1332. Remove Palindromic Subsequences

Easy Two Pointers String

Problem Description

This problem requires you to remove palindromic subsequences from a string `s` that consists solely of the letters 'a' and 'b'. A *subsequence* is any sequence that can be derived from the string by deleting some (or no) characters without changing the order of the remaining characters. Note that subsequences are not required to be contiguous portions of the original string.

A subsequence is *palindromic* if it reads the same forward and backward. For example, "aba" is a palindromic subsequence because it reads the same from the left and the right.

The goal is to determine the minimum number of steps to make the string empty by repeatedly removing palindromic subsequences. Each step involves removing exactly one palindromic subsequence from `s`.

Intuition

The intuitive leap is to realize that, since the string `s` only contains 'a' and 'b', there are at most two types of palindromic subsequences that you can form: one comprising entirely of 'a's and the other consisting solely of 'b's.

Each of these single-type subsequences is by default palindromic. All the 'a's in the string form a palindrome since they are identical characters, and so do all the 'b's. Therefore, we can remove all 'a's in one step and all 'b's in another. This reasoning leads us to the conclusion that at most two steps are needed to make any string `s` empty.

To optimize the number of steps, we observe that if the original string `s` itself is a palindrome, then the entire string `s` is also a palindromic subsequence. Consequently, we can remove the entire string in one step.

The algorithm checks if the string `s` is a palindrome:

- If it is, we know we can remove the entire string in one step, hence `1` is returned.
- If it is not, the optimal strategy is to remove all 'a's in one step and all 'b's in another, resulting in `2` steps, so `2` is returned.

Solution Approach

The solution approach is straightforward as it relies on the characteristics of the input string and the definition of a palindrome.

Algorithm

1. Check if the input string `s` is a palindrome.
   - This is done by comparing the string to its reverse. In Python, the reverse of a string can be obtained with the slice notation `s[::-1]`.
   - The comparison `s == s[::-1]` will be `True` if `s` is a palindrome and `False` otherwise.
2. If `s` is a palindrome, return `1` indicating that only one step is needed to remove the entire string `s` since the whole string is one palindromic subsequence.
3. If `s` is not a palindrome, return `2`.
   - In the worst case, where no single large palindromic sequence can be found, the string can be emptied in two steps: first by removing all occurrences of 'a' and then all occurrences of 'b' (or vice versa). Since `s` only contains 'a' and 'b', there is no instance where it would take more than two steps.

Data Structures

- No additional data structures are needed for this implementation.

Patterns

- This solution uses the two-pointer pattern implicitly by leveraging Python's ability to reverse strings to check for palindromicity—and it does so in a constant time operation with O(n) time complexity for the comparison, where n is the length of the string `s`.

Code

The Python code encapsulates the above logic into a class method, as is common for LeetCode problem solutions. Here is the provided code snippet:

```
class Solution:
    def removePalindromeSub(self, s: str) -> int:
        # If the string is already a palindrome, we can remove it in one step.
        if s == s[::-1]:
            return 1
        # If not, then two steps will always be sufficient.
        return 2
```

This method takes the string `s` as an input and uses a single line of logic to determine the number of steps required to remove all characters by removing palindromic subsequences.

Time Complexity

The time complexity of this function is O(n) due to the need to check each character in the string to determine if it is a palindrome. However, since the string contains at most two distinct characters, the result is bounded by a constant number of steps—either 1 or 2.

Example Walkthrough

Consider the string `s = "ababa"`. Let's walk through the solution approach to determine the minimum number of steps required to remove all palindromic subsequences from `s`.

1. First, we check if `s` is a palindrome by comparing it with its reverse. The reverse of `s` is "ababa", which is exactly the same as the original string `s`. Therefore, `s` is a palindrome.
2. Since `s` itself is a palindromic sequence, we can remove the whole string `s` in one step according to our algorithm. This makes it efficient as we do not need to look for individual palindromic subsequences.
3. Following the algorithm logic, we would return `1`, indicating that it only takes a single step to remove the palindromic subsequence (which in this case is the entire string) to make `s` empty.

