2840. Check if Strings Can be Made Equal With Operations II

Medium Hash Table String Sorting Leetcode Link

Problem Description

The problem provides us with two strings `s1` and `s2` of the same length, which contain only lowercase English letters. We are allowed to perform a specific kind of operation to try to make these two strings equal. This operation involves selecting any two indices `i` and `j` within a string such that `i < j` and the gap between them (`j - i`) is an even number. We can then swap the characters at these two indices. Our task is to determine whether it is possible to make the two strings equal by applying this operation any number of times to either of the strings.

Intuition

To understand the solution approach, consider this: since we can only swap characters at indices with an even difference, each character initially at an even index can only be swapped with another character at an even index. The same goes for characters at odd indices. This implies that no matter how many operations we perform, the set of characters at the even indices and the set of characters at the odd indices will remain the same for each respective string.

Given this, if we want to make `s1` equal to `s2`, the sorted sequence of characters at the even indices must be the same for both strings, and similarly, the sorted sequence of characters at the odd indices must also be the same for both strings.

The provided solution reflects this intuition. It checks whether the sorted list of characters at even indices (`s1[::2]` and `s2[::2]`) and at odd indices (`s1[1::2]` and `s2[1::2]`) are equal for both `s1` and `s2`. If both the even and odd sequences match, the function returns `true`, meaning the two strings can be made equal; otherwise, it returns `false`.

Solution Approach

The implementation of the solution is straightforward given the intuition behind the problem.

Here's a breakdown of the implementation using Python:

1. `s1[::2]`: This part of the code takes the string `s1` and slices it to get every second character starting from index 0 (all the even indices).
2. `sorted(s1[::2])`: Sorted is a built-in Python function that takes an iterable and returns a new list with the elements sorted. In this case, it sorts the characters from `s1` that are at the even indices.
3. `s1[1::2]`: This does the same as step 1 but for odd indices, starting at index 1.
4. `sorted(s1[1::2])`: Again, sorted is used to sort the list of characters at odd indices for `s1`.

Now, this is done for both strings `s1` and `s2`.

The reason behind using sorting is that if `s1` can be transformed into `s2` by swapping characters, the sorted order of the characters at odd and even indices must be the same for both strings after any number of swaps, since a swap does not change the relative order of the characters.

5. `return sorted(s1[::2]) == sorted(s2[::2]) and sorted(s1[1::2]) == sorted(s2[1::2])`: This line of code checks two conditions:
   - That the sorted characters at even indices in both strings are identical.
   - That the sorted characters at odd indices in both strings are identical.

If both conditions are `True`, the function returns `True`, indicating that the strings can indeed be made equal by applying the aforementioned operation. If any of the conditions are not met, it returns `False`, meaning it's impossible to make the strings equal using the allowed operations.

Data structures used:

- **Lists**: The sorted function returns a new list of sorted items. Lists are fundamental Python data structures for maintaining ordered collections of items.

Patterns used:

- **Two-pointer approach**: Implicitly, the solution uses a kind of two-pointer technique where we consider the characters at even and odd index positions as two separate sequences.
- **Sorting**: Sorting is a common pattern used when we want to arrange data to easily compare elements or check for equality, as seen here.

By employing this approach, the solution ensures an efficient and effective way to determine if the strings can be made equal without having to simulate any swaps.

Example Walkthrough

Let's take two strings, `s1 = "aabb"` and `s2 = "abab"`, and walk through the solution approach to determine if `s1` can be transformed into `s2` using the specified operations:

1. First, we look at the even-indexed characters of `s1` and `s2`. For `s1` (`"aabb"`), the characters at the even indices are `"a"` and `"a"` (indices 0 and 2). For `s2` (`"abab"`), the characters at the even indices are `"a"` and `"b"` (indices 0 and 2).
   - `s1` even indices: `s1[::2]` → `"aa"`
   - `s2` even indices: `s2[::2]` → `"ab"`
2. We sort these characters to see if a transformation is possible:
   - `sorted(s1[::2])` → `"aa"`
   - `sorted(s2[::2])` → `"ab"`
3. Now, let's examine the odd-indexed characters. In `s1` (`"aabb"`), the characters at the odd indices are `"b"` and `"b"` (indices 1 and 3). In `s2` (`"abab"`), the characters at the odd indices are `"b"` and `"a"` (indices 1 and 3).
   - `s1` odd indices: `s1[1::2]` → `"bb"`
   - `s2` odd indices: `s2[1::2]` → `"ba"`
4. Sort these characters as well:
   - `sorted(s1[1::2])` → `"bb"`
   - `sorted(s2[1::2])` → `"ab"`
5. Finally, we compare the sorted characters at both the even and odd indices for `s1` and `s2`.
   - For even indices: `"aa"` (from `s1`) is not equal to `"ab"` (from `s2`)
   - For odd indices: `"bb"` (from `s1`) is not equal to `"ab"` (from `s2`)
6. Since the sorted sequences of `s1` and `s2` at both even and odd indices are not matching, we can conclude that it's impossible to make the two strings equal by swapping characters. Therefore, the result is `False`.

This example clearly shows that if the sorted sets of both even and odd indices characters are not equal, then there's no operation that can be performed to make the strings match.

