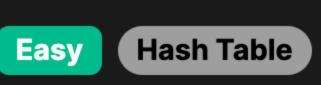
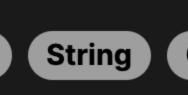
1790. Check if One String Swap Can Make Strings Equal







Problem Description

Given two strings s1 and s2 of equal length, our task is to determine if we can make them equal by performing at most one string swap on exactly one of the strings. A string swap involves choosing any two indices within one string and swapping the characters at these indices. We are to return true if the strings can be equalized in this manner or false if it's not possible.

Intuition

To solve this problem, we must consider that if two strings are to be equal after at most one swap, then there can only be two characters that differ between them. If there are no characters that differ, the strings are already equal. If there is only one pair of characters that differ, swapping them in either of the strings would make the strings equal. However, if there are more than two characters that differ, it will not be possible to make the strings equal with just one swap.

The solution approach employs a simple iteration wherein we traverse both strings s1 and s2 using a single for loop. Throughout

Solution Approach

the iteration, we keep a count (cnt) of the positions where the characters in s1 and s2 are different. Additionally, we make use of two variables c1 and c2 to hold the pair of characters that were different on the first occurrence. In Python, the zip function is utilized to iterate over characters from both strings simultaneously, providing a convenient way to

compare characters at the same indices from s1 and s2. When a difference is found (i.e., characters a from s1 and b from s2 are not the same), the cnt counter is incremented. The algorithm considers three main scenarios:

If after the traversal, the value of cnt is zero, it means that all the characters at corresponding indices are the same, and hence, s1 is already equal to s2. Thus we can return True.

- If cnt becomes greater than 2 at any point during the iteration, it means there are too many differences to correct with a single swap, and the function immediately returns False.
- Lastly, if the function encounters exactly two differences (cnt equals 2), it checks if the current pair of differing characters (a, b) could be swapped with the first pair (c1, c2) to make the strings equal. This is assessed by checking if the current
- differing character from s1 (c1). If this condition doesn't hold, then it's not possible with a single swap to make the strings equal, and we return False. At the end of the iteration, we also need to return False if cnt is exactly one, because a single difference can't be rectified with a swap. Hence, we can conclude that True is only returned if cnt is exactly 0 or 2, signifying that no swaps or exactly one swap can

character from s1 (a) equals the first differing character from s2 (c2), and the current character from s2 (b) equals the first

equalize the strings. **Example Walkthrough**

determine if a single swap can make these two strings equal. We start iterating through both strings from the first character: s1[0] = 'c' and s2[0] = 'c'. Since the characters are the

same, we move on. The same goes for the second, third, and fourth characters: s1[1] = 'o' and s2[1] = 'o', s1[2] = 'n' and s2[2] = 'n',

Let's consider a small example with the strings s1 = "converse" and s2 = "conserve". As per the description, we need to

- When we reach the fifth character, we notice a difference: s1[4] = 'e' and s2[4] = 's'. This is our first difference, so cnt is incremented to 1 and we store the differing characters in variables: c1 = 'e' and c2 = 's'.
- The sixth characters are equal again: s1[5] = 'r' and s2[5] = 'r'.

At the seventh character, there's another difference: s1[6] = 's' and s2[6] = 'e'. We increment cnt to 2 and note this

Since cnt equals 2, we now check if the characters s1[6] and s2[4] would form a valid swap with the first pair, c1 and c2. We

def areAlmostEqual(self, string1: str, string2: str) -> bool:

for char_string1, char_string2 in zip(string1, string2):

if char_string1 != char_string2:

if (++mismatchCount > 2 ||

return false;

return mismatchCount != 1;

if (charA != charB) {

};

return false;

difference_count += 1

return False

If characters don't match, increase the difference count.

remains 2 by the end of the iteration and no further discrepancies arise.

second pair of differing characters.

s1[3] = 'v' and s2[3] = 'v'. All are equal; hence cnt remains 0.

- find that s1[6] equals c2, and s2[6] equals c1, meaning swapping s1[4] with s1[6] will make s1 equal to s2. The remaining characters s1[7] = 'e' and s2[7] = 'r', s1[8] = 'r' and s2[8] = 'v' are also equal, confirming that cnt
- Because cnt is exactly 2 and the single swap condition is met, we return True. The single swap needed would be to swap s1[4] with s1[6] resulting in s1 becoming conserve, which is equal to s2.

Using this approach allows us to go through each character of both strings and efficiently determine whether a single swap can

make the two strings equal without any unnecessary comparisons or operations.

Python

Initialize the count of different characters and placeholders for characters that differ.

