1750. Minimum Length of String After Deleting Similar Ends

Medium Two Pointers String

Problem Description

The task involves a string `s` composed solely of the characters `'a'`, `'b'`, and `'c'`. There's an iterative operation defined where we can do the following:

1. Choose a non-empty prefix from the beginning of the string, where all characters in the prefix are identical.
2. Choose a non-empty suffix from the end of the string, where all characters in the suffix are identical.
3. The chosen prefix and suffix should not overlap.
4. The characters in both the prefix and suffix must match.
5. Delete both the prefix and suffix from the string.

Our goal is to repeatedly perform these operations in order to achieve the shortest possible length of the string `s`, and then return that minimum length. It is worth noting that we may choose to perform this operation zero times if it does not contribute to reducing the string's length.

Intuition

To efficiently reduce the length of the string, we utilize a two-pointer strategy:

- One pointer starts at the beginning of the string (`i`) and the other at the end (`j`).
- We check if the characters at both pointers `i` and `j` are equal, since we can only remove matching prefixes and suffixes.
- If they match, we proceed inward from both the prefix and the suffix, ensuring that we're only looking at identical characters which could be potentially removed.
- After skipping over all identical characters in both the prefix and suffix, we move our pointers past these characters (`i` and `j` are incremented and decremented respectively) to essentially 'delete' them from the string.
- We repeat this process until the characters at the positions of `i` and `j` don't match or until `i` is not less than `j`, at which point no further operation can reduce the string's length.

The remaining characters between `i` and `j` (inclusive) represent the irreducible core of the string, and the length of this section is our answer. If `i` surpasses `j`, it means the entire string can be deleted, resulting in a length of 0.

By using this approach, we navigate towards the center of the string, removing matching pairs of prefixes and suffixes, and arrive at the solution in an optimal way with respect to time complexity.

Solution Approach

The solution is based on a two-pointer approach, a popular strategy used to solve problems involving sequences like arrays and strings. Here, we walk through the implementation step by step:

1. Initialize two pointers, `i` at the beginning of the string (index 0) and `j` at the end of the string (`len(s) - 1`).
2. Use a `while` loop to continue the process as long as `i` is less than `j` and the character at `i` is the same as the character at `j`.
3. If the characters at `i` and `i + 1` are the same, increment `i`. This is done using another `while` loop to skip over all identical characters in the prefix.
4. Similarly, if the characters at `j` and `j - 1` are the same, decrement `j`. A `while` loop is used here as well to skip all identical characters in the suffix.
5. Once we've moved past all identical characters at both ends, we increment `i` and decrement `j` once more to 'remove' the matching prefix and suffix.
6. Repeat steps 2 through 5 until characters at `i` and `j` do not match or `i >= j`.
7. Finally, calculate the length of the remaining string. If `i` has crossed `j` (meaning the entire string can be removed), the length is `0`. Otherwise, it's `j - i + 1`.

The code efficiently reduces the length of the string using the defined operation. This solution is effective because it works outward from both ends of the string and avoids unnecessary comparisons. The complexity of this algorithm is O(n), where n is the length of the string, since in the worst case, the algorithm might have to iterate over each character at most twice—once from the beginning and once from the end.

Here's a small snippet demonstrating the increment and decrement operations for the two pointers which are crucial in the implementation:

```
while i < j and s[i] == s[j]:
    while i + 1 < j and s[i] == s[i + 1]:
        i += 1
    while i < j - 1 and s[j - 1] == s[j]:
        j -= 1
    i, j = i + 1, j - 1
```

The `while` loops inside check for continuous characters that are identical at both the start and end (prefix and suffix), and the final line within the loop updates the pointers to simulate the removal of these characters. The result is the length of the section that cannot be reduced any further, calculated by `max(0, j - i + 1)`.

Example Walkthrough

Consider the string `s = "aaabccccba"`. We'll use the solution approach to find the shortest possible length after repeatedly removing matching prefixes and suffixes.

1. Initialize two pointers, `i` at index `0` and `j` at index `9` (since `len(s) - 1 = 10 - 1 = 9`).
2. Start a `while` loop where `i < j` and `s[i] == s[j]`. Initially, `s[i]` is `'a'` and `s[j]` is `'a'`, so the condition holds.
3. Increment `i` to skip over identical characters in the prefix. The characters at index `0`, `1`, and `2` are all `'a'`, so `i` now moves to index `3`.
4. Decrement `j` to skip over identical characters in the suffix. The character at index `9` is `'a'` and at index `8` is `'b'`, so `j` stays at index `9`.
5. Now increment `i` once more and decrement `j` once to 'remove' the `'a'` prefix and suffix. `i` now moves to index `4`, and `j` moves to index `8`.
6. We check if `s[i] == s[j]` again. At this point, `s[i]` is `'b'` and `s[j]` is `'b'`, so we repeat steps 3 through 5:
   - Increment `i` past all `'b'`s. Since `s[i]` is `'b'` only at index `4`, `i` moves to index `5`.
   - Decrement `j` past all `'b'`s. Since `s[j]` is `'b'` at indexes `8` and `7`, `j` moves to index `6`.
   - Now, remove the `'b'` prefix and suffix by setting `i` to `6` and `j` to `5`.
7. Our loop condition is no longer true since `i >= j`, we exit the loop.
8. The remaining string length is `j - i + 1`, which gives us `5 - 6 + 1 = 0`. Since `i` has crossed `j`, the entire string can be deleted, resulting in a minimum length of `0`.

This example walks through the two-pointer approach, illustrating how pointers `i` and `j` are used to reduce the string's length by removing identical prefixes and suffixes. The method proves to be optimal as it quickly narrows down to the core part of the string that cannot be reduced any further.

