1153. String Transforms Into Another String

#### Hash Table String Hard

transform str1 into str2. The transformation follows a specific rule where in one conversion step, we can choose any character in str1 and convert every occurrence of that character into any other lowercase English character. The question is whether str1 can be transformed into str2 after zero or more such conversions.

In this problem, we are given two strings str1 and str2 that are of the same length. We need to determine if it's possible to

**Problem Description** 

Intuition The intuition behind the solution is to check if a one-to-one mapping exists from each character in str1 to each character in str2.

If we find that one character in str1 maps to multiple characters in str2, the transformation is not possible, because one

character in str1 can only be transformed into one character in str2. Another key observation is that if str2 consists of all 26

English lowercase characters, the transformation would not be possible because there wouldn't be any available characters left

1. Immediately return True if str1 is already equal to str2, because no transformations are needed. 2. Check if str2 has all 26 lowercase characters. If it does, return False because we can't make any character in str1 map uniquely to an

The approach, therefore, is to:

to map a character from str1 that isn't already in str2.

Here is the step-by-step approach to the implementation:

unmapped character in str2. 3. Create a dictionary to keep track of the mappings from characters in str1 to characters in str2. 4. Iterate through pairs of corresponding characters from str1 and str2. For each pair of characters: • If we encounter a character from str1 that hasn't been mapped yet, add the mapping to the dictionary.

o If we encounter a character from str1 that has been mapped but the mapping doesn't match the current character from str2, it means a

single character in str1 is mapping to multiple characters in str2, so we return False. 5. If we successfully map every character from str1 to a character in str2 without conflicts, and we don't run into the situation where str2 has all 26 characters, we return True.

- Solution Approach The solution to the problem uses a dictionary as a data structure to maintain a mapping from characters in str1 to characters in

str2. The pattern used here is akin to a graph mapping problem where each vertex (character from str1) should map to a unique vertex (character from str2).

## Check for Equality: The first check is to see if str1 and str2 are the same, in which case the function returns True - because nothing needs to be done.

if str1 == str2:

return True

Check for Maximum Characters in str2: We determine the number of distinct characters in str2 by converting it into a set and counting the elements. If all 26 lowercase letters are present, it's impossible to map any character from str1 to a new character, as there are no available characters left. Hence, return False.

**Dictionary Mapping**: We create an empty dictionary d where each key-value pair will represent a character mapping from

**Iterating Over Characters**: We use the zip function to iterate over pairs of corresponding characters from str1 and str2.

**if** len(set(str2)) == 26: return False

str1 to str2.

for a, b in zip(str1, str2):

if a not in d:

**Example Walkthrough** 

steps.

characters in str2, so we return False.

from str1 are successfully mapped to str2, and the transformation is possible.

mapping strategy, validating the constraints imposed by the problem statement.

**Character Mapping:** We start mapping the characters using dictionary d.

For the first pair (a, c), since a is not in d, we add d[a] = c.

The fifth pair (c, e) is also matching because d[c] is indeed e.

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

# If the input strings are equal, no conversion is needed.

# as there would be no spare character to map a transition.

# If the loop completes with no conflicts, conversion is possible.

**Character Mapping:** Here we determine the mapping for each character: If a character a from str1 is encountered for the first time, we create a new entry in the dictionary d with a as the key and b (the corresponding character from str2) as the value.

If the character a has already been assigned a mapping, we check to see if the stored value (previous mapping) matches

the current character b from str2. If they do not match, it means one character from str1 is trying to map to different

Successfully Mapped: If the loop completes without hitting a mismatching mapping, we return True, meaning all characters

d[a] = belif d[a] != b: return False

To summarize, this implementation's algorithm checks the feasibility of character transformation using a simple dictionary

Let's illustrate the solution approach with a simple example. Assume we're given two strings: str1 = "aabcc" and str2 = "ccdee".

Check for Equality: We first check if str1 is equal to str2. In this case, str1 is not equal to str2, so we continue with the next

Check for Maximum Characters in str2: We examine str2 to see if it contains all 26 lowercase letters. str2 = "ccdee" only

**Dictionary Mapping:** We create an empty dictionary d to hold our character mappings from str1 to str2. **Iterating Over Characters**: Using the zip function we pair up characters from both strings:

has 3 unique characters ('c', 'd', 'e'), so it doesn't contain all 26 characters. We proceed to the mapping phase.

