2207. Maximize Number of Subsequences in a String

Medium | Greedy | String | Prefix Sum | Leetcode Link

Problem Description

You are given a string `text` which is indexed from `0`. You are also given another string `pattern`, which is of length `2` and also indexed from `0`. Both strings contain only lowercase English letters. You have the option to add one character either `pattern[0]` or `pattern[1]` to any position in the `text` string exactly once. This includes the possibility of adding the character at the start or the end of the `text`. Your task is to figure out the maximum number of times the `pattern` can be found as a subsequence in the new string after adding one character.

To clarify, a subsequence is a sequence that can be derived from another sequence by deleting some characters (or none) without altering the order of the remaining characters.

For example, given `text = "ababc"` and `pattern = "ab"`, if we add "a" to the `text` to form "aababc", we can now see the subsequence "ab" occurs 4 times (indices 0-1, 0-3, 2-3, and 2-5).

Intuition

The intuition behind the solution lies in understanding what a subsequence is and how the addition of a character impacts the count of subsequences. The key insight is that by adding a character from the `pattern` to the `text`, we can increase the occurrences of that `pattern` as a subsequence.

We can track the occurrences of `pattern[0]` and the potential matches of the `pattern` as we iterate through the `text`. Each time we encounter the second character of the `pattern` in the text (`pattern[1]`), we know that any previous occurrences of `pattern[0]` could form a new subsequence match with it. We keep a cumulative count of `pattern[0]`'s occurrences as we scan through the `text`, adding to the answer whenever we see `pattern[1]`.

Finally, we add the larger of the counts of `pattern[0]` and `pattern[1]` to `ans` because we can insert one additional character `pattern[0]` or `pattern[1]` to the text (at the best place possible), which will create the most additional subsequences.

This approach allows us to efficiently calculate the maximum number of subsequence counts possible by traversing the string only once.

Solution Approach

The solution uses a two-pass approach with a counter to keep track of occurrences of characters from the `pattern`.

1. Initialize a variable `ans` to keep track of the number of subsequences of `pattern` found in `text`.
2. Create a `Counter` object named `cnt` that will hold the frequency of each character we encounter in `text`.
3. Iterate over each character `c` in the `text` string.
   - If the current character `c` is the same as the second character in `pattern` (`pattern[1]`), we increment `ans` by the number of times the first character of `pattern` (`pattern[0]`) has appeared so far. This is because each occurrence of `pattern[0]` before `pattern[1]` could form a new valid subsequence with `pattern[1]`.
   - Update `cnt[c]` by incrementing it by `1` to keep the count of each character.
4. After the iteration is complete, add to `ans` the maximum frequency between `cnt[pattern[0]]` and `cnt[pattern[1]]`. We can add one instance of either `pattern[0]` or `pattern[1]` anywhere in `text` to maximize the subsequence count. Adding the character from the pattern with the highest frequency will yield the highest number of subsequences.
5. Return the final answer `ans`, which represents the maximum number of times `pattern` can occur as a subsequence in the modified `text`.

Let's look at an example to better understand the approach:

- `text = "ababc"`, `pattern = "ab"`
- As we iterate through `text`, we count occurrences of 'a' and 'b'.
- When we reach the first 'b' at index 1, `ans` is increased by the count of 'a' seen so far, which is 1 (`cnt['a'] = 1`).
- As we continue, every time we encounter 'b', we add the count of 'a's seen so far to `ans`.
- After the loop, we can add one more 'a' or 'b' (choosing 'a' is better in this case, as `cnt['a']` > `cnt['b']`), adding `cnt['a']` (which is 2) to `ans`.
- Finally, `ans` becomes `4`, which is the maximum subsequence count after the addition.

The use of a Counter allows us to efficiently keep track of character frequencies, and the single pass approach with an additional step minimizes the time complexity, resulting in an elegant and effective solution.

Example Walkthrough

Let's apply the solution approach to a small example where `text = "bbaca"` and `pattern = "ba"`.

