[Click on the logo to find **Problem Statement**]

Method 1: Sorting

Intuition

The provided code determines whether two input strings, 's' and 't', are anagrams by sorting their characters and comparing the sorted results. The intuition behind this approach lies in the fact that anagrams have the same set of characters but in a different order. By sorting both strings, the code transforms them into representations where anagrams are guaranteed to be equal. If the sorted versions of 's' and 't' match, the function returns True, indicating that the input strings are indeed anagrams. Otherwise, it returns False. While this method is effective, it incurs a time complexity of O(n log n) due to the sorting operation, where 'n' is the length of the input strings, and a space complexity of O(n) for storing the sorted versions of the strings.

Approach

1. Sorting Operation:
    - The code begins by sorting the characters of both input strings 's' and 't' using the sorted() function.
2. Anagram Comparison:
    - After sorting, the code compares the sorted representations of 's' and 't' using the equality operator (==).
    - If the sorted versions are equal, it implies that the input strings are anagrams, and the function returns True.
3. Return False for Non-Anagrams:
    - If the sorted versions do not match, the function returns False, indicating that the input strings are not anagrams.
4. Time Complexity:

    - The time complexity is O(n log n), where 'n' is the length of the input strings. This is due to the sorting operation, which dominates the overall time complexity.

5. Space Complexity:

    - The space complexity is O(n), as additional space is required to store the sorted versions of 's' and 't'.

6. Intuition:

    - Anagrams share the same set of characters but in a different order. Sorting both strings transforms them into comparable representations, and if the sorted versions match, the strings are considered anagrams.

Complexity

- Time complexity: O(n log n)
- Space complexity: O(n)

Code

```
class Solution:
    def isAnagram(self, s: str, t: str) -> bool:
        sorted_c = sorted(s)
        sorted_t = sorted(t)
        if sorted_c == sorted_t:
            return True
        return False
```

Method 2: Hashmap

Intuition

The provided code determines whether two input strings, 's' and 't', are anagrams by utilizing hash maps to count the frequency of each character in both strings. The approach involves iterating through each character in the strings and updating the respective hash maps. If the lengths of the strings differ, they cannot be anagrams, and the function returns False. Otherwise, the code compares the two hash maps. If they are identical, it implies that the character frequencies match, and the function returns True, indicating that the input strings are anagrams. This method has a time complexity of O(n), where 'n' is the length of the input strings, and a space complexity of O(n) in the worst case, considering the number of distinct characters. Overall, the approach provides an efficient way to check for anagrams by leveraging hash maps to track character frequencies.

Approach

1. Length Check:
    - The code starts by comparing the lengths of the input strings 's' and 't'. If they are not equal, the strings cannot be anagrams, and the function returns False.
2. Hash Maps for Character Frequency:
    - Two hash maps, `s_hashmap` and `t_hashmap`, are initialized to store the frequency of each character in the respective strings.
3. Character Frequency Counting Loop:
    - The code iterates through each character in the strings using a loop.
    - For each character in 's', its frequency is updated in `s_hashmap`. The `get` method is used to retrieve the current count, and 1 is added.
    - Similarly, for each character in 't', its frequency is updated in `t_hashmap`.
4. Comparison of Hash Maps:
    - After processing both strings, the code compares the two hash maps (`s_hashmap` and `t_hashmap`) using the equality operator.
    - If the hash maps are not identical, it means that the character frequencies differ, and the function returns False.
5. Return True for Anagrams:
    - If the lengths match, and the hash maps are identical, the function returns True, indicating that the input strings are anagrams.

Complexity

- Time complexity: O(n)
- Space complexity: O(n)

Code

```python
class Solution:
    def isAnagram(self, s: str, t: str) -> bool:
        if len(s) != len(t):
            return False

        s_hashmap = {}
        t_hashmap = {}

        for i in range(len(s)):
            s_hashmap[s[i]] = 1 + s_hashmap.get(s[i], 0)
            t_hashmap[t[i]] = 1 + t_hashmap.get(t[i], 0)

        if s_hashmap != t_hashmap:
            return False
        return True
```

If you want to see more solutions to coding problems, you can visit:

GitHub