Valid Anagram

Given two strings *s* and *t*, write a function to determine if *t* is an anagram of *s*.

For example,
s = "anagram", *t* = "nagaram", return true.
s = "rat", *t* = "car", return false.

Note:
You may assume the string contains only lowercase alphabets.

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

Solution 1

The idea is simple. It creates a size 26 int arrays as buckets for each letter in alphabet. It increments the bucket value with String s and decrement with string t. So if they are anagrams, all buckets should remain with initial value which is zero. So just checking that and return

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

written by vimukthi original link here

Solution 2

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

written by abner original link here

Hash Table

This idea uses a hash table to record the times of appearances of each letter in the two strings `s` and `t`. For each letter in `s`, it increases the counter by `1` while for each letter in `t`, it decreases the counter by `1`. Finally, all the counters will be `0` if they two are anagrams of each other.

The first implementation uses the built-in `unordered_map` and takes 36 ms.

```
class Solution {
public:
    bool isAnagram(string s, string t) {
        if (s.length() != t.length()) return false;
        int n = s.length();
        unordered_map<char, int> counts;
        for (int i = 0; i < n; i++) {
            counts[s[i]]++;
            counts[t[i]]--;
        }
        for (auto count : counts)
            if (count.second) return false;
        return true;
    }
};
```

Since the problem statement says that "the string contains only lowercase alphabets", we can simply use an array to simulate the `unordered_map` and speed up the code. The following implementation takes 12 ms.

```
class Solution {
public:
    bool isAnagram(string s, string t) {
        if (s.length() != t.length()) return false;
        int n = s.length();
        int counts[26] = {0};
        for (int i = 0; i < n; i++) {
            counts[s[i] - 'a']++;
            counts[t[i] - 'a']--;
        }
        for (int i = 0; i < 26; i++)
            if (counts[i]) return false;
        return true;
    }
};
```

Sorting

For two anagrams, once they are sorted in a fixed order, they will become the same.

This code is much shorter (this idea can be done in just 1 line using Python as here). However, it takes much longer time --- 76 ms in C++.

```
class Solution {
public:
    bool isAnagram(string s, string t) {
        sort(s.begin(), s.end());
        sort(t.begin(), t.end());
        return s == t;
    }
};
```

written by jianchao.li.fighter original link here

From Leetcoder.