1. We initialize `ans = 0` because initially, we haven't found any subsequences of the `pattern` yet.
2. We create a `Counter` object `cnt` to keep track of occurrences of characters in `text`. It is initially empty.
3. We start iterating through the `text`:
   - We encounter the first 'b'. It's not the second character of `pattern`, so we just update `cnt['b']` to 1.
   - We encounter the second 'b'. Again, it's not `pattern[1]`, so we update `cnt['b']` to 2.
   - We encounter 'a', and since it is `pattern[0]`, we update `cnt['a']` to 1 but do not alter `ans` since 'a' is not the second character in the `pattern`.
   - Next, we encounter 'c'. It's neither `pattern[0]` nor `pattern[1]`, so we continue.
   - Finally, we encounter another 'a'. We update `cnt['a']` to 2.
4. As we continue scanning, we find the last character 'b', which is `pattern[1]`. Now we add to `ans` the count of 'a' seen so far because 'a' followed by this 'b' can form another subsequence "ba". The `ans` is incremented by `cnt['a']`, which is 2. So now, `ans = 2`.
5. After the iteration is over, we look at the counts of 'b' and 'a' in `cnt`. We have `cnt['b'] = 2` and `cnt['a'] = 2`. We can add one more character to `text`. To maximize the subsequences, we should choose to add the character with the maximum count.
6. In this case, both `cnt['b']` and `cnt['a']` are the same, so we can choose either. Let's choose to add another 'a'.
7. By adding an 'a', we will be able to create new subsequences "ba" with all existing 'b's. Therefore, we add `cnt['b']` to `ans`, which results in an additional 2 subsequences.

So, the final answer `ans` is now `2 + 2 = 4`, which represents the maximum number of times `pattern = "ba"` can occur as a subsequence in the modified `text` after adding one extra 'a'.

This walkthrough demonstrates the process of calculating the number of subsequences of a given pattern in a text by iteratively counting characters, leveraging the subsequence definition, and maximizing the outcome by intelligently adding a character to the text based on the counts obtained.

Python Solution

```python
from collections import Counter

class Solution:
    def maximumSubsequenceCount(self, text: str, pattern: str) -> int:
        # Initialize the total count of subsequences found
        total_subsequence_count = 0

        # Create a counter to store the frequency of characters encountered
        character_frequency = Counter()

        # Iterate through all characters in the text
        for character in text:
            # If the current character matches the second character of the given pattern
            if character == pattern[1]:
                # Increment the subsequence count by the frequency of the first pattern character seen so far
                total_subsequence_count += character_frequency[pattern[0]]

            # Increment the frequency of the current character
            character_frequency[character] += 1

        # Add the maximum frequency between the first and second pattern characters
        # This accounts for the option to add a pattern character before or after the text
        total_subsequence_count += max(character_frequency[pattern[0]], character_frequency[pattern[1]])

        # Return the total count of pattern subsequences that can be found or added in the text
        return total_subsequence_count

```

Java Solution

```java
public class Solution {

    /**
     * Calculates the maximum number of times a given pattern appears as a subsequence
     * in the given text by potentially adding either character of the pattern at the beginning or end.
     *
     * @param text The input text in which subsequences are to be counted.
     * @param pattern The pattern consisting of two characters to be looked for as a subsequence.
     * @return The maximum number of times the pattern can occur as a subsequence.
     */
    public long maximumSubsequenceCount(String text, String pattern) {
        // Array to track the frequency of each character in the text
        int[] charCount = new int[26];

        // Extract the two characters from the pattern
        char firstPatternChar = pattern.charAt(0);
        char secondPatternChar = pattern.charAt(1);

        // This will hold the number of times the pattern occurs as a subsequence
        long totalCount = 0;

        // Iterate over each character in the text
        for (char currentChar : text.toCharArray()) {
            // If the current char matches the second char of the pattern,
            // increment the pattern occurrence count by the number of occurrences
            // of the first pattern character that have been seen so far.
            if (currentChar == secondPatternChar) {
                totalCount += charCount[firstPatternChar - 'a'];
            }
            // Update the count of the current character in our tracking array.
            charCount[currentChar - 'a']++;
        }

        // Increase the totalCount by the max frequency of appearing of either of the pattern characters.
        // This is because we can add one character (either the first or the second in the pattern)
        // to the start or the end of the text to increase the count of subsequences by that amount.
        totalCount += Math.max(charCount[firstPatternChar - 'a'], charCount[secondPatternChar - 'a']);

        // Return the total number of times the pattern can occur as a subsequence.
        return totalCount;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    long long maximumSubsequenceCount(string text, string pattern) {
        long long totalCount = 0; // This will hold the total count of desired subsequences
        char firstPatternChar = pattern[0]; // The first character in the pattern
        char secondPatternChar = pattern[1]; // The second character in the pattern

        // Initialize a count array for all letters, assuming English lower-case letters only
        vector<int> charCount(26, 0);

        // Iterate over each character in the text string
        for (char& currentChar : text) {
            // If the current char is the second in the pattern, add the count of the first pattern char seen so far
            if (currentChar == secondPatternChar) {
                totalCount += charCount[firstPatternChar - 'a'];
            }
            // Increment the count of the currentChar in the charCount vector
            charCount[currentChar - 'a']++;
        }

        // We can add either one of the pattern characters to either the beginning or the end of the string.
        // We choose the character that gives us more subsequences.
        // So, add the max between the occurrences of the two pattern characters to totalCount.
        totalCount += max(charCount[firstPatternChar - 'a'], charCount[secondPatternChar - 'a']);

        return totalCount; // Return the final total count of subsequences.
    }
};

