389. Find the Difference

Easy | Bit Manipulation | Hash Table | String | Sorting

Problem Description

You are given two strings, `s` and `t`. The string `t` is formed by taking the string `s`, performing a random shuffle of its characters, and then adding one additional character at a random position in the string. Your task is to find the letter that was added to `t`.

Intuition

The intuition behind the solution is to use a character count approach. First, the solution counts the frequency of each character in string `s`. Then it iterates through string `t` and decrements the count for each character encountered. Since string `t` contains all characters from `s` plus one extra, the moment we find a character in `t` such that its count drops below zero, we know that this is the extra character that was added—it wasn't present in `s` in the same quantity.

Therefore, by using a hash map or an array for counting the characters, we can efficiently find the added character.

Solution Approach

The solution provided uses Python's `Counter` class from the `collections` module, which is essentially a hash map or dictionary that maps elements to the number of occurrences in a given iterable.

Here's a step-by-step breakdown of how the algorithm works:

1. `cnt = Counter(s)`: We initialize `cnt` with the counts of each character in string `s`.
2. We then iterate over each character `c` in string `t` with `for c in t:`. As `t` includes every character from `s` plus one additional character, we expect all counts to be zero or one by the end of this iteration.
3. Inside the loop, we decrement the count of the current character `cnt[c] -= 1`. Since `t` should have exactly the same characters as `s`, except for the added one, most of the counts should eventually become zero.
4. The condition `if cnt[c] < 0:` checks if the count of the current character goes below zero. This indicates that `c` has appeared one more time in `t` than it has in `s`, specifically identifying it as the extra character.
5. At this point, the algorithm immediately returns `c` with `return c`, which is the character that was added to `t`.

By leveraging the `Counter` data structure, the algorithm can operate efficiently and arrive at the solution in O(n) time, where n is the length of the string `t`.

Example Walkthrough

Let's assume the given strings are `s = "aabc"` and `t = "aacbc"`. We need to identify the letter that was added to `t`.

Following the steps provided in the solution approach:

1. We first initialize `cnt` with the counts of each character in string `s`. That gives us a `Counter` object with counts `{ 'a': 2, 'b': 1, 'c': 1 }`.
2. We then iterate over each character in `t`. The sequence of actions in the loop would look like this:
   - For `c = 'a'`, decrement `cnt[c]` to `{ 'a': 1, 'b': 1, 'c': 1 }`.
   - For `c = 'a'` again, decrement `cnt[c]` to `{ 'a': 0, 'b': 1, 'c': 1 }`.
   - For `c = 'c'`, decrement `cnt[c]` to `{ 'a': 0, 'b': 0, 'c': 0 }`.
   - For `c = 'b'`, decrement `cnt[c]` to `{ 'a': 0, 'b': 0, 'c': 0 }`.
   - Lastly, for `c = 'c'`, decrement `cnt[c]` which would result in `{ 'a': 0, 'b': 0, 'c': -1 }`.
3. As soon as we decrement a count and it drops below zero, we've found our character. In this example, when we encounter the second 'c' in `t`, `cnt[c]` becomes `-1`, which means 'c' is the added character.

Thus, we return 'c' as the character that was added to `t`. The algorithm effectively determines that 'c' is the letter that was added, doing so with a simple linear scan and character count adjustment.

Solution Implementation

Python

```python
from collections import Counter

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

        # Iterate through each character in string 't'
        for char in t:
            # Decrement the count for the current character
            char_count[char] -= 1

            # If count becomes less than zero, it means this character is the extra one
            if char_count[char] < 0:
                return char

        # The function assumes that it is guaranteed there is an answer
        # If all characters match, this line would theoretically not be reached
```

Java

```java
class Solution {
    public char findTheDifference(String s, String t) {
        // Array to store the count of each alphabet character in string s
        int[] count = new int[26];

        // Increment the count for each character in the first string s
        for (int i = 0; i < s.length(); i++) {
            count[s.charAt(i) - 'a']++;
        }

        // Iterate over the second string t
        for (int i = 0; i < t.length(); i++) {
            // Decrease the count for the current character
            // If the count becomes negative, we've found the extra character in t
            if (--count[t.charAt(i) - 'a'] < 0) {
                return t.charAt(i);
            }
        }

        // This return statement is a placeholder; the code logic guarantees that the function will return from within the loop.
        // Since we know string t has one extra character, the loop will always return that extra character before this point.
        // Thus, the code will never reach this statement, but Java syntax requires a return statement here.
        return '\0';
    }
}
```

C++

```cpp
class Solution {
public:
    char findTheDifference(string s, string t) {
        int charCounts[26] = {0}; // Initialize an array to keep track of the character counts from 'a' to 'z'.

        // Increment the count for each character in string 's'.
        for (char& c : s) {
            charCounts[c - 'a']++; // 'c - 'a'' translates char c to an index (0-25) corresponding to chars 'a'-'z'.
        }

        // Decrement the count for each character in string 't'.
        for (char& c : t) {
            charCounts[c - 'a']--; // Similarly, translate char c to the correct index.

            // If the count goes below zero, it means 't' has an extra character that is not in 's'
            if (charCounts[c - 'a'] < 0) {
                return c; // Return the extra character.
            }
        }

        // If no extra character is found, this shouldn't happen as per the problem statement,
        // but we return a space as a default return value.
        return ' ';
    }
};
```

TypeScript

```typescript
function findTheDifference(source: string, target: string): string {
    // Convert the 'target' string into an array of its characters,
    // then use the 'reduce' function to accumulate the sum of the char codes.
    const targetCharCodeSum = [...target].reduce((runningTotal, currentChar) =>
        runningTotal + currentChar.charCodeAt(0), 0
    );

    // Convert the 'source' string into an array of its characters,
    // then use the 'reduce' function to accumulate the sum of the char codes.
    const sourceCharCodeSum = [...source].reduce((runningTotal, currentChar) =>
        runningTotal + currentChar.charCodeAt(0), 0
    );

    // Find the difference in the accumulated char code sums between the 'target' and 'source' strings.
    // This difference is the char code of the added letter in the 'target' string.
    const charCodeDifference = targetCharCodeSum - sourceCharCodeSum;

    // Convert the char code of the added letter to a string and return it.
    return String.fromCharCode(charCodeDifference);
}
```

```python
from collections import Counter

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

        # Iterate through each character in string 't'
        for char in t:
            # Decrement the count for the current character
            char_count[char] -= 1

            # If count becomes less than zero, it means this character is the extra one
            if char_count[char] < 0:
                return char

        # The function assumes that it is guaranteed there is an answer
        # If all characters match, this line would theoretically not be reached
```

Time and Space Complexity

Time Complexity

The time complexity of the given code is `O(n)` where `n` is the length of the string `t`. This is because the code iterates over each character of the string `t` exactly once. Utilizing the `Counter` from the `collections` module on string `s` has a time complexity of `O(m)`, where `m` is the length of string `s`. Thus, the overall time complexity of the algorithm is `O(m + n)`.

Space Complexity

The space complexity of the function is also `O(n)`, with `n` being the size of the alphabet used in the strings because the `Counter` object `cnt` will store the frequency of each character present in string `s`. In the worst-case scenario where all characters are unique, the counter will need to store an entry for each character. Since the alphabet size is generally fixed and small (e.g., 26 for lowercase English letters), the space complexity could also be considered `O(1)` if we consider the alphabet size as a constant factor.