2186. Minimum Number of Steps to Make Two Strings Anagram II

Medium Hash Table String Counting

Leetcode Link

Problem Description

In this problem, we are given two strings `s` and `t`. The task is to perform a series of steps with the goal of making these two strings anagrams of each other. An anagram is a rearrangement of letters to form a different word or phrase, typically using all the original letters exactly once; for example, 'listen' is an anagram of 'silent'.

The only allowed action in each step we can take is to append any character we want to the end of either string `s` or string `t`. The main objective is to determine the fewest number of such steps required to make the strings anagrams of each other. There is no restriction on how many times a character can be added, and there is no need to add the same character to both strings.

Intuition

The intuition behind the solution comes from the definition of an anagram. Since both `s` and `t` must have the same characters in the same amounts to be anagrams of each other, what we ultimately need to do is to balance out the character counts in both strings.

To find the minimum number of steps required, we should:

1. Count the frequency of each character in string `s`. This is because we need to know how many of each character we have to begin with and what we may need to add to `t` to balance the two strings.
2. Subtract the frequency of each character that appears in `t` from the frequency count obtained from `s`. When we do this, the count for a character will increase if `t` has fewer instances of that character than `s`, decrease if `t` has more of that character, or remain the same if both have an equal number. This gives us a measure of the deficit or surplus of characters in `t` relative to `s`.
3. To make `s` and `t` anagrams of each other, we need to make up for the character deficits between the two strings - in other words, we need to convert the negative counts to zero by appending characters to `t`, and reduce the positive counts to zero by appending characters to `s`. We don't actually append the characters; we just need to count how many characters would need to be appended.
4. We sum the absolute values of these counts, which gives us the total number of characters that need to be added to either string; in other words, the minimum number of steps required to make the strings anagrams.

The given solution achieves exactly this with an efficient approach utilizing the `Counter` from Python's collections module to tally the characters quickly and a loop to adjust the counts based on the second string `t`.

Solution Approach

The solution uses the `Counter` class from Python's `collections` module, which is a subclass of dict. It is designed to count objects, a special data structure that is ideal for tallying a hashable object (in our case, the characters in the string `s`).

Here's how the implementation works step by step:

1. `cnt = Counter(s)`: We instantiate a `Counter` object for the first string `s`. This counts the frequency of each character in `s` and stores it as key-value pairs, where the key is the character, and the value is the number of times that character occurs in `s`.
2. `for c in t: cnt[c] -= 1`: We iterate over each character `c` in the second string `t`. For every character, we decrement its count in the `Counter` by one. If `t` has more of a particular character than `s`, the count for that character will become negative, representing a surplus of that character in `t`. If `t` has fewer of that character, the count for that character will either decrease towards zero (if `s` had more initially) or stay negative.
3. `return sum(abs(v) for v in cnt.values())`: After adjusting the counts based on both strings, we compute the sum of the absolute values of these counts. This is because we want the total number of characters that are in surplus or deficit—regardless of the string to which they need to be added—to make `s` and `t` anagrams. As explained in the intuition, negative counts indicate a surplus in `t` (and thus a need to add characters to `s`), while positive counts indicate a surplus in `s` (and a need to add characters to `t`). The absolute value treats both cases the same way, as simply characters to be added.

The use of `abs()` function in step 3 is crucial because we want to ignore whether the final count is positive or negative for our purpose. We are only interested in how many characters in total are out of balance.

This approach offers an efficient and straightforward solution to the problem: it leverages Python's `Counter` to handle the calculations in a condensed manner without the need for lots of loops or conditionals, and the final result is simply the sum of absolute differences in character counts between the two strings.

Example Walkthrough

Let's walk through a small example using the solution approach described above. Suppose we have `s = "abc"` and `t = "xabc"` as our strings, and we want to make them anagrams of each other.

Using the solution approach:

1. **Count the characters in `s`:** First, we will create a `Counter` object for string `s`. After this step, `cnt` will look like `{'a': 1, 'b': 1, 'c': 1}` since these are the frequencies of the characters in `s`.
2. **Adjust counts based on `t`:** Next, we'll iterate through the second string `t` and update the counts in our `Counter` object. For each character in `t`, we will decrease the corresponding count in `cnt`. After processing `t`, which has `x` that doesn't exist in `s` and all the other characters exactly once, `cnt` will be updated like this:
   - `cnt['a']` will be `1 - 1 = 0`
   - `cnt['b']` will be `1 - 1 = 0`
   - `cnt['c']` will be `1 - 1 = 0`
   - `cnt['x']` will be `0 - 1 = -1` (since 'x' was not in `s` to begin with, it starts at zero and gets decreased)
3. **Compute the number of steps needed to make `s` and `t` anagrams:** After the second step, our counter `cnt` shows that `s` and `t` are already anagrams for characters `a`, `b`, and `c` (because their counts are 0), but we have an extra 'x' in string `t` that needs to be balanced. Thus, `cnt` is `{'a': 0, 'b': 0, 'c': 0, 'x': -1}` now. We are interested in the sum of the absolute values of the counts:
   - `abs(0) + abs(0) + abs(0) + abs(-1) = 0 + 0 + 0 + 1 = 1`

So, in this example, only 1 step is required to make `s` and `t` anagrams of each other, and that step is to append an 'x' to `s`.

