1446. Consecutive Characters

Easy String

Problem Description

The problem is about finding the maximum length of a substring in a given string `s` where all characters in the substring are the same. This length is referred to as the "power" of the string. A substring is a contiguous sequence of characters within a string. The uniqueness here means that within this substring, there should be no varying characters. It is a sequence of the same character repeated.

For example, in the string `"aaabccc"`, the power would be 3, since the longest substring where the same character is repeated is `"ccc"`.

The problem asks for a function that processes the input string `s` and outputs the integer power of that string.

Intuition

The intuition behind the solution relies on iterating through the string and keeping track of the current sequence of identical characters. To achieve that, we analyze each pair of adjacent characters in the string.

We need two variables: one to keep track of the current substring's length of consecutive identical characters (`t`) and another to keep a record of the maximum length found so far (`ans`).

1. Initialize `ans` and `t` to 1, because the minimum power for any non-empty string is 1 (any individual character counts as a substring of power 1).
2. Iterate through adjacent pairs of characters in the string `s`. In Python, this can be conveniently done by using the `pairwise` utility from the `itertools` module. However, since this utility is not mentioned in the problem statement and it is not available before Python 3.10, we can manually compare elements at indices `i` and `i+1` while iterating with a normal loop from `0` to `len(s) - 1`.
3. For each pair `(a, b)` of adjacent characters, check if they are the same:
   - If they are the same, increment the temporary substring length `t` by 1.
   - Update the `ans` with the maximum of the current `ans` and the new `t`.
4. If the characters `a` and `b` are different, reset `t` to 1 because we have encountered a different character and thus need to start a new substring count.
5. Continue this process until the end of the string is reached.
6. Return the recorded maximum length `ans` as the power of the string.

Solution Approach

The solution provided is straightforward and relies on a simple iteration. It does not require any complex data structures or algorithms. The core pattern used here is a linear scan across the input string, leveraging a sliding window approach to keep track of the current substring of identical characters.

Here's how the implementation unfolds:

- We initiate the answer (`ans`) and a temporary count (`t`) both set to 1. The minimal power for any string is 1, as any standalone character is a valid substring. These variables will keep track of the maximum power discovered so far and the current sequence length, respectively.
- The `for` loop in the code iterates over each pair of adjacent characters in the string `s`.

  ```
  for a, b in pairwise(s):
  ```

  This is accomplished by utilizing the `pairwise` function, which iterates the string such that in each iteration, `a` and `b` hold a pair of adjacent characters. The `pairwise` function, introduced in Python 3.10, effectively generates a sequence of tuples containing `(s[i], s[i+1])` for `i` ranging from `0` to `len(s) - 2`. If `pairwise` is not available, it would be necessary to create these pairs manually using index-based iteration.
- The core logic takes place inside this loop, checking whether each consecutive pair of elements are the same:
  - If `a == b`, it means we are still looking at a substring of identical characters, so we increment our temporary count `t` and update the maximum power `ans` if necessary:

    ```
    t += 1
    ans = max(ans, t)
    ```

  - When `a != b`, we encounter a different character that breaks the current sequence. Therefore, we reset `t` to 1 to start counting a new sequence:

    ```
    else:
        t = 1
    ```
- Once the loop has finished, we've scanned the whole string and determined the maximum length of a substring with only one unique character, providing us with the power of the string `s`.
- Finally, the function returns the value of `ans`, which is the maximum power that we were looking for:

  ```
  return ans
  ```

This approach is effective because it only requires a single pass through the string, making it an (O(n)) solution, where (n) is the length of the input string. The space complexity is (O(1)) as we use only a few variables regardless of the input size.

Example Walkthrough

Let's illustrate the solution approach with a small example. Consider the string `s = "aabbb"`.

- We start by initializing `ans` and `t` to 1. The string `s` has a minimum substring power of 1 by default because even a single character is considered a substring.
- We enter the loop and compare each pair of adjacent characters:
  - We compare `s[0]` (which is `'a'`) with `s[1]` (also `'a'`). Since they are the same, we increment `t` to 2. We also update `ans` to 2 because it's greater than the initial value of 1.
  - Next, we compare `s[1]` with `s[2]`, but `s[2]` is `'b'`, so they are different. We reset `t` to 1 as we are now starting to count a new sequence of characters.
  - Moving on, we compare `s[2]` (which is `'b'`) with `s[3]` (also `'b'`). They match, so `t` is incremented to 2.
  - We compare `s[3]` with `s[4]`, and again they are the same (`'b'`), so `t` goes up to 3. We compare `ans` with the new `t`, and since 3 is greater than the current `ans` value of 2, we update `ans` to 3.
- After the loop is done, we've gone through the entire string and the maximum `t` value we encountered was 3. Since there are no more characters to check, `ans` is already the maximum power of the string, which is 3.

In this example, the substring with the highest power is `"bbb"`, which has a power of 3, and that's what our function correctly returns.

