1657. Determine if Two Strings Are Close

Medium · Hash Table · String · Sorting

Problem Description

The problem introduces a concept called "close" strings. Two strings are "close" if one can be obtained from the other by performing any of the following operations, any number of times:

- **Operation 1:** Swap any two existing characters in the string. For example, changing "abce" to "aebc" by swapping 'b' with 'e'.
- **Operation 2:** Transform every occurrence of one existing character into another existing character and vice versa. For instance, changing "aacabb" to "bbcbaa" by swapping all 'a' characters with 'b' characters, and all 'b' characters with 'a' characters.

The goal is to determine if two given strings, `word1` and `word2`, can be considered "close" by applying these operations.

Intuition

To solve this problem, we need to understand that the operations allowed don't change the frequency of characters, only their positions or representations. Therefore, two "close" strings must have the same set of characters and the same frequency of each character, although the characters themselves can be different.

The crucial realization is that operation 1 allows us to reorder characters in any fashion, making the relative order of characters inconsequential. Operation 2 allows us to transform characters into each other, given that both characters exist in both strings. The consequence of this is:

- Both `word1` and `word2` must contain the same unique characters - they must have the same set of keys in their character counts (Counter).
- Both `word1` and `word2` must have the same character frequencies, which implies, after sorting their frequency counts, these should match.

With these constraints in mind, the solution approach is to count the frequency of each character in `word1` and `word2` using Python's `Counter`, then compare the sorted values of the counters to check for matching frequencies, as well as compare the sets of keys to ensure that both words contain the same unique characters.

If both the sorted values of the counts and the sets of unique characters match, we return `true`. Otherwise, we return `false`.

Solution Approach

The solution takes the following approach:

1. Use Python's `Counter` from the `collections` module to count the frequency of each character in both `word1` and `word2`. `Counter` is a dictionary subclass that's designed to count hashable objects. It's a collection where elements are stored as dictionary keys and their counts are stored as dictionary values.
2. Check if the set of keys from both Counters is the same—`set(cnt1.keys()) == set(cnt2.keys())`. This checks whether `word1` and `word2` contain exactly the same unique characters. This is a requirement because operation 2 can only be performed if a character exists in both strings.
3. Compare the sorted values of both counters—`sorted(cnt1.values()) == sorted(cnt2.values())`. This step checks if `word1` and `word2` have the same frequency of characters (the same count of each character). Sorting is crucial here since we need to compare frequencies regardless of the character they are associated with.

If both conditions are met, then `word1` and `word2` are "close", and the function returns `True`. If any of the conditions is not met, the function returns `False`. These comparisons effectively implement the constraints of the two operations without having to manually perform the operations themselves.

Here's a step-by-step breakdown of how the provided solution achieves this:

```python
class Solution:
    def closeStrings(self, word1: str, word2: str) -> bool:
        # Count the frequency of each character in both words
        cnt1, cnt2 = Counter(word1), Counter(word2)

        # If the sets of unique characters (keys of the counters) are the same and
        # the sorted lists of character counts (values of the counters) are the same,
        # then the strings are "close"
        return sorted(cnt1.values()) == sorted(cnt2.values()) and set(cnt1.keys()) == set(cnt2.keys())
```

The algorithmic complexity of this solution is mainly dependent on the sorting of the counted values, which is O(n log n) where n is the number of unique characters. Comparing the sets of keys essentially happens in linear time, O(m), where m is the length of the string.

Example Walkthrough

Let's take an example to illustrate the solution approach with two strings, word1 = "baba" and word2 = "abab".

1. We first count the frequency of each character in both words. The `Counter` for `word1` is `{ 'b': 2, 'a': 2 }` and for `word2` is `{ 'a': 2, 'b': 2 }`. This shows that both 'a' and 'b' appear twice in both strings.
2. We then compare the sets of keys from both dictionaries—`set(cnt1.keys()) == set(cnt2.keys())`. For our example, this means comparing `{ 'b', 'a' } == { 'a', 'b' }`, which evaluates to `True` because the set operation takes care of the order, and both sets contain the same elements.
3. For the final step, we compare the sorted values of both counters—`sorted(cnt1.values()) == sorted(cnt2.values())`. In this case, `sorted([2, 2]) == sorted([2, 2])`. The sorted list of character counts for both `word1` and `word2` is `[2, 2]`, which means that both strings have the same frequency for their characters.

