443. String Compression

Medium Two Pointers String

Problem Description

The problem involves compressing a sequence of characters into a string where each group of consecutive, identical characters is represented by the character, followed by the number of occurrences if the occurrences are more than 1. For example, "aabccc" would compress to "a2bc3".

Here's how the algorithm works:

- We iterate over the provided array, `chars`.
- For each unique character, we find the number of times it repeats consecutively.
- We then overwrite the original `chars` array with the character followed by the number of occurrences if more than one.
- We have to return the length of the array after the modifications.
- The space complexity needs to be constant; therefore, we cannot use any additional arrays or strings for our computations.

The challenge is to edit the array in place since we aren't allowed to return a separate compressed string but instead must modify the `chars` array directly.

Intuition

To solve this problem, we maintain two pointers and a counter:

- The first pointer (`i`) marks the start of a sequence of identical characters.
- The second pointer (`j`) is used to find the end of this sequence by iterating through the array until a different character is found.
- As we identify the consecutive characters, we store the current character and its count (if necessary) into the array, keeping track of where we're writing with a `k` index.

The process operates as follows:

- Iterate over each character with the `i` pointer.
- Use the `j` pointer to count how many times the character at position `i` repeats consecutively.
- Write the character at the `i` index position into the `k` index of the `chars` list.
- Increase `k` by 1 to move to the next position in the list.
- If the number of repetitions is greater than 1, convert it into a string and iterate over each character of this string count.
- For each digit of the count, write it at the `k` index of the `chars` list and increment `k`.
- Move `i` to the position where `j` has ended up to process the next set of unique characters.
- Continue this process until you have finished iterating over the entire array.
- Lastly, return `k` as the new length of the compressed `chars` array.

The solution ensures the use of constant space by overwriting the original list with no additional list or string allocation.

Solution Approach

The solution to the problem of compressing consecutive characters into a shorter representation involves a straightforward approach with a focus on in-place array manipulation to keep the space complexity constant. The key idea is to use a "read-write" mechanism facilitated by pointer manipulation in the array.

Here is a step-by-step breakdown of the implementation details:

- **Two Pointers Technique**: We initialize two pointers: `i` and `k`. Pointer `i` is utilized to identify contiguous blocks of the same character, and `k` is used to write the compressed form of these blocks back into the `chars` list.
- **Iteration Over the Characters**: As we iterate over the `chars` with pointer `i`, we aim to identify groups of the same character and count their occurrences.
- **Counting Group Occurrences**: We initialize another pointer `j` that starts from `i + 1` and increments until it finds a character different from `chars[i]`. The difference `j - i` then gives us the count of consecutive characters.
- **Writing the Character**: Regardless of the count, we always write the character in consideration to the `chars` list at `chars[k]` and increment `k`.
- **Writing the Count**: If the count is greater than one (meaning there are repetitions), we convert the count to a string (`str(j - i)`) and iterate over each character of this count string, writing the digits sequentially into the `chars` list at subsequent `k` positions, incrementing `k` after each write.
- **Updating Pointers**: After handling one set of consecutive characters, we set `i` to the current position of `j`, effectively jumping to the next new character or the end of the list.
- **Returning the New Length**: Once we reach the end of the list with pointer `i`, `k` points to the next unwritten position in `chars` and thus represents the length of the compressed character list. We then return `k`.

This approach does not require sorting or any additional data structures, but it efficiently leverages the given array itself to store the result. By overwriting the original array, we maintain a space complexity of O(1), aside from the space needed to hold the input itself. The complexity is achieved by utilizing the in-place rewriting of the array, a key requirement specified in the problem description.

In terms of time complexity, each character in `chars` is read and written at least once. In the worst case, each write operation for counts might take O(log n) time (if we represent the count in decimal form and n is the number of occurrences), but since count increments for each group of identical characters need to be written at most once, it doesn't affect the overall linear time complexity of O(n), where n is the total number of characters in `chars`.