By following this step-by-step process, we ensure that, as we traverse the string, we keep the count of consecutive identical characters with `t` and always remember the maximum such count in `ans`. Once the traversal is complete, `ans` contains our final result, which is the power of string `s`.

Solution Implementation

Python

```python
class Solution:
    def max_power(self, s: str) -> int:
        # Initialize the maximum power and temporary power count to 1
        max_power = temp_power = 1

        # Go through each pair of adjacent characters in the string
        for i in range(1, len(s)):
            # If the current character is the same as the previous one
            if s[i] == s[i - 1]:
                # Increment the temporary power count
                temp_power += 1
                # Update the maximum power if the new temporary power is higher
                max_power = max(max_power, temp_power)
            else:
                # Reset the temporary power count for a new character sequence
                temp_power = 1

        # Return the maximum power found
        return max_power
```

Java

```java
class Solution {

    /**
     * Calculates the maximum power of a string. The power of the string is
     * the maximum length of a non-empty substring that contains only one unique character.
     * @param s the input string
     * @return the maximum power of the string
     */
    public int maxPower(String s) {
        // Initialize the maximum power to 1, since a single char has a power of 1
        int maxPower = 1;
        // Temporary variable to track the current sequence length
        int currentSequenceLength = 1;

        // Iterate over the string starting from the second character
        for (int i = 1; i < s.length(); ++i) {
            // Check if the current character is the same as the previous one
            if (s.charAt(i) == s.charAt(i - 1)) {
                // If so, increment the current sequence length
                currentSequenceLength++;
                // Update the maximum power if the current sequence is longer
                maxPower = Math.max(maxPower, currentSequenceLength);
            } else {
                // Reset the current sequence length if the character changes
                currentSequenceLength = 1;
            }
        }
        // Return the calculated maximum power
        return maxPower;
    }
}
```

C++

```cpp
class Solution {
public:
    // Function to find the longest substring where all characters are the same
    int maxPower(string s) {
        int max_power = 1; // Initialize the maximum power to 1
        int current_count = 1; // Initialize the current consecutive character count to 1

        // Loop through the string starting from the second character
        for (int i = 1; i < s.size(); ++i) {
            // Check if the current character is the same as the previous one
            if (s[i] == s[i - 1]) {
                // Increase the current consecutive count
                ++current_count;
                // Update the maximum power if the current count is larger
                max_power = max(max_power, current_count);
            } else {
                // Reset the current count when encountering a different character
                current_count = 1;
            }
        }

        return max_power; // Return the maximum power found
    }
};
```

TypeScript

```typescript
// This function calculates the maximum consecutive identical character count in a string.
// @param s - The input string to be analyzed.
// @returns The length of the longest consecutive sequence of identical characters.
function maxPower(s: string): number {
    // Initialize the answer (max consecutive length) to 1, as any non-empty string will have at least a count of 1.
    let maxConsecutiveLength = 1;
    // Start with a temporary count of 1 for the first character.
    let currentCount = 1;

    // Iterate through the string starting from the second character.
    for (let i = 1; i < s.length; ++i) {
        // Check if the current character is the same as the previous one.
        if (s[i] === s[i - 1]) {
            // If so, increment the temporary count.
            currentCount++;
            // Update the maximum consecutive length if the current count exceeds it.
            maxConsecutiveLength = Math.max(maxConsecutiveLength, currentCount);
        } else {
            // If the current character is different, reset temporary count to 1.
            currentCount = 1;
        }
    }
    // Return the maximum consecutive length found.
    return maxConsecutiveLength;
}
```

```python
class Solution:
    def max_power(self, s: str) -> int:
        # Initialize the maximum power and temporary power count to 1
        max_power = temp_power = 1

        # Go through each pair of adjacent characters in the string
        for i in range(1, len(s)):
            # If the current character is the same as the previous one
            if s[i] == s[i - 1]:
                # Increment the temporary power count
                temp_power += 1
                # Update the maximum power if the new temporary power is higher
                max_power = max(max_power, temp_power)
            else:
                # Reset the temporary power count for a new character sequence
                temp_power = 1

        # Return the maximum power found
        return max_power
```

Time and Space Complexity

The given Python code is designed to find the maximum power of a string, which is defined as the maximum length of a non-empty substring that contains only one unique character.

Here is an analysis of its time and space complexities:

Time Complexity:

The time complexity of the function is dictated by the single for loop over the adjacent elements produced by the `pairwise` function. The `pairwise` function creates an iterator that will produce `n-1` pairs, where `n` is the length of the string `s`.

The loop runs exactly `n-1` times if `n` is the length of the string `s`. Each iteration performs a constant time operation; either incrementing `t`, updating `ans` with the `max` function, or resetting `t` to `1`. Therefore, the time complexity is `O(n)`, where `n` is the length of the string.

Space Complexity:

The space complexity of the function is `O(1)`. The reason is that the amount of extra memory used does not depend on the size of the input string. It only uses a fixed number of variables (`ans` and `t`), and `pairwise` (assuming it is similar to `itertools.pairwise`) generates pairs using an iterator, which doesn't consume additional memory proportional to the input size.