38. Count and Say

Medium String

Problem Description

The "count-and-say" problem requires us to generate a sequence of digit strings where each term is derived from its predecessor by a process that mimics "counting and saying" the numbers of consecutive digits. Specifically, it follows these rules:

- `countAndSay(1)` is defined to be the string `"1"`.
- For `countAndSay(n)` where `n > 1`, we consider the digit string of `countAndSay(n-1)`, then "count and say" the digits to form a new string.

To "count and say" the string, we go through the string from left to right, identify consecutive runs of the same digit, and for each run, we first say the quantity of digits, followed by the digit itself. This is then concatenated to give us the next number in the sequence.

For example, if we have a digital string "3322251":

1. There are two '3's, so we say "23".
2. Then we have three '2's, so we say "32".
3. Next is one '5', so we say "15".
4. Lastly, there's one '1', so we say "11".

Putting it all together, we "say" "23321511", which becomes our "count and say" string for the next number in the sequence.

The challenge asks us to find the nth term of this sequence given a positive integer `n`.

Intuition

The intuition for solving this problem lies in understanding the "count and say" mechanism—specifically, how each term in the sequence is derived from the previous one. Here's how we can approach the solution:

1. Initialize the sequence with the base case—"1" for `countAndSay(1)`.
2. To generate each subsequent term, iterate over the current string while keeping track of the count of identical consecutive digits encountered.
3. We move along the string and, for each unique digit (or when the next digit is different from the current), we append the count followed by the digit to the result for this iteration.
4. The result of this iteration becomes the input for the next, allowing us to build up the sequence term by term.

Solution Approach

The implementation of the solution utilizes a straightforward iterative approach, which transitions from one term to the next until the nth term is reached. Here's a step-by-step breakdown of the algorithm using the solution code provided:

1. Initialize a string `s` with the initial value `"1"`, representing the first term in the sequence.
2. Use a loop to iterate from the second term up to the nth term (thus, the loop runs `n-1` times). In each iteration, a new term is built from the previous one.
3. Within the loop, we introduce two pointers, `i` and `j`. The pointer `i` marks the start of a sequence of identical digits, and `j` is used to find the end of this sequence.
4. Start another loop to traverse the current string `s`. For each unique sequence of repeated characters:
   - Use the inner loop to increment `j` until the characters at indices `i` and `j` are different or `j` reaches the end of `s`.
   - The difference `j - i` gives the count of consecutive identical digits. This count, converted to a string, is appended to the temporary list `t`.
   - Append the actual digit (converted to a string) to the list `t`.
   - Update `i` to `j` to start counting the next sequence.
5. After we have iterated over the entire string `s` and recorded the counts and digits in the list `t`, we use `''.join(t)` to concatenate the elements of `t` into a new string, which represents the next term.
6. Assign this new string back to `s`, which will now be used as the base for the next iteration if any.
7. Continue this process until all `n-1` iterations are completed.
8. The final string `s` after the last iteration is the nth term of the sequence, which is returned by the function.

No additional data structures are used in this solution other than a list to build the strings for each iteration, which is a space-efficient way to construct strings in Python. This approach ensures that the solution runs in a scalable manner—with each iteration only depending on the previous term.

Additionally, the efficiency of this approach comes from the use of the two pointer technique, which reduces unnecessary re-scanning of the digits and allows the algorithm to process the string in a single pass during each iteration.

Example Walkthrough

Let's clarify the solution approach with a small example by finding `countAndSay(4)`.

1. According to the base case, `countAndSay(1)` is `"1"`.
2. To find `countAndSay(2)`, we look at the previous term which is `"1"`. We have one '1', so we say "11". Hence, `countAndSay(2)` is `"11"`.
3. Next, to compute `countAndSay(3)`, we consider the previous term `"11"`. We have two '1's, thus we say "21". So, `countAndSay(3)` is `"21"`.
4. To determine `countAndSay(4)`, we use the term from the previous step, `"21"`. We have one '2', and one '1', so we say "12" followed by "11". This gives us `countAndSay(4)` as `"1211"`.

Breaking down the "count and say" for `"21"` to get `"1211"`:

- We start with the first digit '2', which only appears once, so we say "12".
- Next, we go to the digit '1', which also appears once, so we say "11".

We concatenate "12" and "11" to form "1211", yielding the term `countAndSay(4)`. If we were to continue this process for `countAndSay(5)`, we would apply the same counting and saying method to the string "1211".

