1737. Change Minimum Characters to Satisfy One of Three Conditions

Medium Hash Table String Counting Prefix Sum Leetcode Link

Problem Description

The problem provides us with two strings, a and b, both of which only contain lowercase letters. We are allowed to perform operations on these strings where we can change any single character in either string to any other lowercase letter. The goal is to satisfy one of three conditions through the fewest possible number of such operations:

1. Every letter in string a is alphabetically strictly less than every letter in string b. This means, for example, if a contains letters up to 'x', then b should only contain letters 'y' or 'z'.
2. Every letter in string b is alphabetically strictly less than every letter in a. Following the same logic, if b consists of letters up to 'c', then a should only contain letters from 'd' onwards.
3. Both strings consist of only one distinct letter each. This means that both strings a and b could be turned into strings of a single repeating letter, which could be the same or different in each string.

The objective of the problem is to return the *minimum* number of such operations needed to meet one of these conditions as efficiently as possible.

Intuition

Understanding this problem requires identifying that we are essentially looking at the relationship between the letters of the two strings and determining the cost (number of operations) to achieve alignment under one of the given conditions. There are a few core ideas to consider for the solution:

1. **Comparing letter frequencies**: Since we care about the relative ordering of letters across the strings, counting the frequencies of each letter in both a and b provides a good starting point. This counting allows us to know how many changes we need to make to satisfy the conditions.

2. **Calculating the cost for the three conditions**: To satisfy condition (1) and (2), we need to ensure that one string only has letters that are less than the letters in the other string. For this, we check the cumulative frequency of letters from 'a' to 'z' and make sure that the sum of frequencies of one string from one letter is less than the sum of frequencies of the other string starting from the next letter in the alphabet. The intuition for condition (3) is to find the most common letter between the two strings and change all other letters to match it.

3. **Using Prefix Sum**: For the first two conditions, we can use a prefix sum technique. By calculating prefix sums, we cumulatively add letter frequencies and compare them across both strings. This allows us to efficiently compute the required changes as we iterate through the alphabet.

4. **Minimizing the operations**: To achieve the lowest cost, we need to check all possible scenarios for transformation. We perform this action by iterating through the possible letter divisions and also by calculating the cost for the third condition. Then, out of all these possibilities, we need to choose the one with the minimum number of operations.

Considering these points, the solution code maintains a global ans variable to store the minimum number of operations. It defines a helper function f which calculates the operation cost of making string a less than string b for all positions in the alphabet and updates the global minimum ans. The function then calls this helper function twice – once for ensuring a is less than b, and once for the opposite. For the third condition, it also computes the least cost for making both a and b consist of only one distinct letter, and updates the minimum operations needed. Finally, it returns the minimum operations as the answer.

Solution Approach

The solution uses the prefix sum algorithm approach, where cumulative computations are performed to make the operations efficient. Let's break down the implementation steps and the logic behind them:

1. **Frequency Counts**: The solution starts by computing the frequency of each letter in both a and b. This is done using two arrays cnt1 and cnt2, with 26 elements each (for each letter of the alphabet), initialized to 0. As we iterate over each string, we update the frequency count for the corresponding letters.

2. **Initialize Minimum Operations Variable**: The variable ans is initialized to the sum of lengths of a and b, which represents the worst-case scenario where we have to change every character in both strings.

3. **Optimizing for Condition 3**: Before checking for the other two conditions, the solution first tries to find the minimum number of operations required to satisfy the third condition. This is done by calculating m + n - c1 - c2 for each corresponding count of letters c1 and c2 in cnt1 and cnt2 arrays. The minimum of these values is the optimal result for the third condition, which is then compared with the current ans to update it if lesser.

4. **Defining the Helper Function 'f'**: The function f is defined to calculate the number of operations needed to make one string strictly less than the other. It loops through the counts starting from 1 to 25, which represent the positions where we can make the split. Then, it calculates the number of operations as the sum of the suffices of cnt1 starting from i and the prefixes of cnt2 up to i. The global ans is then updated if this calculated number of operations is less than the current ans.

