

242. Valid Anagram

EasyHash TableStringSorting

Problem Description

The problem presents two input strings, `s` and `t`, and asks to determine whether `t` is an anagram of `s`. An **anagram** is defined as a word or phrase that is created by rearranging the letters of another word or phrase, using all the original letters exactly once. The goal is to return `true` if `t` is an anagram of `s`, and `false` otherwise. This implies that if `t` is an anagram of `s`, both strings should have the same letters with the exact amount of occurrences for each letter.

Intuition

To solve this problem efficiently, we can use a counting approach. The intuition comes from the definition of an anagram – the strings must have the same characters in the same quantities. First, we can quickly check if `s` and `t` are of the same length. If they're not, `t` cannot be an anagram of `s` and we can immediately return `false`.

Once we've established that `s` and `t` are of equal length, the next step is to count the occurrences of each character in `s`. We can do this efficiently by using a hash table (in Python, this could be a `Counter` from the `collections` module). The hash table will map each character to the number of times it appears in `s`.

After setting up the counts for `s`, we traverse the string `t`. For each character in `t`, we decrease its corresponding count in the hash table by one. If at any point we find that a character's count goes below zero, this means that `t` contains more of that character than `s` does, which violates the anagram property. In this case, we can return `false`.

If we can successfully traverse all characters in `t` without any counts going below zero, it means that `t` has the same characters in the same quantities as `s`. Hence, `t` is an anagram of `s`, and we return `true`.

Solution Approach

The solution approach utilizes a hash table to keep track of the characters in string `s`. The use of a hash table allows us to efficiently map characters to the number of times they occur in `s`. This is a common pattern when we need to count instances of items, such as characters in a string.

The solution proceeds with the following steps:

- Check Length:** We compare the lengths of `s` and `t`. If they are different, we immediately return `false` because an anagram must have the same number of characters.
- Initialize Counter:** We initialize a `Counter` from the `collections` module in Python for the first string `s`. This `Counter` object effectively creates the hash table that maps each character to its count in the string.
- Character Traversal & Counting:** We then iterate over each character `c` in the second string `t` and decrement the count of `c` in our `Counter` by one for each occurrence. The `Counter` object allows us to do this in constant time for each character.
- Check Negative Counts:** We check if the decremented count of any character goes below zero. If this happens, it means that `t` has an extra occurrence of a character that does not match the count in `s`, and we return `false`.
- Return True:** If we go through the entire string `t` without encountering a negative count, this means that `t` has the exact same characters in the same amounts as `s`, confirming that `t` is an anagram of `s`. Thus, we return `true`.

By using a hash table, we achieve a linear time complexity of $O(n)$, where n is the length of the strings, since we go through each string once. This makes the algorithm efficient for large strings. The solution combines the steps of setting up the `Counter`, iterating over `t`, and verifying the counts into a concise and effective approach.

Example Walkthrough

Let's consider an example where `s = "listen"` and `t = "silent"`. We want to determine if `t` is an anagram of `s` using the solution approach described above.

- Check Length:** We check the lengths of both `s` and `t`. In this case, both strings are of length 6. As they are equal, we proceed to the next step.
- Initialize Counter:** We utilize a `Counter` from the `collections` module in Python to tally the characters in string `s`. So, we get the counts as follows:

```
1 {'l': 1, 'i': 1, 's': 1, 't': 1, 'e': 1, 'n': 1}
```
- Character Traversal & Counting:** We iterate over each character in string `t`.
 - Decrement count for 's': `{'l': 1, 'i': 1, 's': 0, 't': 1, 'e': 1, 'n': 1}`
 - Decrement count for 'l': `{'l': 1, 'i': 0, 's': 0, 't': 1, 'e': 1, 'n': 1}`
 - Decrement count for 'i': `{'l': 0, 'i': 0, 's': 0, 't': 1, 'e': 1, 'n': 1}`
 - Decrement count for 'e': `{'l': 0, 'i': 0, 's': 0, 't': 1, 'e': 0, 'n': 1}`
 - Decrement count for 'n': `{'l': 0, 'i': 0, 's': 0, 't': 1, 'e': 0, 'n': 0}`
 - Decrement count for 't': `{'l': 0, 'i': 0, 's': 0, 't': 0, 'e': 0, 'n': 0}`
- Check Negative Counts:** Throughout the traversal, we check the counts after each decrement. No count goes below zero, meaning there are no extra occurrences of any character.
- Return True:** After traversing `t`, all the counts are exactly zero, which means that `t` has precisely the same characters in the same amounts as `s`. Therefore, according to the solution approach, we return `true`, confirming that `t` is an anagram of `s`.

