2330. Valid Palindrome IV

Medium Two Pointers String

Leetcode Link

Problem Description

You are given a 0-indexed string `s` made up of lowercase English letters. To address the problem, you are required to determine if the string `s` can be turned into a palindrome through exactly one or two operations. In this context, an operation is defined as changing any single character in the string `s` to any other character. The expected return value is a boolean `true` if it is possible to make `s` a palindrome with one or two changes, and `false` if it is not.

A palindrome is a string that reads the same forward and backward. For example, the strings "racecar" and "abba" are palindromes.

Intuition

The intuition behind the solution is straightforward. To check if a string is a palindrome, you compare characters from the outside in, meaning you compare the first and the last characters, the second and the second-to-last characters, and so on. If they're all the same, the string is a palindrome. However, this problem allows for one or two operations, meaning you are allowed to make up to two mismatches and still consider the string potentially a palindrome, as you can change one of the mismatching characters to match the other.

The solution approach, therefore, is to iterate over the string from both ends towards the center. If the characters in the corresponding positions don't match, we increment a counter. Since we are allowed up to two operations, we can tolerate up to two mismatches (incrementing the counter each time). If we finish the iteration and the counter is less than or equal to two, we return `true`. Otherwise, we return `false`.

The proposed solution does this efficiently by utilizing two pointers, `i` starting from the beginning of the string (0) and `j` from the end (`len(s) - 1`). With each iteration, these pointers move towards each other, and the comparison is made between `s[i]` and `s[j]`. The counting mechanism described above (`cnt += s[i] != s[j]`) increments `cnt` only when there's a mismatch. This process continues while `i` is less than `j` to ensure every character pair is checked exactly once.

Solution Approach

The implementation of the solution utilizes a simple yet effective two-pointer approach commonly used in array and string manipulation problems. Let's break down the steps, referring to the provided code solution:

1. Initialize two pointers, `i` and `j`, where `i` starts at `0` (the beginning of the string `s`) and `j` starts at `len(s) - 1` (the end of the string). These pointers will move towards the center of the string, comparing the characters at `s[i]` and `s[j]`.

2. Initialize a counter `cnt` to `0`. This counter will keep track of the number of mismatches between the characters at the current pointers.

3. Begin a while loop that runs as long as `i < j`. Since we are comparing characters from both ends of the string towards the center, the loop should terminate when the pointers meet or cross over.

4. Inside the while loop, increment the counter `cnt` if the characters pointed by `i` and `j` are not the same, indicating a mismatch. The code does this with the line `cnt += s[i] != s[j]`, which is a concise way of writing an if-else statement that increments `cnt` by `1` when the condition `s[i] != s[j]` is `True`.

5. After checking for a mismatch, move the pointers closer to the center of the string: `i` moves right (`i + 1`), and `j` moves left (`j - 1`).

The data structures used here are elemental—the string `s`, the integer counter `cnt`, and the integer pointers `i` and `j`. There's no need for more complex structures, as the problem is solved with direct access to string characters and simple arithmetic.

Finally, after exiting the while loop (meaning all the character pairs have been checked), evaluate the `cnt` counter. If it's less than or equal to `2`, which means that zero, one, or two operations can turn `s` into a palindrome, then return `true`. Otherwise, if more than two mismatches were found, the string cannot be made into a palindrome with the allowed number of operations, so return `false`.

In summary, the algorithm's efficiency lies in its linear time complexity, `O(n)`, where `n` is the length of the string `s`. It only traverses the string once and performs a constant amount of work at each step.

Example Walkthrough

Let's consider a small example where the string `s` is `"racebcr"`. According to the problem statement, we are allowed to use one or two operations to change any character to make `s` a palindrome.

1. Initialize `i = 0` and `j = len(s) - 1`, which is `j = 6`.
2. Initialize the mismatch counter, `cnt = 0`.

We compare characters of `s` starting from the ends and moving towards the center:

- Compare `s[i]` with `s[j]` (`s[0]` with `s[6]`), which are "r" and "r". They match, so `cnt` remains `0`.
- Move `i` to `1` and `j` to `5` and compare `s[1]` with `s[5]` ("a" with "c"). They don't match, so increment `cnt` to `1`.
- Move `i` to `2` and `j` to `4` and compare `s[2]` with `s[4]` ("c" with "b"). They don't match, so increment `cnt` to `2`.
- Move `i` to `3` and `j` to `3`. Since `i` now equals `j`, we've reached the middle of the string.

