680. Valid Palindrome II

Easy Greedy Two Pointers String

Problem Description

The problem requires determining if a given string `s` can be made into a palindrome by removing at most one character. A palindrome is a word, phrase, number, or other sequences of characters which reads the same forward and backward, ignoring spaces, punctuation, and capitalization. For example, 'radar' is a palindrome, while 'radio' is not. However, 'radio' can become a palindrome by removing the 'i', which becomes 'rado', and 'raod' is a palindrome because it reads the same backward as forward. The challenge here is to decide whether such a removal is possible to achieve a palindrome with at most one deletion.

Intuition

To understand the given solution, we should recognize two things:

1. If a string does not need any character removal to be a palindrome, it means that the characters at the start and end of the string (and so on moving inward) match.
2. If one mismatch is found, we get a choice - to remove a character from either the left or the right at the point of mismatch and check if the resulting substrings could form a palindrome.

The solution uses a two-pointer approach:

- Start with two pointers, one at the beginning (`i`) and one at the end (`j`) of the string.
- Move both pointers towards the center, checking if the characters are the same.
- If a mismatch is discovered, there are two substrings to check: one without the character at `i` and one without the character at `j`. We use the helper function `check()` to verify if either substrings can form a palindrome.
- If we can successfully verify one substring as a palindrome, we conclude that the original string can be a palindrome after removing at most one character.
- We continue checking until we either find a proper substring that is a palindrome or exit because we have checked all characters without violating the palindrome property.

The key is the helper function `check(i, j)` which checks if the substring from `i` to `j` is a palindrome by iterating through the substring and comparing characters at the start and end, moving inwards.

By carefully applying the `check()` function only when a mismatch is found, the given algorithm efficiently decides whether one can obtain a palindrome with at most one deletion.

Solution Approach

The solution's core relies on a two-pointer approach while also incorporating a helper method to reduce redundancy. Here's the process step-by-step:

1. **Initial Two-pointer Setup:** Initialize two pointers, `i` and `j`, representing the start and the end of the string `s`. These two pointers will progressively move towards the center.
2. **First Pass - Checking Palindrome:** Move both pointers towards the center, implied by `i < j`. If the characters at `i` and `j` match (`s[i] == s[j]`), we can safely continue. This loop continues until a mismatch is found (when `s[i] != s[j]`) or until the entire string is checked.
3. **Handling Mismatches - Utilizing the Helper Function:** Upon encountering a mismatch, the solution must determine whether omitting one of the characters can lead to a palindrome. This is where the helper method, `check(i, j)`, comes into play. When `s[i] != s[j]`, two checks are made:
   - One check leaves out the character at the end (`check(i, j - 1)`), assuming this might be the extra character causing a non-palindrome.
   - The other check omits the character at the start (`check(i + 1, j)`), assuming this might be the non-matching odd one out.
4. **The Helper Method - `check(i, j)`:** The method takes two indices and checks if the substring between them (`s[i]` to `s[j]`) is a palindrome. It uses the same two-pointer technique, now applying it within the narrower range. It returns `True` if a palindrome is detected, `False` if not.
5. **Single Character Removal Decision:** After calling the `check()` method for both possible single removals, we use logical OR (`or`) to combine the results. If either case returns `True`, the original string can be converted to a palindrome by removing one character. The function then returns `True`.
6. **Completing the Loop:** If no mismatch is found, the loop ends, and we can assume the string is already a palindrome or can be made into one with a single removal (might be a character at the start or the end which doesn't interfere with the palindrome property).
7. **Final Return:** If we reach outside the loop without returning `False`, the string must be a palindrome, and the function returns `True`.

This approach provides an optimal solution as it only scans the string once and performs the minimum necessary comparisons, obeying the constraints set by the problem (at most one character removal).

Example Walkthrough

Let's use the string `s = "abca"` to illustrate the solution approach. We need to determine if it's possible to make `s` into a palindrome by removing at most one character.

1. **Initial Two-pointer Setup:** We start with two pointers, `i = 0` (pointing to 'a') and `j = 3` (pointing to 'a'). The string looks like this: `abca`, with `i` at the first character and `j` at the last.
2. **First Pass - Checking Palindrome:** We compare `s[i]` and `s[j]`. Since `s[0]` is 'a' and `s[3]` is also 'a', there's a match, so we move both `i` and `j` towards each other (now `i = 1, j = 2`).
3. **Finding a Mismatch:** Now `i = 1` is pointing to 'b' and `j = 2` is pointing to 'c'. They don't match (`s[i] != s[j]`). We need to check if removing one character makes it a palindrome. We call the `check()` function twice as follows:
   - `check(1, 1)`: This omits the character at `j` (the 'c') and checks if `ab` is a palindrome.
   - `check(2, 3)`: This omits the character at `i` (the 'b') and checks if `ca` is a palindrome.
4. **The Helper Method - `check(i, j)`:** When we run `check(1, 1)`, it instantly returns `True` as it's effectively checking a single character 'b', which is trivially a palindrome. We don't need to perform `check(2, 3)` as we've already found a potential palindrome by removing 'c'.
5. **Single Character Removal Decision:** Because `check(1, 1)` returned `True`, we have confirmed that by removing one character ('c'), the string `s` could be turned into a palindrome. Therefore, for the input `abca`, the function would return `True`.
6. **Completing the Loop:** In this example, the mismatch was found, and the `check()` function indicated a palindrome is possible, so the loop is completed successfully.
7. **Final Return:** Since we found that a single removal can lead to a palindrome, the function would return `True` for the string `s = "abca"`.

