2839. Check if Strings Can be Made Equal With Operations I

Easy String

Problem Description

In this problem, we are given two strings `s1` and `s2`, each exactly 4 characters long, consisting of lowercase English letters. The task is to determine if we can transform string `s1` into string `s2` through a series of operations. The operation allowed in this context is to swap any two characters in either string, given that these characters are exactly two indices apart (that is, there is one character between them).

The goal is to decide if it's possible to make `s1` identical to `s2` by applying this operation zero or more times on any of the strings and return `true` if it is possible, or `false` otherwise.

Intuition

Upon examining the operation allowed, we realize that it only enables us to swap characters that are two indices apart. For a 4-character string, this means we could swap the 1st character with the 3rd, and the 2nd character with the 4th, but not any other pairs. Therefore, the characters that start at even indices (0 and 2 in zero-based indexing) can only be moved within their own set of positions, and the same goes for characters at odd indices (1 and 3).

To decide if it's possible to make the two strings equal, we only need to check if, within these two groups of indices (even and odd), the sets of characters are the same for both `s1` and `s2`.

The reference solution approach leverages precisely this observation. It separately sorts the characters located at even and odd indices for both `s1` and `s2` and checks if they are equal. If both the even and odd indexed characters match after sorting, that means `s1` can be transformed into `s2` through the allowed operation.

Here's why:

- The even indexed characters can only be swapped among themselves. The same applies to the odd indexed characters.
- Whether `s1` can be transformed into `s2` does not depend on the initial order of the characters within the even and odd index groups, just on the presence of the same characters.
- Sorting the characters in a specific group (even or odd) gives their ordered sequence based on the character values, regardless of their initial order.
- If the sequences match after sorting for both even and odd indexed groups, it affirms that one can achieve one string from the other through the swapping operation.

Solution Approach

The implementation of the solution uses Python's list slicing and sorting capabilities. The key to the solution lies in separating the characters based on their index being even or odd and then comparing these subsets from `s1` and `s2`. Here are the main components of this approach:

- **List Slicing**: In Python, slicing is a feature that allows for accessing parts of sequences like strings, lists, or tuples. In the given solution, `s1[::2]` and `s2[::2]` are used to access characters from `s1` and `s2` that are at even indices, which are 0 and 2 for a 4-character string. Similarly, `s1[1::2]` and `s2[1::2]` are used to access the characters at odd indices, which are 1 and 3.
- **Sorting**: Sorting is used to rearrange the characters in a specific order. This is done to easily compare whether the same set of characters are present in both `s1` and `s2` for the respective positions (even and odd). The `sorted()` function in Python returns a new sorted list from the elements of any iterable.
- **Equality Check**: After sorting, the two lists of characters (each for the even and odd positions) are compared using the equality operator `==`. If two lists are equal, it means they contain the same elements in the same order.

Now looking at the code implementation based on this approach:

```
class Solution:
    def canBeEqual(self, s1: str, s2: str) -> bool:
        # Sort and compare characters at even indices of s1 and s2
        even_positions_match = sorted(s1[::2]) == sorted(s2[::2])
        # Sort and compare characters at odd indices of s1 and s2
        odd_positions_match = sorted(s1[1::2]) == sorted(s2[1::2])
        # Return True if both matches are True, otherwise False
        return even_positions_match and odd_positions_match
```

The two separate boolean variables `even_positions_match` and `odd_positions_match` store the result of whether or not the characters at even and odd positions, respectively, match between `s1` and `s2`. The function finally returns `True` if both sets of positions match after sort, which indicates the operation can be performed to make the strings equal, otherwise, it returns `False`.

In terms of complexity, since the strings are of constant length, both the space and time complexity of the operations on the strings are also constant, O(1), regardless of the actual implementation detail of the `sorted()` function.

Example Walkthrough

Let's take an example to understand how the solution approach works. Suppose `s1` is `abcd` and `s2` is `cbad`. We wish to know if `s1` can be transformed into `s2` by swapping any two characters that are two indices apart in either string.

