1653. Minimum Deletions to Make String Balanced

Medium Stack String Dynamic Programming Leetcode Link

Problem Description

In this problem, we are given a string `s` which contains only two types of characters: 'a' and 'b'. Our goal is to make the string "balanced" by removing characters from the string. A string is considered to be "balanced" if there are no occurrences where a 'b' is followed by an 'a' at any point later in the string.

The objective is to find the minimum number of deletions required to achieve this balance. In other words, after all the deletions, anywhere you look in the string from left to right, if you see the character 'b', you should not expect to find the character 'a' following it anywhere in the rest of the string.

Intuition

To solve the problem, we need to understand what makes the string unbalanced. The string becomes unbalanced when there is a 'b' that occurs before an 'a' because the problem's condition is that no 'b' should come before an 'a'. So, we essentially need to either remove such 'b's or the 'a's that make them violate the condition.

A straightforward approach might be to remove all 'b's before an 'a', but this might not be optimal, as there might be a large number of 'b's followed by very few 'a's. Similarly, removing all 'a's after a 'b' may also not be optimal for the opposite reason. Hence, we need an approach that efficiently finds the balance by considering the distribution of 'a's and 'b's in the string.

Dynamic programming can be used to find such an optimal solution, but it might be too complex for this scenario. Observing that we only need minimal deletions, we can use a simple iterative approach. We'll go through the string and keep track of the count of 'b's we've seen so far. Each time we encounter an 'a', we have a choice - we can either delete this 'a' or all the 'b's before it. We choose the option that minimizes the total deletions.

The intuitive insight is that, as we scan the string, if we keep track of the number of deletions and the count of 'b's, we can always decide the best option at each step. This accumulates to the minimum number of deletions needed to balance the string by the end of the iteration over the string.

Solution Approach

The solution uses a simple linear scan approach that leverages two variables, `b` and `ans`, to track the minimum number of deletions needed.

Here's a step-by-step explanation of the algorithm used in the solution:

1. Initialize two variables: `b` to keep track of the number of 'b' characters encountered and `ans` to hold the minimum number of deletions required so far. Both are set to 0 at the start.
2. Iterate over each character `c` in the string `s`:
   - If `c` is 'b', increment the count of `b` since it might need to be deleted later if an 'a' follows.
   - If `c` is 'a', we have a decision to make. We can either delete this 'a' or all the 'b's we have encountered before. To make an optimal choice, we update `ans` to be the minimum of `ans + 1` (which represents deleting this 'a') and `b` (which represents deleting all the 'b's encountered so far).
3. The `min` function ensures that we choose the best possible outcome at each step, whether it's deleting the current 'a' or the 'b's counted previously.
4. Upon completion of the iteration through the string, `ans` will hold the minimum number of deletions needed to make the string balanced.

This algorithm is efficient because it only goes through the string once, resulting in a time complexity of `O(n)`, where `n` is the length of the string. No additional data structures are used, and only constant extra space is required, making the space complexity `O(1)`.

Example Walkthrough

Let's consider the string `s = "bbaba"`. We will use the solution approach to determine the minimum number of deletions required to make this string balanced.

1. Initialize `b = 0` and `ans = 0`. No characters have been processed, so no deletions are necessary yet.
2. Process the first character (from left to right):
   - `c = 'b'` → Increment `b` to `1`. We may need to delete this 'b' later if an 'a' appears.
   - As it stands, `b = 1` and `ans = 0`.
3. Process the second character:
   - `c = 'b'` → Increment `b` to `2`. So far, we have not seen an 'a', so no decision needs to be made.
   - `b = 2` and `ans = 0`.
4. Process the third character:
   - `c = 'a'` → We have a choice: delete this 'a' or delete the two 'b's encountered before. We choose the option that requires the least deletions.
   - `ans = min(ans + 1, b) = min(1, 2) = 1`. We decide to delete this 'a' as it involves fewer deletions.
   - `b = 2` and `ans = 1`.
5. Process the fourth character:
   - `c = 'b'` → Increment `b` to `3` because we may need to delete this 'b' too if another 'a' appears.
   - Keep `b = 3` and `ans = 1`.
6. Process the fifth (and last) character:
   - `c = 'a'` → Again, we must decide whether to delete this 'a' or all previous 'b's. Again, choose the option with fewer deletions.
   - `ans = min(ans + 1, b) = min(2, 3) = 2`. We delete this 'a'.
   - Final `b = 3` and `ans = 2`.

