Compress String LeetCode

Given a vector of characters **chars**, your task is to compress it in-place by replacing consecutive occurrences of each character with that character followed by the count of consecutive occurrences. If the compressed string is not smaller than the original string, then the function should return the original length of the string.

Example:

Input: chars = ['a', 'a', 'b', 'b', 'c', 'c', 'c']

Output: The new length of the compressed array: 6

Explanation: The original array is compressed as follows: ['a', '2', 'b', '2', 'c', '3']. The new length of the compressed array is 6.

Approach 1: Function to compress the characters in the given vector

1. The **compress** function takes a reference to a vector of characters **chars** as input and returns the length of the compressed vector.
2. It uses two pointers, **slow** and **fast**, to traverse the characters in the vector.
3. The **slow** pointer points to the beginning of a group of consecutive characters, and the **fast** pointer moves forward until it reaches a different character or the end of the vector.
4. Inside the while loop, the function writes the character at **slow** index to the compressed vector at **ansIndex**.
5. It calculates the count of consecutive occurrences of the character by subtracting **slow** from **fast**.
6. If the count is greater than 1, it converts the count to a string and writes the count digits as characters to the compressed vector.
7. After processing a group of characters, it moves **slow** to the next group and repeats the process until the end of the vector is reached.
8. Finally, it returns **ansIndex**, which represents the length of the compressed vector.
9. **Time Complexity: O(n)**
   - **The outer while loop iterates through the chars vector once, so it has a time complexity of O(n).**
   - The inner while loop also iterates through a group of consecutive characters, but it does not cover the entire length of the vector. Hence, its time complexity is considered constant.
   - Overall, the time complexity of the compress function is O(n).

10. **Space Complexity: O(1)**
    - **The compress function performs the compression in-place without using any extra space**. It modifies the input vector directly, so the space usage remains constant regardless of the size of the input vector. Hence, the space complexity is O(1).

Approach 2: Modified version of the compress function with optimized loop condition

1. The **compressModified** function has the same logic as the **compress** function, but with a slightly modified loop condition.
2. It uses a **for** loop with **fast** starting from **slow + 1** and the condition **fast <= chars.size()**.
3. By using **fast <= chars.size()**, it ensures that the final group of characters is processed.
4. Rest of the code in the function is similar to the **compress** function.
5. **Time Complexity: O(n)**
    - **The for loop iterates through the chars vector once, so it has a time complexity of O(n).**
    - Similar to the compress function, the inner if statement has a constant time complexity since it covers a group of consecutive characters, not the entire length of the vector.
    - Overall, the time complexity of the compressModified function is O(n).

1. **Space Complexity: O(1)**
    - **The compressModified function also performs the compression in-place without using any extra space.** It modifies the input vector directly, so the space usage remains constant regardless of the size of the input vector. Thus, the space complexity is O(1).

Which approach to use:

Both approaches (**compress** and **compressModified**) achieve the same result and have the same time and space complexity. You can choose either of them based on your preference or coding style. The **compressModified** approach has a slightly modified loop condition, which includes the final group of characters.