

valid anagram leetcode solution

Valid anagram LeetCode solution is a common problem encountered by many programmers, especially those preparing for technical interviews or participating in coding competitions. 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. The challenge presented in platforms like LeetCode often involves determining whether two given strings are anagrams of each other. In this article, we will explore the problem, discuss various approaches to solve it, and provide a detailed solution in Python.

Understanding the Problem

The problem statement can be summarized as follows:

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

An anagram must satisfy the following conditions:

- Both strings must have the same length.
- Both strings must contain the same characters in the same frequency.

For example:

- Input: `s = "anagram", t = "nagaram"` -> Output: `True`
- Input: `s = "rat", t = "car"` -> Output: `False`

Approaches to Solve the Problem

There are multiple ways to determine if two strings are anagrams. Below we will discuss three common approaches:

1. Sorting Method

One straightforward way to check if two strings are anagrams is to sort both strings and compare them. If the sorted versions are the same, then the strings are anagrams.

Steps:

1. Check if the lengths of the two strings are equal.
2. Sort both strings.
3. Compare the sorted strings.

Time Complexity: $O(n \log n)$ due to the sorting operation.

2. Frequency Counting Using a Hash Map

Another efficient approach involves counting the frequency of each character in both strings and comparing the counts.

Steps:

1. Check if the lengths of the two strings are equal.
2. Create a frequency map for characters in the first string.
3. Decrease the count for characters found in the second string.
4. If all counts return to zero, the strings are anagrams.

Time Complexity: $O(n)$

3. Frequency Counting Using an Array

This method is similar to the Hash Map method but uses a fixed-size array instead. This is a good optimization because the number of possible characters is limited (e.g., lowercase English letters).

Steps:

1. Check if the lengths of the two strings are equal.
2. Create an array of size 26 (for each letter in the English alphabet).
3. Increment the count for each character in the first string and decrement for the second string.
4. If all counts are zero, the strings are anagrams.

Time Complexity: $O(n)$

Implementing the Solution

Let's implement the solution using the second method, the frequency counting with a hash map, and the third method using an array.

Solution using Hash Map

Here's a Python implementation using a hash map:

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

 count = {}
```

```

for char in s:
 count[char] = count.get(char, 0) + 1

for char in t:
 if char not in count:
 return False
 count[char] -= 1
 if count[char] < 0:
 return False

return True
'''

```

## Solution using Array

Now, let's see the implementation using an array:

```

'''python
def isAnagram(s: str, t: str) -> bool:
 if len(s) != len(t):
 return False

 count = [0] * 26 Assuming only lowercase letters

 for char in s:
 count[ord(char) - ord('a')] += 1

 for char in t:
 count[ord(char) - ord('a')] -= 1
 if count[ord(char) - ord('a')] < 0:
 return False

 return True
'''

```

## Testing the Function

It is essential to test our function with various cases to ensure its correctness. Here are some test cases:

```

'''python
Test cases
print(isAnagram("anagram", "nagaram")) True
print(isAnagram("rat", "car")) False
print(isAnagram("listen", "silent")) True
print(isAnagram("evil", "vile")) True
print(isAnagram("fluster", "restful")) True
'''

```

```
print(isAnagram("abc", "ab")) False
```
```

Conclusion

The problem of determining whether two strings are anagrams of each other can be solved efficiently with various approaches. The sorting method is simple but not the most optimal. The frequency counting methods, whether using a hash map or an array, provide a more efficient solution with linear time complexity.

In competitive programming and technical interviews, understanding the underlying principles and being able to articulate your thought process is as important as arriving at the correct solution. Therefore, practicing problems like the valid anagram on platforms such as LeetCode can significantly enhance your problem-solving skills and prepare you for future challenges.

Frequently Asked Questions

What is the problem statement for the 'Valid Anagram' LeetCode challenge?

The 'Valid Anagram' problem requires you to determine if two strings are anagrams of each other, meaning they contain the same characters in the same frequency but possibly in a different order.

What are common approaches to solve the 'Valid Anagram' problem?

Common approaches include sorting both strings and comparing them, or using a frequency count with a hash map or an array to track the number of occurrences of each character.

How can the sorting approach be implemented in Python for 'Valid Anagram'?

You can implement the sorting approach by using the 'sorted()' function on both strings and comparing the results: `return sorted(s1) == sorted(s2)`.

What is the time complexity of the frequency count method for 'Valid Anagram'?

The time complexity of the frequency count method is $O(n)$, where n is the length of the strings, because you are iterating through the characters to

build the frequency count.

Are there any edge cases to consider when solving the 'Valid Anagram' problem?

Yes, you should consider edge cases such as empty strings, strings of different lengths, and cases with special characters or spaces, as these can affect whether the strings are anagrams.

[Valid Anagram Leetcode Solution](#)

Valid Anagram Leetcode Solution

Related Articles

- [viviscal hair therapy stress relief](#)
- [vagus nerve breathing exercises](#)
- [vocabulary power plus level 10 answer key](#)

[Back to Home](#)