Given the example string `bbaba`, the minimum number of deletions required to balance the string, by the solution approach, is `2`. We delete the two 'a's encountered to prevent any 'b' from being followed by an 'a' later in the string.

Python Solution

```python
class Solution:
    def minimumDeletions(self, s: str) -> int:
        # Initialize the answer to 0 and a counter for 'b' characters
        minimum_deletions = 0
        count_b = 0

        # Loop through each character in the string
        for char in s:
            if char == 'b':
                # If the current character is 'b', increment the 'b' counter
                count_b += 1
            else:
                # If the current character is 'a', compute the minimum of:
                # 1. Current minimum deletions + 1 (assuming deleting current 'a')
                # 2. Number of 'b' characters encountered so far
                # This step ensures we take the minimum deletions needed to keep the string without 'ba' substring
                minimum_deletions = min(minimum_deletions + 1, count_b)

        # Return the computed minimum deletions
        return minimum_deletions
```

Java Solution

```java
class Solution {
    // Method to calculate the minimum number of deletions to make string 's' sorted
    public int minimumDeletions(String s) {
        int length = s.length(); // Store the length of the string 's'
        int minDeletions = 0; // Initialize the minimum number of deletions to 0
        int countB = 0; // Initialize the count of 'b' characters to 0

        // Loop through every character in the string
        for (int i = 0; i < length; ++i) {
            // Check if the current character is 'b'
            if (s.charAt(i) == 'b') {
                // Increment the count of 'b' characters
                ++countB;
            } else {
                // It's an 'a' so compute the minimum between
                // deleting the current 'a' plus any previous minimum deletions
                // and the count of 'b' characters seen so far
                minDeletions = Math.min(minDeletions + 1, countB);
            }
        }
        // Return the computed minimum number of deletions
        return minDeletions;
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    int minimumDeletions(string s) {
        int deletionsRequired = 0; // Tracks the number of deletions required
        int countB = 0;            // Counts the number of 'b's encountered

        // Iterate over each character in the string
        for (char& c : s) {
            if (c == 'b') {
                // When a 'b' is found, increment the count of 'b's
                ++countB;
            } else {
                // If we encounter an 'a', decide whether to delete this 'a' or
                // any 'b' encountered so far to get a sorted string 'aa..bb...'
                // The min function chooses the smaller of deletion an 'a' or any previous 'b'
                deletionsRequired = std::min(deletionsRequired + 1, countB);
            }
        }

        // Return the minimum number of deletions required to sort the string
        return deletionsRequired;
    }
};
```

Typescript Solution

```typescript
function minimumDeletions(s: string): number {
    const stringLength = s.length; // Length of the input string
    let deletionCount = 0; // Tracks the minimum number of deletions needed
    countB = 0; // Counts the number of 'b' characters encountered

    for (let i = 0; i < stringLength; ++i) {
        if (s.charAt(i) === 'b') {
            // If the current character is a 'b', increment the 'b' count
            ++countB;
        } else {
            // If the current character is 'a', calculate the minimum between:
            // 1. Incrementing the deletion count (as if deleting the 'a')
            // 2. The current count of 'b's (indicating deletion of all 'b's encountered so far)
            deletionCount = Math.min(deletionCount + 1, countB);
        }
    }

    // Return the minimum number of deletions found
    return deletionCount;
}
```

Time and Space Complexity

The given Python code defines a function `minimumDeletions` which takes a string `s` and returns the minimum number of deletions required to make the string good. A string is considered good if there are no instances of 'b' that come after an 'a'. The algorithm uses two variables `ans` and `b` to track the minimum deletions required and the count of 'b's encountered respectively. It iterates through the string a single time and, therefore, has a linear relationship regarding the number of elements (characters in this case) in the input string.

Time Complexity

The time complexity of the function is `O(n)`, where `n` is the length of the input string `s`. This is because the algorithm iterates through each character in the string exactly once.

Space Complexity

The space complexity of the function is `O(1)`. It only uses a fixed number of variables (`ans` and `b`), which do not grow with the input size, meaning the amount of space used remains constant regardless of the input size.