859. Buddy Strings

Easy Hash Table String

Leetcode Link

Problem Description

The problem provides two strings, `s` and `goal`, and asks to determine if it's possible to make them equal by performing exactly one swap of two characters within the string `s`. Swapping characters involves taking any two positions `i` and `j` in the string (where `i` is different from `j`) and exchanging the characters at these positions. The goal is to return `true` if `s` can be made equal to `goal` after one such swap, otherwise `false`.

Intuition

To solve this problem, we first address some basic checks before we move on to character swaps:

1. **Length Check**: If the lengths of `s` and `goal` aren't the same, it's impossible for one to become the other with a single swap.
2. **Character Frequency Check**: If `s` and `goal` don't have the same frequency of characters, one cannot become the other, as a single swap doesn't affect the frequency of characters.

After these initial checks, we look for the differences between `s` and `goal`:

- **Exact 2 Differences**: If there are precisely two positions at which `s` and `goal` differ, these could potentially be the two characters we need to swap to make the strings equal. For instance, if `s = "ab"` and `goal = "ba"`, swapping these two characters would make the strings equal.
- **Zero Differences with Duplicates**: If there are no differences, we need at least one character in `s` that occurs more than once. This way, swapping the duplicates won't alter the string but will satisfy the condition of making a swap. For example, if `s = "aa"` and `goal = "aa"`, we can swap the two 'a's to meet the requirement.

The solution returns `true` if either condition is fulfilled - a single swap can rectify exactly 2 differences, or no differences exist and at least one character has a duplicate in `s`. Otherwise, it returns `false`.

Solution Approach

The implementation of the solution adheres to the intuition described earlier and uses a couple of steps to determine if we can swap two letters in the string `s` to match `goal`. Here's how the solution is accomplished:

1. Length Check:
   - First, we compare the lengths of `s` and `goal` using `len(s)` and `len(goal)`. If they are different, we immediately return `False`.
2. Character Frequency Check:
   - Two `Counter` objects from the `collections` module are created, one for each string. The `Counter` objects `cnt1` and `cnt2` count the frequency of each character in `s` and `goal`, respectively.
   - We then compare these `Counter` objects. If they are not equal, it means that `s` and `goal` have different sets of characters or different character frequencies, so we return `False`.
3. Differing Characters:
   - We iterate through `s` and `goal` concurrently, checking for characters at the same indices that are not equal. This is done using a comprehension expression that checks `s[i] != goal[i]` for each `i` from `0` to `n-1`.
   - We sum the total number of differences found, and if the sum is exactly `2`, it implies there is a pair of characters that can be swapped to make `s` equal to `goal`.
4. Zero Differences with Duplicates:
   - If there are no differences (`diff == 0`), we check if any character in `s` has a count greater than `1` using `any(v > 1 for v in cnt1.values())`. This would mean that there is at least one duplicate character that can be swapped.
5. Return Value:
   - The function returns `True` if the sum of differing characters `diff` is exactly `2` or if there is no difference and there is at least one duplicate character. Otherwise, it returns `False`.

The overall solution makes use of Python's dictionary properties for quick character frequency checks, and the efficiency of set operations for comparing the two `Counter` objects. The integration of these checks allows the function to quickly determine whether a single swap can equate two strings, making the solution both concise and effective.

Example Walkthrough

Let's consider a small example to illustrate the solution approach using the strings `s = "xy"` and `goal = "yx"`. We want to determine if making one swap in `s` can make it equal to `goal`.

Step 1: Length Check
`len(s) == len(goal)`
Both strings have the same length of 2 characters.

Step 2: Character Frequency Check
`Counter(s)` produces `Counter({'x': 1, 'y': 1})`,
`Counter(goal)` produces `Counter({'y': 1, 'x': 1})`.
Comparing these counts, we see they match, which means `s` and `goal` have the same characters with the same frequency.

Step 3: Differing Characters
As `s[0] != goal[0]` ('x' != 'y') and `s[1] != goal[1]` ('y' != 'x'),
we have exactly two positions where `s` and `goal` differ.

Step 4: Zero Differences with Duplicates
This step is only relevant if there were no differences identified in the earlier step. As we have found two differing characters, this step can be skipped.

Step 5: Return Value
Since there are exactly two differences, we can swap the characters 'x' and 'y' in string `s` to make it equal to `goal`.
Thus, the function should return `True`.

Applying these steps to our example `s = "xy"` and `goal = "yx"` confirms that the solution approach correctly yields a `True` result, as a single swap is indeed sufficient to make `s` equal to `goal`.

Python Solution

```python
from collections import Counter

class Solution:
    def buddyStrings(self, a: str, b: str) -> bool:
        # Lengths of both strings
        len_a, len_b = len(a), len(b)

        # If lengths are not equal, they cannot be buddy strings
        if len_a != len_b:
            return False

        # Count characters in both strings
        counter_a, counter_b = Counter(a), Counter(b)

        # If character counts are not the same, then it's not a simple swap case
        if counter_a != counter_b:
            return False

        # Count the number of positions where the two strings differ
        difference_count = sum(1 for i in range(len_a) if a[i] != b[i])

        # Return True if there are exactly two differences
        # (which can be swapped to make the strings equal)
        # Or if there's no difference and there are duplicate characters in the string
        # (which can be swapped with each other while keeping the string the same)
        return difference_count == 2 or (difference_count == 0 and any(value > 1 for value in counter_a.values()))
```