Python Solution

```python
class Solution:
    def check_strings(self, s1: str, s2: str) -> bool:
        """
        Check if strings s1 and s2 have the same characters at even and odd indices
        respectively, after sorting those characters.

        Parameters:
        s1 (str): First input string to compare.
        s2 (str): Second input string to compare.

        Returns:
        bool: True if both even and odd indexed characters of s1 and s2 match after sorting,
              otherwise False.
        """

        # Extract characters from even indices and sort them, compare whether they are the same for s1 and s2.
        even_index_chars_match = sorted(s1[::2]) == sorted(s2[::2])

        # Extract characters from odd indices and sort them, compare whether they are the same for s1 and s2.
        odd_index_chars_match = sorted(s1[1::2]) == sorted(s2[1::2])

        # Both even and odd indexed characters should match for the condition to be true.
        return even_index_chars_match and odd_index_chars_match

# Example of how the function can be used:
# solution = Solution()
# result = solution.check_strings("a1b2c", "1abc2")
# print(result) # Output will be either True or False based on the method's logic

```

Java Solution

```java
class Solution {

    public boolean checkStrings(String s1, String s2) {
        // Define a 2D array to hold the count of each character in both strings.
        // One row for counting characters at even indices and another for odd indices.
        int[][] charCount = new int[2][26];

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

            // Decrement the count of the current character in s2 in the same part
            // of the array used for s1 character counting.
            charCount[i & 1][s2.charAt(i) - 'a']--;
        }

        // Check the count arrays for each alphabet.
        // The goal is to determine if s1 and s2 have the same number of each character
        // at the corresponding even and odd indices.
        for (int i = 0; i < 26; ++i) {
            if (charCount[0][i] != 0 || charCount[1][i] != 0) {
                // If any count is not 0, s1 and s2 do not have the same characters
                // at the same positions, hence return false.
                return false;
            }
        }

        // If all counts are 0, both strings have the same characters at each index
        // (even and odd), and thus the method returns true.
        return true;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // This method checks if two strings s1 and s2 can become equal by swapping any
    // of the characters an arbitrary number of times, but only between the same
    // parity indices (i.e., swap s[i] with s[j] only if i and j are both even or both odd).
    bool checkStrings(string s1, string s2) {
        // `counts` is a 2D vector that keeps track of the frequency of each character in s1 and s2.
        // The first index (0 or 1) represents the parity (even or odd position in the string),
        // the second index represents the characters ('a' to 'z').
        vector<vector<int>> counts(2, vector<int>(26, 0)); // Initializing to 0

        // Loop through the strings
        for (int i = 0; i < s1.size(); ++i) {
            // Increment the count for the character at the current index in s1
            // based on its parity (even or odd index)
            ++counts[i & 1][s1[i] - 'a'];

            // Decrement the count for the character at the current index in s2
            // based on its parity (even or odd index)
            --counts[i & 1][s2[i] - 'a'];
        }

        // Check the counts for each character. If any count is non-zero after the
        // above operations, the strings cannot be made equal by swapping characters,
        // hence return false.
        for (int i = 0; i < 26; ++i) {
            if (counts[0][i] != 0 || counts[1][i] != 0) {
                return false;
            }
        }

        // If we reach this point, it means all character counts are balanced between s1 and s2
        // for both parities, so we can return true.
        return true;
    }
};

```

Typescript Solution

```typescript
function checkStrings(s1: string, s2: string): boolean {
    // Initialize a 2x26 matrix to count the occurrence of each letter in both strings.
    // The first dimension represents whether the character index is even or odd.
    const counts: number[][] = Array.from({ length: 2 }, () => new Array(26).fill(0));

    // Iterate over the characters of the first string to update the counts.
    for (let index = 0; index < s1.length; ++index) {
        // Incrementing the count of the current character for the respective string index parity
        counts[index & 1][s1.charCodeAt(index) - 'a'.charCodeAt(0)]++;
        // Decrementing the count of the current character for the s2 string with the same index parity
        counts[index & 1][s2.charCodeAt(index) - 'a'.charCodeAt(0)]--;
    }

    // Iterate over the alphabet letters to check their counts.
    for (let i = 0; i < 26; ++i) {
        if (counts[0][i] !== 0 || counts[1][i] !== 0) {
            // If any count doesn't cancel out, strings don't have the same character frequency per parity.
            return false;
        }
    }

    // If all counts cancel out, the strings have the same character frequency per index parity.
    return true;
}

```

Time and Space Complexity

The time complexity of the code is `O(n log n)`, where `n` is the length of the strings `s1` and `s2`. This is because the code performs sorting operations on the even and odd indexed characters separately. Sorting an array is typically `O(l log l)`, where `l` is the length of the array being sorted. Since the code sorts half of the characters in the strings at a time (even and odd indexed), each sort operation works on approximately `n/2` elements, leading to a complexity of `O((n/2) log (n/2))`. Simplifying this expression (since constants can be ignored in Big O notation), we still conclude `O(n log n)`.

The space complexity of the code is `O(n)`. When sorting, a new list is created with the characters to be sorted. For both the even and odd indexed characters, this means that a list of size roughly `n/2` is created twice. This leads to a space requirement of `O(n)`, as the extra space needed for sorting does not depend on the size of the input, but on a list created based on half the size of the input, which is still linear with `n`.