2937. Make Three Strings Equal

Easy String

Problem Description

You are provided with three strings, `s1`, `s2`, and `s3`, and you can perform a particular operation on them any number of times. The operation allows you to choose one of these strings and delete its rightmost character, but only if the chosen string is at least 2 characters long. The goal is to determine the fewest number of these operations required to make all three strings identical. If it is impossible to make the three strings equal through any number of operations, the answer should be `-1`.

Intuition

When looking for a way to make the three strings equal by deleting characters, we must focus on their commonalities. Since we can only delete characters from the end of a string, the strings could only be made equal if they share a common prefix. The longest common prefix that the strings share will determine the operations. By iterating from the start of the strings and comparing characters at the same index across all three strings, we can identify the length of the common prefix.

Solution Approach

The solution provided uses a straightforward, brute force approach to determine how we can make the strings equal by trimming them from the right. We do not need any complex data structures or algorithms for this; we rely on basic string manipulation and iteration.

To begin with, we calculate the sum `s` of the lengths of the three strings. This sum gives us the total number of characters we have in the strings initially.

We then identify the shortest string length `n` by taking the minimum length of the three strings, since the longest possible common prefix cannot exceed the length of the shortest string.

With these initial calculations out of the way, we iterate over each string up to the length of the shortest string (index `0` to `n - 1`). On each iteration, we compare the characters of `s1`, `s2`, and `s3` at the current index.

If we find that the characters at index `i` are not identical in all three strings, this means that the length of the common prefix is `i`. In this case, we calculate the number of operations as `s - 3 * i`, which essentially subtracts three times the length of the common prefix (because we would have to delete the remaining characters from each string beyond the common prefix).

But if we find that the first characters themselves do not match (`i == 0`), we immediately return `-1` since no amount of operations will make the strings equal.

If the loop completes without finding any mismatch, this implies that all strings share a common prefix of length `n`. Therefore, we return `s - 3 * n` since the total number of operations needed is to delete every character beyond the length of the shortest string, from all three strings.

This approach does not require any special data structures, as it simply involves comparing string characters and basic arithmetic to calculate the required value. The solution is an enumeration strategy that covers all potential cases, ensuring correctness by exhaustively checking each character until the end of the shortest string is reached or a mismatch is encountered.

Example Walkthrough

Let's say we have three strings:

- `s1` = "abcde"
- `s2` = "abfg"
- `s3` = "abcef"

Now, we need to apply the solution approach to find out the fewest number of operations required to make all three strings identical:

1. Sum the lengths of `s1`, `s2`, and `s3`: `sum = 5 + 4 + 5 = 14`
2. Identify the shortest string length `n` (in this case, it's `s2` with length `4`).
3. Start iterating from index 0 to index n-1 (0 to 3 in this case) and compare the characters of `s1`, `s2`, and `s3` at each index.
4. Index 0: All strings have 'a' - so continue.
5. Index 1: All strings have 'b' - so continue.
6. Index 2: `s1` has 'c', `s2` has 'f', and `s3` has 'c'. Here, the characters do not match, so the length of the common prefix is `2`.
7. Calculate the operation count: `operations = sum - 3 * length_of_prefix = 14 - 3 * 2 = 14 - 6 = 8`.

Thus, we need a minimum of 8 operations to make the strings identical, which means we need to delete the last two characters of `s1` and the last two characters of `s3`, and then delete all characters of `s2` beyond the common prefix "ab".

Solution Implementation

Python

```python
class Solution:
    def findMinimumOperations(self, string1: str, string2: str, string3: str) -> int:
        # Calculate the sum of lengths of all strings
        total_length = len(string1) + len(string2) + len(string3)

        # Find the minimum length among the three strings
        minimum_length = min(len(string1), len(string2), len(string3))

        # Iterate over the strings up to the minimum length to check for common prefix
        for index in range(minimum_length):
            # If the characters at the current position are different,
            # there is no common prefix at this index or beyond
            if not string1[index] == string2[index] == string3[index]:
                return -1 if index == 0 else total_length - 3 * index

        # If the loop completes, the substrings up to 'minimum length' are the same
        # hence, we subtract the length of the common prefix for each string
        return total_length - 3 * minimum_length
```

