1003. Check If Word Is Valid After Substitutions

Medium Stack String

Problem Description

This problem presents a string manipulation task where you are given a string `s` and need to determine if it can be considered "valid" based on a specific operation. In this context, a valid string is one that can be formed by repeatedly inserting the substring `"abc"` at any position within an initially empty string `t`. Therefore, you can imagine starting with nothing and each time inserting "abc" somewhere into the string you're building until it matches the string `s` provided.

To put it simply, you are to verify if string `s` can be entirely composed of the substring `"abc"` repeated multiple times, possibly interspersed within each other but always in the correct order.

Intuition

The approach to solving this problem is nicely suited to a stack data structure because stacks are good at checking for patterns that should emerge in the reverse order in which they are read. Each time an `"abc"` pattern is detected within the string `s`, it is as if we can remove that pattern because it could represent one of the inserts that we previously made while constructing it.

Here's the intuition step-by-step:

1. We iterate through each character of the string `s`.
2. We add each character to a stack `t`. The use of a stack allows us to monitor the most recent characters that have been added and see if they form the sequence `"abc"`.
3. After adding each character, we check if the last three characters of the stack `t` are `"abc"`. If they are, it means we have found a valid sequence that could have been inserted into our initially empty string `t`, and we can remove this sequence from the stack.
4. We repeat the above steps until we have processed the entire string `s`.
5. At the end, if the string `s` is valid (meaning composed only of `"abc"` substrings), our stack `t` should be empty because every `"abc"` sequence would have been removed as it was detected.

The solution leverages the fact that if an `"abc"` pattern is found at any point, it can be eliminated as it represents a valid sequence that builds up the string `s`. If at the end there are characters left in the stack that cannot form the string `"abc"`, then the initial string `s` could not have been formed exclusively by inserting `"abc"`, and therefore it is not valid.

Solution Approach

The solution to this problem makes use of a simple but efficient data structure - the stack. A stack follows a Last In, First Out (LIFO) principle, which means that the last element added is the first one to be removed. This is perfect for the problem at hand, where we need to continuously check the last few elements for a specific pattern (`"abc"`).

Let's walk through the implementation as outlined in the reference solution:

1. **Length Check**: Right off the bat, the solution employs a quick check to see if the length of the input string `s` is divisible by 3. If it's not, the input can't possibly be a valid string since the patter `"abc"` is 3 characters long. This is executed with `if len(s) % 3:` which returns `False` if the condition is met.

2. **Initialization of the Stack**: A Python list named `t` is used here as the stack. This list will store characters from the input string.

3. **Iterating through String `s`**:

   - The algorithm iterates through every character `c` in the string `s`.
   - Each character is appended to the top of the stack `t`.

4. **Pattern Detection and Removal**:

   - After each character is added, the solution checks if the stack's size is at least 3 and if the top three elements form the string `"abc"`.
   - If they do, it means that a valid sequence has been found, and it removes the top three elements from the stack. The check and removal is succinctly performed by `if ''.join(t[-3:]) == 'abc': t[-3:] = []`.

5. **Final Check**:

   - After the iteration is complete, there's one final check to see whether the stack is empty.
   - If the stack is empty, this indicates that all elements of the string `s` formed valid sequences of `"abc"` and were removed during the iteration, which means the input string is valid. This results in a return value of `True`.
   - If there are any elements left on the stack, then there are characters which did not form the pattern `"abc"` correctly. As such, the input string `s` is not valid and `False` is returned.

In conclusion, this approach cleverly uses a stack to keep track of and identify valid patterns (`"abc"`) through iteration and pattern matching which when found, are removed, simulating the building process described in the problem statement.

Example Walkthrough

Let's consider an example where the string `s` is `"aabbcc"`. We need to determine if this string can be constructed by repeatedly inserting the substring `"abc"`.

1. **Length Check**: Our string `s` has a length of 6, which is divisible by 3. This passes our first check.

2. **Initialization of the Stack**: We initialize an empty stack `t`.

3. **Iterating through String `s`**: We process characters of `s` one by one:

   - We insert 'a' into stack `t`.
   - Now our stack `t` looks like `['a']`.
   - Next, we insert 'a' again.
   - Now our stack `t` looks like `['a', 'a']`.
   - We insert 'b'.
   - Now our stack `t` looks like `['a', 'a', 'b']`.
   - We insert another 'b'.
   - Now our stack `t` looks like `['a', 'a', 'b', 'b']`.
   - We insert 'c'.
   - Now our stack `t` looks like `['a', 'a', 'b', 'b', 'c']`.
   - Finally, we insert another 'c'.
   - Now our stack `t` looks like `['a', 'a', 'b', 'b', 'c', 'c']`.

4. **Pattern Detection and Removal**:

   - We observe that after adding each character, we don't find a sequence `"abc"` at the top of the stack at any point during this process. The stack never reaches a point where the top three elements are `"abc"`.

5. **Final Check**:

   - After processing all characters, we find that the stack `t` is full of characters and contains `['a', 'a', 'b', 'b', 'c', 'c']`.
   - Clearly, these cannot form the required `"abc"` pattern and hence cannot be removed.
   - Since our stack is not empty, the string `s` cannot be formed solely by inserting the substring `"abc"`.

