

1347. Minimum Number of Steps to Make Two Strings Anagram

MediumHash TableStringCounting

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:

- We first create a count (frequency) of all the characters in `s` using something like a `Counter` or a dictionary.
- We then iterate through each character in `t`, checking against our previously made frequency count.
- 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.
- 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.
- 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:

- 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.
- 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`.
- 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`.
- 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`.
- 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.
- 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`.

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

```
1 Counter('anagram') = {'a': 3, 'n': 1, 'g': 1, 'r': 1, 'm': 1}
```
- We start with `ans = 0`. This `ans` keeps track of the required changes.
- 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 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.
- 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

```
1 from collections import Counter
2
3 class Solution:
4     def minSteps(self, s: str, t: str) -> int:
5         # Create an array to count the frequency of each character in string 's'
6         char_count = Counter(s)
7
8         # Initialize a variable to count the number of steps
9         steps = 0
10
11        # Iterate over each character in string 't'
12        for char in t:
13            # If the character is in the counter and count is greater than 0
14            if char_count[char] > 0:
15                # Decrease the count for that character since it can be transformed
16                char_count[char] -= 1
17            else:
18                # If the character is not present or count is 0,
19                # increase the steps needed, as this requires an additional operation
20                steps += 1
21
22        # Return the total number of steps required to make 't' equal to 's'
23        return steps
24
```

Java Solution

```
1 class Solution {
2     public int minSteps(String s, String t) {
3         // Initialize an array to count the frequency of each character in the string s
4         int[] charFrequency = new int[26];
5
6         // Populate the character frequency array with the count of each character in s
7         for (int i = 0; i < s.length(); ++i) {
8             charFrequency[s.charAt(i) - 'a']++;
9         }
10
11        // This variable will keep track of the minimum number of steps (character changes)
12        int minSteps = 0;
13
14        // Iterate over the string t and decrease the frequency of each character in the charFrequency array
15        for (int i = 0; i < t.length(); ++i) {
16            // If the character's frequency after decrementing is negative,
17            // it means that t has an extra occurrence of this character
18            // that is not matched by a character in s.
19            if (--charFrequency[t.charAt(i) - 'a'] < 0) {
20                // Since this character is extra and unneeded, increase the minSteps
21                minSteps++;
22            }
23        }
24
25        // Return the total minimum number of steps to make t an anagram of s
26        return minSteps;
27    }
28 }
```

C++ Solution

```
1 class Solution {
2 public:
3     // Function to find the minimum number of steps required to make the strings 's' and 't' anagrams
4     int minSteps(string s, string t) {
5         // Array to count the frequency of each character in the string 's'
6         int charCounts[26] = {0};
7
8         // Increment the frequency of each character found in the string 's'
9         for (char& ch : s) {
10             charCounts[ch - 'a']++;
11         }
12
13        // This will count the number of extra characters in the string 't' that aren't in string 's' or are in abundance
14        int extraChars = 0;
15
16        // Loop over the string 't'
17        for (char& ch : t) {
18            // 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
19            extraChars += --charCounts[ch - 'a'] < 0;
20        }
21
22        // The answer is the number of extra characters which need to be changed in 't' to make it an anagram of 's'
23        return extraChars;
24    }
25 };
26
```

Typescript Solution

```
1 // Function definition using TypeScript with explicit input and output types
2 /**
3  * Determines the minimum number of steps to make two strings anagrams by replacing letters.
4  * @param {string} stringOne - The first string to be compared.
5  * @param {string} stringTwo - The second string to be compared.
6  * @returns {number} The minimum number of steps required to make the strings anagrams.
7  */
8 const minSteps = (stringOne: string, stringTwo: string): number => {
9     // Initialize an array to count character frequencies for stringOne
10    const charCount = new Array(26).fill(0);
11
12    // Populate the character frequency array for stringOne
13    for (const char of stringOne) {
14        const index = char.charCodeAt(0) - 'a'.charCodeAt(0);
15        charCount[index]++;
16    }
17
18    // Initialize a counter for the number of steps needed
19    let steps = 0;
20
21    // Iterate through stringTwo and decrement the corresponding character count from stringOne
22    for (const char of stringTwo) {
23        const index = char.charCodeAt(0) - 'a'.charCodeAt(0);
24        // Increment steps if character count falls below zero, which indicates a character in stringTwo not present in stringOne
25        steps += --charCount[index] < 0 ? 1 : 0;
26    }
27
28    // Return the total number of steps required to make the two strings anagrams
29    return steps;
30 };
31
```

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`.

- 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`.
- 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.