Java Solution

```java
class Solution {
    public boolean buddyStrings(String s, String goal) {
        int lengthS = s.length(), lengthGoal = goal.length();

        // If the lengths are not equal, they can't be buddy strings
        if (lengthS != lengthGoal) {
            return false;
        }

        // If there are differences in characters, we will count them
        int diffCounter = 0;

        // Arrays to count occurrences of each character in s and goal
        int[] charCountS = new int[26];
        int[] charCountGoal = new int[26];

        for (int i = 0; i < lengthGoal; ++i) {
            int charS = s.charAt(i), charGoal = goal.charAt(i);

            // Increment character counts
            ++charCountS[charS - 'a'];
            ++charCountGoal[charGoal - 'a'];

            // If characters at this position differ, increment differences
            if (charS != charGoal) {
                ++diffCounter;
            }
        }

        // To track if we find any character that occurs more than once
        boolean duplicateCharFound = false;

        for (int i = 0; i < 26; ++i) {
            // If character counts differ, they can't be buddy strings
            if (charCountS[i] != charCountGoal[i]) {
                return false;
            }

            // Check if there's any character that occurs more than once
            if (charCountS[i] > 1) {
                duplicateCharFound = true;
            }
        }

        // The strings can be buddy strings if there are exactly two differences
        // or no differences but at least one duplicate character in either string
        return differences == 2 || (differences == 0 && duplicateCharFound);
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    // Define the buddyStrings function to check if two strings can become equal by swapping exactly one pair of characters
    bool buddyStrings(string sInput, string goalInput) {
        int lengthS = sInput.size(), lengthGoal = goalInput.size();
        // If string lengths must match, otherwise it is not possible to swap just one pair
        if (lengthS != lengthGoal) return false;

        // Counter to keep track of differences
        int diffCounter = 0;
        // Counters to store frequency of characters in both strings
        vector<int> freqS(26, 0);
        vector<int> freqGoal(26, 0);

        // Iterate through both strings to fill freq arrays and count differences
        for (int i = 0; i < lengthGoal; ++i) {
            ++freqS[sInput[i] - 'a'];
            ++freqGoal[goalInput[i] - 'a'];
            if (sInput[i] != goalInput[i]) ++diffCounter; // Increment diffCounter when characters are not same
        }

        // Duplicate found flag, initially false
        bool hasDuplicate = false;
        // Check if the strings have different frequency of any character
        for (int i = 0; i < 26; ++i) {
            if (freqS[i] != freqGoal[i]) return false; // Frequencies must match for a valid swap
            if (freqS[i] > 1) hasDuplicate = true;     // If any character occurs more than once, we can potentially swap duplicates
        }

        // Valid buddy strings have either:
        // 2 differences (swap those and strings become equal)
        // No differences but at least one duplicate character (swap duplicates and strings remain equal)
        return diffCounter == 2 || (diffCounter == 0 && hasDuplicate);
    }
};
```

Typescript Solution

```typescript
function buddyStrings(inputString: string, goalString: string): boolean {
    // Lengths of the input strings
    const inputLength = inputString.length;
    const goalLength = goalString.length;

    // If lengths are not equal, strings cannot be buddy strings
    if (inputLength !== goalLength) {
        return false;
    }

    // Arrays to hold character counts for each string
    const charCountInput = new Array(26).fill(0);
    const charCountGoal = new Array(26).fill(0);

    // Variable to count the number of positions where characters differ
    let differences = 0;

    // Iterate over the strings and populate character counts
    for (let i = 0; i < goalLength; ++i) {
        charCountInput[inputString.charCodeAt(i) - 'a'.charCodeAt(0)]++;
        charCountGoal[goalString.charCodeAt(i) - 'a'.charCodeAt(0)]++;

        // If characters at the same position differ, increment differences
        if (inputString[i] !== goalString[i]) {
            ++differences;
        }
    }

    // Compare character counts for both strings
    for (let i = 0; i < 26; ++i) {
        if (charCountInput[i] !== charCountGoal[i]) {
            // If counts do not match, strings cannot be buddy strings
            return false;
        }
    }

    // Return true if there are exactly two differences or no differences but at least one character with more than one occurrence
    return differences === 2 || (differences === 0 && charCountInput.some(count => count > 1));
}
```

Time and Space Complexity

Time Complexity

The time complexity of the code is determined by several factors:

1. The length comparison of `s` and `goal` strings which takes `O(1)` time since length can be checked in constant time in Python.
2. The construction of `Counter` objects for `s` and `goal` is `O(m)` and `O(n)` respectively, where `m` and `n` are the lengths of the strings. Since `m` is equal to `n`, it simplifies to `O(n)`.
3. The comparison of the two `Counter` objects is `O(n)` because it involves comparing the count of each unique character from both strings.
4. The calculation of `diff`, which involves iterating through both strings and comparing characters, is `O(n)`.

Since all these steps are sequential, the overall time complexity is `O(n) + O(n) + O(n) + O(n) = O(n)`, where `n` is the length of the input strings.

Space Complexity

The space complexity is based on the additional space required by the algorithm which is primarily due to the `Counter` objects:

1. Two `Counter` objects for `s` and `goal`, each of which will have at most `k` unique characters where `k` is the size of the character set used in the strings. The space complexity for this part is `O(k)`.
2. The additional space for the variable `diff` is negligible, `O(1)`.

If we assume a fixed character set (like the ASCII set), `k` could be considered constant and the complexity is `O(1)` regarding the character set. However, the more precise way to describe it would be `O(k)` based on the size of the character set.

Therefore, the total space complexity of the algorithm can be expressed as `O(k)`.