In this example, the final stack not being empty indicates that the input string `"aabbcc"` is not valid based on our criteria. Therefore, the function would return `False`.

Solution Implementation

Python

```python
class Solution:
    def isValid(self, s: str) -> bool:
        # Early check to ensure the length of the string is a multiple of 3
        if len(s) % 3:
            return False

        # Initialize an empty list that will simulate a stack
        stack = []

        # Iterate over each character in the string
        for char in s:
            # Add the current character to the stack
            stack.append(char)

            # Check if the last 3 characters in the stack form the string 'abc'
            if ''.join(stack[-3:]) == 'abc':
                # If they do, pop the last 3 characters from the stack
                stack[-3:] = []

        # If the stack is empty after processing the entire string, return True
        # This indicates that the string was composed entirely of 'abc' sequences
        return not stack
```

Java

```java
class Solution {

    /**
     * Checks if the input string is valid by applying the following rule recursively:
     * if the string contains the substring "abc", it removes this substring and continues.
     * The string is valid if it can be reduced to an empty string using this rule.
     *
     * @param s The input string to be validated.
     * @return true if the string is valid, false otherwise.
     */
    public boolean isValid(String s) {
        // If the length of the string is not a multiple of 3, it cannot be valid
        if (s.length() % 3 != 0) {
            return false;
        }

        // Using StringBuilder for efficient modification of the string
        StringBuilder stringBuilder = new StringBuilder();

        // Iterate over the characters of the input string
        for (char character : s.toCharArray()) {
            // Append the current character to the stringBuilder
            stringBuilder.append(character);

            // Check if the last 3 characters form the substring "abc"
            if (stringBuilder.length() >= 3 && "abc".equals(stringBuilder.substring(stringBuilder.length() - 3))) {
                // If we found "abc", delete it from the stringBuilder
                stringBuilder.delete(stringBuilder.length() - 3, stringBuilder.length());
            }
        }

        // If stringBuilder is empty, all occurrences of "abc" have been removed and the string is valid
        return stringBuilder.length() == 0;
    }
}
```

C++

```cpp
class Solution {
public:
    // Function to check if a given string is valid, following specified constraints
    bool isValid(string s) {
        // Return false if the string length is not a multiple of 3
        if (s.size() % 3 != 0) {
            return false;
        }

        // Use a variable `temp` to store the intermediate string states
        string temp;

        // Iterate over each character in the input string
        for (char c : s) {
            // Append the current character to `temp`
            temp.push_back(c);

            // Check if the last three characters in `temp` form the string "abc"
            if (temp.size() >= 3 && temp.substr(temp.size() - 3, 3) == "abc") {
                // If "abc" is found, erase the last three characters from `temp`
                temp.erase(temp.end() - 3, temp.end());
            }
        }

        // If `temp` is empty, all occurrences of "abc" have been resolved; return true
        // If `temp` is not empty, the string is invalid; return false
        return temp.empty();
    }
};
```

TypeScript

```typescript
// Function to check if the given string can be fully reduced by successive removal of substring "abc"
function isValid(s: string): boolean {
    // If the string length is not a multiple of 3, it can't be fully reduced
    if (s.length % 3 !== 0) {
        return false;
    }

    // Temporary stack to hold characters for processing
    const tempStack: string[] = [];

    // Iterate over each character in the string
    for (const char of s) {
        // Push the current character onto the temp stack
        tempStack.push(char);

        // Check if the top 3 elements of the stack form the substring "abc"
        if (tempStack.slice(-3).join('') === 'abc') {
            // If they do, pop these 3 characters off the stack
            tempStack.splice(-3);
        }
    }

    // If the temp stack is empty, then the string is valid and can be fully reduced
    return tempStack.length === 0;
}
```

```python
class Solution:
    def isValid(self, s: str) -> bool:
        # Early check to ensure the length of the string is a multiple of 3
        if len(s) % 3:
            return False

        # Initialize an empty list that will simulate a stack
        stack = []

        # Iterate over each character in the string
        for char in s:
            # Add the current character to the stack
            stack.append(char)

            # Check if the last 3 characters in the stack form the string 'abc'
            if ''.join(stack[-3:]) == 'abc':
                # If they do, pop the last 3 characters from the stack
                stack[-3:] = []

        # If the stack is empty after processing the entire string, return True
        # This indicates that the string was composed entirely of 'abc' sequences
        return not stack
```

Time and Space Complexity

The provided Python code defines a method `isValid` which takes a string `s` as input and returns a boolean indicating whether the input string can be reduced to an empty string by repeatedly deleting the substring `"abc"`.

Time Complexity

The time complexity of the function is `O(n)` where `n` is the length of the input string `s`. This is because the function iterates through each character of the string exactly once, and the inner check—if the last three characters form the substring `"abc"`—is done in constant time, as it involves only a fixed-size (i.e., 3 characters) comparison and slice assignment. Therefore, the iteration dominates the runtime, resulting in a linear complexity relative to the length of the string.

Space Complexity

The space complexity of the function is also `O(n)`, as it potentially stores all characters of the string in the list `t` if the input does not contain any `"abc"` substrings to remove. In the worst-case scenario, where the string `s` is made up entirely of characters none of which combine to form `"abc"`, the list `t` will grow to the same size as the string `s`. Therefore, the space used by the list `t` scales linearly with the input size.