1247. Minimum Swaps to Make Strings Equal

Medium Greedy Math String

Problem Description

In this problem, we are given two strings `s1` and `s2`. Both strings have the same length and contain only the characters `'x'` and `'y'`. Our objective is to make the two strings identical by performing swaps between the characters of the two strings. A swap involves taking one character from `s1` and one character from `s2` and exchanging them. The goal is to determine the minimum number of swaps required to make the two strings identical. If it's not possible to make the strings equal, we must return `-1`.

Intuition

The solution is based on counting how many characters are out of place when comparing both strings. We look for characters that are `'x'` in `s1` and `'y'` in `s2`, and vice versa, since these are the only ones that need to be swapped to make the strings equal. There are two types of mismatches:

- An `'xy'` mismatch: where `s1` has an `'x'` and `s2` has a `'y'` at the same position.
- A `'yx'` mismatch: where `s1` has a `'y'` and `s2` has an `'x'` at the same position.

1. If we have an even number of `'xy'` and `'yx'` mismatches, we can always solve the problem.
2. We can swap odd mismatches within themselves, for instance, an odd count of `'xy'` mismatches can be made even by performing one swap inside the `'xy'` set (changing two `'xy'` to two `'yx'`). This adds one more swap to our answer for each odd count.
3. Each pair of mismatches ('xy' with 'yx') can be solved with one swap, so the pair count contributes directly to the total number of swaps.

The algorithm checks if there is an even total number of mismatches. If it's odd, we return `-1` because we would always end up with one character that cannot be matched. If it's even, we sum the half of each count (because a swap fixes two mismatches) and case for odd counts, which contributes one additional swap for each.

Solution Approach

The implementation of the solution uses a simple counter-based approach without any sophisticated data structures or patterns. The core of the solution revolves around counting the discrepancies between the two strings and categorizing them as `xy` or `yx`, followed by calculating the number of swaps needed based on these counts.

Let's walk through the steps present in the Python code provided:

1. Initialize two counters `xy` and `yx` to 0. These will hold the counts for mismatches where `s1[i]` is 'x' and `s2[i]` is 'y', and where `s1[i]` is 'y' and `s2[i]` is 'x', respectively.
2. Iterate over both strings in parallel by using the `zip` function, which allows us to obtain pairs of characters (a, b) from `s1` and `s2`.
3. For each pair (a, b), increment the `xy` counter if `a < b`, which means we have an 'x' in `s1` and a 'y' in `s2`. Similarly, increment the `yx` counter if `a > b`, which indicates a 'y' in `s1` and an 'x' in `s2`.
4. Once we've counted all mismatches, we check whether the sum of `xy` and `yx` is even. If it's not (`(xy + yx) % 2` is truthy), we return `-1` because this means it's impossible to make the strings equal.
5. If the sum is even, we calculate the number of swaps. We calculate `xy // 2` to see how many pairs of 'xy' mismatches can be swapped directly, which also applies to `yx // 2` pairs of 'yx' mismatches. Additionally, for each type of mismatch, if there's an odd one out (checked by `xy % 2` and `yx % 2`), it requires an extra swap.
6. The minimum number of swaps required is the sum of these values, which is `xy // 2 + yx // 2 + xy % 2 + yx % 2`, meaning the number of direct swaps for pairs plus the extra swaps for any remaining mismatch.

The logic applied here uses basic arithmetic and logic to deduce the number of required actions to align the two strings. It's a pragmatic approach that is both efficient and clear in its purpose.

Example Walkthrough

Let's illustrate the solution approach with a small example. Consider two strings `s1 = "xyyx"` and `s2 = "yxyx"`.

Following the steps of the solution:

1. Initialize counters `xy` and `yx` to 0.
2. Iterate over both strings `s1` and `s2` and compare the corresponding characters using `zip`:
   - For the first pair (`s1[0] = 'x'`, `s2[0] = 'y'`), since `s1[0] < s2[0]`, increment `xy` to 1.
   - For the second pair (`s1[1] = 'y'`, `s2[1] = 'x'`), `s1[1] > s2[1]`, so increment `yx` to 1.
   - The third pair (`s1[2] = 'y'`, `s2[2] = 'y'`) matches, so no counter is incremented.
   - For the fourth pair (`s1[3] = 'x'`, `s2[3] = 'x'`), again they match, so no counter is increased.

