680. Valid Palindrome II **Greedy** Two Pointers String Easy

Problem Description

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.

The problem requires determining if a given string s can be made into a palindrome by removing at most one character. A

Intuition

character.

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.
- 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

• Move both pointers towards the center, checking if the characters are the same.

- 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.

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

pointers will progressively move towards the center.

Solution Approach

process step-by-step:

palindrome is detected, False if not.

palindrome by removing at most one character.

move both i and j towards each other (now i = 1, j = 2).

most one character, 'c' in this case, leading to the palindrome "aba".

if string[left] != string[right]:

left, right = left + 1, right - 1

Iterate while the two pointers don't cross each other

If the characters at the current pointers don't match

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

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

// Check if the substring skipping one character on the left is a palindrome

// Check if the substring skipping one character on the right is a palindrome

return isPalindrome(s, left + 1, right) || isPalindrome(s, left, right - 1);

return False

if string[left] != string[right]:

if (s.charAt(left) != s.charAt(right)) {

// If no mismatched characters found, it's already a palindrome

private boolean isPalindrome(String s, int startIndex, int endIndex) {

for (int i = startIndex, i = endIndex; i < i; ++i, --i) {</pre>

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

// Traverse the string from both ends towards the center

// Helper function to determine if a given string is a palindrome

// Traverse the string from both ends towards the center

// If a mismatch is found, the string is not a palindrome

for (let i = 0, j = s.length - 1; i < j; ++i, --j) {

for (let i = 0, i = s.length -1; i < i; ++i, --i) {

// Function to determine if the string can become a palindrome by removing at most one character

return isPalindrome(s.slice(i, j)) || isPalindrome(s.slice(i + 1, j + 1));

// If a mismatch is found, try to remove one character at either end

// If no mismatches are found or the string is a palindrome, return true

// Check if removing from the start or the end makes a palindrome

Helper function to check if substring string[left:right+1] is a palindrome

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

// Helper method to check whether a substring defined by its indices is a palindrome

// If any pair of characters is not equal, it's not a palindrome

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.

Initial Two-pointer Setup: Initialize two pointers, i and j, representing the start and the end of the string s. These two

- 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:
- The other check omits the character at the start (check(i + 1, j)), assuming this might be the non-matching odd one out. 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

∘ One check leaves out the character at the end (check(i, j - 1)), assuming this might be the extra character causing a non-palindrome.

character. The function then returns True. 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

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

Final Return: If we reach outside the loop without returning False, the string must be a palindrome, and the function returns

- 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**
 - 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. 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

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

character 'b', which is trivially a palindrome. We don't need to perform check(2, 3) as we've already found a potential

Completing the Loop: In this example, the mismatch was found, and the check() function indicated a palindrome is possible,

Final Return: Since we found that a single removal can lead to a palindrome, the function would return True for the string s =

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

check if removing one character makes it a palindrome. We call the check() function twice as follows:

palindrome by removing 'c'.

Solution Implementation

return True

while left < right:</pre>

Python

class Solution:

so the loop is completed successfully.

property).

True.

 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. The Helper Method - check(i, j): When we run check(1, 1), it instantly returns True as it's effectively checking a single

- 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.
- "abca". This walkthrough demonstrates how we can efficiently determine that the string "abca" can become a palindrome by removing at
- 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:</pre>

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

```
// 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) {</pre>
        // If two characters are not equal, try to skip a character either from left or right
```

return true;

--right;

function validPalindrome(s: string): boolean {

function isPalindrome(s: string): boolean {

return false;

if (s.charAt(i) !== s.charAt(j)) {

def validPalindrome(self, string: str) -> bool:

return False

if string[left] != string[right]:

if string[left] != string[right]:

left. right = left + 1, right - 1

Iterate while the two pointers don't cross each other

If the characters at the current pointers don't match

def is palindrome(left, right):

while left < right:</pre>

return True

while left < right:</pre>

if (s.charAt(i) !== s.charAt(j)) {

// Iterate over the substring

if (s.charAt(i) != s.charAt(j)) {

class Solution {

return True

Java

```
return false;
        // No mismatches found, it's a palindrome
        return true;
C++
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) {</pre>
            // 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) {</pre>
            if (s[left] != s[right]) {
                return false; // Return false if a mismatch is found
            ++left;
```

// If no mismatches are found, the string is a palindrome return true;

class Solution:

};

TypeScript

return true;

```
# 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
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.
```

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 0(1) time.

Time Complexity:

• If a mismatch is found, there are two calls to the helper function check, each with a worst-case time complexity of 0(n/2), which simplifies to 0(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 0(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.