Your approach is almost correct, but there is a small mistake in the second loop condition. Specifically, in the backward loop, you have used i < n in the condition, which should be i >= 0 in order to properly iterate backward from the last index (n-1) to the first index (0).

Here's the corrected version of your code:

Correct Code:

```
class Solution {

public:

   vector<int> shortestToChar(string s, char c) {

      int n = s.size();

      vector<int> answer(n, INT_MAX); // Initialize the answer array with max values.

      int lastPos = -1;


      // Forward pass

      for(int i = 0; i < n; i++) {

         if(s[i] == c) {

            lastPos = i; // Update the last position of character 'c'

         }

         if(lastPos != -1) {

            answer[i] = i - lastPos; // Update the distance from the last occurrence of 'c'

         }

      }


      lastPos = -1;


      // Backward pass

      for(int i = n - 1; i >= 0; i--) { // Corrected condition for backward iteration

         if(s[i] == c) {

            lastPos = i; // Update the last position of character 'c' while iterating backward

         }
```

```
        if(lastPos != -1) {
            answer[i] = min(answer[i], lastPos - i); // Get the minimum distance from either forward or backward
        }
    }

    return answer;
  }
};
```

Explanation:

1. **Forward Pass**:
    - The first loop (from left to right) checks the string and finds the distance from the closest previous occurrence of the character c.
    - We update the answer[i] for each index if c is found before it.
2. **Backward Pass**:
    - The second loop (from right to left) makes sure that we also check the closest occurrence of c after each index.
    - We then update answer[i] with the minimum distance between the previous calculated value and the new distance from the closest occurrence of c during the backward pass.

Complexity:

- **Time Complexity**:
  Both passes (forward and backward) take **O(n)** time, where n is the length of the string.

Thus, the overall **Time Complexity** is **O(n)**.

- **Space Complexity**:
  We are using an array answer of size n to store the shortest distances. This gives us **O(n)** space complexity.

Conclusion:

This solution is optimal, and with the correct backward loop condition, it efficiently computes the shortest distance to the character c for each character in the string.

Summary: Optimized Approach for Finding Shortest Distance to Character

---

Problem Statement

Given a string s and a character c, you need to compute the shortest distance from each character in s to the nearest occurrence of c.

Mistakes in Original Code

1. **Assignment Instead of Comparison**
    - **Mistake**: Used = instead of == for comparison in conditions:
    - if (lastPos = -1)
    - **Fix**: Correct the condition to compare:
    - if (lastPos == -1)
2. **Incorrect Logic for Distance Update**
    - **Mistake**: Incorrectly updating answer[i] when lastPos == -1:
    - if (lastPos = -1)
    - **Fix**: Only update the distance if lastPos is not -1 (i.e., a valid position is found):
    - if (lastPos != -1)
3. **Incorrect Loop Condition for Backward Traversal**
    - **Mistake**: The backward loop uses the wrong condition (i < n):
    - for (int i = n - 1; i < n; i--)
    - **Fix**: Use the correct condition to iterate backward (i >= 0):
    - for (int i = n - 1; i >= 0; i--)
4. **Missing Semicolon**
    - **Mistake**: Missing semicolon in the backward loop:
    - answer[i] = min(answer[i], i - lastPos)
    - **Fix**: Add a semicolon:
    - answer[i] = min(answer[i], i - lastPos);

Optimized Approach

Two-Pass Algorithm

1. **Forward Pass**
    - Traverse the string from left to right.
    - Keep track of the last seen position of character c using the variable lastPos.
    - Update the distance to the nearest occurrence of c for each character.
2. **Backward Pass**
    - Traverse the string from right to left.
    - Update the distance by comparing the current distance with the newly computed distance from the nearest occurrence of c encountered during the backward traversal.

Final Correct Code

```
class Solution {
public:
    vector<int> shortestToChar(string s, char c) {
        int n = s.size();
        vector<int> answer(n, INT_MAX);
        int lastPos = -1;

        // Forward pass
        for (int i = 0; i < n; i++) {
            if (s[i] == c) {
                lastPos = i;
            }
            if (lastPos != -1) {
                answer[i] = i - lastPos;
            }
        }

        // Backward pass
```

```
        lastPos = -1;
        for (int i = n - 1; i >= 0; i--) {
            if (s[i] == c) {
                lastPos = i;
            }
            if (lastPos != -1) {
                answer[i] = min(answer[i], lastPos - i);
            }
        }

        return answer;
    }
};
```

Time Complexity

1. **Forward Pass**:
    - You iterate through the string once, performing constant time operations for each character.
    - **Time Complexity**: **O(n)**, where n is the length of the string s.
2. **Backward Pass**:
    - Similarly, you traverse the string once in reverse, performing constant time operations.
    - **Time Complexity**: **O(n)**.

Overall Time Complexity:
Since both passes are linear, the overall time complexity is **O(n)**.

Space Complexity

1. **Answer Array**:
    - We store the result in a vector answer of size n, where n is the length of the string.
    - **Space Complexity**: **O(n)**.

2. **Other Variables**:
    - Variables like lastPos and loop counters take constant space, i.e., **O(1)**.

Overall Space Complexity:
The space complexity is **O(n)** because of the space required for the answer array.

Execution Example

Input:

s = "loveleetcode";

c = 'e';

Output:

[3, 2, 1, 0, 1, 0, 0, 1, 2, 2, 1, 0]

Explanation:

1. **Forward Pass**
    - From left to right, for each character, calculate the distance to the most recent occurrence of c.
    - After forward pass, the result looks like:
      [3, 2, 1, 0, 1, 0, 0, 1, 2, INT_MAX, INT_MAX, INT_MAX].
2. **Backward Pass**
    - From right to left, update the result based on the nearest occurrence of c from the right side.
    - Final result:
      [3, 2, 1, 0, 1, 0, 0, 1, 2, 2, 1, 0].

Key Points to Remember

1. **Two Passes**:
   The problem is solved using two passes: one from left to right and one from right to left. This ensures that we compute the shortest distance for each character in the string.
2. **Handling Edge Cases**:
    - If the character c is found at the start or end of the string, the algorithm correctly handles those positions.
    - This approach ensures that the distance is always the minimum possible value.

3. **Optimal Time Complexity**:
   The solution runs in linear time **O(n)**, making it efficient even for large strings.
4. **Space Complexity**:
   The space complexity is **O(n)** due to the answer array used to store the shortest distances for each character in the string.

Common Mistakes to Avoid

1. Confusing assignment (=) with comparison (==).
2. Incorrect loop conditions, especially when traversing the string backward.
3. Forgetting to update the result in both the forward and backward passes.

By understanding the problem and applying the optimized approach with two passes, we can efficiently solve this problem in **O(n)** time and **O(n)** space.