383. Ransom Note

Easy | Hash Table | String | Counting

Leetcode Link

Problem Description

The problem presents us with two strings: `ransomNote` and `magazine`. Our task is to determine if it's possible to construct the `ransomNote` string using the letters from the `magazine` string. There are certain rules to follow for this construction:

- Each letter in the `magazine` can be used only once.
- The order of the letters in the `ransomNote` does not have to match the order in the `magazine`.
- If all letters in the `ransomNote` can be matched to letters in `magazine`, taking into account the frequency of each letter, then the output should be `true`.
- Conversely, if there are any letters in the `ransomNote` which appear more times than they do in the `magazine`, or if there are any letters in the `ransomNote` that don't appear in the `magazine` at all, the output should be `false`.

Considering the above points, an immediate intuitive answer might seem complex since it implicitly involves tracking the count of each character in both `ransomNote` and `magazine`. The heart of the problem lies in efficiently checking the availability of each character needed to form the `ransomNote`.

Intuition

When considering the constraints of the problem, we naturally lean towards solutions that involve counting the frequency of each letter. The key, intuitive insight here is the realization that if `magazine` contains all the letters required by `ransomNote` in the necessary quantities, then we should be able to "use up" these letters from the `magazine` without any shortage.

To arrive at the solution, we think of keeping a tally of all the letters in `magazine`. A sensible way to achieve this is by using a hash table or a fixed-size array (since the alphabet is of fixed size - 26 letters for English). For each character in `magazine`, we increase its count in our tally. We then decrement these counts for each letter in `ransomNote`. If at any point the count for a particular letter drops below zero, it means `ransomNote` requires more of that letter than `magazine` can supply, and hence, we cannot construct `ransomNote` with the given `magazine`. The final result hinges on this check. If the entire `ransomNote` is processed without any count going negative, it's possible to create `ransomNote` from `magazine`; otherwise, it is not.

Solution Approach

The provided solution approach utilizes a hash table data structure, more specifically a `Counter`, which comes from Python's standard library `collections`. The hash table is particularly suited for this problem as it allows us to efficiently keep track of the counts of individual characters.

Here is how the algorithm works step by step:

1. First, we initialize the hash table by passing the `magazine` string to the `Counter`, which will count the occurrences of each character in `magazine` and store them in the `cnt` hash table.
2. Next, we iterate through each character `c` in the `ransomNote` string. For each such character, we perform two operations:
   - We decrement the count of the character `c` in the hash table `cnt`.
   - We check if the count of the character `c` falls below `0` after the decrement. If it does, it indicates that `magazine` does not have enough occurrences of character `c` to match the requirements of `ransomNote`, and we return `False`.
3. If we can iterate through all characters in `ransomNote` without the count of any character falling below `0`, it implies that `magazine` contains sufficient characters to construct `ransomNote`. Therefore, we return `True`.

Using the hash table for frequency counting is efficient as both insertion and lookup operations are on average O(1) complexity. The solution's overall time complexity is O(n+m), where `n` is the length of `ransomNote` and `m` is the length of `magazine`, since we have to iterate over both strings entirely. The space complexity is O(1) - even though we are using extra space for the hash table, it's size depends on the size of the character set (which is fixed), not the size of the input. In this case, it is the 26 letters of the English alphabet, hence it is a constant space overhead.

Using a `Counter` is particularly nifty here because it wraps up the necessary operations of initializing the frequency counts from `magazine`, decrementing with each character from `ransomNote`, and performing the check for a negative amount in a concise and readable manner. This indicates a great example of choosing the right data structures to simplify the implementation of an algorithm.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach:

Imagine the `ransomNote` is `"aabbcc"` and the `magazine` is `"abccbaac"`. We want to know if we can construct the ransom note using the letters from the magazine.

Following the solution approach:

1. We initialize the hash table `cnt` using `Counter` with the `magazine` string:

   ```
   cnt = Counter("abccbaac")
   // The counter will have counts for letters as follows: {'a': 3, 'b': 2, 'c': 3}
   ```

2. Next, we iterate through each character `c` in the `ransomNote` string `"aabbcc"` and update the hash table `cnt`:

   - We take the first character 'a' from the `ransomNote`:
     - Decrement the count of 'a' in `cnt` by 1: `Counter({'a': 2, 'b': 2, 'c': 3})`
     - Count is not below 0, we move to the next character.
   - We take the second character 'a':
     - Decrement the count again: `Counter({'a': 1, 'b': 2, 'c': 3})`
     - Count is not below 0, we move to the next character.
   - We take the first 'b':
     - Decrement the count of 'b': `Counter({'a': 1, 'b': 1, 'c': 3})`
     - Count is not below 0, we move on.
   - We take the second 'b':
     - Decrement the count: `Counter({'a': 1, 'b': 0, 'c': 3})`
     - Count is not below 0, so we continue.
   - We take the first 'c':
     - Decrement the count of 'c': `Counter({'a': 1, 'b': 0, 'c': 2})`
     - Count is not below 0, move to the next character.
   - We take the second 'c':
     - Decrement the count: `Counter({'a': 1, 'b': 0, 'c': 1})`
     - Count is not below 0.