Solution Implementation

Python

```
class Solution:
    def minimumLength(self, s: str) -> int:
        # Initialize two pointers at the start and end of the string
        left_pointer, right_pointer = 0, len(s) - 1

        # Loop while the left pointer is less than the right pointer and
        # the characters at both pointers are the same
        while left_pointer < right_pointer and s[left_pointer] == s[right_pointer]:

            # Move the left pointer to the right as long as the next character is the same
            while left_pointer + 1 < right_pointer and s[left_pointer] == s[left_pointer + 1]:
                left_pointer += 1

            # Move the right pointer to the left as long as the previous character is the same
            while left_pointer < right_pointer - 1 and s[right_pointer - 1] == s[right_pointer]:
                right_pointer -= 1

            # After moving pointers inward, increment left and decrement right to find a new character
            left_pointer, right_pointer = left_pointer + 1, right_pointer - 1

        # Calculate the remaining length of the string and return it
        # The maximum is used to ensure a non-negative length is returned
        return max(0, right_pointer - left_pointer + 1)
```

Java

```
class Solution {
    public int minimumLength(String s) {
        // Initialize two pointers representing the beginning and end of the string
        int leftIndex = 0, rightIndex = s.length() - 1;

        // Continue looping while the left pointer is less than the right pointer
        // and the characters at both pointers are the same
        while (leftIndex < rightIndex && s.charAt(leftIndex) == s.charAt(rightIndex)) {
            // Increment the left pointer if the current and next characters are the same
            while (leftIndex + 1 < rightIndex && s.charAt(leftIndex) == s.charAt(leftIndex + 1)) {
                leftIndex++;
            }
            // Decrement the right pointer if the current and previous characters are the same
            while (leftIndex < rightIndex - 1 && s.charAt(rightIndex) == s.charAt(rightIndex - 1)) {
                rightIndex--;
            }
            // Move both pointers towards the center
            leftIndex++;
            rightIndex--;
        }

        // Calculate the remaining string length and ensure it's not negative
        // Return 0 if the string has been completely reduced, otherwise return the length
        return Math.max(0, rightIndex - leftIndex + 1);
    }
}
```

C++

```
class Solution {
public:
    int minimumLength(string s) {
        // Initialize two pointers, one at the start and one at the end of the string.
        int start = 0, end = s.size() - 1;

        // Continue with the loop until the two pointers meet or cross each other.
        while (start < end && s[start] == s[end]) {
            // Skip all identical characters from the start.
            while (start + 1 < end && s[start] == s[start + 1]) {
                ++start;
            }
            // Skip all identical characters from the end.
            while (start < end - 1 && s[end] == s[end - 1]) {
                --end;
            }
            // Move the start and end pointers towards each other to the next distinct character.
            ++start;
            --end;
        }

        // Calculate and return the remaining length of the string.
        // If pointers have crossed, return 0 as there's no remaining string.
        return max(0, end - start + 1);
    }
};
```

TypeScript

```
function minimumLength(s: string): number {
    // Initialize the starting index (left side of the string).
    let startIndex = 0;
    // Initialize the ending index (right side of the string).
    let endIndex = s.length - 1;

    // Iterate over the string as long as the starting index is less than the ending index
    // and the characters at these indices match.
    while (startIndex < endIndex && s[startIndex] === s[endIndex]) {
        // Move the starting index to the right, skipping all similar adjacent characters.
        while (startIndex + 1 < endIndex && s[startIndex] === s[startIndex + 1]) {
            startIndex++;
        }
        // Move the ending index to the left, skipping all similar adjacent characters.
        while (startIndex < endIndex - 1 && s[endIndex - 1] === s[endIndex]) {
            endIndex--;
        }

        // Once all adjacent similar characters are skipped, narrow down the search space
        // by incrementing the starting index and decrementing the ending index.
        startIndex++;
        endIndex--;
    }

    // Calculate the remaining length of the string by subtracting the
    // starting index from the ending index and adding 1, making sure it is
    // not less than 0 when the whole string is deleted.
    return Math.max(0, endIndex - startIndex + 1);
}
```

```
class Solution:
    def minimumLength(self, s: str) -> int:
        # Initialize two pointers at the start and end of the string
        left_pointer, right_pointer = 0, len(s) - 1

        # Loop while the left pointer is less than the right pointer and
        # the characters at both pointers are the same
        while left_pointer < right_pointer and s[left_pointer] == s[right_pointer]:

            # Move the left pointer to the right as long as the next character is the same
            while left_pointer + 1 < right_pointer and s[left_pointer] == s[left_pointer + 1]:
                left_pointer += 1

            # Move the right pointer to the left as long as the previous character is the same
            while left_pointer < right_pointer - 1 and s[right_pointer - 1] == s[right_pointer]:
                right_pointer -= 1

            # After moving pointers inward, increment left and decrement right to find a new character
            left_pointer, right_pointer = left_pointer + 1, right_pointer - 1

        # Calculate the remaining length of the string and return it
        # The maximum is used to ensure a non-negative length is returned
        return max(0, right_pointer - left_pointer + 1)
```

Time and Space Complexity

The time complexity of the code is `O(n)` where `n` is the length of the input string `s`. This is because the algorithm iterates through the characters of the string at most twice - once when increasing `i` and once when decreasing `j`. Even though there are nested while loops, they do not result in more than `O(n)` time since each element is considered at most twice.

The space complexity of the code is `O(1)`. This is because the algorithm only uses a fixed number of variables (`i` and `j`) and does not allocate any additional space that is dependent on the size of the input.