Now we can slice and sort the characters from `s1` and `s2` based on their indices being even or odd and compare them:

- For the even indices (0 and 2):
  - `s1` has the characters `'a'` and `'c'` at these indices.
  - `s2` has the characters `'c'` and `'a'`.

  Sort these characters and we get the following:
  - `s1` sorted even characters: `'ac'` → `'ac'`
  - `s2` sorted even characters: `'ca'` → `'ac'`

  Since `'ac'` is equal to `'ac'`, the even-indexed characters match.
- For the odd indices (1 and 3):
  - `s1` has the characters `'b'` and `'d'` at these indices.
  - `s2` has the characters `'b'` and `'d'`.

  Sort these characters as well:
  - `s1` sorted odd characters: `'bd'` → `'bd'`
  - `s2` sorted odd characters: `'bd'` → `'bd'`

  Since `'bd'` is equal to `'bd'`, the odd-indexed characters also match.

Since the sorted subsets for both the even and odd indices from `s1` match those in `s2`, we can confirm that `s1` can indeed be transformed into `s2` using the allowed operation.

Therefore, when we use the described method on our example, the `canBeEqual` function in the Solution class would return `True`. Here's how the class would process our example:

```
# Instantiate the solution class
solution = Solution()
# Call the canBeEqual function with s1 and s2
result = solution.canBeEqual('abcd', 'cbad')
# Output the result
print(result)  # This would print True to the console
```

By applying the solution's approach to our example, we are able to walk through the steps to determine that `s1` can be transformed into `s2` as per the defined operation.

Solution Implementation

Python

```
class Solution:
    def canBeEqual(self, target: str, arr: str) -> bool:
        # Check if the even-indexed characters of both strings (when sorted) are equal
        even_index_equal = sorted(target[::2]) == sorted(arr[::2])

        # Check if the odd-indexed characters of both strings (when sorted) are equal
        odd_index_equal = sorted(target[1::2]) == sorted(arr[1::2])

        # Return True if both even and odd indexed characters are equal, otherwise False
        return even_index_equal and odd_index_equal

# The code checks if strings 'target' and 'arr' can be made equal by rearranging the characters,
# under the condition that characters at even and odd indices are compared separately.
```

Java

```
class Solution {
    // Method to determine if two strings can be made equal by reordering characters
    public boolean canBeEqual(String s1, String s2) {
        // Counter arrays to store the frequency of each character at even and odd indices
        int[][] count = new int[2][26];

        // Iterate over the characters of the strings
        for (int i = 0; i < s1.length(); ++i) {
            // Increase the count for the i-th character of s1 in the respective count array (even or odd)
            ++count[i & 1][s1.charAt(i) - 'a'];

            // Decrease the count for the i-th character of s2 in the respective count array (even or odd)
            --count[i & 1][s2.charAt(i) - 'a'];
        }

        // Check each character's frequency to ensure both strings s1 and s2 are equal
        for (int i = 0; i < 26; ++i) {
            // If any character count does not match in even or odd index counts, strings cannot be equal
            if (count[0][i] != 0 || count[1][i] != 0) {
                return false;
            }
        }

        // If all character counts match, the strings can be made equal by reordering
        return true;
    }
}
```

C++

```
#include <vector>
#include <string>

class Solution {
public:
    // Function to determine if two strings can be made equal by rearranging characters
    bool canBeEqual(std::string str1, std::string str2) {
        // Create a 2D vector to keep character counts for odd and even indices separately
        std::vector<std::vector<int>> charCounts(2, std::vector<int>(26, 0));

        // Increment/decrement character counts based on their occurrences at odd/even positions
        for (int i = 0; i < str1.size(); ++i) {
            // Increment count for current character in str1 at index i
            ++charCounts[i & 1][str1[i] - 'a']; // i & 1 alternates between 0 and 1 for even and odd indices
            // Decrement count for current character in str2 at index i
            --charCounts[i & 1][str2[i] - 'a'];
        }

        // Check if all counts are zero, which means strings can be rearranged to be equal
        for (int i = 0; i < 26; ++i) {
            // If there's any non-zero count, strings cannot be equal
            if (charCounts[0][i] != 0 || charCounts[1][i] != 0) {
                return false;
            }
        }

        // All counts are zero, strings can be made equal
        return true;
    }
};
```

