2531. Make Number of Distinct Characters Equal

Medium Hash Table String Counting

Leetcode Link

Problem Description

In this problem, we are given two zero-indexed (meaning indexing starts from 0) strings `word1` and `word2`. We are allowed to perform a move which involves choosing two indices `i` from `word1` and `j` from `word2` (both indices must be within the bounds of their respective strings) and swapping the characters at these positions, i.e., `word1[i]` and `word2[j]`.

The objective is to determine if it is possible to equalize the number of distinct characters in both `word1` and `word2` using exactly one such move. If this is possible, our function should return `true`. Otherwise, it should return `false`.

Intuition

The intuition behind the solution is based on counting characters in each string and then considering every possible swap to see if it can lead us to the goal of equalizing the distinct number of characters in both strings with just one move.

To keep track of the characters, we use two arrays `cnt1` and `cnt2` of size 26 (assuming the input strings consist of lowercase alphabetic characters only). Each array corresponds to counting occurrences of characters in `word1` and `word2`, respectively.

Here's the thinking process to arrive at the solution:

- We first count how many times each character appears in both `word1` and `word2`.
- Then, we iterate through the counts of both `cnt1` and `cnt2`. For every pair of characters `(i, j)` where `cnt1[i]` and `cnt2[j]` are both non-zero (indicating that character `i` is available to swap in `word1` and character `j` in `word2`), we emulate a swap.
- After the virtual swap, we calculate the number of distinct characters currently present in `cnt1` and `cnt2`.
- We check if the number of distinct characters is now equal in both. If it is, we return `true`.
- If it isn't, we revert the swap back to its original state and continue with the next possible swap pair.

The reason we try every possible swap is that the distinct number of characters is influenced by both the characters being swapped in and out. So, to find out if any swap can achieve our goal, we have to examine each possibility.

Solution Approach

The implementation of the solution follows a simple yet efficient brute-force approach to verify whether a single swap can make the number of distinct characters equal in both strings. Here's a walk-through:

- **Data Structures**: Two arrays `cnt1` and `cnt2` are utilized, each with a length of 26 corresponding to the 26 letters in the English alphabet. These arrays are used to count the occurrences of each character in `word1` and `word2` respectively.

- **Counting Characters**: Iterate over `word1` and `word2` to count each character. This is done by converting the character to its ASCII value with `ord(c)`, subtracting the ASCII value of `'a'` to normalize the index to 0-25, and incrementing the count at that index in `cnt1` or `cnt2`. This looks like the following:

```python
for c in word1:
    cnt1[ord(c) - ord('a')] += 1
for c in word2:
    cnt2[ord(c) - ord('a')] += 1
```

- **Attempting Swaps**: The next step is to consider every possible swap. For this, nested loops are used to go over every index `i` in `cnt1` and every index `j` in `cnt2`. If `cnt1[i]` and `cnt2[j]` are both greater than 0 (meaning both characters are present in their respective strings), a virtual swap is performed:

```python
cnt1[i], cnt2[j] = cnt1[i] - 1, cnt2[j] - 1
cnt1[j], cnt2[i] = cnt1[j] + 1, cnt2[i] + 1
```

  This emulates moving one occurrence of the `i-th` character from `word1` to `word2` and one occurrence of the `j-th` character from `word2` to `word1`.

- **Checking Distinct Characters**: After emulating the swap, a check is performed to determine if the number of distinct characters in both arrays is the same. This is done by summing up the count of indices greater than 0 in both arrays:

```python
if sum(v > 0 for v in cnt1) == sum(v > 0 for v in cnt2):
    return True
```

- **Reverting Swap**: If the numbers are not equal, the swap is reverted:

```python
cnt1[i], cnt2[j] = cnt1[i] + 1, cnt2[j] + 1
cnt1[j], cnt2[i] = cnt1[j] - 1, cnt2[i] - 1
```

  And the algorithm continues to the next possible swap.

- **Result**: If a successful swap that balances the number of distinct characters is found, the function returns `True`. If no such swap is found after all possibilities have been considered, the function eventually returns `False`.

This algorithm is efficient in the sense that it goes through a limited set of possible swaps (at most 26 * 26) and requires no extra space besides the two counting arrays.

Example Walkthrough

Let's illustrate the solution approach with a small example:

Suppose we have `word1 = "abc"` and `word2 = "def"`. Our goal is to determine if we can equalize the number of distinct characters in both words by performing exactly one swap.