Since both conditions are satisfied, we conclude that word1 and word2 are indeed "close" as per the problem's definition. The function would return `True`.

In a case where word1 = "baba" and word2 = "abbc", this is how the approach would result in a `False`:

1. `Counter(word1)` gives `{ 'b': 2, 'a': 2 }` and `Counter(word2)` gives `{ 'a': 1, 'b': 2, 'c': 1 }`.
2. When we compare the sets of the keys `{ 'b', 'a' }` and `{ 'a', 'b', 'c' }`, we get `False` because `word2` contains an extra character 'c'.

Since the sets of keys don't match, we don't even need to compare the values. The function would return `False`, indicating word1 and word2 are not "close".

Solution Implementation

Python

```python
from collections import Counter  # Import Counter from the collections module

class Solution:
    # Method to determine if two words can be made equal by swapping
    # characters an arbitrary number of times under the condition that
    # each character can only be swapped with the same character.
    def closeStrings(self, word1: str, word2: str) -> bool:
        # Calculate the frequency of each character in both words
        frequency_word1 = Counter(word1)
        frequency_word2 = Counter(word2)

        # Get the frequency values (counts of characters) from both words and sort them
        # Sorting is needed to compare if the words have the same frequency distribution
        sorted_values_word1 = sorted(frequency_word1.values())
        sorted_values_word2 = sorted(frequency_word2.values())

        # Compare the sets of keys (unique characters) from both words to check if
        # both words contain exactly the same characters
        keys_match = set(frequency_word1.keys()) == set(frequency_word2.keys())

        # Check both conditions: character counts must match when sorted and
        # the sets of characters present in both words must be the same
        return sorted_values_word1 == sorted_values_word2 and keys_match
```

Java

```java
class Solution {

    public boolean closeStrings(String word1, String word2) {
        // Frequency arrays for each character 'a' through 'z'.
        int[] freq1 = new int[26];
        int[] freq2 = new int[26];

        // Calculate the frequency of each character in word1.
        for (int i = 0; i < word1.length(); ++i) {
            freq1[word1.charAt(i) - 'a']++;
        }

        // Calculate the frequency of each character in word2.
        for (int i = 0; i < word2.length(); ++i) {
            freq2[word2.charAt(i) - 'a']++;
        }

        // Check if there's a character that exists in one word but not the other.
        for (int i = 0; i < 26; ++i) {
            if ((freq1[i] > 0 && freq2[i] == 0) || (freq2[i] > 0 && freq1[i] == 0)) {
                return false; // Words can't be close strings if a character is not shared.
            }
        }

        // Sort the frequency arrays to compare the frequency distribution.
        Arrays.sort(freq1);
        Arrays.sort(freq2);

        // Compare the sorted frequency arrays.
        for (int i = 0; i < 26; ++i) {
            if (freq1[i] != freq2[i]) {
                return false; // If frequencies don't match, words aren't close strings.
            }
        }

        // If all checks pass, the words are close strings.
        return true;
    }
}
```

C++

```cpp
#include <algorithm> // Include algorithm header for std::sort
#include <array>     // Include array header for std::array

class Solution {
public:
    // Function to determine if two strings are close.
    // Two strings are close if you can attain one from the other using operations of
    // swapping any two characters (often or rarely used) or transforming one character into another.
    bool closeStrings(std::string word1, std::string word2) {
        // We are using std::array for fixed size character frequency count
        std::array<int, 26> charCount1 = {}; // Character frequency count for word1
        std::array<int, 26> charCount2 = {}; // Character frequency count for word2

        // Count the frequency of each character in word1
        for (char c : word1) {
            ++charCount1[c - 'a'];
        }

        // Count the frequency of each character in word2
        for (char c : word2) {
            ++charCount2[c - 'a'];
        }

        // Check the presence of characters in both words;
        // a character must appear in both words to be close, or not at all
        for (int i = 0; i < 26; ++i) {
            bool charPresentWord1 = charCount1[i] > 0;
            bool charPresentWord2 = charCount2[i] > 0;

            // If a character is present in one word but not the other, they're not close
            if ((charPresentWord1 && !charPresentWord2) || (!charPresentWord1 && charPresentWord2)) {
                return false;
            }
        }

        // Sort the character counts to compare the frequency of characters
        std::sort(charCount1.begin(), charCount1.end());
        std::sort(charCount2.begin(), charCount2.end());

        // Check if both words have the same character frequency distribution
        // If they differ at any point, the words are not close
        for (int i = 0; i < 26; ++i) {
            if (charCount1[i] != charCount2[i]) {
                return false;
            }
        }

        // If all checks pass, the words are close
        return true;
    }
};
```

