, string_length):
                # Increment the count of the current character
                char_counter[s[j]] += 1
                # Calculate the beauty of the current substring by finding the
                # difference between the max and min frequency of characters
                current_beauty = max(char_counter.values()) - min(char_counter.values())
                # Add the beauty of the current substring to the total beauty
                total_beauty += current_beauty
        # Return the total beauty of all substrings
        return total_beauty

```

Java Solution

```java
class Solution {
    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) {
            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) {
                // 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;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // This function calculates the beauty sum of a string.
    int beautySum(string s) {
        int sum = 0; // Initialize sum to store the beauty sum result.
        int n = s.size(); // Get the size of the string.
        int charCounts[26]; // Array to count occurrences of each character (a-z).

        // Iterate over the string starting with substrings of length 1 to n.
        for (int start = 0; start < n; ++start) {
            memset(charCounts, 0, sizeof charCounts); // Reset character counts for each new starting point.
            // Explore all substrings starting at 'start' and ending at 'end'.
            for (int end = start; end < n; ++end) {
                // Increment the count of the current character.
                ++charCounts[s[end] - 'a'];

                // Initialize max and min occurrences of characters found so far.
                int minFreq = 1000, maxFreq = 0;
                // Iterate over the counts to find the max and min frequencies.
                for (int count : charCounts) {
                    if (count > 0) { // Only consider characters that appear in the substring.
                        minFreq = min(minFreq, count);
                        maxFreq = max(maxFreq, count);
                    }
                }
                // 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 Solution

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

```

Time and Space Complexity

Time Complexity

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

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

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

Space Complexity

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 `O(k)` where `k` is the number of unique characters in the string `s`.