Check if there are more than 2 differences or if the swap doesn't make strings equal

if difference_count > 2 or (difference_count == 2 and (char_string1 != char2 or char_string2 != char1)):

difference_count = 0 char1 = char2 = None# Iterate through characters of both strings in parallel.

class Solution:

Solution Implementation

```
# Record the first set of different characters.
                char1, char2 = char_string1, char_string2
       # If there's exactly one difference, the strings cannot be made equal with one swap.
        return difference_count != 1
Java
class Solution {
    public boolean areAlmostEqual(String s1, String s2) {
       // Initialize a counter for the number of non-matching character pairs between s1 and s2.
       int mismatchCount = 0;
       // Initialize variables to store characters from non-matching character pairs.
       char firstCharFromS1 = 0, firstCharFromS2 = 0;
       // Iterate over the strings to compare characters at each index.
        for (int i = 0; i < s1.length(); ++i) {</pre>
           // Get the current characters from each string.
            char charFromS1 = s1.charAt(i), charFromS2 = s2.charAt(i);
           // Check for non-matching characters
           if (charFromS1 != charFromS2) {
```

// If this is the second mismatch but does not form a transposable pair with the first mismatch, return false

// Increment the difference counter and check if it exceeds 2. If it does, return false as more than one swap wor

if (++differenceCount > 2 || (differenceCount == 2 && (charA != charFromStr2 || charB != charFromStr1))) {

// Update the characters that were found to mismatch for comparison when the next mismatch occurs

(mismatchCount == 2 && !(charFromS1 == firstCharFromS2 && charFromS2 == firstCharFromS1))) {

// If more than two non-matching pairs, strings are already not almost equal.

// If there is exactly one mismatch, strings cannot be made equal by a single swap.

```
// Store the characters from the first non-matching character pair.
firstCharFromS1 = charFromS1;
firstCharFromS2 = charFromS2;
```

```
C++
class Solution {
public:
   bool areAlmostEqual(string str1, string str2) {
       // Initialize a counter to track the number of positions where str1 and str2 differ
       int differenceCount = 0;
       // Variables to store the characters from each string when a mismatch is found
       char charFromStr1 = 0, charFromStr2 = 0;
```

// Iterate through both strings to compare character by character

// If there is a mismatch, we'll need to check further

charFromStr1 = charA, charFromStr2 = charB;

// then the strings cannot be made equal with one swap.

char1, char2 = char_string1, char_string2

become equal by swapping at most one pair of characters in one of the strings.

If there's exactly one difference, the strings cannot be made equal with one swap.

for (int index = 0; index < str1.size(); ++index) {</pre>

char charA = str1[index], charB = str2[index];

// Strings are almost equal if there were zero or two mismatches.

```
// If there was exactly one mismatch, strings cannot be made equal with a single swap
       // Return true if difference count is 0 or 2 (since they can be equal after exactly one swap); otherwise, return false
       return differenceCount != 1;
TypeScript
// This function checks if two strings are almost equal by allowing one swap of two characters in one string
function areAlmostEqual(string1: string, string2: string): boolean {
    let firstMismatchedCharFromS1: string; // to store the character from string1 involved in the first mismatch
    let firstMismatchedCharFromS2: string; // to store the character from string2 involved in the first mismatch
    let mismatchCount = 0; // to keep track of the number of mismatches found
   // Loop over each character in the strings to check for mismatches
   for (let i = 0; i < string1.length; ++i) {</pre>
       const charFromS1 = string1.charAt(i);
       const charFromS2 = string2.charAt(i);
       // If a mismatch is found
       if (charFromS1 !== charFromS2) {
           mismatchCount++; // we increment the mismatch counter
           // If more than two mismatches are found, or if at the second mismatch the
```

// mismatching characters are not the transposed characters from the first mismatch,

```
if (mismatchCount > 2 || (mismatchCount === 2 && (charFromS1 !== firstMismatchedCharFromS2 || charFromS2 !== firstMis
                  return false;
              // If this is the first mismatch encountered, store the mismatching characters
              if (mismatchCount === 1) {
                  firstMismatchedCharFromS1 = charFromS1;
                  firstMismatchedCharFromS2 = charFromS2;
      // Strings are considered almost equal if there are no mismatches or exactly two mismatches
      return mismatchCount !== 1;
class Solution:
   def areAlmostEqual(self, string1: str, string2: str) -> bool:
       # Initialize the count of different characters and placeholders for characters that differ.
       difference_count = 0
        char1 = char2 = None
       # Iterate through characters of both strings in parallel.
        for char_string1, char_string2 in zip(string1, string2):
           # If characters don't match, increase the difference count.
           if char_string1 != char_string2:
               difference_count += 1
               # Check if there are more than 2 differences or if the swap doesn't make strings equal
               if difference_count > 2 or (difference_count == 2 and (char_string1 != char2 or char_string2 != char1)):
                    return False
               # Record the first set of different characters.
```

The code provided implements a function to check if two strings are almost equal. That means they are equal or they can

Time and Space Complexity

return difference_count != 1

the code iterates over each character of the strings exactly once through the use of the zip function. The conditional statement inside the loop has constant-time complexity checks (0(1)), thus, they don't affect the overall linear

The time complexity of the given code is O(n), where n is the length of the strings s1 and s2. This time complexity arises because

time complexity of iterating through the strings. **Space Complexity:**

Time Complexity:

The space complexity of the given code is 0(1). No additional space that scales with the input size is required. The variables cnt, c1, and c2 use constant space, only storing a fixed number of elements (at most two characters and a counter) regardless of the input size.

Since the code operates in-place, checking the characters of the input strings without creating any additional data structures or recursive calls, the space complexity remains constant.