# 1347. Minimum Number of Steps to Make Two Strings Anagram

Medium | Hash Table | String | Counting

Leetcode Link

## Problem Description

In this problem, you have two strings `s` and `t` of equal length. You're tasked with converting string `t` into an anagram of string `s`. An anagram is defined as a different arrangement of letters that are exactly the same in both strings. The way to do this is by choosing any character from `t` and replacing it with another character. The goal is to achieve this in the minimum number of steps. A step is defined as one instance of changing a character in `t`. The main question is to find out the least amount of steps necessary to turn `t` into an anagram of `s`.

## Intuition

The intuitive approach to solving this problem is based on tracking the frequency of each character in both strings, and then determining how many characters in `t` do not match the frequency needed to make an anagram of `s`. Here's how we arrive at the solution:

1. We first create a count (frequency) of all the characters in `s` using something like a `Counter` or a dictionary.
2. We then iterate through each character in `t`, checking against our previously made frequency count.
3. If the character in `t` is found in the frequency count, and the count for that character is greater than 0, it means we already have this character in `s`, so we decrement the count for that character.
4. If the character is not found or its count is already 0, it means this character is extra or not needed, so this is a character that needs to be replaced. Hence, we increment our answer which represents the number of changes needed.
5. After iterating through all characters in `t`, the value of our answer will represent the minimum number of steps required to make `t` an anagram of `s`.

## Solution Approach

The implementation of the solution can be explained with the following steps:

1. Import the `Counter` class from the `collections` module. `Counter` is a subclass of dictionary used for counting hashable objects, subtly simplifying the process of frequency computation for us.

2. Define the `minSteps` function which accepts two strings `s` and `t`, representing the initial and target strings, where we want to convert string `t` into an anagram of string `s`.

3. Instantiate a `Counter` for the string `s` to get the frequency of each character in `s`. The resulting `cnt` is a dictionary-like object where `cnt[c]` is the count of occurrences of character `c` in string `s`.

4. Initialize a variable `ans` to zero, which will be used to keep track of the number of changes we need to make in `t` to turn it into an anagram of `s`.

5. Iterate over each character `c` in `t`:

   - If `cnt[c]` is greater than zero, it implies that character `c` is present in `s` and we haven't matched all occurrences of `c` in `s` yet, so we decrease `cnt[c]` by one, indicating that we've matched this instance of `c` in `t` with an instance in `s`.

   - If `cnt[c]` is zero or `c` is not present, it implies that `s` does not require this character or we already have enough of this character to match `s` (i.e., excess characters in `t`), and therefore, we increment `ans` by one since this character needs to be replaced.

6. After the for-loop terminates, `ans` stores the total number of replacements needed for `t` to become an anagram of `s`, and this value is returned.

Using these steps, the solution efficiently computes the minimum number of steps to make `t` an anagram of `s`, taking advantage of hash maps (in this case, the `Counter` object) for fast access and update of character frequencies.

### Example Walkthrough

Let's consider a small example where `s = "anagram"` and `t = "mangaar"`. Our goal is to determine the minimum number of steps needed to transform `t` into an anagram of `s`.

1. First, using the `Counter` class to count the frequency of characters in `s`, we would get:

```
Counter('anagram') = {'a': 3, 'n': 1, 'g': 1, 'r': 1, 'm': 1}
```

2. We start with `ans = 0`. This `ans` keeps track of the required changes.

3. We examine each character in `t`:

   - For the first character 'm', `Counter` for `s` has `{'m': 1}`. We match 'm' in `t` to one in `s`, so we decrement by 1: now `Counter` is `{'m': 0}`.
   - The next character is 'a', `Counter` has `{'a': 3}`. After decrementing, it becomes `{'a': 2}`.
   - The following two characters are 'n' and 'g', `Counter` for 'n' is `{'n': 0}` and for 'g' is `{'g': 0}` after decrementing since both are present only once in `s`.
   - Next, we have two 'a's, but `Counter` shows `{'a': 2}`. After matching both, `Counter` updates to `{'a': 0}`. So far, no steps required as all characters matched.
   - The final letter in `t` is 'r', which is already matched in `s` (`Counter` for 'r' `{'r': 0}`). Since it's extra, we need to replace it, incrementing `ans` to 1.

4. After examining all characters in `t`, our `ans` equals 1. This means that we only need to replace one character in `t` to make it an anagram of `s`.

So, in this example, we only need a single step: replace the extra 'r' in `t` with a missing 'm' to make `t` an anagram of `s`.

