Documentetion

Intuition

This problem requires finding the extra character added to string t after shuffling string s. A straightforward way to solve this is by comparing character frequencies, but that requires extra space. Instead, we can use the XOR operation, which has unique properties that make it well-suited for this problem. The key insight is that when we XOR the same character twice, it cancels out (e.g., a ^ a = 0). Since string t contains all characters of s plus one additional character, XORing all characters in s + t will leave only the extra character. This approach provides an efficient and elegant solution without needing additional data structures.

Understanding XOR in This Context

XOR (exclusive OR) is a bitwise operation that follows specific mathematical rules:

1. Any number XORed with itself results in zero (x ^ x = 0).
2. Any number XORed with zero remains unchanged (x ^ 0 = x).
3. XOR is both commutative and associative, meaning order does not matter (a ^ b ^ c = c ^ a ^ b).

These properties help us efficiently determine the extra character in t. Since every character in s appears twice (once in s, once in t), their XOR results in zero. The only character left after XORing all characters in s + t is the additional character that was added.

Step-by-Step Approach

To implement this approach, we initialize a variable result to zero. We then iterate through each character in both s and t, converting them to their ASCII values using the ord() function and applying the XOR (^) operation. Because XOR of two identical numbers results in zero, the characters appearing in both s and t cancel each other out. Since t contains exactly one extra character, this is the only value left in result after performing XOR on all characters. Finally, we convert this ASCII value back to a character using the chr() function and return it.

Example Walkthrough

Consider an example where s = "abcd" and t = "abcde". The character sequence for XORing would be:

a ^ a ^ b ^ b ^ c ^ c ^ d ^ d ^ e = e

All matching characters cancel out, leaving only e, which is the extra character. Another example is s = "" and t = "y". Since s is empty, XORing y with zero still results in y, which is the correct answer. This demonstrates how XOR effectively isolates the additional character.

Time and Space Complexity Analysis

The time complexity of this approach is **O(n)**, where n is the length of string s. This is because we iterate through s + t exactly once. The space complexity is **O(1)** since we only use a single integer variable (result) to store the XOR result. Unlike other methods that rely on hash tables or arrays to store character counts, this approach does not require additional memory beyond a few integer operations.

Alternative Approaches

A commonly used alternative approach is to count the frequency of each character in s and t using a dictionary or an array. This method involves iterating through s to store counts and then iterating through t to find the extra character. While effective, it requires additional space (O(1) for an array of 26 letters or O(n) for a dictionary). Another approach involves sorting both s and t and comparing them character by character, but this has a higher time complexity of **O(n log n)** due to sorting.

Final Thoughts

Using XOR to solve this problem is both optimal and elegant. It eliminates the need for extra memory while maintaining an efficient **O(n)** runtime. This approach leverages fundamental bitwise properties to find the extra character with minimal computation. By understanding how XOR works and applying it strategically, we can solve problems like this in a highly efficient manner. This method is widely applicable in scenarios where we need to detect a single anomaly in a dataset with paired values.