2825. Make String a Subsequence Using Cyclic Increments

Medium Two Pointers String Leetcode Link

Problem Description

The problem presented requires determining whether `str2` can become a subsequence of `str1` by performing a specific operation on `str1`. The operation involves selecting a set of indices in `str1` and incrementing the character at each index to the next character in a cyclic manner. This means that the alphabet is considered to be circular and incrementing 'z' would result in 'a'.

A subsequence is defined as a sequence that can be derived from another sequence by deleting some or no elements without changing the order of the remaining elements. The goal is to verify if by applying the operation at most once, `str2` can be made a subsequence of `str1`.

Note that the problem specifies that the operation can be performed at most once, meaning you cannot perform this operation multiple times on `str1`. The string `str2` must either already be a subsequence of `str1` or be one operation away from being a subsequence.

Intuition

The solution approach is based on the idea that for `str2` to be a subsequence of `str1`, each character in `str2` must appear in `str1` in the same order, with the possibility of characters in `str2` being one cyclic increment away from the characters in `str1`.

To implement this, iterate through each character in `str1` and simulate the operation. For each character in `str1`, determine what the next cyclic character would be ('z' to 'a', and any other character to its successor). If the current character in `str1`, or its cyclic successor, matches the current character in `str2`, then that means this character is in the correct position, or one operation away from it, to form a subsequence.

Using a pointer `i`, keep track of the position in `str2` that you are trying to match with `str1`. Initialize this pointer to 0, and move it forward through `str2` each time you find a match or a potential match after a cyclic increment in `str1`.

If you reach the end of `str2` by advancing this pointer throughout the iteration, it means that all characters in `str2` are present in `str1` in order or one operation away, and `str2` is (or can become) a subsequence of `str1`. If the pointer `i` is equal to the length of `str2` by the end of the iteration through `str1`, return `true`; otherwise, `false`.

This approach allows checking whether `str2` can be obtained by performing at most one operation on `str1`, complying with the problem's constraints.

Solution Approach

The implementation of the solution in Python is straightforward and does not require the use of any complex data structures or patterns. It mainly utilizes basic control structures and string operations to achieve the goal.

Here is a step-by-step walk-through of the provided solution code:

1. A method `canMakeSubsequence` is defined in the `Solution` class. It takes two parameters: `str1` and `str2` (the input strings).
2. An index variable `i` is initialized to `0`. This variable is used to keep track of the current position in `str2` that we are trying to match against `str1`.
3. A `for` loop is used to iterate over each character `c` in `str1`.
4. Inside the loop, a new variable `d` is calculated. It contains the next character cyclically after `c`. If `c` is 'z', `d` would be 'a', otherwise `d` is the character that comes after `c` in the ASCII table, obtained by `chr(ord(c) + 1)`.
5. The loop checks if `i` is still within the bounds of `str2` (`i < len(str2)`), and if the current character of `str2` (`str2[i]`) matches either the current character `c` from `str1`, or the next cyclic character `d`. The `in` operator checks membership within a tuple made of `c` and `d`.
6. If a match or potential match after a cyclic increment is found, `i` is incremented by `1`, signifying that we have found the current character of `str2` in the `str1` or one operation away from it.
7. After the loop finishes, the algorithm checks whether `i` has advanced to the end of `str2` by comparing `i` with the length of `str2`. If they are equal, it means `str2` can be a subsequence of `str1` after performing the operation at most once. Thus, it returns `true`.
8. If the end of `str2` has not been reached, it indicates that `str2` cannot be made a subsequence of `str1` with at most one operation, so the method returns `false`.

This solution is efficient because it requires only a single pass through `str1` and stops as soon as all characters in `str2` are accounted for. Since there are no additional data structures used, the space complexity is O(1), and the time complexity is O(n), where n is the length of `str1`.

Example Walkthrough

Let's illustrate the solution approach with an example using `str1 = "abcde"` and `str2 = "axz"`. We want to determine whether we can transform `str2` into a subsequence of `str1` by performing the character increment operation at most once.

1. We begin with `i = 0`, which represents the position in `str2` that we're looking to match in `str1`.
2. As we iterate through `str1`, we compare the characters in `str2` with the characters in `str1` and their cyclic successor.
3. We start with the first character of `str1`, which is `a`. We see that `str2[i]` is also `a`. Since they match, we can increment `i` to `1`.
4. We proceed to the second character in `str1`, which is `b`. We check its cyclic successor, which is `c`. Neither match `str2[i]`, which is now `x`, so we don't increment `i`.
5. We move to the third character in `str1`, `c`. Its successor is `d`, and `str2[i]` is `x`. There's no match, so we leave `i` unchanged.
6. Next is `d` in `str1`, with its successor being `e`. `str2[i]` is still `x`. No match, so `i` remains at `1`.
7. The final character in `str1` is `e`, and its successor is `f` (cyclic incrementing from `z` to `a`, but regular increment otherwise). Yet again, there's no match with `str2[i]`, which remains `x`.
8. We exit the loop and compare `i` with the length of `str2`. We see that `i` is still `1`, but the length of `str2` is `3`. Since `i` does not equal the length of `str2`, we cannot make `str2` a subsequence of `str1` with at most one operation, and we return `false`.