At this point, the `cnt` equals `2`, which is the maximum number of mismatches allowed for us to potentially be able to make the string a palindrome with one or two changes. Therefore, according to our algorithm, it is possible to make the string `s` a palindrome with two operations. Specifically, we could change the second character from "a" to "c" and the fifth character from "b" to "c" to make the string `s` into "racecar", which is a palindrome.

Hence, the function should return `true` for the input string `"racebcr"`.

This walkthrough demonstrates how the two-pointer approach is used to efficiently determine if a string can be turned into a palindrome by changing one or two of its characters.

Python Solution

```python
class Solution:
    def makePalindrome(self, s: str) -> bool:
        # Initialize two pointers, left and right,
        # to start from the beginning and end of the string, respectively.
        left, right = 0, len(s) - 1

        # Initialize a counter to keep track of non-matching character pairs.
        mismatch_count = 0

        # Iterate through the string until the two pointers meet or cross.
        while left < right:
            # If the characters at the current pointers do not match,
            # increment the mismatch counter.
            if s[left] != s[right]:
                mismatch_count += 1

            # Move the pointers closer to the center.
            left += 1
            right -= 1

        # Return True if the number of non-matching pairs is less than or equal to 2.
        # In such a case, by deleting or replacing up to one character,
        # the string can be made into a palindrome.
        return mismatch_count <= 2

```

Java Solution

```java
class Solution {
    public boolean makePalindrome(String s) {
        // Counter for the number of mismatches between characters
        int mismatchCount = 0;
        // Two pointers to compare characters from both ends of the string
        int left = 0;
        int right = s.length() - 1;

        // Iterate over the string until the two pointers meet or cross
        while (left < right) {
            // Check if the characters at the current pointers do not match
            if (s.charAt(left) != s.charAt(right)) {
                // Increment the mismatch counter
                ++mismatchCount;
                // If mismatches are more than 1, it's not possible to make palindrome
                // by changing just one character
                if (mismatchCount > 1) {
                    return false;
                }
            }
            // Move the pointers towards the center
            ++left;
            --right;
        }

        // The string can be made into a palindrome if there's at most one mismatch
        // because we can change one character to match the other
        return true;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // Function to check if a string can be made palindrome by changing at most two characters
    bool makePalindrome(string str) {
        int mismatchCount = 0; // Counter for the number of mismatched character pairs
        int left = 0;                // Left pointer starting from the beginning of the string
        int right = str.size() - 1; // Right pointer starting from the end of the string

        // Loop through the string until the left pointer meets or crosses the right pointer
        while (left < right) {
            // If the characters at the current pointers do not match
            if (str[left] != str[right]) {
                mismatchCount++; // Increment the mismatch counter
            }

            // Move the pointers closer towards the center of the string
            left++;
            right--;
        }

        // The string can be made palindrome if there are at most 2 mismatches (or none)
        return mismatchCount <= 2;
    }
};

```

Typescript Solution

```typescript
function makePalindrome(s: string): boolean {
    let mismatchCount = 0; // Initialize count of non-matching character pairs
    let leftIndex = 0; // Start pointer for the left side of the string
    let rightIndex = s.length - 1; // Start pointer for the right side of the string

    // Loop from both ends of the string towards the center
    while (leftIndex < rightIndex) {
        // If characters at leftIndex and rightIndex do not match
        if (s[leftIndex] !== s[rightIndex]) {
            mismatchCount++; // Increment the count of mismatches
        }
        leftIndex++; // Move the left pointer towards the center
        rightIndex--; // Move the right pointer towards the center
    }

    // A string can be made palindrome by changing at most one character,
    // which might affect at most two of these character pairs (one at each end)
    return mismatchCount <= 1;
}

```

Time and Space Complexity

The provided code snippet checks if a given string can be transformed into a palindrome by modifying at most two characters.

Time Complexity:

The time complexity of the code is `O(n)` where `n` is the length of the input string `s`. This is because the while loop will iterate at most `n/2` times (once for each pair of characters from the start and the end of the string until the middle is reached), performing constant time operations within each iteration.

Space Complexity:

The space complexity of the code is `O(1)` because the amount of extra space used does not depend on the size of the input string. The space used is constant as it only involves two index variables (`i` and `j`), and a counter variable (`cnt`).