   After this step, `xy` equals 1, and `yx` equals 1.
3. Check if the sum of `xy` and `yx` is even: `1 + 1 = 2` which is even, so continue.
4. Calculate the minimum number of swaps:
   - Pairs of 'xy' mismatches that can be swapped directly: `xy // 2 = 1 // 2 = 0`.
   - Pairs of 'yx' mismatches that can be swapped directly: `yx // 2 = 1 // 2 = 0`.
   - An extra swap is needed for the one 'xy' mismatch: `xy % 2 = 1 % 2 = 1`.
   - An extra swap is needed for the one 'yx' mismatch: `yx % 2 = 1 % 2 = 1`.
5. The total number of swaps needed is: `xy // 2 + yx // 2 + xy % 2 + yx % 2 = 0 + 0 + 1 + 1 = 2`.

Therefore, for `s1 = "xyyx"` and `s2 = "yxyx"`, it takes a minimum of 2 swaps to make the strings identical. The swaps that could be performed would be between the first character of `s1` with the second character of `s2`, and the second character of `s1` with the first character of `s2`. After these swaps, both `s1` and `s2` will be `"xyxy"`.

Solution Implementation

Python

```python
class Solution:
    def minimumSwap(self, s1: str, s2: str) -> int:
        # Counters for "x-y" and "y-x" pairs
        count_xy = count_yx = 0

        # Iterate over characters of both strings simultaneously
        for char1, char2 in zip(s1, s2):
            # If we find a "x-y" pair, increment count_xy
            if char1 == 'x' and char2 == 'y':
                count_xy += 1
            # If we find a "y-x" pair, increment count_yx
            elif char1 == 'y' and char2 == 'x':
                count_yx += 1

        # If the sum of both counts is odd, we can't make them equal, thus return -1
        if (count_xy + count_yx) % 2 != 0:
            return -1

        # For pairs "xy" or "yx" that match directly, each pair takes one swap
        swaps = count_xy // 2 + count_yx // 2

        # For the remaining unpaired "xy" or "yx", they take two swaps to balance
        # As there would be one "xy" and one "yx" remaining for an even total of mismatches
        swaps += 2 * (count_xy % 2)

        return swaps
```

Java

```java
class Solution {
    public int minimumSwap(String s1, String s2) {
        // Counters for 'x' in s1 and 'y' in s2, and 'y' in s1 and 'x' in s2
        int countXY = 0, countYX = 0;

        // Loop through the strings to count 'xy' and 'yx' pairs
        for (int i = 0; i < s1.length(); i++) {
            char charS1 = s1.charAt(i), charS2 = s2.charAt(i);
            // If s1 has 'x' and s2 has 'y', increment countXY
            if (charS1 == 'x' && charS2 == 'y') {
                countXY++;
            }
            // If s1 has 'y' and s2 has 'x', increment countYX
            if (charS1 == 'y' && charS2 == 'x') {
                countYX++;
            }
        }

        // If the sum of countXY and countYX is odd, return -1, as it's impossible to swap to equality
        if ((countXY + countYX) % 2 == 1) {
            return -1;
        }

        // The minimum swaps is the sum of:
        // - Half the pairs of 'xy' and 'yx', as two swaps can solve two pairs,
        // - and one swap for each of the remaining 'xy' or 'yx' if there is an odd count.
        return countXY / 2 + countYX / 2 + countXY % 2 + countYX % 2;
    }
}
```