5. **Applying Prefix Sum**: The helper function f is then called twice with reversed parameters to account for both cases where a is less than b and where b is less than a.

6. **Returning the Result**: After both calls to f, the accumulated lowest ans is the minimum number of operations needed to reach the goal and is then returned by the function.

The algorithm makes use of both frequency counting for direct comparison and prefix sums to enable efficient calculation of optimal splits between the strings. This combination of techniques is why the solution is categorized under the 'Prefix Sum' approach, providing a balance between computational complexity and memory usage.

Example Walkthrough

Let's consider two example strings a and b where a = "abc" and b = "xyy". Now, let's walk through the solution approach to demonstrate how we would reach one of the stated conditions using the minimum number of operations:

1. **Frequency Counts**: We count the frequencies of each letter in strings a and b.
   - cnt1 for string a would be: [1, 1, 1, ...] (rest are 0)
   - cnt2 for string b would be: [0, 0, ... , 1, 2] (rest are 0)
2. **Initialize Minimum Operations Variable**: ans is initialized to the worst-case scenario, so it would be ans = (length of a) + (length of b) = 6.
3. **Optimizing for Condition 3**: We check for the third condition:
   - For letter 'a', the cost would be m + n - c1 - c2 = 3 + 3 - 1 - 0 = 5.
   - For letter 'b', it's not present in b, so cost is still 6.
   - For letter 'c', the cost would be 5 as well.
   - For letter 'x', the cost is 3 + 3 - 0 - 1 = 5.
   - For letter 'y', the cost is 3 + 3 - 0 - 2 = 4. Thus, the lowest cost to meet the third condition is to turn both strings into "yyy", which requires 4 operations.
4. **Defining the Helper Function 'f'**: To satisfy condition 1 and 2, we use the helper function f to determine the minimum operations needed:
   - To satisfy condition 1, we need to ensure that a < b. One possible split is between 'c' and 'x'.
   - To satisfy condition 2, b < a, which cannot be achieved given the current string values since b already contains 'x' and 'y'.
5. **Applying Prefix Sum**: We use prefix sum to accumulate frequency counts and calculate the minimum operations for each possible split:
   - For a < b, imagine a split between 'c' and 'd'. We need to convert 'x' and 'y' into a letter after 'c', which takes 3 operations. However, this is not better than the result from the third condition.
   - For b < a, no such split can yield a result better than what we have from condition 3.
6. **Returning the Result**: After the calculations, we've determined that the least number of operations required is 4, which allows us to turn both strings into "yyy", thus satisfying condition 3.

In this example, the solution completes the transformation with a minimum of 4 operations, which is less than any other combinations we could create to satisfy condition 1 or 2.

Python Solution

```python
class Solution:
    def minCharacters(self, a: str, b: str) -> int:
        # Helper function to compute the minimum number of changes needed
        # so that:
        # 1. All letters in a are strictly less than any letter in b, or
        # 2. All letters in b are strictly less than any letter in a.
        def calculate_min_changes(count_a, count_b):
            for i in range(1, 26):  # Exclude 'a' by starting from 1 as it cannot be greater than any other letter
                # Calculate the number of changes needed by summing the number of characters
                # in count_a that need to be increased and the number of characters in count_b
                # that need to be decreased. This ensures all chars in a are less than those in b.
                changes = sum(count_a[i:]) + sum(count_b[:i])
                # Update the global answer if the local count is less
                nonlocal min_changes
                min_changes = min(min_changes, changes)

        # Get the lengths of both strings
        len_a, len_b = len(a), len(b)

        # Initialize character frequency counters for both strings
        count_a = [0] * 26
        count_b = [0] * 26

        # Fill the character frequency counters for both strings
        for char in a:
            count_a[ord(char) - ord('a')] += 1
        for char in b:
            count_b[ord(char) - ord('a')] += 1

        # Initialize the minimum number of changes to the sum of lengths
        # of both strings assuming all characters are changed
        min_changes = len_a + len_b

        # Check the third condition where you make both strings equal.
        # This is done by selecting the minimum change that can be done
        # by converting all characters of both strings to any character
        for count_a_char, count_b_char in zip(count_a, count_b):
            min_changes = min(min_changes, len_a + len_b - count_a_char - count_b_char)

        # Calculate the number of changes needed as per the helper function
        calculate_min_changes(count_a, count_b)
        calculate_min_changes(count_b, count_a)

        # Return the minimum number of changes needed
        return min_changes
```

