2645. Minimum Additions to Make Valid String

Medium Stack Greedy String Dynamic Programming

Problem Description

The given problem presents a scenario where you are working with strings and the goal is to make the given input string `word` valid by inserting additional letters. The valid form of the string is defined as one that can be generated by repeatedly concatenating the sub-string "abc". For example, "abcabc" and "abc" are valid strings, but "abbc" and "ac" are not.

To solve this problem, you have to identify the minimum number of letters that need to be inserted into the given string `word` to make it valid. You are permitted to insert the letters "a", "b", or "c".

To clarify, if the `word` is "aabcbc", you would need to insert "b" before the first "a" to make it "abcabc", which is a valid string. So, the output in this case would be 1, as only one insertion is necessary.

Intuition

The intuition behind the solution revolves around tracking the expected sequence of the characters "a", "b", and "c". As the valid string should have "abc" repeating, we can impose this sequence on the given `word` and count the number of deviations from it.

The core idea is to iterate through the given `word` and check if the current character matches the expected character in the sequence "abc". If it doesn't match, this means we would have to insert the expected character, and thus we increment our count of insertions. We continue to do this throughout the string, tracking whether we are expecting an "a", "b", or "c" at each position and incrementing the insertion count each time the actual character does not match the expected one. By the end of this process, we will have a count that represents the minimum insertions required to make the string valid.

In special cases, such as the end of the string, we may need to add extra characters depending on what the last character is. If it is not "c", we conclude the sequence with the necessary characters to complete the "abc" pattern.

For instance, if the last processed character of the word was "a", we need to insert "bc" to end the string with a complete "abc" pattern; thus we would add 2 to our count. Similarly, if it was "b", we only need to insert "c", adding 1 to our count. Hence, the insertion count after processing the full string gives us the answer.

Solution Approach

The solution approach leverages a straightforward iteration pattern along with a simple modulo operation to track the expected sequence of characters, "a", "b", and "c". Here, we detail the steps and logic used in the provided solution code. The solution does not depend on complicated data structures or algorithms; instead, it uses basic control structures.

1. Initialize two pointers `i` (to track the sequence "abc") and `j` (to iterate through the given `word`), as well as a variable `ans` to keep count of the minimum insertions required.
2. `i` is used to reference the index of the characters in the string `s`, which is "abc". It cycles through 0, 1, 2, and then back to 0, by using the expression `(i + 1) % 3`.
3. Start iterating through the given `word` using the `j` pointer, and compare each character in `word` at index `j` with the character in `s` at index `i`.
4. If the characters do not match (`word[j] != s[i]`), it means an insertion is needed to maintain the valid "abc" pattern, so increment `ans`.
5. When the characters do match (`word[j] == s[i]`), move to the next character in `word` by incrementing `j`. The `i` pointer gets updated to the expected next character in the sequence regardless of whether there was a match or not, to model the fact that an insertion was made or a correct character was observed.
6. After the main loop, if the last character of `word` is not 'c', it means that the current `abc` sequence hasn't been completed. Hence, we need to add characters to complete the sequence. If the last character is 'b', only one more character ('c') is needed, else if it's 'a', two characters ('bc') are required.
7. Finally, the function returns the value of `ans`, which after these steps, contains the minimum number of insertions required to make the `word` a valid string.

An important aspect to note is the simplicity of the solution. There is no use of complex data structures; the entire logic revolves around two pointers and comparisons, making the approach elegant and efficient with a time complexity of O(n), where n is the length of the `word`.

Example Walkthrough

Let's apply the solution approach to a small example, where the input `word` is "abab".

1. Initialize `i` to point to the start of the sequence, and `j` to point at the first character of `word`. `ans` is initialized to 0.
   - `i` points to 'a' in the sequence "abc".
   - `j` points to 'a' in `word`.
   - `ans` is 0.
2. Iterate through each character in `word`:
   - Compare `word[j]` ('a') with `s[i]` ('a'). They match, so no insertion is needed. Increment `i` using `(i + 1) % 3`, which sets `i` to point to 'b'.
   - Move to the next character in `word` ('b'), `j` becomes 1.
   - Compare `word[j]` ('b') with `s[i]` ('b'). They match again, so increment `i` to point to 'c'.
   - Move to next character in `word` ('a'), `j` becomes 2.
   - Compare `word[j]` ('a') with `s[i]` ('c'). They do not match, so an insertion is needed. Increment `ans` to 1, and update `i` to point to 'a'.
   - As we haven't advanced `j` due to the non-match, compare `word[j]` ('a') with `s[i]` ('a') again. They match, so now increment `i` (pointing to 'b'), and increment `j` to point to the last character in `word` ('b').
   - Compare `word[j]` ('b') with `s[i]` ('b'). They match, so increment `i` to point to 'c'.
   - Since we've reached the end of `word`, we look at the last character ('b'). It is not 'c', so we conclude the sequence needs a 'c'. Increment `ans` to 2.
3. By the end of the iteration, `ans` indicates that we need 2 insertions for `word` to become "abcabc", which is a valid string by the problem definition.