The main takeaway from the solution is the effective use of pointers and the in-place writing technique to transform the array without the use of additional memory, which is a common constraint in more complex problems involving arrays.

Example Walkthrough

Let's use the string `chars = ['a', 'a', 'b', 'b', 'c', 'c', 'c']` to walk through the solution approach:

1. **Initialization**: We start with pointers `i`, `j`, and `k` all set to 0. The `chars` array looks like this: `['a', 'a', 'b', 'b', 'c', 'c', 'c']`.
2. **Iteration and Counting**: The `i` pointer is at the first `a`. We advance `j` to find the end of the `a` group, which is at index 2. Therefore, we have 2 `a`'s in a row.
3. **Compression Writing**: We write `'a'` into `chars[k]` and increment `k`; `chars` now looks like this: `['a', 'a', 'b', 'b', 'c', 'c', 'c']` with `k` pointing at index 1.
4. **Writing the Count**: The count is 2, so we write `'2'` into `chars[k]` and increment `k` again; `chars` looks like: `['a', '2', 'b', 'b', 'c', 'c', 'c']`.
5. **Updating Pointers**: We move the `i` pointer where `j` stopped, which is index 2, the first `b`.
6. Repeat Steps 2-5 for character `b`: `j` will stop at index 4 and we write `'b'` and then `'2'` into the array. The array now looks like: `['a', '2', 'b', '2', 'c', 'c', 'c']`, and `k` is at index 4.
7. Repeat Steps 2-5 for character `c`: `j` moves to the end of the array since there are no more characters after `c`. We write `'c'` and then `'3'` into the array. The final array is `['a', '2', 'b', '2', 'c', '3', 'c']`, and `k` would be at index 6.
8. **Final Step**: Since we've reached the end of the `chars` array with pointer `i`, we can now return `k` as the length of the compressed list. However, due to the previous compressions, there is a leftover `'c'`, which we do not count. The final compressed length is 6, and the compressed array is `['a', '2', 'b', '2', 'c', '3']`.

In summary, the `chars` array, which initially included 7 characters, has been compressed in place to a length of 6, representing the sequence "a2b2c3". The algorithm executed in constant space, as required by the problem definition.

Solution Implementation

Python

```python
class Solution:
    def compress(self, chars: List[str]) -> int:
        # Start pointers at the beginning of the list
        read, write, length = 0, 0, len(chars)

        # Process the entire character list
        while read < length:
            # Move read pointer to end of current character sequence
            read_next = read + 1
            while read_next < length and chars[read_next] == chars[read]:
                read_next += 1

            # Write the current character to the write pointer
            chars[write] = chars[read]
            write += 1

            # If we have a sequence longer than 1
            if read_next - read > 1:
                # Convert count to string and write each digit
                count = str(read_next - read)
                for char in count:
                    chars[write] = char
                    write += 1

            # Move read pointer to next new character
            read = read_next

        # Return the length of the compressed list
        return write
```

Java

```java
class Solution {
    public int compress(char[] chars) {
        int writeIndex = 0; // Tracks where to write in the array
        int length = chars.length; // Total length of the input array

        // Start processing each sequence of characters
        for (int start = 0; start < length; ) {
            // 'start' is the beginning of a sequence; 'end' will be one past the last char
            int end = start + 1;

            // Expand the sequence to include all identical contiguous characters
            while (end < length && chars[end] == chars[start]) {
                end++;
            }

            // Write the character that the sequence consists of
            chars[writeIndex++] = chars[start];

            // If the sequence is longer than 1, write the count of characters
            if (end - start > 1) {
                String count = String.valueOf(end - start); // convert count to string
                for (char c : count.toCharArray()) { // iterate over each character in the count
                    chars[writeIndex++] = c; // write count characters to the result array
                }
            }

            // Move to the next sequence
            start = end;
        }

        // writeIndex represents the length of the the compressed string within the array
        return writeIndex;
    }
}
```

