1153. String Transforms Into Another String

Hard Hash Table String Leetcode Link

Problem Description

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 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.

--

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

The approach, therefore, is to:

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 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.
   - 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`).

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

1. **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
   ```

2. **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`.

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

3. **Dictionary Mapping**: We create an empty dictionary `d` where each key-value pair will represent a character mapping from `str1` to `str2`.

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

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

5. **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 characters in `str2`, so we return `False`.

     ```
     if a not in d:
         d[a] = b
     elif d[a] != b:
         return False
     ```

6. **Successfully Mapped**: If the loop completes without hitting a mismatching mapping, we return `True`, meaning all characters from `str1` are successfully mapped to `str2`, and the transformation is possible.

To summarize, this implementation's algorithm checks the feasibility of character transformation using a simple dictionary mapping strategy, validating the constraints imposed by the problem statement.

Example Walkthrough

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

1. **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 steps.

2. **Check for Maximum Characters in `str2`**: We examine `str2` to see if it contains all 26 lowercase letters. `str2 = "ccdee"` only has 3 unique characters (`'c'`, `'d'`, `'e'`), so it doesn't contain all 26 characters. We proceed to the mapping phase.

3. **Dictionary Mapping**: We create an empty dictionary `d` to hold our character mappings from `str1` to `str2`.

4. **Iterating Over Characters**: Using the `zip` function we pair up characters from both strings:

   - Pair 1: `(a, c)`
   - Pair 2: `(a, c)`
   - Pair 3: `(b, d)`
   - Pair 4: `(c, e)`
   - Pair 5: `(c, e)`
5. **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 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`.
   - The fifth pair `(c, e)` is also matching because `d[c]` is indeed `e`.
6. **Successfully Mapped**: As we have gone through all the character pairs without conflict, and since `str2` does not contain all 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 approach.

Python Solution

```python
class Solution:
    def canConvert(self, string1: str, string2: str) -> bool:
        # If the input strings are equal, no conversion is needed.
        if string1 == string2:
            return True

        # If `string2` has all 26 letters, there's no way to convert `string1` to `string2`
        # as there would be no spare character to map a transition.
        if len(set(string2)) == 26:
            return False

        # Mapping dictionary to track corresponding characters from `string1` to `string2`.
        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.
            if char1 not in mapping_dict:
                mapping_dict[char1] = char2
            # If char1 is already mapped to a different character, conversion isn't possible.
            elif mapping_dict[char1] != char2:
                return False

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

```

Java Solution

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

        // Tracks the count of unique characters in 'str2'.
        int uniqueCharsCount = 0;
        // Array to store the frequency of characters in 'str2'.
        int[] charFrequency = new int[26];
        // Length of strings 'str1' and 'str2'.
        int length = str1.length();

        // Count the occurrences of each character in 'str2'.
        for (int i = 0; i < length; ++i) {
            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) {
            // 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++ Solution

```cpp
#include <string>
using namespace std;

class Solution {
public:
    // Function to determine if it's possible to convert 'str1' into 'str2'
    bool canConvert(string str1, string str2) {
        // If both strings are equal, no conversion is needed
        if (str1 == str2) {
            return true;
        }

        // Count the occurrences of each character in 'str2'
        int charCount[26] = {0};
        // Number of distinct characters in 'str2'
        int distinctChars = 0;

        // Populate the character count array for 'str2' and count distinct chars
        for (char c : str2) {
            if (++charCount[c - 'a'] == 1) {
                ++distinctChars;
            }
        }

        // If there are 26 distinct characters, no conversion is possible
        if (distinctChars == 26) {
            return false;
        }

        // Array to keep track of character mappings from 'str1' to 'str2'
        int charMapping[26] = {0};

        // Verify if the characters can be mapped from 'str1' to 'str2'
        for (int i = 0; i < str1.size(); ++i) {
            int indexStr1 = str1[i] - 'a';
            int indexStr2 = str2[i] - 'a';
            // 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 Solution

```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 and Space Complexity

Time Complexity

The function `canConvert` consists of several operations. Let's analyze them step by step:

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

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

Hence, the overall space complexity is `O(1)`, as both the set and the dictionary are bounded by a constant maximum size of 26.