First, we initialize two arrays to count the occurrences of each character, `cnt1` and `cnt2`. After iterating through each word:

`cnt1` (for `word1`) would be `[1, 1, 1, 0, 0, 0, ..., 0]` corresponding to [a, b, c, ..., z]. `cnt2` (for `word2`) would be `[0, 0, 0, 1, 1, 1, ..., 0]` also corresponding to [a, b, c, ..., z].

Now, we attempt every possible swap between characters in `word1` and `word2`:

1. Trying to swap `a` from `word1` with `d` from `word2`

We adjust our count arrays to reflect this potential swap:

- `cnt1` becomes `[0, 1, 1, 1, 0, 0, ..., 0]`
- `cnt2` becomes `[1, 0, 0, 0, 1, 1, ..., 0]`

We now check if the number of distinct characters in both is the same:

- `cnt1` has 3 distinct characters (`b`, `c`, `d`)
- `cnt2` has 3 distinct characters (`a`, `e`, `f`)

Since the number of distinct characters is equal, we return `True`.

However, if we need to continue to illustrate the rest of the process, we would revert this virtual swap and try other possibilities:

Revert back to original counts:

- `cnt1` becomes `[1, 1, 1, 0, 0, 0, ..., 0]`
- `cnt2` becomes `[0, 0, 0, 1, 1, 1, ..., 0]`

2. Trying to swap `a` from `word1` with `e` from `word2`

... and so on for every character in word1 vs every character in word2.

For the sake of this example, we already found a swap that works, so there's no need to continue. The function would now return `True`. If a swap could not equalize the number of distinct characters, then we would finally return `False` after exhausting all the combinations.

Python Solution

```python
class Solution:
    def isItPossible(self, word1: str, word2: str) -> bool:
        # Count the frequency of each character in both words
        count1 = [0] * 26
        count2 = [0] * 26

        # Update the character frequency for word1
        for char in word1:
            count1[ord(char) - ord('a')] += 1

        # Update the character frequency for word2
        for char in word2:
            count2[ord(char) - ord('a')] += 1

        # Try swapping frequencies and check if both words can have the same character set
        for i, frequency1 in enumerate(count1):
            for j, frequency2 in enumerate(count2):
                # Only proceed with the swap if both frequencies are non-zero
                if frequency1 and frequency2:
                    # Decrement and increment the frequencies at position i and j respectively
                    count1[i], count2[j] = count1[i] - 1, count2[j] - 1
                    count1[j], count2[i] = count1[j] + 1, count2[i] + 1

                    # Calculate the number of distinct characters current in each word
                    distinct_in_word1 = sum(v > 0 for v in count1)
                    distinct_in_word2 = sum(v > 0 for v in count2)

                    # If both words have the same number of distinct characters, return True
                    if distinct_in_word1 == distinct_in_word2:
                        return True

                    # Revert the changes if the above condition is not met
                    count1[i], count2[j] = count1[i] + 1, count2[j] + 1
                    count1[j], count2[i] = count1[j] - 1, count2[i] - 1

        # If no swaps can result in both words having the same character set, return False
        return False
```

Java Solution

```java
class Solution {
    public boolean isItPossible(String word1, String word2) {
        // Create arrays to count the frequency of each character in both strings
        int[] countWord1 = new int[26];
        int[] countWord2 = new int[26];

        // Count the frequency of each character in word1
        for (int i = 0; i < word1.length(); ++i) {
            countWord1[word1.charAt(i) - 'a']++;
        }

        // Count the frequency of each character in word2
        for (int i = 0; i < word2.length(); ++i) {
            countWord2[word2.charAt(i) - 'a']++;
        }

        // Iterate over all pairs of characters
        for (int i = 0; i < 26; ++i) {
            for (int j = 0; j < 26; ++j) {
                // If both characters are present in their respective words
                if (countWord1[i] > 0 && countWord2[j] > 0) {
                    // Simulate swapping the characters
                    countWord1[i]--;
                    countWord2[j]--;
                    countWord1[j]++;
                    countWord2[i]++;

                    // Check if the frequency distribution matches after the swap
                    int delta = 0; // Delta will store the net difference in frequencies
                    for (int k = 0; k < 26; ++k) {
                        if (countWord1[k] > 0) {
                            delta++;
                        }
                        if (countWord2[k] > 0) {
                            delta--;
                        }
                    }

                    // If delta is zero, it means that the frequency distribution matches
                    if (delta == 0) {
                        return true;
                    }

                    // Undo the swap operation as it did not lead to a match
                    countWord1[i]++;
                    countWord2[j]++;
                    countWord1[j]--;
                    countWord2[i]--;
                }
            }
        }

        // If no swaps resulted in a match, return false
        return false;
    }
}
```