Java Solution

```java
class Solution {
    private int minimumOperations;

    // Method to find the minimum number of characters you have to change
    // to make a and b strings satisfying at least one of the conditions given.
    public int minCharacters(String a, String b) {
        int lengthA = a.length();
        int lengthB = b.length();

        // Arrays to hold the character count for both strings.
        int[] countA = new int[26];
        int[] countB = new int[26];

        // Count the occurrence of each character in string a.
        for (int i = 0; i < lengthA; ++i) {
            countA[a.charAt(i) - 'a']++;
        }

        // Count the occurrence of each character in string b.
        for (int i = 0; i < lengthB; ++i) {
            countB[b.charAt(i) - 'a']++;
        }

        // Initialize the answer to the total number of characters in both strings.
        minimumOperations = lengthA + lengthB;

        // Condition 3: Change all characters in any string to one character.
        // Find the minimum changes required by making all characters same in either a or b.
        for (int i = 0; i < 26; ++i) {
            minimumOperations = Math.min(minimumOperations, lengthA + lengthB - countA[i] - countB[i]);
        }

        // Try changing both strings to satisfy condition 1 and 2
        calculateMinimumChanges(countA, countB);
        calculateMinimumChanges(countB, countA);

        return minimumOperations;
    }

    // Method to calculate minimum changes to make all characters in a < all characters in b and vice versa
    private void calculateMinimumChanges(int[] count1, int[] count2) {
        for (int i = 1; i < 26; ++i) {
            // Count the number of elements that need to be changed in count1 and count2
            int changeCount = 0;
            // Count changes needed for making all elements in count1 < character at position i.
            for (int j = i; j < 26; ++j) {
                changeCount += count1[j];
            }
            // Count changes needed for making all elements in count2 >= character at position i.
            for (int j = 0; j < i; ++j) {
                changeCount += count2[j];
            }
            // Update the minimum change count.
            minimumOperations = Math.min(minimumOperations, changeCount);
        }
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    int minCharacters(string a, string b) {
        // Calculate the size of strings a and b
        int sizeA = a.size(), sizeB = b.size();

        // Initialize character frequency vectors for a and b with zeros
        vector<int> freqA(26, 0);
        vector<int> freqB(26, 0);

        // Count the frequency of each character in string a
        for (char& c : a) {
            freqA[c - 'a']++;
        }

        // Count the frequency of each character in string b
        for (char& c : b) {
            freqB[c - 'a']++;
        }

        // Initialize the answer with the sum of the sizes of a and b
        int answer = sizeA + sizeB;

        // Check condition 3, reducing counts to make both strings anagrams
        for (int i = 0; i < 26; ++i) {
            answer = min(answer, sizeA + sizeB - freqA[i] - freqB[i]);
        }

        // Helper lambda function to calculate the minimum number of operations
        // needed for conditions 1 and 2
        auto calculateMinOperations = [&](vector<int>& cnt1, vector<int>& cnt2) {
            // Iterate over every possible division of the alphabet
            for (int i = 1; i < 26; ++i) {
                int operations = 0;
                // Count letters in cnt1 that must be increased (to become strictly greater)
                for (int j = i; j < 26; ++j) {
                    operations += cnt1[j];
                }
                // Count the letters in cnt2 that must be reduced (to become strictly less)
                for (int j = 0; j < i; ++j) {
                    operations += cnt2[j];
                }
                // Update the answer with the minimal operations needed
                answer = min(answer, operations);
            }
        };

        // Check condition 1, making 'a' letters strictly less than 'b' letters
        calculateMinOperations(freqA, freqB);

        // Check condition 2, making 'b' letters strictly less than 'a' letters
        calculateMinOperations(freqB, freqA);

        // Return the minimum number of operations found
        return answer;
    }
};
```