This walkthrough demonstrates how we can efficiently determine that the string "abca" can become a palindrome by removing at most one character, 'c' in this case, leading to the palindrome "aba".

Python Solution

```python
class Solution:
    def validPalindrome(self, string: str) -> bool:
        # Helper function to check if substring string[left:right+1] is a palindrome
        def is_palindrome(left, right):
            while left < right:
                if string[left] != string[right]:
                    return False
                left, right = left + 1, right - 1
            return True

        left, right = 0, len(string) - 1  # Initialize pointers at both ends of the string

        # Iterate while the two pointers don't cross each other
        while left < right:
            # If the characters at the current pointers don't match
            if string[left] != string[right]:
                # Check for palindrome by removing one character - either from the left or right
                # If either case returns true, the function returns true
                return is_palindrome(left, right - 1) or is_palindrome(left + 1, right)
            # Move both pointers towards the center
            left, right = left + 1, right - 1

        # If the string is a palindrome or can be made into one by removing a single character
        return True
```

Java Solution

```java
class Solution {

    // This method checks if a string can be a palindrome after at most one deletion.
    public boolean validPalindrome(String s) {
        // Iterate from both ends towards the center
        for (int left = 0, right = s.length() - 1; left < right; ++left, --right) {
            // If two characters are not equal, try to skip a character either from left or right
            if (s.charAt(left) != s.charAt(right)) {
                // Check if the substring skipping one character on the left is a palindrome
                // or
                // Check if the substring skipping one character on the right is a palindrome
                return isPalindrome(s, left + 1, right) || isPalindrome(s, left, right - 1);
            }
        }
        // If no mismatched characters found, it's already a palindrome
        return true;
    }

    // Helper method to check whether a substring defined by its indices is a palindrome
    private boolean isPalindrome(String s, int startIndex, int endIndex) {
        // Iterate over the substring
        for (int i = startIndex, j = endIndex; i < j; ++i, --j) {
            // If any pair of characters is not equal, it's not a palindrome
            if (s.charAt(i) != s.charAt(j)) {
                return false;
            }
        }
        // No mismatches found, it's a palindrome
        return true;
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    // Function to check whether a given string can be a palindrome by removing at most one character
    bool validPalindrome(string s) {
        int left = 0, right = s.size() - 1;

        // Iterate from both ends towards the center
        while (left < right) {
            // If mismatch is found, check for the remaining substrings
            if (s[left] != s[right]) {
                // Check if the substrings skipping one character each are palindromes
                return isPalindrome(s, left + 1, right) || isPalindrome(s, left, right - 1);
            }
            ++left;
            --right;
        }

        // The string is a palindrome if no mismatches are found
        return true;
    }

private:
    // Helper function to check if a substring is a palindrome
    bool isPalindrome(const string& s, int left, int right) {
        // Check for equality from both ends towards the center
        while (left < right) {
            if (s[left] != s[right]) {
                return false; // Return false if a mismatch is found
            }
            ++left;
            --right;
        }

        return true; // Return true if no mismatches are found
    }
};
```

Typescript Solution

```typescript
// Function to determine if the string can become a palindrome by removing at most one character
function validPalindrome(s: string): boolean {
    // Traverse the string from both ends towards the center
    for (let i = 0, j = s.length - 1; i < j; ++i, --j) {
        // If a mismatch is found, try to remove one character at either end
        if (s.charAt(i) !== s.charAt(j)) {
            // Check if removing from the start or the end makes a palindrome
            return isPalindrome(s.slice(i, j)) || isPalindrome(s.slice(i + 1, j + 1));
        }
    }
    // If no mismatches are found or the string is a palindrome, return true
    return true;
}

// Helper function to determine if a given string is a palindrome
function isPalindrome(s: string): boolean {
    // Traverse the string from both ends towards the center
    for (let i = 0, j = s.length - 1; i < j; ++i, --j) {
        // If a mismatch is found, the string is not a palindrome
        if (s.charAt(i) !== s.charAt(j)) {
            return false;
        }
    }
    // If no mismatches are found, the string is a palindrome
    return true;
}
```

Time and Space Complexity

The given Python code defines a method `validPalindrome` to determine if a given string can be considered a palindrome by removing at most one character.

Time Complexity:

The time complexity of the main function is generally `O(n)`, where `n` is the length of the input string `s`. This is because the function includes a while loop that iterates over each character of the string at most twice - once in the main `while` loop and once in the `check` function, which is called at most twice if a non-matching pair is found.

To break it down:

- The main while loop iterates over the string `s`, comparing characters from `i` to `j`. Each loop iteration takes `O(1)` time.
- If a mismatch is found, there are two calls to the helper function `check`, each with a worst-case time complexity of `O(n/2)`, which simplifies to `O(n)`.

In the worst case, you compare up to `n - 1` characters twice (once for each call of `check`), so the upper bound is `2 * (n - 1)` operations, which still results in a linear time complexity: `O(n)`.

Space Complexity:

The space complexity of the code is `O(1)`. No additional space is proportional to the input size is being used, aside from a constant number of integer variables to keep track of indices. The `check` function is called with different indices but does not use any extra space apart from the input string `s`, which is passed by reference and not copied.