2272. Substring With Largest Variance

Hard | Array | Dynamic Programming | Leetcode Link

Problem Description

The problem focuses on finding the largest variance within any substring of a given string `s`, which contains only lowercase English letters. The variance of a string is defined as the largest difference in the number of occurrences between any two distinct characters in it. It's important to consider that the characters could be the same or different.

To clarify, let's say we're given a string like "abcde". In this string, each character appears exactly once, so the variance between any two characters is zero. However, if we had a string like "aabbcc", the variance could be higher. If we choose the substring "aab", the variance would be 2 because there are two 'a' characters and none of the 'b' character (or vice versa). We need to consider all possible substrings to find the one with the largest variance.

Intuition

The intuition behind the solution lies in realizing that the variance could only be maximized if we consider pairs of different characters in the string, as those pairs are the ones that can actually have a non-zero variance. So the first step is to iterate over all possible pairs of characters (permutations of two distinct letters from the 26 lowercase English characters).

Once a pair of characters is selected (let's say 'a' and 'b'), we apply a two-pass scanning algorithm on the string to track the difference in the occurrence counts between 'a' and 'b'. We keep two counters: one (`f[0]`) resets every time we encounter an 'a' after a 'b', while the other (`f[1]`) continues to accumulate or starts over from `f[0] - 1` when a 'b' is encountered after at least one 'a'. We look to maximize the difference (`f[1]`), which is essentially the variance, while scanning.

In the end, we take the maximum recorded value during all scans as the answer. This approach works because by focusing on pairs of characters at a time, we are able to isolate the potential substrings that can provide the maximum variance, rather than getting lost in the overwhelming number of all possible substrings.

Along with tracking the occurrences of 'a' and 'b', there's also an understanding that we include a subtraction operation when we encounter a 'b', making sure we are actually counting a valid substring with at least one 'a' when 'b' appears, since variance requires both characters to be present. A `-inf` is used to handle cases where a valid substring does not exist yet. By considering every single position in the string as a potential starting or ending point for substrings with maximum variance between two specific characters, the algorithm guarantees that no possible substring is overlooked, thus ensuring the correctness of the solution.

Solution Approach

The implementation employs a brute force strategy enhanced with a smart iteration mechanism and memory optimization to calculate the maximum variance for all substrings of the given string. The key elements of this approach involve permutations, two comparison variables, and a linear scan through the string.

Permutations of Characters

Firstly, the algorithm generates all possible permutations of two distinct characters from the set of lowercase English letters. This is done using `permutations(ascii_lowercase, 2)` which loops through all pairs of two distinct characters since we're looking for the difference in occurrences between two potentially different characters.

Tracking Variance with Two Counters

In the solution, an array `f` with two elements is used:

- `f[0]` serves as a resettable counter for occurrences of character `a` since the last occurrence of `b`.
- `f[1]` keeps track of the accumulated variance between `a` and `b`.

This dichotomy allows the algorithm to keep track of two scenarios simultaneously:

- Accumulating occurrences of `a` when no `b` has interfered (handled by `f[0]`).
- Capturing the 'variance' until a reset is needed due to a `b` occurrence (handled by `f[1]`).

Both `f[0]` and `f[1]` are reset to zero if a `b` appears directly after a `b`. However, when an `a` is followed by a `b`, `f[1]` takes the role of capturing the current variance by taking either the current accumulated variance minus one, or the variance just after the last `a` minus one, determined by `max(f[1] - 1, f[0] - 1)`.

This handling ensures that at least one `a` is present before a `b` is counted, which is necessary for a valid substring.

Maximal Variance Calculation

As we scan the string `s`, for each character `c`, the algorithm performs the following actions:

- If `c` is `a`, we increment both `f[0]` and `f[1]`.
- If `c` is `b`, we decrement `f[1]` and choose the larger of the two: `f[1]` or `f[0] - 1`, effectively either continuing with the current substring or starting a new substring that excludes the last `a`. `f[0]` is reset to zero as we can't start a new substring with just `b`.

After each update, we check if `f[1]` exceeds the current `ans`. If it does, we update `ans` to the value of `f[1]`. The algorithm avoids false starts with `-inf`, ensuring that `f[1]` only records valid substrings.

The loop through permutations of characters ensures that we consider every potential character disparity. The nested loop through the string ensures that we consider every potential substring for each character pair. Eventually, `ans` will contain the maximum variance achievable among all substrings of `s`.