C++

```cpp
class Solution {
public:
    // Function to calculate the minimum number of swaps to make two strings equal
    int minimumSwap(string s1, string s2) {
        int countXY = 0; // Number of occurrences where 'x' in s1 is aligned with 'y' in s2
        int countYX = 0; // Number of occurrences where 'y' in s1 is aligned with 'x' in s2

        // Loop through both strings character by character
        for (int i = 0; i < s1.size(); ++i) {
            char charFromS1 = s1[i], charFromS2 = s2[i];
            if (charFromS1 == 'x' && charFromS2 == 'y') {
                // Increment count for 'x' in s1 and 'y' in s2
                countXY++;
            } else if (charFromS1 == 'y' && charFromS2 == 'x') {
                // Increment count for 'y' in s1 and 'x' in s2
                countYX++;
            }
        }

        // If the sum of countXY and countYX is odd, it's not possible to make the strings equal
        if ((countXY + countYX) % 2 != 0) {
            return -1; // Return -1 to indicate the impossibility
        }

        // The number of swaps is calculated by adding:
        // - Half of countXY because two 'xy' pairs can be corrected by a single swap
        // - Half of countYX for the same reason as above
        // - The remainder of countXY divided by 2, because one 'xy' cannot be solved without an extra 'yx' pair
        //   Similarly, one cannot solve an extra 'yx' without an extra 'xy' so countYX % 2 is also needed
        return countXY / 2 + countYX / 2 + countXY % 2 + countYX % 2;
    }
};
```

TypeScript

```typescript
function minimumSwap(s1: string, s2: string): number {
    let countXY = 0, // count of 'x' in s1 and 'y' in s2 at the same position
        countYX = 0; // count of 'y' in s1 and 'x' in s2 at the same position

    // Iterate through the strings to count 'xy' and 'yx' pairs
    for (let i = 0; i < s1.length; ++i) {
        const charS1 = s1[i],
              charS2 = s2[i];

        if (charS1 === 'x' && charS2 === 'y') {
            ++countXY; // Increase count for 'xy' pair
        }
        if (charS1 === 'y' && charS2 === 'x') {
            ++countYX; // Increase count for 'yx' pair
        }
    }

    // Check if the sum of 'xy' and 'yx' pairs is odd, return -1 since it's not possible to swap to equality
    if ((countXY + countYX) % 2 === 1) {
        return -1;
    }

    // Calculate minimum swaps:
    // Each pair of 'xy' or 'yx' requires one swap
    // If there's an odd number of 'xy' or 'yx', it takes two additional swaps to fix both
    return Math.floor(countXY / 2) + Math.floor(countYX / 2) + (countXY % 2) + (countYX % 2);
}
```

```python
class Solution:
    def minimumSwap(self, s1: str, s2: str) -> int:
        # Counters for "x-y" and "y-x" pairs
        count_xy = count_yx = 0

        # Iterate over characters of both strings simultaneously
        for char1, char2 in zip(s1, s2):
            # If we find a "x-y" pair, increment count_xy
            if char1 == 'x' and char2 == 'y':
                count_xy += 1
            # If we find a "y-x" pair, increment count_yx
            elif char1 == 'y' and char2 == 'x':
                count_yx += 1

        # If the sum of both counts is odd, we can't make them equal, thus return -1
        if (count_xy + count_yx) % 2 != 0:
            return -1

        # For pairs "xy" or "yx" that match directly, each pair takes one swap
        swaps = count_xy // 2 + count_yx // 2

        # For the remaining unpaired "xy" or "yx", they take two swaps to balance
        # As there would be one "xy" and one "yx" remaining for an even total of mismatches
        swaps += 2 * (count_xy % 2)

        return swaps
```

Time and Space Complexity

Time Complexity

The time complexity of the given function is `O(n)`, where `n` is the length of the strings `s1` and `s2`. This is because the function consists of a single loop that iterates through all characters of `s1` and `s2` in a pairwise manner using the `zip` function. Within the loop, only constant time operations are performed (comparison and addition).

Space Complexity

The space complexity of the given function is `O(1)`. There are only a fixed number of integer variables (`xy` and `yx`) initialized and used for counting the occurrences, which do not depend on the size of the input strings. Thus, the amount of additional memory required remains constant regardless of the input size.