TypeScript

```
/**
 * Checks if two strings can be made equal by swapping characters.
 *
 * @param {string} str1 - The first string to be compared.
 * @param {string} str2 - The second string to be compared.
 * @returns {boolean} - Returns true if str1 can be made equal to str2, otherwise false.
 */
function canBeEqual(str1: string, str2: string): boolean {
    // Initialize a 2x26 array to count the frequency of each alphabet letter in odd/even positions.
    const charCounts: number[][] = Array.from({ length: 2 }, () => Array.from({ length: 26 }, () => 0));

    // Iterate over letters in the strings and update the counts.
    for (let i = 0; i < str1.length; ++i) {
        // Increment the count for the current character in str1 at position i (even or odd).
        ++charCounts[i & 1][str1.charCodeAt(i) - 'a'.charCodeAt(0)];
        // Decrement the count for the current character in str2 at the same position.
        --charCounts[i & 1][str2.charCodeAt(i) - 'a'.charCodeAt(0)];
    }

    // Check if there are any non-zero counts i.e. unmatched characters in either string.
    for (let i = 0; i < 26; ++i) {
        if (charCounts[0][i] !== 0 || charCounts[1][i] !== 0) {
            // If there's an unmatched character, the strings cannot be made equal.
            return false;
        }
    }

    // If all counts are zero, the strings can be made equal.
    return true;
}
```

```
class Solution:
    def canBeEqual(self, target: str, arr: str) -> bool:
        # Check if the even-indexed characters of both strings (when sorted) are equal
        even_index_equal = sorted(target[::2]) == sorted(arr[::2])

        # Check if the odd-indexed characters of both strings (when sorted) are equal
        odd_index_equal = sorted(target[1::2]) == sorted(arr[1::2])

        # Return True if both even and odd indexed characters are equal, otherwise False
        return even_index_equal and odd_index_equal

# The code checks if strings 'target' and 'arr' can be made equal by rearranging the characters,
# under the condition that characters at even and odd indices are compared separately.
```

Time and Space Complexity

Time Complexity

The time complexity of the provided algorithm is driven by the sorting operations performed on sliced strings. Python uses Timsort, an optimized merge sort that, in the general case, has a time complexity of `O(n log n)`.

Since the string `s1` is sliced into two halves (`s1[::2]` and `s1[1::2]`), and the same is done for `s2`, these operations can be considered to work on roughly half the length of the input strings each. For a string of length `n`, each slice would have a size of about `n/2`.

The `sorted()` function is then applied to each slice, resulting in a time complexity of `O(n/2 * log(n/2))` for each sort operation. There are four such sort operations, two for `s1` and two for `s2`.

Multiplying this by four (since there are four slices), the overall time complexity is `4 * O(n/2 * log(n/2))`, which simplifies to `O(n log n)` because the constants do not change the order of growth.

Space Complexity

The space complexity is driven by the space required to store the sorted slices. Sorting a list in Python is not done in-place (when using the `sorted()` function); rather, it creates a new list. Therefore, for a slice of `n/2` elements, the space required is `O(n/2)`.

Since we sort four slices independently, and assuming only the largest of these affects the overall space complexity, we could argue that the space complexity is `O(n)`. However, all these sorts are not in memory at the same time unless in the case of concurrent execution, which isn't specified here. Therefore, the space complexity is `O(n/2)`, which simplifies to `O(n)` because dropping constants is standard in Big O notation.