Python Solution

```python
class Solution:
    def countAndSay(self, n: int) -> str:
        # Initialize the sequence with the first term.
        sequence = '1'

        # Build the sequence up to the n-th term.
        for _ in range(n - 1):
            # Initialize index for the current sequence
            index = 0
            # Create a temporary list to hold new term
            temp_sequence = []

            # Iterate through the current sequence to build the next sequence
            while index < len(sequence):
                # Initialize a count index
                count_index = index
                # Count the number of same digits
                while count_index < len(sequence) and sequence[count_index] == sequence[index]:
                    count_index += 1

                # Append the count and the digit itself to the temp_sequence list
                temp_sequence.append(str(count_index - index))
                temp_sequence.append(sequence[index])

                # Move to the next different digit
                index = count_index

            # Join the temp_sequence list to make a string and assign it as the new sequence
            sequence = ''.join(temp_sequence)

        # After the loop, sequence variable holds the n-th term of the sequence
        return sequence
```

Java Solution

```java
class Solution {
    public String countAndSay(int n) {
        // The initial string is "1".
        String current = "1";

        // Iterate until we reach the requested sequence iteration (n).
        while (--n > 0) {
            // StringBuilder to build the next sequence.
            StringBuilder nextSequence = new StringBuilder();

            // Loop through the characters of the current sequence.
            for (int i = 0; i < current.length();) {
                int count = 0; // Initialize a counter for the character grouping.
                char ch = current.charAt(i); // Current character to be counted.

                // Count consecutive similar characters.
                while (i < current.length() && current.charAt(i) == ch) {
                    ++i;
                    count++;
                }

                // Append the count and the character to the next sequence.
                nextSequence.append(count).append(ch);
            }

            // Prepare for the next iteration by updating the current sequence.
            current = nextSequence.toString();
        }

        // Return the final sequence after 'n' iterations.
        return current;
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    // This function generates the n-th term in the "count and say" sequence.
    string countAndSay(int n) {
        // The first term in the sequence is "1"
        string currentTerm = "1";

        // Build the sequence up to the n-th term
        while (--n > 0) {
            string nextTerm = ""; // This will be the next term in the sequence

            // Process the current term character by character
            for (int i = 0; i < currentTerm.size();) {
                int countIndex = i;
                // Count how many times the same digit appears consecutively
                while (countIndex < currentTerm.size() && currentTerm[countIndex] == currentTerm[i]) {
                    ++countIndex;
                }

                // Append the count (number of times the digit appears) and the digit itself
                nextTerm += to_string(countIndex - i);
                nextTerm += currentTerm[i];

                // Move to the next different digit
                i = countIndex;
            }

            // The next term becomes the current term for the next iteration
            currentTerm = nextTerm;
        }

        // Return the n-th term in the sequence
        return currentTerm;
    }
};
```

Typescript Solution

```typescript
// A function to implement the 'count-and-say' sequence.
// @param {number} n - The position in the count-and-say sequence to generate the string for.
// @returns {string} - The n-th term in the count-and-say sequence.
function countAndSay(n: number): string {
    // Initialize the sequence with the first term.
    let sequence = '1';

    // Generate the sequence from the second term to the nth term.
    for (let i = 1; i < n; i++) {
        // Temporary variable to build the next term in the sequence.
        let nextTerm = '';

        // Initialize the current character to the first character of the sequence.
        let currentCharacter = sequence[0];

        // Initialize a counter for occurrences of the current character.
        let count = 1;

        // Loop through the sequence starting from the second character.
        for (let j = 1; j < sequence.length; j++) {
            // If the current character is different from the next character in the sequence, append the count and character to the r
            if (sequence[j] !== currentCharacter) {
                nextTerm += `${count}${currentCharacter}`;
                currentCharacter = sequence[j];
                count = 0; // Reset count for the new character.
            }

            // Increment count for every occurrence of the current character.
            count++;
        }

        // Append the count and character for the last segment of the sequence.
        nextTerm += `${count}${currentCharacter}`;

        // Update the sequence to the newly generated term.
        sequence = nextTerm;
    }

    // Return the nth term in the count-and-say sequence.
    return sequence;
}
```

Time and Space Complexity

The provided Python code implements the "Count and Say" problem. It constructs a sequence by reading the previous sequence and counting the number of digits in groups.

To analyze the time complexity, let's consider the worst-case scenario:

- The `for` loop runs `n - 1` times.
- Inside the loop, there is a `while` loop which could, in the worst case, run for the length of the string `s`. The length of `s` can grow significantly with each iteration.
- Assume the length of the string `s` at iteration `i` is `len(s_i)`, then the time complexity would roughly be `O(len(s_1) + len(s_2) + ... + len(s_{n-1}))`.
- It is hard to express this in terms of `n` only because the series growth rate is not linear. However, the rate at which `s` grows can be approximated to be exponential in the worst case. Therefore, we can consider the worst-case time complexity to be an exponential function of `n`.

For space complexity:

- The space complexity is driven by the storage of the string `s`.
- In each iteration, a new string `t` is created, which can be twice as long as the current string `s`, since it records both the count and the value for each group of characters.
- At the end of each iteration, `s` takes the value of `t`, which means that the space complexity will be proportional to the maximum length of `s`.
- Given the potential exponential growth of `s`, the space complexity can also be considered exponential in terms of `n`, in the worst-case scenario.

In summary:

- Time complexity: `O(2^(n-1))` in the worst case (where `2^(n-1)` represents the exponential growth of the string length).
- Space complexity: `O(2^(n-1))` in the worst case, which is driven by the length of the string `s`.