Typescript Solution

```typescript
function minCharacters(a: string, b: string): number {
    const lengthA = a.length,
        lengthB = b.length;
    let frequencyA = new Array(26).fill(0); // Array to store frequency of each character in string 'a'
    let frequencyB = new Array(26).fill(0); // Array to store frequency of each character in string 'b'
    const baseCharCode = 'a'.charCodeAt(0); // Base ASCII value for lowercase 'a'

    // Count characters in string 'a'
    for (let char of a) {
        frequencyA[char.charCodeAt(0) - baseCharCode]++;
    }
    // Count characters in string 'b'
    for (let char of b) {
        frequencyB[char.charCodeAt(0) - baseCharCode]++;
    }

    let prefixSumA = 0, // Prefix sum of frequencies in 'a'
        prefixSumB = 0; // Prefix sum of frequencies in 'b'
    let answer = lengthA + lengthB; // Initialize answer with the maximum possible value: sum of the lengths of 'a' and 'b'

    // Iterate over the first 25 letters to find the minimum number of changes
    for (let i = 0; i < 25; i++) {
        prefixSumA += frequencyA[i];
        prefixSumB += frequencyB[i];

        const case1 = lengthA - prefixSumA + prefixSumB; // Characters to change to make all chars in 'a' < chars in 'b'
        const case2 = prefixSumA + lengthB - prefixSumB; // Characters to change to make all chars in 'a' > chars in 'b'
        const case3 = lengthA + lengthB - frequencyA[i] - frequencyB[i]; // Characters to change to make all chars in 'a' and 'b' t

        // Find the minimum among the current answer and these three cases
        answer = Math.min(answer, case1, case2, case3);
    }
    // Check for the last character 'z' separately
    answer = Math.min(answer, lengthA + lengthB - frequencyA[25] - frequencyB[25]);

    return answer; // Return the minimum number of operations
}
```

Time and Space Complexity

Time Complexity

The core of the time complexity analysis lies in several parts of the code:

1. Counting characters in strings a and b. This involves iterating through each string once, resulting in O(m) for string a and O(n) for string b, where m and n are the lengths of the strings respectively.

2. The loop `for i in range(1, 26):` in function f runs 25 times (it doesn't include i=0 because changing all characters to 'a' is not allowed). Within this loop, there are two sums being calculated: sum(cnt1[i:]) and sum(cnt2[:i]). Summing operations are O(26), since they are the sums of a constant array of 26 possible characters. Therefore, despite seeming like a nested loop, this part remains constant O(26).

3. The min function used repeatedly could be considered O(1) for each operation, but it is used multiple times within different loops.

So, joining these parts together:

- The initial character counts are O(m+n)
- The min operation for equalizing characters is inside a loop that runs 26 times, so it's O(26)
- The f function is called twice and runs a loop of 25 iterations with constant-time operations inside, so it's O(25 * 2).

Considering these components, the overall time complexity is dominated by the character counts and the operations within the f function, so we estimate it as O(m + n + 26 + 25 * 2), which simplifies to O(m + n) because m and n could be significantly larger than the constants (26 and 50).

Space Complexity

The space complexity is calculated by considering the additional space used by the program excluding the input:

1. Two arrays cnt1 and cnt2 of size 26 each to store character frequencies, which give us a space of O(26 * 2).
2. Integer variables m, n, and ans which use a constant space, hence O(1).

The space complexity therefore amounts to O(52) which simplifies to O(1), as the space used is constant and does not depend on the input size.