### ○ Pair 4: (c, e) ∘ Pair 5: (c, e)

approach.

**Python** 

class Solution:

if string1 == string2:

if len(set(string2)) == 26:

return False

return True

return true;

int uniqueCharsCount = 0;

int[] charFrequency = new int[26];

// Length of strings 'str1' and 'str2'.

if char1 not in mapping\_dict:

mapping\_dict[char1] = char2

elif mapping\_dict[char1] != char2:

// Tracks the count of unique characters in 'str2'.

// Array to store the frequency of characters in 'str2'.

// Function to determine if it's possible to convert 'str1' into 'str2'

// Populate the character count array for 'str2' and count distinct chars

// If there are 26 distinct characters, no conversion is possible

// Array to keep track of character mappings from 'str1' to 'str2'

// Verify if the characters can be mapped from 'str1' to 'str2'

// If both strings are equal, no conversion is needed

// Count the occurrences of each character in 'str2'

// Number of distinct characters in 'str2'

if (++charCount[c - 'a'] == 1) {

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

int indexStr1 = str1[i] - 'a';

int indexStr2 = str2[i] - 'a';

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

for char1, char2 in zip(string1, string2):

mapping\_dict[char1] = char2

elif mapping\_dict[char1] != char2:

if char1 not in mapping\_dict:

# If the input strings are equal, no conversion is needed.

# as there would be no spare character to map a transition.

# Iterate over both strings to populate the mapping dictionary.

# If the loop completes with no conflicts, conversion is possible.

# If `string2` has all 26 letters, there's no way to convert `string1` to `string2`

# Mapping dictionary to track corresponding characters from `string1` to `string2`.

# If char1 is encountered for the first time, add it to the mapping\_dict.

# If char1 is already mapped to a different character, conversion isn't possible.

++distinctChars;

bool canConvert(string str1, string str2) {

**if** (str1 == str2) {

return true;

int charCount[26] = {0};

int distinctChars = 0;

for (char c : str2) {

if (distinctChars == 26) {

int charMapping[26] = {0};

return false;

return True

return False

∘ Pair 1: (a, c)

∘ Pair 2: (a, c)

∘ Pair 3: (b, d)

• The second pair (a, c) again has a. We find that d[a] is c, which matches, so we continue. The third pair (b, d), since b is not in d, we add d[b] = d. The fourth pair (c, e), c is not in d, we add d[c] = e.

26 characters, the mapping is successful. Therefore, we can transform str1 into str2 following the defined rules.

In conclusion, this example confirms that str1 can be transformed into str2 using the algorithm explained in the solution

Successfully Mapped: As we have gone through all the character pairs without conflict, and since str2 does not contain all

Solution Implementation

# If `string2` has all 26 letters, there's no way to convert `string1` to `string2`

# Mapping dictionary to track corresponding characters from `string1` to `string2`.

# If char1 is already mapped to a different character, conversion isn't possible.

mapping\_dict = {} # Iterate over both strings to populate the mapping dictionary. for char1, char2 in zip(string1, string2): # If char1 is encountered for the first time, add it to the mapping\_dict.

```
class Solution {
    public boolean canConvert(String str1, String str2) {
       // If strings are equal, no conversion is required.
       if (str1.equals(str2)) {
```