Solution Implementation

Python

```python
class Solution:
    def add_minimum(self, word: str) -> int:
        # Pattern to be matched
        pattern = 'abc'

        # Initialize counter for additional characters and the length of the word
        additional_chars = 0
        word_length = len(word)

        # Initialize pointers for word and pattern
        pattern_index = 0
        word_index = 0

        # Iterate through the word
        while word_index < word_length:
            # If the current character does not match the pattern character
            if word[word_index] != pattern[pattern_index]:
                # Increment the counter for additional characters
                additional_chars += 1
            else:
                # Move to the next character in the word if there is a match
                word_index += 1

            # Move to the next character in the pattern, wrapping around as needed
            pattern_index = (pattern_index + 1) % 3

        # After the last character, ensure the word ends with 'c'
        # If the word ends with 'b', only 1 additional character ('c') is needed
        # If the word ends with 'a', 2 additional characters ('bc') are needed
        if word[-1] != 'c':
            additional_chars += 1 if word[-1] == 'b' else 2

        # Return the total number of additional characters needed
        return additional_chars
```

Java

```java
class Solution {
    // Method to calculate the minimum number of characters to add
    public int addMinimum(String word) {
        // Reference string to compare with
        String reference = "abc";
        // Initialize the count of additional characters
        int count = 0;
        // Length of the input word
        int wordLength = word.length();

        // Loop through the word and reference in synchronization
        for (int refIndex = 0, wordIndex = 0; wordIndex < wordLength; refIndex = (refIndex + 1) % 3) {
            // If characters do not match, increment the count
            if (word.charAt(wordIndex) != reference.charAt(refIndex)) {
                count++;
            } else {
                // If characters match, move to the next character in word
                wordIndex++;
            }
        }

        // After processing the main loops, ensure the last character of 'word' is 'c'
        if (word.charAt(wordLength - 1) != 'c') {
            // If the last character is 'b', only one character ('c') needs to be added
            count += word.charAt(wordLength - 1) == 'b' ? 1 : 2;
            // If the last character is not 'b' (thus it must be 'a'), two characters ('b' and 'c') need to be added
        }

        // Return the total number of characters that need to be added
        return count;
    }
}
```

C++

```cpp
class Solution {
public:
    int addMinimum(string word) {
        // The pattern we want to follow is "abc". We'll iterate through the characters
        // of the input word and check their alignment with this pattern.
        string pattern = "abc";

        int modifications = 0; // Count of modifications required to align with the pattern.
        int wordLength = word.size(); // The length of the input word.

        // Iterate through the input word, using 'i' for the pattern index and 'j' for the word index.
        for (int i = 0, j = 0; j < wordLength; i = (i + 1) % 3) {
            // If the current character in the word does not match the current character in the pattern.
            if (word[j] != pattern[i]) {
                // We need to perform a modification (increment the counter)
                ++modifications;
            } else {
                // If it matches, we move to the next character in the word.
                ++j;
            }
        }

        // After iterating through the word, we need to ensure the last character matches the pattern.
        // If the last character is not 'c' we need additional modifications:
        // If it's 'b', we need to add 'c' so just 1 modification.
        // If it's 'a' or any other character, we need to add 'bc' so 2 modifications.
        if (word[wordLength - 1] != 'c') {
            modifications += (word[wordLength - 1] == 'b' ? 1 : 2);
        }

        // Return the total number of modifications required.
        return modifications;
    }
};
```

TypeScript

```typescript
// This function takes a string "word" and calculates the minimum number of
// characters that need to be added to make sure that no 'a', 'b', or 'c' is
// immediately followed by the identical character and the sequence 'abc' is not
// present.
function addMinimum(word: string): number {
    // Define a sequence 'abc' to be used for comparison
    const sequence: string = 'abc';
    let additionsNeeded: number = 0; // Counter for the number of additions needed
    const wordLength: number = word.length; // Length of the input word

    // Loop through the input word and the sequence in parallel until the
    // end of the word is reached.
    for (let seqIndex = 0, wordIndex = 0; wordIndex < wordLength; seqIndex = (seqIndex + 1) % 3) {
        if (word[wordIndex] !== sequence[seqIndex]) {
            // Increment additionsNeeded when the characters don't match
            additionsNeeded++;
        } else {
            // Move to the next character in the word if there is a match
            wordIndex++;
        }
    }

    // After processing the entire word, if the word ends with 'b', 'c'
    // needs to be added. If it ends with 'a', both 'b' and 'c' need to be
    // added to avoid creating the sequence 'abc' or having identical
    // characters next to each other.
    if (word[wordLength - 1] === 'b') {
        additionsNeeded++;
    } else if (word[wordLength - 1] === 'a') {
        additionsNeeded += 2;
    }

    // Return the total number of additions needed
    return additionsNeeded;
}
```

```python
class Solution:
    def add_minimum(self, word: str) -> int:
        # Pattern to be matched
        pattern = 'abc'

        # Initialize counter for additional characters and the length of the word
        additional_chars = 0
        word_length = len(word)

        # Initialize pointers for word and pattern
        pattern_index = 0
        word_index = 0

        # Iterate through the word
        while word_index < word_length:
            # If the current character does not match the pattern character
            if word[word_index] != pattern[pattern_index]:
                # Increment the counter for additional characters
                additional_chars += 1
            else:
                # Move to the next character in the word if there is a match
                word_index += 1

            # Move to the next character in the pattern, wrapping around as needed
            pattern_index = (pattern_index + 1) % 3

        # After the last character, ensure the word ends with 'c'
        # If the word ends with 'b', only 1 additional character ('c') is needed
        # If the word ends with 'a', 2 additional characters ('bc') are needed
        if word[-1] != 'c':
            additional_chars += 1 if word[-1] == 'b' else 2

        # Return the total number of additional characters needed
        return additional_chars
```

Time and Space Complexity

The time complexity of the `addMinimum` function is `O(n)`, where `n` is the length of the input string `word`. The function contains a single while loop that iterates over each character of the word exactly once. The operations inside the loop have a constant cost, as they involve only basic arithmetic and comparisons. The final check after the loop is also a constant time operation.

The space complexity of the function is `O(1)`. The function uses a finite number of integer variables (`ans`, `n`, `i`, `j`) and a constant string `s`, with no dependence on the size of the input. Therefore, the amount of space used does not scale with `n`, and it stays constant no matter how large the input string is.