The space complexity of the solution is `O(1)` as we only maintain a few variables and a pair of counters, and the time complexity is `O(26*26*n)` where `n` is the length of the string, due to the nested loops—26 for each character in the permutations, and `n` for the iterations over the string.

Example Walkthrough

Let's illustrate the solution approach using a small example. Suppose we are given the string `s = "abbab"`. Our task is to find the largest variance within any substring of `s`.

Firstly, we generate all possible permutations of two distinct characters from the set of lowercase English letters. For simplicity, let's consider only a few pairs here, such as ('a', 'b') and ('b', 'a'), since these are the only characters in our example string.

We initiate a variable `ans` to store the maximum variance found. Initially, `ans = -inf`.

Pair ('a', 'b'):

1. Initialize the counters `f[0]` and `f[1]` to 0.
2. Loop through each character `c` in `s = "abbab"`:
   - For `c = 'a'`: Increment both `f[0]` and `f[1]` (since we initially consider the occurrence of 'a' without 'b' involved).
   - For `c = 'b'`: Decrement `f[1]` and then set `f[1]` to the maximum of `f[1]` and `f[0] - 1`. Reset `f[0]` to 0.

Let's go through the string "abbab":

- Start with 'a': `f[0] = 1`, `f[1] = 1`.
- Next, 'b': `f[1] = max(f[1] - 1, f[0] - 1) = 0`, `f[0] = 0`. Update `ans = max(ans, f[1]) = 1`.
- Next, 'b': `f[1] = max(f[1] - 1, f[0] - 1) = -1`, since there are no 'a' in front, we don't update `ans`.
- Next, 'a': `f[0] = 1`, `f[1] = max(f[1] + 1, f[0]) = 1`.
- Last, 'b': `f[1] = max(f[1] - 1, f[0] - 1) = 0`. Update `ans = max(ans, f[1]) = 1`.

Pair ('b', 'a'):

Following the same steps but reversing the roles of 'a' and 'b', we would not find a variance larger than what we already have from the pair ('a', 'b').

After considering these permutations, we find that the maximum variance is `1`, which we may have obtained from the substring "ab" or "ba" in the string "abbab".

In a complete implementation, we would evaluate all permutations of character pairs, but in this small example with only 'a' and 'b' present, it is unnecessary.

The process effectively scans through every substring of `s` and calculates the variance for pairs of characters, updating `ans` when a higher variance is found. The time complexity is mainly driven by the nested loops: there are 26*26 permutations of character pairs, and for each pair, we scan the entire string `s` of length `n`, resulting in an `O(26*26*n)` time complexity.

Python Solution

```python
from itertools import permutations
from string import ascii_lowercase
from math import inf

class Solution:
    def largestVariance(self, s: str) -> int:
        # Initialize the answer to 0
        max_variance = 0

        # Iterate through all ordered pairs of distinct lowercase ascii characters
        for char_a, char_b in permutations(ascii_lowercase, 2):

            # Initialize the frequency counters for the current pair, where:
            # freq_before_b records the number of 'a' characters seen so far without any 'b' in between
            # max_freq_after_b records the maximum frequency difference after at least one 'b' was found
            freq_before_b = 0
            max_freq_after_b = -inf

            # Iterate over each character in the string
            for current_char in s:
                if current_char == char_a:
                    # Increment the frequency counter of 'a' when 'a' is found
                    freq_before_b += 1
                    max_freq_after_b += 1
                elif current_char == char_b:
                    # Decrement the frequency counter of 'b' when 'b' is found; reset freq_before_b
                    max_freq_after_b = max(max_freq_after_b - 1, freq_before_b - 1)
                    freq_before_b = 0

                # Update the max_variance if we find a new maximum
                if max_variance < max_freq_after_b:
                    max_variance = max_freq_after_b

        # Return the largest variance found
        return max_variance