3. Since we have gone through all the characters in `ransomNote` without any count falling below 0, we can conclude that it's possible to construct `ransomNote` from the `magazine`.

Thus, the result for the example would be `True`.

This example illustrates how the solution approach uses `Counter` to efficiently manage character counts from the magazine, and then check against the characters required to form the ransom note.

Python Solution

```python
from collections import Counter

class Solution:
    def canConstruct(self, ransom_note: str, magazine: str) -> bool:
        # Create a counter object for all characters in the magazine
        char_count = Counter(magazine)

        # Check each character in the ransom note
        for char in ransom_note:
            # Decrement the count for this character in the counter
            char_count[char] -= 1

            # If count goes below zero, we cannot construct the note from the magazine
            if char_count[char] < 0:
                return False

        # If we haven't returned False, we can construct the ransom note
        return True

```

Java Solution

```java
class Solution {

    public boolean canConstruct(String ransomNote, String magazine) {
        // Array to count occurrences of each letter in the magazine.
        int[] letterCounts = new int[26];

        // Populate the letterCounts array with the count of each character in the magazine.
        for (int i = 0; i < magazine.length(); i++) {
            // Increment the count of the current character.
            letterCounts[magazine.charAt(i) - 'a']++;
        }

        // Check if the ransom note can be constructed using the letters in the magazine.
        for (int i = 0; i < ransomNote.length(); i++) {
            // Decrement the count of the current character, as it is used in the ransom note.
            if (--letterCounts[ransomNote.charAt(i) - 'a'] < 0) {
                // If any letter in the ransom note is in deficit, return false.
                return false;
            }
        }

        // If all letters are accounted for, return true.
        return true;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // Function to check if ransomNote can be constructed from magazine
    bool canConstruct(string ransomNote, string magazine) {
        // Create a frequency array for each letter in the alphabet
        int letterCount[26] = {};

        // Iterate through each character in the magazine
        for (char& letter : magazine) {
            // Increase the count for the corresponding letter
            // 'a' maps to index 0, 'b' to 1, ..., 'z' to 25
            ++letterCount[letter - 'a'];
        }

        // Iterate through each character in the ransomNote
        for (char& letter : ransomNote) {
            // Decrease the count for the corresponding letter
            if (--letterCount[letter - 'a'] < 0) {
                // If the count goes negative, magazine doesn't have enough of this letter
                return false;
            }
        }

        // If we've gone through the entire note without issues, the note can be constructed
        return true;
    }
};

```

Typescript Solution

```typescript
function canConstruct(ransomNote: string, magazine: string): boolean {
    // Initialize an array to keep track of the frequency of each letter in the magazine
    const letterCount = new Array(26).fill(0);

    // Iterate over each character in the 'magazine' string
    for (const char of magazine) {
        // Increment the count corresponding to the current character
        letterCount[char.charCodeAt(0) - 'a'.charCodeAt(0)]++;
    }

    // Iterate over each character in the 'ransomNote' string
    for (const char of ransomNote) {
        // Get the index of the current character in 'letterCount' array
        // and decrement the frequency count of the corresponding character
        const index = char.charCodeAt(0) - 'a'.charCodeAt(0);
        letterCount[index]--;

        // If the frequency count goes negative, we do not have enough of this character in the magazine
        if (letterCount[index] < 0) {
            return false;
        }
    }

    // If we have not returned false by this point, it means we have enough characters for the ransomNote
    return true;
}

```

Time and Space Complexity

The time complexity of the function `canConstruct` is `O(m + n)`, where `m` is the length of the `ransomNote` string and `n` is the length of the `magazine` string. This is because the function first counts the occurrences of each character in the magazine string, which takes `O(n)` time, and then iterates through each character in the ransom note, which takes `O(m)` time. Each character decrement and comparison is an `O(1)` operation, thus the total time for the loop is `O(m)`. Combined, it leads to `O(m + n)` time complexity.

The space complexity of the function is `O(C)`, where `C` is the size of the character set involved in the magazine and ransom note. Given that these are likely to be letters from the English alphabet, the size of the character set `C` is 26. This fixed size of the character set means the space taken to store the counts is independent of the lengths of the input strings and is hence constant.