Documentation on String Compression

Table of Contents

1. Problem Statement

Given an array of characters chars, the task is to compress it using the following rules:

- Replace each consecutive group of repeating characters with the character followed by the group's length.
- If a character appears only once, do not append a number.
- The modified array should be stored in-place in chars, and the function should return the new length of the modified array.
- The algorithm should use only constant extra space.

Constraints:

- $1 \leq chars.length \leq 2000$
- chars[i] is an English letter (lowercase/uppercase), a digit, or a symbol.

Example 1:

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

Output: Return 6, with chars modified to ["a","2","b","2","c","3"]

Example 2:

Input: chars = ["a"]

Output: Return 1, with chars modified to ["a"]

Example 3:

Input:chars = ["a","b","b","b","b","b","b","b","b","b","b","b","b"]

Output: Return 4, with chars modified to ["a","b","1","2"]

2. Intuition

The goal is to traverse the array and count consecutive occurrences of each character. Instead of storing the result in a separate string, we directly modify the chars array. The key observations are:

- If a character appears once, we keep it unchanged.
- If a character appears more than once, we append the frequency as separate digits.
- The challenge is ensuring in-place modifications with minimal extra space.

3. Key Observations

- We can use two pointers:
    - read pointer to traverse chars and count occurrences.
    - write pointer to update chars with compressed values.
- Digit conversion: Since the count can be more than 9 (e.g., "bbbbbbbbbbbb" → "b12"), we need to store numbers digit by digit.

4. Approach

i. Initialize Pointers
    a. write = 0: Keeps track of where to write in chars.
    b. read = 0: Traverses through chars to count occurrences.

ii. Traverse the Array
    a. Identify a group of consecutive repeating characters.
    b. Store the character at write and move write forward.
    c. If the group has more than 1 character, convert the count to a string and store each digit separately.

iii. Return the Modified Length
    a. The function returns write, representing the length of the modified chars array.

5. Edge Cases

- Single character array:
    - Example: ["a"] → Output: ["a"]
- All unique characters:
    - Example: ["a","b","c","d"] → Output: ["a","b","c","d"]
- Large groups:
    - Example: ["a","a","a","a","a","a","a","a","a","a","a"] → Output: ["a","1","1"]
- Mixed cases & symbols:
    - Example: ["$","$","$","A","A","B","B","B","B"] → Output: ["$","3","A","2","B","4"]

6. **Complexity Analysis**

Time Complexity:

- O(N), where N is the length of chars, as each character is processed once for reading and once for writing.

Space Complexity:

- O(1), as we modify chars in-place without using additional space.

7. **Alternative Approaches**

Using a New List (Not In-Place)

Instead of modifying chars, store the result in a new list and return its length.
However, this approach violates the problem constraints (constant space).

Using String Concatenation (Inefficient)

Concatenating strings during compression (s += char + str(count)) is inefficient due to immutable string operations.

8. **Test Cases**

```
# Test Case 1
chars = ["a","a","b","b","c","c","c"]
print(Solution().compress(chars))  # Output: 6, chars = ["a","2","b","2","c","3"]

# Test Case 2
chars = ["a"]
print(Solution().compress(chars))  # Output: 1, chars = ["a"]

# Test Case 3
chars = ["a","b","b","b","b","b","b","b","b","b","b","b","b"]
print(Solution().compress(chars))  # Output: 4, chars = ["a","b","1","2"]
```

```
# Test Case 4 (Edge Case - Large Group)
chars = ["a"] * 12  # ["a","a",...,"a"] (12 times)
print(Solution().compress(chars))  # Output: 3, chars = ["a","1","2"]

# Test Case 5 (Mixed characters)
chars = ["$","$","$","A","A","B","B","B","B"]
print(Solution().compress(chars))  # Output: 6, chars = ["$","3","A","2","B","4"]
```

9. Final Thoughts

- The two-pointer approach efficiently compresses the string in-place.
- The algorithm ensures O(N) time complexity and O(1) space complexity.
- It correctly handles single characters, large groups, and mixed symbols.
- This solution meets the constant space requirement, making it optimal.