Python Solution

```
1 from collections import Counter
2
3 class Solution:
4     def isAnagram(self, s: str, t: str) -> bool:
5         # If the lengths of the strings are not equal, they cannot be anagrams
6         if len(s) != len(t):
7             return False
8
9         # Create a counter dictionary for all characters in the first string 's'
10        char_count = Counter(s)
11
12        # Iterate over each character in the second string 't'
13        for char in t:
14            # Decrease the count of the character in the counter dictionary
15            char_count[char] -= 1
16
17            # If the count of a character goes below zero,
18            # it means 't' has more occurrences of that character than 's',
19            # so 's' and 't' cannot be anagrams
20            if char_count[char] < 0:
21                return False
22
23        # If we've not returned False till now, 's' and 't' are anagrams
24        return True
25
```

Java Solution

```
1 class Solution {
2     public boolean isAnagram(String s, String t) {
3         // Check if both strings are of equal length
4         if (s.length() != t.length()) {
5             return false; // If not equal, they can't be anagrams
6         }
7
8         // Create an array to store the frequency of each letter
9         int[] letterCount = new int[26];
10
11        // Iterate over each character of both strings
12        for (int i = 0; i < s.length(); i++) {
13            // Increment the count for each letter in string s
14            letterCount[s.charAt(i) - 'a']++;
15            // Decrement the count for each letter in string t
16            letterCount[t.charAt(i) - 'a']--;
17        }
18
19        // Check if all counts are zero, indicating anagrams
20        for (int count : letterCount) {
21            if (count != 0) {
22                return false; // If any count is not zero, s and t are not anagrams
23            }
24        }
25
26        // If all counts are zero, then s and t are anagrams
27        return true;
28    }
29 }
30
```

C++ Solution

```
1 #include <string>
2 #include <vector>
3 #include <algorithm>
4
5 class Solution {
6 public:
7     // Function to determine if two strings are anagrams of each other
8     bool isAnagram(std::string s, std::string t) {
9         // If the strings are not of the same size, they cannot be anagrams
10        if (s.size() != t.size()) {
11            return false;
12        }
13
14        // Create a vector of 26 elements to count the occurrences of each letter
15        std::vector<int> charCounts(26, 0);
16
17        // Increment and decrement counts for each character in both strings respectively
18        for (int i = 0; i < s.size(); ++i) {
19            ++charCounts[s[i] - 'a']; // Increment count for the current char in string s
20            --charCounts[t[i] - 'a']; // Decrement count for the current char in string t
21        }
22
23        // Check if all counts are zero, if so, strings are anagrams, otherwise, they are not
24        return std::all_of(charCounts.begin(), charCounts.end(), [](int count) {
25            return count == 0;
26        });
27    }
28 };
29
```

Typescript Solution

```
1 function isAnagram(source: string, target: string): boolean {
2     // Check if both strings are of equal length; if not, they cannot be anagrams
3     if (source.length !== target.length) {
4         return false;
5     }
6
7     // Create an array of 26 elements to represent counts of each letter in the alphabet
8     const letterCounts = new Array(26).fill(0);
9
10    for (let i = 0; i < source.length; ++i) {
11        // Calculate the count for each letter in the source string (increment)
12        letterCounts[source.charCodeAt(i) - 'a'.charCodeAt(0)]++;
13
14        // Calculate the count for each letter in the target string (decrement)
15        letterCounts[target.charCodeAt(i) - 'a'.charCodeAt(0)]--;
16    }
17
18    // Check if all counts return to zero; if so, the strings are anagrams of each other
19    return letterCounts.every(count => count === 0);
20 }
21
```

Time and Space Complexity

The time complexity of the provided code is $O(n)$ where n represents the length of the string `s` (and `t`, since they are of the same length for the comparison to be valid). This complexity arises because the code iterates over each character in both `s` and `t` exactly once.

The space complexity of the code is $O(C)$ where C is the size of the character set used in the strings. In this context, since we are typically dealing with lowercase English letters, C is equal to 26. The space complexity comes from the use of a counter to store the frequency of each character in string `s`.