This approach highlights the power of using a `Counter` object to efficiently tackle this problem and provides a simple and clear Example Walkthrough showing how one would apply the solution to a real-world example.

Python Solution

```python
from collections import Counter

class Solution:
    def minSteps(self, s: str, t: str) -> int:
        # Create a Counter object for the string s to keep track of character frequencies
        char_count = Counter(s)

        # Decrease the count in the Counter for every character in string t
        for char in t:
            char_count[char] -= 1

        # Sum the absolute values of the counts in the Counter.
        # This gives the total number of characters that are different
        # between s and t, which is the number of steps required.
        return sum(abs(value) for value in char_count.values())

```

Java Solution

```java
class Solution {
    public int minSteps(String s, String t) {
        // Initialize an array to store the character counts for the alphabet
        int[] charCounts = new int[26];

        // Increment count for each character in string s
        for (char c : s.toCharArray()) {
            charCounts[c - 'a']++;
        }

        // Decrement count for each character in string t
        for (char c : t.toCharArray()) {
            charCounts[c - 'a']--;
        }

        // Initialize a variable to store the minimum number of steps required
        int minSteps = 0;

        // Sum the absolute values of the differences in character counts
        for (int count : charCounts) {
            minSteps += Math.abs(count);
        }

        // Return the total number of steps to make t an anagram of s
        return minSteps;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <string>
#include <cmath> // Include cmath for std::abs

class Solution {
public:
    // Function to find the minimum number of steps to make two strings anagrams
    int minSteps(string s, string t) {

        // Create a vector to store the count of each character in the alphabet
        vector<int> charCount(26, 0);

        // Increment the count for each character in string 's'
        for (char& c : s) {
            ++charCount[c - 'a'];
        }

        // Decrement the count for each character in string 't'
        for (char& c : t) {
            --charCount[c - 'a'];
        }

        // Initialize the answer variable to count the number of steps
        int steps = 0;

        // Sum the absolute values of the counts
        // This represents the minimum number of character changes needed
        for (int count : charCount) {
            steps += std::abs(count);
        }

        // Return the total number of steps required
        return steps;
    }
};

```

Typescript Solution

```typescript
// Function to determine the minimum number of steps to make two strings anagrams of each other
function minSteps(s: string, t: string): number {
    // Initialize an array to count character occurrences
    const charCounts: number[] = new Array(128).fill(0);

    // Count the occurrences of each character in string `s`
    for (const char of s) {
        charCounts[char.charCodeAt(0)]++;
    }

    // Subtract the occurrences of each character in string `t`
    for (const char of t) {
        charCounts[char.charCodeAt(0)]--;
    }

    // Calculate the total number of steps required to make the strings anagrams
    let totalSteps = 0;
    for (const count of charCounts) {
        totalSteps += Math.abs(count);
    }

    // Return the total number of steps
    return totalSteps;
}

```

Time and Space Complexity

Time complexity

The time complexity of the provided code is derived from iterating over both strings `s` and `t` and the summation of the values in the counter `cnt`.

1. `cnt = Counter(s)`: Creating a counter for string `s` has a time complexity of $O(n)$ where n is the length of string `s`.
2. The for-loop iterating over string `t`: The loop runs for the length of string `t`, which is m. Therefore, the time complexity for this part is $O(m)$.
3. The summation `sum(abs(v) for v in cnt.values())`: This operation is dependent on the number of unique characters in `s`. In the worst case, each character is unique, and the complexity of this summation would be $O(u)$, where u is the number of unique characters, which is at most n.

Summing up, if n is the length of string `s` and m is the length of string `t`, the total time complexity is $O(n + m + u)$. Since $u <= n$, we can simplify this to $O(n + m)$.

Space complexity

The space complexity is mainly due to the storage required for the counter `cnt`.

1. `cnt = Counter(s)`: The counter for string `s` needs space for each unique character. In the worst case, this is $O(u)$ where u is the number of unique characters in `s`, which is at most n characters.

Thus, the space complexity of the algorithm is $O(u)$, and considering $u <= n$, it simplifies to $O(n)$.