```

Typescript Solution

```typescript
// Function to calculate the maximum number of subsequences with a given pattern in a text
function maximumSubsequenceCount(text: string, pattern: string): number {
    let totalCount = 0; // This will hold the total count of desired subsequences
    let firstPatternChar = pattern[0]; // The first character in the pattern
    let secondPatternChar = pattern[1]; // The second character in the pattern

    // Initialize a count array for all letters. In TypeScript, a Map is often more appropriate.
    let charCount: Map<string, number> = new Map<string, number>();

    // Iterate over each character in the text string
    for (let currentChar of text) {
        // If the current char is the second in the pattern, add the count of the first pattern char seen so far
        if (currentChar === secondPatternChar) {
            totalCount += (charCount.get(firstPatternChar) || 0);
        }
        // Increment the count of the currentChar in the charCount map
        charCount.set(currentChar, (charCount.get(currentChar) || 0) + 1);
    }

    // We can add either one of the pattern characters to either the beginning or the end of the string.
    // We choose the character that gives us more subsequences.
    // So, add the maximum between the occurrences of the two pattern characters to totalCount.
    totalCount += Math.max(charCount.get(firstPatternChar) || 0, charCount.get(secondPatternChar) || 0);

    return totalCount; // Return the final total count of subsequences.
}

// Usage of the function
const result = maximumSubsequenceCount("exampletext", "et");
console.log(result); // This would print the result of the function call to the console

```

Time and Space Complexity

The given Python code calculates the number of times a certain pattern of two characters can be inserted into a text such that the subsequence count of that pattern is maximized.

Time Complexity

The time complexity of the function is determined by a single pass over the text string `text`, which has length `n`. During this pass, for each character `c` in `text`, the code updates the `Counter` object `cnt` and in some cases increments the answer `ans`.

Accessing and updating the count in `Counter` for each character is generally O(1) complexity assuming a good hash function (as `Counter` is a type of `dict` in Python, and dictionary access is generally considered constant time).

The increment of `ans` based on `cnt[pattern[0]]` also occurs in constant time.

Therefore, because we have a single pass over the `text` with constant-time operations within the loop, the overall time complexity of the function is `O(n)`, where `n` is the length of the input string `text`.

Space Complexity

The space complexity is determined by the additional space used which is mainly the `Counter` object `cnt`. In the worst case, `cnt` could store a count for every unique character in `text`. The number of unique characters in `text` could be up to the size of the character set but is typically much smaller.

If we only consider the length of `text`, the space complexity would be `O(u)` where `u` is the number of unique characters in the `text`, which is less than or equal to `n`.

However, since the space taken up by `cnt` does not scale with the size of the input `text`, but rather with the number of unique characters, it might also be fair to consider it `O(1)` space, under the assumption that the character set size is fixed and not very large (e.g., ASCII characters).

So, the space complexity is either `O(u)` or `O(1)`, depending on whether you count the fixed size of the character set as constant or not.