```

Java Solution

```java
class Solution {
    public int largestVariance(String s) {
        // Length of the input string.
        int length = s.length();
        // Variable to store the maximum variance found so far.
        int maxVariance = 0;

        // Iterate over all possible pairs of characters (a and b are different).
        for (char firstChar = 'a'; firstChar <= 'z'; ++firstChar) {
            for (char secondChar = 'a'; secondChar <= 'z'; ++secondChar) {
                if (firstChar == secondChar) {
                    // If both characters are the same, skip this iteration.
                    continue;
                }

                // Array to keep track of the frequency of character 'a'
                // f[0] is the streak of 'a's, f[1] is the max variance for the current window.
                // Initialize with 0 for streak and -n for variance because variance cannot be less than -n.
                int[] frequency = new int[] {0, -length};

                // Iterate over each character in the string.
                for (int i = 0; i < length; ++i) {
                    if (s.charAt(i) == firstChar) {
                        // If the current character is 'a', increase both frequencies.
                        frequency[0]++;
                        frequency[1]++;
                    } else if (s.charAt(i) == secondChar) {
                        // If the current character is 'b', calculate the variance.
                        frequency[1] = Math.max(frequency[0] - 1, frequency[1] - 1);
                        // Reset the streak of 'a's because 'b' is encountered.
                        frequency[0] = 0;
                    }
                    // Update the maximum variance found.
                    maxVariance = Math.max(maxVariance, frequency[1]);
                }
            }
        }

        // Return the largest variance found.
        return maxVariance;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // Method to find the largest variance of a substring for a given string
    int largestVariance(string s) {
        // Length of the input string
        int n = s.size();
        // Initialize the maximum variance to 0
        int maxVariance = 0;

        // Iterate over all possible character pairs (a, b) where a != b
        for (char a = 'a'; a <= 'z'; ++a) {
            for (char b = 'a'; b <= 'z'; ++b) {
                // Skip if both characters are the same
                if (a == b) continue;

                // Initialize frequency counters for characters a and b
                // f[0] will track frequency of a, f[1] tracks max variance
                int frequencies[2] = {0, -n};

                // Iterate over each character in the string
                for (char currentChar : s) {
                    // Increment frequency count if a matches the current character
                    if (currentChar == a) {
                        frequencies[0]++;
                        frequencies[1]++;
                    }
                    // Decrement frequency count and update variance if b matches
                    else if (currentChar == b) {
                        frequencies[1] = max(frequencies[1] - 1, frequencies[0] - 1);
                        frequencies[0] = 0;
                    }
                    // Update the maximum variance found so far
                    maxVariance = max(maxVariance, frequencies[1]);
                }
            }
        }
        // Return the final maximum variance
        return maxVariance;
    }
};

```

Typescript Solution

```typescript
// Function to find the largest variance of a substring for a given string
function largestVariance(s: string): number {
    // Length of the input string
    let n: number = s.length;
    // Initialize the maximum variance to 0
    let maxVariance: number = 0;

    // Iterate over all possible character pairs (a, b) where a != b
    for (let a = 'a'.charCodeAt(0); a <= 'z'.charCodeAt(0); ++a) {
        for (let b = 'a'.charCodeAt(0); b <= 'z'.charCodeAt(0); ++b) {
            // Skip if both characters are the same
            if (a === b) continue;

            // Initialize frequency counters for characters 'a' and 'b'
            let frequencies: number[] = [0, -n];

            // Iterate over each character in the string
            for (let currentChar of s) {
                // Increment frequency count if 'a' matches the current character
                if (currentChar.charCodeAt(0) === a) {
                    frequencies[0]++;
                    frequencies[1]++;
                }
                // Decrement frequency count and update variance if 'b' matches
                else if (currentChar.charCodeAt(0) === b) {
                    frequencies[1] = Math.max(frequencies[1] - 1, frequencies[0] - 1);
                    frequencies[0] = 0;
                }
                // Update the maximum variance found so far
                maxVariance = Math.max(maxVariance, frequencies[1]);
            }
        }
    }
    // Return the final maximum variance
    return maxVariance;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the code can be analyzed based on the operations it performs:

1. The function `permutations(ascii_lowercase, 2)` generates all possible permutations of two distinct lowercase letters from the English alphabet. Since there are 26 letters in the English alphabet, there are `26 * 25` such permutations because after choosing the first letter, there are 25 remaining letters to form a pair.
2. The function then iterates through each character in the string `s` for every pair of letters (`a`, `b`). The length of the string `s` is `n`.
3. Within the nested loop, there are constant-time operations (comparisons, assignments, `max`, addition, and subtraction).

Given the above steps, the time complexity is `O(26 * 25 * n)`, which simplifies to `O(n)` because the number of permutations (`26 * 25`) is a constant.

Space Complexity

The space complexity of the code is determined by:

1. Variables `ans`, `f[0]`, and `f[1]`, which are used to keep track of the current and maximum variance.
2. No additional data structures that grow with input size are used.

Thus, the space complexity of the algorithm is `O(1)` as it only uses a constant amount of space.