In this example, the character `x` in `str2` cannot be matched in `str1` since there is no character that can be incremented cyclically to `x` in `str1`. Therefore, `str2` cannot become a subsequence of `str1` with just one operation according to our given rules.

Python Solution

```python
class Solution:
    def canMakeSubsequence(self, text: str, subsequence: str) -> bool:
        index = 0  # Initialize a pointer for the position in subsequence

        # Loop through each character in text
        for character in text:
            # Determine the next character after 'character' in alphabet, wrap around if it is 'z'
            next_char = 'a' if character == 'z' else chr(ord(character) + 1)

            # Check if the current pointer is within bounds of subsequence
            if index < len(subsequence):
                # If the current character in text is the same as the current character in subsequence
                # Or if it is the next character in the alphabet, move the pointer in subsequence
                if subsequence[index] in (character, next_char):
                    index += 1

        # After the loop, if the pointer has reached the length of subsequence
        # It means the subsequence can be formed, hence return True
        return index == len(subsequence)

```

Java Solution

```java
class Solution {

    /**
     * Checks if str2 is a subsequence of str1 with the character replacement rule.
     * Each character in str1 can remain the same or be replaced by the next
     * character in alphabetical order to match a character in str2.
     *
     * @param str1 The string to be transformed.
     * @param str2 The target subsequence.
     * @return true if str2 is a subsequence of str1 after allowed transformations.
     */
    public boolean canMakeSubsequence(String str1, String str2) {
        int currentIndex = 0; // Pointer into str2 to track our current progress.
        int lengthOfStr2 = str2.length(); // Total length of str2.

        // Iterate through each character of str1.
        for (char currentChar : str1.toCharArray()) {
            // Calculate the next character in the alphabetical order ('z' wraps to 'a').
            char nextChar = currentChar == 'z' ? 'a' : (char) (currentChar + 1);

            // Check if the current character in str1 matches the current or next valid character in str2.
            if (currentIndex < lengthOfStr2 &&
                (str2.charAt(currentIndex) == currentChar || str2.charAt(currentIndex) == nextChar)) {
                currentIndex++; // Move to the next character in str2.
            }
        }

        // str2 is a subsequence of str1 only if we have traversed its entire length.
        return currentIndex == lengthOfStr2;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    bool canMakeSubsequence(string s1, string s2) {
        // Initialize the index for traversing s2
        int index = 0;
        // Get the length of s2 for boundary checks
        int s2Length = s2.size();

        // Iterate over each character in s1
        for (char currentChar : s1) {
            // Determine the next character in the alphabet, wrapping around if 'z' is reached
            char nextChar = currentChar == 'z' ? 'a' : static_cast<char>(currentChar + 1);

            // Check if the current character of s2 matches currentChar or nextChar
            // and ensure we have not exceeded the bounds of s2
            if (index < s2Length && (s2[index] == currentChar || s2[index] == nextChar)) {
                // Move to next character in s2
                ++index;
            }
        }

        // Return true if we have traversed the whole s2, making it a subsequence
        return index == s2Length;
    }
};

```

Typescript Solution

```typescript
function canMakeSubsequence(sourceString: string, targetString: string): boolean {
    // Initialize a pointer to track the characters in targetString
    let pointer = 0;

    // Get the length of targetString for comparison
    const targetLength = targetString.length;

    // Loop through the characters of sourceString to check if a subsequence can be made
    for (const sourceChar of sourceString) {

        // Determine the character that follows sourceChar in the alphabet,
        // wrapping around from 'z' to 'a'
        const nextChar = sourceChar === 'z' ? 'a' : String.fromCharCode(sourceChar.charCodeAt(0) + 1);

        // If the current character in the targetString matches sourceChar
        // or the next character in the alphabetical sequence,
        // increment the pointer to continue with the next character
        if (pointer < targetLength && (targetString[pointer] === sourceChar || targetString[pointer] === nextChar)) {
            pointer++;
        }
    }

    // A subsequence can be made if the pointer has reached the end of targetString
    return pointer === targetLength;
}

```

Time and Space Complexity

The given Python code determines whether `str2` can be formed as a subsequence of `str1` by either taking a character as it is or replacing it with the next character in the alphabet (with 'z' converted to 'a').

Time Complexity

The time complexity of the code is determined by the single loop that iterates over the characters of `str1`. For each character `c` in `str1`:

- A constant-time operation is done to find the next character `d` (except for 'z', which turns to 'a').
- A constant-time operation `in` is used to check if `str2[i]` is one of the two characters `(c, d)`.

Since these operations are constant time, and the loop runs for each character in `str1`, the time complexity is `O(n)`, where `n` is the length of `str1`.

Space Complexity

The space complexity of the code:

- It uses a fixed number of simple variables (`i`, `c`, `d`), which require `O(1)` space.
- No additional data structures are allocated proportionally to the size of the input.

Thus, the overall space complexity of the code is `O(1)` – constant space complexity.