This demonstrates the efficacy of the solution approach: if the entire string is a palindrome, it can be removed in one step. If not, the process would require two steps - one for each character type ('a' and 'b'), resulting in the string being rendered empty.

Solution Implementation

Python

```
class Solution:
    def removePalindromeSub(self, s: str) -> int:
        # Check if the given string 's' is a palindrome by comparing it to its reverse.
        # If 's' is a palindrome, only one operation is needed to remove all characters
        # since a palindrome is composed of either 'a's or 'b's or empty, hence return 1.
        # If 's' is not a palindrome, two operations are needed:
        # one to remove all 'a's and one to remove all 'b's, hence return 2.
        return 1 if s == s[::-1] else 2
```

Java

```
class Solution {

    // Method to determine the minimum number of steps to remove all characters from a palindrome or substrings
    public int removePalindromeSub(String str) {
        // Two pointer approach to check if the string is a palindrome
        for (int start = 0, end = str.length() - 1; start < end; ++start, --end) {
            // If the characters at the start and end do not match, the string is not a palindrome
            if (str.charAt(start) != str.charAt(end)) {
                // If the string is not a palindrome, it requires 2 operations:
                // One to remove all 'a's and one to remove all 'b's.
                return 2;
            }
        }
        // If the loop completes, then the string is a palindrome
        // Since the string contains only 'a's and 'b's, any palindrome can be removed in one operation.
        return 1;
    }
}
```

C++

```
class Solution {
public:
    // Function to determine the minimum number of steps to make the string empty by
    // removing palindrome subsequences. The function returns either 1 or 2
    // because any string consists of only 2 different characters: 'a' and 'b'.
    int removePalindromeSub(string s) {
        // Two pointers approach to check if the string is a palindrome
        for (int left = 0, right = s.size() - 1; left < right; ++left, --right) {
            // If characters at the left and right don't match,
            // then it's not a palindrome and requires 2 steps to remove ('a's and 'b's separately)
            if (s[left] != s[right]) {
                return 2;
            }
        }
        // If the string is a palindrome, it can be removed in one step.
        return 1;
    }
};
```

TypeScript

```
/**
 * The function removePalindromeSub determines the minimum number of steps
 * to make the string empty by removing palindromic subsequences.
 * The function returns 1 if the entire string is a palindrome (as it can be removed in one step),
 * otherwise, it returns 2 since any string of just 'a's and 'b's can be removed in two steps:
 * one for 'a's and one for 'b's.
 * @param {string} str - The input string consisting of only 'a' and 'b' characters.
 * @returns {number} The minimum number of steps to make the string empty.
 */
function removePalindromeSub(str: string): number {
    // Iterate from the start and end of the string towards the center,
    // checking if the string is a palindrome.
    for (let startIndex = 0, endIndex = str.length - 1; startIndex < endIndex; startIndex++, endIndex--) {
        // If a non-matching pair is found, the whole string is not a palindrome.
        // Therefore, it will take 2 steps to remove all 'a's and 'b's.
        if (str[startIndex] !== str[endIndex]) {
            return 2;
        }
    }
    // If the whole string is a palindrome, it can be removed in a single step.
    return 1;
}
```

```
class Solution:
    def removePalindromeSub(self, s: str) -> int:
        # Check if the given string 's' is a palindrome by comparing it to its reverse.
        # If 's' is a palindrome, only one operation is needed to remove all characters
        # since a palindrome is composed of either 'a's or 'b's or empty, hence return 1.
        # If 's' is not a palindrome, two operations are needed:
        # one to remove all 'a's and one to remove all 'b's, hence return 2.
        return 1 if s == s[::-1] else 2
```

Time and Space Complexity

Time Complexity

The time complexity of the code is `O(n)` where `n` is the length of the string `s`. This time complexity arises from the need to check whether the string `s` is a palindrome or not which involves a full traversal and reversal operation (`s[::-1]`).

Space Complexity

The space complexity of the code is `O(n)` as well, because when the string `s` is reversed, it creates a new string of the same size as `s`, thus requiring additional space that grows linearly with the input size.