C++ Solution

```cpp
#include <string>

class Solution {
public:
    bool isItPossible(std::string word1, std::string word2) {
        // Arrays to store letter frequencies for both words
        int count1[26] = {0};
        int count2[26] = {0};

        // Populate frequency arrays for word1
        for (char c : word1) {
            ++count1[c - 'a'];
        }

        // Populate frequency arrays for word2
        for (char c : word2) {
            ++count2[c - 'a'];
        }

        // Iterate over each letter in the alphabet
        for (int i = 0; i < 26; ++i) {
            // Iterate over each letter in the alphabet
            for (int j = 0; j < 26; ++j) {
                // Check if current letters are present in both words
                if (count1[i] > 0 && count2[j] > 0) {
                    // Simulate swapping the letters by updating counts
                    --count1[i];
                    --count2[j];
                    ++count1[j];
                    ++count2[i];

                    // Variable to track the sum of differences in frequencies
                    int difference = 0;
                    // Check if the frequencies match for each letter
                    for (int k = 0; k < 26; ++k) {
                        if (count1[k] > 0) {
                            ++difference;
                        }
                        if (count2[k] > 0) {
                            --difference;
                        }
                    }

                    // If the sum of differences is zero, words can be made identical
                    if (difference == 0) {
                        return true;
                    }

                    // Undo the simulated swap
                    ++count1[i];
                    ++count2[j];
                    --count1[j];
                    --count2[i];
                }
            }
        }

        // If no swap can make the words identical, return false
        return false;
    }
};
```

Typescript Solution

```typescript
function isItPossible(word1: string, word2: string): boolean {
    // Arrays to store letter frequencies for both words
    let count1: number[] = new Array(26).fill(0);
    let count2: number[] = new Array(26).fill(0);

    // Populate frequency arrays for word1
    for (let c of word1) {
        count1[c.charCodeAt(0) - 'a'.charCodeAt(0)]++;
    }

    // Populate frequency arrays for word2
    for (let c of word2) {
        count2[c.charCodeAt(0) - 'a'.charCodeAt(0)]++;
    }

    // Iterate over each letter in the alphabet
    for (let i = 0; i < 26; ++i) {
        // Iterate over each letter in the alphabet
        for (let j = 0; j < 26; ++j) {
            // Check if current letters are present in both words
            if (count1[i] > 0 && count2[j] > 0) {
                // Simulate swapping the letters by updating counts
                count1[i]--;
                count2[j]--;
                count1[j]++;
                count2[i]++;

                // Variable to track the sum of differences in frequencies
                let difference = 0;
                // Check if the frequencies match for each letter
                for (let k = 0; k < 26; ++k) {
                    if (count1[k] > 0) {
                        difference++;
                    }
                    if (count2[k] > 0) {
                        difference--;
                    }
                }

                // If the sum of differences is zero, words can be made identical
                if (difference === 0) {
                    return true;
                }

                // Undo the simulated swap
                count1[i]++;
                count2[j]++;
                count1[j]--;
                count2[i]--;
            }
        }
    }

    // If no swap can make the words identical, return false
    return false;
}
```

Time and Space Complexity

Time Complexity

The provided code iterates over each character in `word1` and `word2`, counting the frequency of every character. Then it has nested loops where it iterates over the counts of characters in `cnt1` and `cnt2` arrays (size 26 for each letter of the alphabet) and performs operations to check if swapping characters could make the frequency of non-zero characters in the count arrays equal.

The time complexity of counting characters is `O(n)` for each word, where `n` is the length of the word. However, the nested loops create a bigger time complexity issue. There are 26 possible characters, leading to 26*26 comparisons in the worst case which, is `O(26^2)` or simply `O(1)` since 26 is a constant and does not change with the input size.

Combining these, the overall time complexity is primarily affected by the character counting, so `O(n)` for `word1` plus `O(m)` for `word2`, where `n` and `m` are the lengths of `word1` and `word2` respectively. Since the 26*26 operations are constant time and do not scale with `n` or `m`, the insignificant additional constant time doesn't affect the overall complexity.

The total time complexity is `O(n + m)`, where `n` is the length of `word1` and `m` is the length of `word2`.

Space Complexity

The space complexity is much simpler to analyze. The space required by the algorithm is the space for the two count arrays `cnt1` and `cnt2` which hold the frequency of each character. As these arrays have a fixed size of 26, regardless of the input size, the space complexity is `O(1)`.

Putting it all together, the space complexity is `O(1)` because it only requires fixed space for the frequency counts and a few variables for iteration and comparison, which does not scale with the input size.