★★★★★ Average Rating: 4.80 (121 votes)

Solution

Approach #1 (Sorting) [Accepted]

Algorithm

An anagram is produced by rearranging the letters of s into t. Therefore, if t is an anagram of s, sorting both strings will result in two identical strings. Furthermore, if s and t have different lengths, t must not be an anagram of s and we can return early.

```
public boolean isAnagram(String s, String t) {
    if (s.length() != t.length()) {
        return false;
    }
    char[] str1 = s.toCharArray();
    char[] str2 = t.toCharArray();
    Arrays.sort(str1);
    Arrays.sort(str2);
    return Arrays.equals(str1, str2);
}
```

Complexity analysis

- Time complexity : $O(n \log n)$. Assume that n is the length of s, sorting costs $O(n \log n)$ and comparing two strings costs $O(n)$. Sorting time dominates and the overall time complexity is $O(n \log n)$.
- Space complexity : $O(1)$. Space depends on the sorting implementation which, usually, costs $O(1)$ auxiliary space if `heapsort` is used. Note that in Java, `toCharArray()` makes a copy of the string so it costs $O(n)$ extra space, but we ignore this for complexity analysis because:
  - It is a language dependent detail.
  - It depends on how the function is designed. For example, the function parameter types can be changed to `char[]`.

Approach #2 (Hash Table) [Accepted]

Algorithm

To examine if t is a rearrangement of s, we can count occurrences of each letter in the two strings and compare them. Since both s and t contain only letters from $a - z$, a simple counter table of size 26 is suffice.

Do we need *two* counter tables for comparison? Actually no, because we could increment the counter for each letter in s and decrement the counter for each letter in t, then check if the counter reaches back to zero.

```
public boolean isAnagram(String s, String t) {
    if (s.length() != t.length()) {
        return false;
    }
    int[] counter = new int[26];
    for (int i = 0; i < s.length(); i++) {
        counter[s.charAt(i) - 'a']++;
        counter[t.charAt(i) - 'a']--;
    }
    for (int count : counter) {
        if (count != 0) {
            return false;
        }
    }
    return true;
}
```

Or we could first increment the counter for s, then decrement the counter for t. If at any point the counter drops below zero, we know that t contains an extra letter not in s and return false immediately.

```
public boolean isAnagram(String s, String t) {
    if (s.length() != t.length()) {
        return false;
    }
    int[] table = new int[26];
    for (int i = 0; i < s.length(); i++) {
        table[s.charAt(i) - 'a']++;
    }
    for (int i = 0; i < t.length(); i++) {
        table[t.charAt(i) - 'a']--;
        if (table[t.charAt(i) - 'a'] < 0) {
            return false;
        }
    }
    return true;
}
```

Complexity analysis

- Time complexity : $O(n)$. Time complexity is $O(n)$ because accessing the counter table is a constant time operation.
- Space complexity : $O(1)$. Although we do use extra space, the space complexity is $O(1)$ because the table's size stays constant no matter how large n is.

Follow up

What if the inputs contain unicode characters? How would you adapt your solution to such case?

Answer

Use a hash table instead of a fixed size counter. Imagine allocating a large size array to fit the entire range of unicode characters, which could go up to more than 1 million. A hash table is a more generic solution and could adapt to any range of characters.

Report Article Issue

Python3

Autocomplete

```
class Solution:
    def isAnagram(self, s: str, t: str) -> bool:


        if len(s) != len(t):
            return False
        else :

            if sorted(s) == sorted(t):
                return True
```