Java

```
int length = str1.length();
       // Count the occurrences of each character in 'str2'.
        for (int i = 0; i < length; ++i) {</pre>
            if (++charFrequency[str2.charAt(i) - 'a'] == 1) {
                // If a character appears for the first time, increase the unique character count.
                ++uniqueCharsCount;
       // If there are 26 unique characters in 'str2', there is no spare character for conversion.
       if (uniqueCharsCount == 26) {
            return false;
       // Array to track the mapping from characters in 'str1' to 'str2'.
       int[] mapping = new int[26];
        // Build the mapping by relating characters in 'str1' to 'str2'.
        for (int i = 0; i < length; ++i) {</pre>
           // Obtain the indices in the alphabet for the current characters being mapped.
           int indexStr1 = str1.charAt(i) - 'a';
            int indexStr2 = str2.charAt(i) - 'a';
           // If it's the first time this character from 'str1' is encountered, map it to the character in 'str2'.
           if (mapping[indexStr1] == 0) {
                // Store the mapping one more than the index since '0' is the default value and cannot be used to represent 'a'.
                mapping[indexStr1] = indexStr2 + 1;
            } else if (mapping[indexStr1] != indexStr2 + 1) {
                // If the character has been seen before and maps to a different character, the conversion is not possible.
                return false;
       // If the loop completes without returning false, the conversion is possible.
       return true;
C++
```

#include <string>

class Solution {

public:

using namespace std;

```
// If the character from 'str1' has not been mapped yet, map it
            if (charMapping[indexStr1] == 0) {
                charMapping[indexStr1] = indexStr2 + 1; // '+1' to differentiate from default value '0'
           // If there is a mismatch in the expected mapping, return false
           else if (charMapping[indexStr1] != indexStr2 + 1) {
                return false;
       // If all characters can be mapped without conflicts, return true
       return true;
TypeScript
// Determines if string 'str1' can be converted to 'str2' by replacing characters.
function canConvert(str1: string, str2: string): boolean {
    // If both strings are equal, no conversion is needed.
    if (str1 === str2) {
       return true;
    // If 'str2' has all possible 26 characters, it's impossible to find an available character to map to.
    if (new Set(str2).size === 26) {
       return false;
    // Mapping from characters of 'str1' to 'str2'.
    const charMap: Map<string, string> = new Map();
    // Iterate over 'str1' and construct the mapping to 'str2'.
    for (const [index, char] of str1.split('').entries()) {
       // If the character is not in the map, add the character with its counterpart from 'str2'.
       if (!charMap.has(char)) {
            charMap.set(char, str2[index]);
       } else if (charMap.get(char) !== str2[index]) {
           // If the mapping is inconsistent, conversion is not possible.
            return false;
   // If we reach this point, conversion is possible.
    return true;
```

# **Time Complexity** The function canConvert consists of several operations. Let's analyze them step by step:

case is generally considered.

Time and Space Complexity

return True

class Solution:

if string1 == string2:

if len(set(string2)) == 26:

return False

return True

return False

mapping\_dict = {}

2. The if len(set(str2)) == 26 creates a set from str2, and it takes O(M) time, where M is the length of str2, because each character must be checked and inserted into the set. Checking the length of the set is 0(1) operation. 3. The loop for a, b in zip(str1, str2) iterates over the characters in str1 and str2, which takes O(N) time as well. 4. Inside the loop, the dictionary d is used to map characters from str1 to str2. Checking if a key is in the dictionary and assigning a value to a key

5. The elif d[a] != b condition is also an O(1) operation. Since these operations are sequential, the overall time complexity is dominated by the iteration of the two strings, leading to an

1. The if str1 == str2 check is an O(N) operation, where N is the length of the strings, as it involves comparing each character in both strings.

is an 0(1) operation on average due to hash table implementation. In the worst case, it could become 0(N) due to collision handling, but average

- average case time complexity of O(N), where N is the length of the strings provided to the function. The set operation bears the same complexity due to the length M of str2, but since both N and M are the lengths of the given strings, and the strings are
- generally considered to be of roughly equal length in this context, O(M) can also be considered O(N) for the sake of complexity analysis.

Therefore, the time complexity of the function is O(N) on average. **Space Complexity** 

The space complexity of the function canConvert can be analyzed as follows:

consist of lowercase English letters. Therefore, at most, the dictionary takes 0(1) space.

- 1. The set created from str2 in if len(set(str2)) == 26 takes O(M) space, where M is the number of unique characters in str2, but since this is limited to a maximum of 26 (the number of letters in the English alphabet), this can also be considered 0(1) space. 2. The dictionary d can have a maximum of 26 key-value pairs since it's a mapping from characters of str1 to str2, and both strings can only
- Hence, the overall space complexity is 0(1), as both the set and the dictionary are bounded by a constant maximum size of 26.