C++

```cpp
class Solution {
public:
    // The compress function takes a vector of characters and compresses it by replacing
    // sequences of repeating characters with the character followed by the count of repeats.
    // It modifies the vector in-place and returns the new length of the vector after compression.
    int compress(vector<char>& chars) {
        int writeIndex = 0; // Initializing write index to track the position to write the next character.
        int size = chars.size(); // Store the size of the input vector.

        // Loop over characters in the vector starting from the first character.
        for (int readStart = 0, readEnd = readStart + 1; readStart < size; ) {
            // Expand the readEnd pointer to include all consecutive identical characters.
            while (readEnd < size && chars[readEnd] == chars[readStart]) {
                ++readEnd;
            }

            // Write the character to the vector.
            chars[writeIndex++] = chars[readStart];

            // If the run of characters is more than one, write the count after the character.
            int runLength = readEnd - readStart;
            if (runLength > 1) {
                // Convert runLength to string and write each digit individually.
                for (char countChar : to_string(runLength)) {
                    chars[writeIndex++] = countChar;
                }
            }

            // Move the readStart to the next character group.
            readStart = readEnd;
        }

        // Return the new length of the vector after compression.
        return writeIndex;
    }
};
```

TypeScript

```typescript
// The compress function takes an array of characters and compresses it by replacing
// sequences of repeating characters with the character followed by the count of repeats.
// It modifies the array in-place and returns the new length of the array after compression.

function compress(chars: string[]): number {
    let writeIndex = 0; // Initializing write index to track the position to write the next character.
    let size = chars.length; // Store the size of the input array.

    // Loop over characters in the array starting from the first character.
    for (let readStart = 0; readStart < size; ) {
        let readEnd = readStart + 1; // Initialize readEnd as the character following readStart.

        // Expand the readEnd pointer to include all consecutive identical characters.
        while (readEnd < size && chars[readEnd] === chars[readStart]) {
            readEnd++;
        }

        // Write the character to the array.
        chars[writeIndex++] = chars[readStart];

        // If the run of characters is more than one, write the count after the character.
        let runLength = readEnd - readStart;
        if (runLength > 1) {
            // Convert runLength to string to iterate its characters and write each digit individually.
            let runLengthStr = runLength.toString();
            for (let i = 0; i < runLengthStr.length; i++) {
                chars[writeIndex++] = runLengthStr[i];
            }
        }

        // Move the readStart to the next unique character.
        readStart = readEnd;
    }

    // Return the new length of the array after compression.
    return writeIndex;
}
```

```python
class Solution:
    def compress(self, chars: List[str]) -> int:
        # Start pointers at the beginning of the list
        read, write, length = 0, 0, len(chars)

        # Process the entire character list
        while read < length:
            # Move read pointer to end of current character sequence
            read_next = read + 1
            while read_next < length and chars[read_next] == chars[read]:
                read_next += 1

            # Write the current character to the write pointer
            chars[write] = chars[read]
            write += 1

            # If we have a sequence longer than 1
            if read_next - read > 1:
                # Convert count to string and write each digit
                count = str(read_next - read)
                for char in count:
                    chars[write] = char
                    write += 1

            # Move read pointer to next new character
            read = read_next

        # Return the length of the compressed list
        return write
```

Time and Space Complexity

Time Complexity

The given Python code follows a two-pointer approach to compress a list of characters in-place.

1. We have two pointers `i` and `j`; `i` is used to traverse the character list, while `j` is used to find the next character different from `chars[i]`.
2. The while loop with `i` as its iterator runs for each unique character in the list, so it runs 'n' times in the worst-case scenario (`n` being the length of the list).
3. The inner while loop increments `j` until a different character is found. In the worst case, where all characters are the same, the loop runs 'n' times.

Hence, in the worst case, where the inner loop is considered for every character, the time complexity would be considered linear with respect to the length of the list, i.e., $O(n)$.

Space Complexity

The algorithm adjusts the characters in the list in-place and uses a fixed number of variables (`i`, `j`, `k`, `n`, `cnt`). It does not utilize any additional data structure that grows with the input size. Therefore, the space complexity of the algorithm is $O(1)$ regardless of the input size.