TypeScript

```typescript
function closeStrings(word1: string, word2: string): boolean {
    // Initialize arrays to count the character frequencies in both words
    const charCount1: number[] = new Array(26).fill(0);
    const charCount2: number[] = new Array(26).fill(0);

    // Count the frequency of each character in the first word
    for (let char of word1) {
        charCount1[char.charCodeAt(0) - 'a'.charCodeAt(0)]++;
    }

    // Count the frequency of each character in the second word
    for (let char of word2) {
        charCount2[char.charCodeAt(0) - 'a'.charCodeAt(0)]++;
    }

    // Check for the presence of characters in both words.
    // Characters must appear in both words or not at all to be considered close.
    for (let i = 0; i < 26; ++i) {
        let charPresentWord1 = charCount1[i] > 0;
        let charPresentWord2 = charCount2[i] > 0;

        // If a character appears in one word but not the other, return false
        if (charPresentWord1 !== charPresentWord2) {
            return false;
        }
    }

    // Sort the frequency counts of characters to compare distributions
    charCount1.sort((a, b) => a - b);
    charCount2.sort((a, b) => a - b);

    // Compare the sorted frequency counts for both words
    // If they differ, the words are not close
    for (let i = 0; i < 26; ++i) {
        if (charCount1[i] !== charCount2[i]) {
            return false;
        }
    }

    // If all checks pass, the words are considered close
    return true;
}
```

```python
from collections import Counter  # Import Counter from the collections module

class Solution:
    # Method to determine if two words can be made equal by swapping
    # characters an arbitrary number of times under the condition that
    # each character can only be swapped with the same character.
    def closeStrings(self, word1: str, word2: str) -> bool:
        # Calculate the frequency of each character in both words
        frequency_word1 = Counter(word1)
        frequency_word2 = Counter(word2)

        # Get the frequency values (counts of characters) from both words and sort them
        # Sorting is needed to compare if the words have the same frequency distribution
        sorted_values_word1 = sorted(frequency_word1.values())
        sorted_values_word2 = sorted(frequency_word2.values())

        # Compare the sets of keys (unique characters) from both words to check if
        # both words contain exactly the same characters
        keys_match = set(frequency_word1.keys()) == set(frequency_word2.keys())

        # Check both conditions: character counts must match when sorted and
        # the sets of characters present in both words must be the same
        return sorted_values_word1 == sorted_values_word2 and keys_match
```

Time and Space Complexity

The given Python function `closeStrings` determines if two words can be made equal by swapping the positions of the letters. This is done by checking two conditions: the sorted values of the character counters for both words are the same and the sets of characters (keys of the counters) in both words are the same.

Time Complexity:

- Creating the counters for `word1` and `word2` has a time complexity of `O(N + M)`, where `N` is the length of `word1` and `M` is the length of `word2`. This is because each word is traversed once to create the counts of characters.
- Sorting the values of the counters has a time complexity of `O(NlogN + MlogM)` in the worst case, because the time complexity of sorting n elements is `O(nlogn)`, and we are sorting the frequency counts of both words.
- Comparing two sorted lists of values has a time complexity of `O(N + M)` in the worst case, assuming `N` and `M` are the numbers of unique characters in `word1` and `word2`, respectively.
- Comparing two sets of keys has a time complexity of `O(N + M)`, because sets are typically implemented as hash tables, and comparing two sets involves checking that every element of one set is in the other (and vice versa), which takes `O(N + M)` time in this case.

Considering the most expensive operations, the overall time complexity is `O(NlogN + MlogM)`, since the logarithmic factors in sorting dominates the linear factors from other operations.

Space Complexity:

- The counters for `word1` and `word2` consume `O(N + M)` space, as they store counts for the characters present in both `word1` and `word2`.
- Sorting the values of the counters requires `O(N + M)` space to hold the sorted lists.
- The sets of keys take `O(N + M)` space as well.

Hence, the overall space complexity is `O(N + M)`, which accounts for the storage of the character counts and the unique characters.