## Python Solution

```python
from collections import Counter

class Solution:
    def minSteps(self, s: str, t: str) -> int:
        # Create a Counter (multiset) for all characters in string 's'
        char_count = Counter(s)

        # Initialize a variable to count the number of steps
        steps = 0

        # Iterate over each character in string 't'
        for char in t:
            # If the character is in the counter and count is greater than 0
            if char_count[char] > 0:
                # Decrease the count for that character since it can be transformed
                char_count[char] -= 1
            else:
                # If the character is not present or count is 0,
                # increase the steps needed, as this requires an additional operation
                steps += 1

        # Return the total number of steps required to make 't' equal to 's'
        return steps

```

## Java Solution

```java
class Solution {
    public int minSteps(String s, String t) {
        // Initialize an array to count the frequency of each character in the string s
        int[] charFrequency = new int[26];

        // Populate the character frequency array with the count of each character in s
        for (int i = 0; i < s.length(); ++i) {
            charFrequency[s.charAt(i) - 'a']++;
        }

        // This variable will keep track of the minimum number of steps (character changes)
        int minSteps = 0;

        // Iterate over the string t and decrease the frequency of each character in the charFrequency array
        for (int i = 0; i < t.length(); ++i) {
            // If the character's frequency after decrementing is negative,
            // it means that t has an extra occurrence of this character
            // that is not matched by a character in s.
            if (--charFrequency[t.charAt(i) - 'a'] < 0) {
                // Since this character is extra and unneeded, increase the minSteps
                minSteps++;
            }
        }
        // Return the total minimum number of steps to make t an anagram of s
        return minSteps;
    }
}

```

## C++ Solution

```cpp
class Solution {
public:
    // Function to find the minimum number of steps required to make the strings 's' and 't' anagrams
    int minSteps(string s, string t) {
        // Array to count the frequency of each character in the string 's'
        int charCounts[26] = {0};

        // Increment the frequency of each character found in the string 's'
        for (char& ch : s) {
            charCounts[ch - 'a']++;
        }

        // This will count the number of extra characters in the string 't' that aren't in string 's' or are in abundance
        int extraChars = 0;

        // Loop over the string 't'
        for (char& ch : t) {
            // If decrementing leads to a negative count, it means 't' has an extra character that 's' doesn't have or that it's in a
            extraChars += --charCounts[ch - 'a'] < 0;
        }

        // The answer is the number of extra characters which need to be changed in 't' to make it an anagram of 's'
        return extraChars;
    }
};

```

## Typescript Solution

```typescript
// Function definition using TypeScript with explicit input and output types
/**
 * Determines the minimum number of steps to make two strings anagrams by replacing letters.
 * @param {string} stringOne - The first string to be compared.
 * @param {string} stringTwo - The second string to be compared.
 * @returns {number} The minimum number of steps required to make the strings anagrams.
 */
const minSteps = (stringOne: string, stringTwo: string): number => {
    // Initialize an array to count character frequencies for stringOne
    const charCount = new Array(26).fill(0);

    // Populate the character frequency array for stringOne
    for (const char of stringOne) {
        const index = char.charCodeAt(0) - 'a'.charCodeAt(0);
        charCount[index]++;
    }

    // Initialize a counter for the number of steps needed
    let steps = 0;

    // Iterate through stringTwo and decrement the corresponding character count from stringOne
    for (const char of stringTwo) {
        const index = char.charCodeAt(0) - 'a'.charCodeAt(0);
        // Increment steps if character count falls below zero, which indicates a character in stringTwo not present in stringOne
        steps += --charCount[index] < 0 ? 1 : 0;
    }

    // Return the total number of steps required to make the two strings anagrams
    return steps;
};

```

## Time and Space Complexity

### Time Complexity

The time complexity of the code is determined by two main operations: the construction of the counter from string `s` and the iteration over string `t`.

1. Constructing `cnt` using `Counter(s)` involves iterating over all characters in `s`, which has a complexity of `O(N)` where `N` is the length of `s`.
2. The iteration over `t` also has a complexity of `O(M)` where `M` is the length of `t`.

Combining these, the total time complexity of the code is `O(N + M)` since the operations are sequential, not nested.

### Space Complexity

The space complexity is primarily dictated by the space required to store the counter `cnt`, which at most contains as many entries as there are unique characters in `s`. If the character set is fixed (like ASCII or Unicode), this can be considered a constant `O(1)`. However, if we consider the size of the character set `U` (which could grow with the input size in some theoretical cases), the space complexity could also be considered `O(U)`.

In practical programming scenarios where `s` and `t` consist of Unicode characters and the upper bound for `U` is the size of the Unicode character set, which is a constant `O(1)` as it doesn't change with the input size.

Combining the consideration for unique characters in the counter with any overhead for the storage structure, the overall space complexity is `O(U)` which, in the context of Unicode, simplifies to `O(1)` constant space complexity.