**Valid Anagram**

Given two strings s and t, return true *if* t *is an anagram of* s*, and* false *otherwise*.

An **Anagram** is a word or phrase formed by rearranging the letters of a different word or phrase, typically using all the original letters exactly once.

**Example 1:**

**Input:** s = "anagram", t = "nagaram"

**Output:** true

**Example 2:**

**Input:** s = "rat", t = "car"

**Output:** false

**Constraints:**

* 1 <= s.length, t.length <= 5 \* 104
* s and t consist of lowercase English letters.

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

/\*\*

\* @param {string} s

\* @param {string} t

\* @return {boolean}

\*/

var isAnagram = function(s, t) {

};

Solution Article

Approach 1: Sorting

**Algorithm**

An anagram is produced by rearranging the letters of s*s* into t*t*. Therefore, if t*t* is an anagram of s*s*, sorting both strings will result in two identical strings. Furthermore, if s*s* and t*t* have different lengths, t*t* must not be an anagram of s*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)*O*(*n*log*n*). Assume that n*n* is the length of s*s*, sorting costs O(n \log n)*O*(*n*log*n*) and comparing two strings costs O(n)*O*(*n*). Sorting time dominates and the overall time complexity is O(n \log n)*O*(*n*log*n*).
* Space complexity: O(1)*O*(1). Space depends on the sorting implementation which, usually, costs O(1)*O*(1) auxiliary space if heapsort is used. Note that in Java, toCharArray() makes a copy of the string so it costs O(n)*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: Frequency Counter

**Algorithm**

To examine if t*t* is a rearrangement of s*s*, we can count occurrences of each letter in the two strings and compare them. We could use a hash table to count the frequency of each letter, however, since both s*s* and t*t* only contain letters from a*a* to z*z*, a simple array of size 26 will suffice.

Do we need two counters for comparison? Actually no, because we can increment the count for each letter in s*s* and decrement the count for each letter in t*t*, and then check if the count for every character is 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*s*, then decrement the counter for t*t*. If at any point the counter drops below zero, we know that t*t* contains an extra letter not in s*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)*O*(*n*). Time complexity is O(n)*O*(*n*) because accessing the counter table is a constant time operation.
* Space complexity: O(1)*O*(1). Although we do use extra space, the space complexity is O(1)*O*(1) because the table's size stays constant no matter how large n*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](http://stackoverflow.com/a/5928054/490463). A hash table is a more generic solution and could adapt to any range of characters.