Java

```java
class Solution {

    /**
     * Find the minimum number of operations to make s1, s2, and s3 not have the same character
     * at index i where 0 <= i < min(s1.length, s2.length, s3.length)
     *
     * @param s1 first string
     * @param s2 second string
     * @param s3 third string
     * @return the minimum number of operations required or -1 if no operation is needed
     */
    public int findMinimumOperations(String s1, String s2, String s3) {
        // Calculate the total length of all strings
        int totalLength = s1.length() + s2.length() + s3.length();
        // Find the length of the shortest string
        int minLength = Math.min(Math.min(s1.length(), s2.length()), s3.length());

        // Iterate over the strings up to the length of the shortest string
        for (int i = 0; i < minLength; ++i) {
            // Check if the characters at the current index are not the same
            if (!(s1.charAt(i) == s2.charAt(i) && s2.charAt(i) == s3.charAt(i))) {
                // If it's the first character that is different, return -1 (no operation needed)
                // Otherwise, return the total number of remaining characters in all strings
                return i == 0 ? -1 : totalLength - 3 * i;
            }
        }

        // If all characters up to the length of the shortest string are the same,
        // return the total number of remaining characters in all strings
        return totalLength - 3 * minLength;
    }
}
```

C++

```cpp
class Solution {
public:
    // Function to find the minimum number of operations to make all strings equal
    // s1, s2, and s3 are the input strings
    int findMinimumOperations(string s1, string s2, string s3) {
        // Calculate the sum of lengths of all three strings
        int total_length = s1.size() + s2.size() + s3.size();
        // Find the length of the smallest string
        int smallest_length = min({s1.size(), s2.size(), s3.size()});
        // Iterate over the range of the smallest string length
        for (int i = 0; i < smallest_length; ++i) {
            // Check if the characters at the current index are not equal in all strings
            if (!(s1[i] == s2[i] && s2[i] == s3[i])) {
                // If the first characters are not equal, we cannot perform the operation
                return i == 0 ? -1 : total_length - 3 * i;
            }
        }
        // If all the characters are equal up to the smallest string length, subtract the
        // corresponding triple count from the total length of all strings
        return total_length - 3 * smallest_length;
    }
};
```

TypeScript

```typescript
// Function to find the minimum number of operations to make strings non-overlapping
function findMinimumOperations(str1: string, str2: string, str3: string): number {
    // Calculate the sum of the lengths of the strings
    const totalLength = str1.length + str2.length + str3.length;

    // Find the length of the smallest string
    const smallestLength = Math.min(str1.length, str2.length, str3.length);

    // Loop through each character up to the length of the smallest string
    for (let i = 0; i < smallestLength; ++i) {
        // If at any position the characters of the three strings are not equal
        if (!(str1[i] === str2[i] && str2[i] === str3[i])) {
            // If no matching character at beginning, return -1 (no removal possible)
            return i === 0 ? -1 : totalLength - 3 * i;
        }
    }

    // If all characters up to the smallest string's length match, return adjusted totalLength
    // This is because if all characters match, the overlap is until the end of the smallest string
    return totalLength - 3 * smallestLength;
}
```

```python
class Solution:
    def findMinimumOperations(self, string1: str, string2: str, string3: str) -> int:
        # Calculate the sum of lengths of all strings
        total_length = len(string1) + len(string2) + len(string3)

        # Find the minimum length among the three strings
        minimum_length = min(len(string1), len(string2), len(string3))

        # Iterate over the strings up to the minimum length to check for common prefix
        for index in range(minimum_length):
            # If the characters at the current position are different,
            # there is no common prefix at this index or beyond

            # Check if this is the first character; if so, there's no common prefix at all
            if not string1[index] == string2[index] == string3[index]:
                return -1 if index == 0 else total_length - 3 * index

        # If the loop completes, the substrings up to 'minimum length' are the same
        # hence, we subtract the length of the common prefix for each string
        return total_length - 3 * minimum_length
```

Time and Space Complexity

The function `findMinimumOperations` is a simple loop that executes up to `n` iterations, where `n` is the minimum length of the input strings `s1`, `s2`, and `s3`. This loop runs only once through the shortest string to check the condition and hence, it operates in linear time with respect to the shortest string length. Therefore, the time complexity is `O(n)`.

Regarding the space complexity, the function uses a fixed number of single-value variables (`s` and `n`) and does not create any data structures that grow with the input size. Thus, the amount of additional memory used is constant, making the space complexity `O(1)`.