2193. Minimum Number of Moves to Make Palindrome

Problem Description

You are given a string `s` that is composed only of lowercase English letters. Your goal is to transform this string into a palindrome using the smallest number of moves possible. A palindrome is a word or phrase that reads the same backward as forward. In each move, you are allowed to select any two adjacent characters in the string and swap them.

The challenge ensures that the given string can always be rearranged into a palindrome, so you don't need to check for the possibility of creating one, only to find the minimum number of moves to do so.

Intuition

The solution approach is based on the principle that, to form a palindrome, the letters must mirror each other from the center of the string outwards. That means characters at the start and end of the string must be the same, then the ones inward from them, and so on, until the middle of the string is reached.

Understanding that, we can work our way inwards from both ends of the string towards the center, making swaps to position each character correctly. At each step, you look for the first occasion of the letter at the `i`-th position from the start of the string, starting from the `i`-th position from the end of the string, where `i` is initially `length of the string - 1`. When you find a matching letter, it indicates that it needs to be swapped multiple times until it reaches the `i`-th position. Each swap is counted as one move.

In some cases, there might be a character that does not have a matching pair (an unmatched character) in its immediate search. When such an unmatched situation is found, it means this character must be at the center of the palindrome. Since we know the string can always be a palindrome, there would be at most one such character. For an unmatched character, you need to bring it to the center. This requires a specific number of moves, which is `n // 2 - i`, considering `n` is the length of the string.

The process is repeated until `i` reaches the center of the string, accumulating the number of moves required to place each character in its rightful position for a palindrome.

By following the above approach, the minimum number of moves required to make the string a palindrome can be determined.

Solution Approach

The solution approach for the problem involves a two-pointer technique, iterating from the outermost characters of the string towards the center, and a greedy strategy to make the minimum number of swaps at each step.

Here's a step-by-step breakdown of the algorithm, based on the provided Python implementation:

1. The string `s` is converted into a mutable list `cs` to move characters around easily since strings in Python are immutable.
2. Two pointers `i` and `j` are set up, where `i` starts at the beginning of the list (`0`) and `j` starts at the end (`n - 1`, where `n` is the length of the string).
3. We initiate a counter `ans` to keep track of the number of moves made.
4. The while loop begins and continues until `i` is less than `j`. Within this loop, we are progressively fixing the characters at the beginning and end of the string while moving towards the center.
5. Inside the loop, we start an inner loop that goes in reverse from `j` to `i`. This reverse loop looks for a character matching `cs[i]` starting from the `j`-th position.
6. If a matching character is found (the `even` flag is set to `True`), further nested loops are used to swap the matching character (`cs[k]`) towards its mirror position at `cs[j]`. For every swap, the counter `ans` is incremented.
7. Once swapping is done, `j` is decremented, effectively shortening the active part of the list that hasn't been fixed yet.
8. If no matching character is found (the `even` flag remains `False`), it means that the current character `cs[i]` is unmatched and should be moved to the center. The number of moves is calculated as `(n // 2) - i` and added to `ans`.
9. Pointer `i` is incremented, and the loop continues until all characters are in the correct position for the palindrome.
10. After the loop completes, the `ans` variable contains the total number of moves needed to make `s` a palindrome.

By using greedy decision-making (choosing the nearest match to swap) and the two-pointer approach, we ensure that we are making swaps that progress towards the desired palindrome with the least number of moves. At the end of the process, the function returns the calculated `ans`, which is the minimum number of moves needed to transform the string into a palindrome.

Example Walkthrough

Let's illustrate the solution approach with a simple example where `s = "aabb"`. Following the steps of the algorithm:

1. First, we convert the string `s` to a list `cs`, so `cs = ['a', 'a', 'b', 'b']`.
2. Initialize two pointers: `i = 0` and `j = 3` (since `n = 4`, and `n - 1 = 3`).
3. Set a counter `ans = 0` to keep track of the number of moves made.
4. The while loop begins with `i < j` (0 < 3) and will continue until `i >= j`.
5. We enter the inner loop starting from `j` and going towards `i`. We are looking for the first occurrence of the character `cs[i]` which is `'a'`.
6. We find that `cs[i]` equals `'a'`, thus setting the `even` flag to `True`.
7. Since `cs[i]` already matches `cs[i]`, no swap is needed, and we do not increment `ans`.
8. We now decrease `j` to `2` and effectively shorten the list to `['a', 'a', 'b']`.
9. Pointers are now `i = 0` and `j = 2`. The inner loop will then try to find a match for `cs[i]` which is `'a'` again.
10. Since `cs[2]` is `'b'` and not a match, and `j` eventually equals `i` with no match found, the `even` flag remains `False`. This signifies `cs[i]` is the unmatched character. However, since `s` does have a match already at `cs[1]`, we know this process completes here without the need for further moves.
11. Increment `i` to `1` and as `i` is now equal to `j`, the loop ends.

For this simple example, there were 0 moves needed because the string was already in a palindromic order.

Let's take a slightly more complex example: `s = "abbaa"`.

1. Convert the string `s` to a list `cs`, so `cs = ['a', 'b', 'b', 'a', 'a']`.
2. Pointers are `i = 0`, `j = 4`.
3. Counter is `ans = 0`.
4. Enter the while loop with `i < j` (0 < 4).
5. Looking for a match for `cs[i]` (which is `'a'`) starting from `cs[j]`.
6. No swap needed immediately as `cs[i]` is `'a'`.
7. Decrease `j` to `3`. Move `i` to `1` since `i` and `j` match.
8. Now look for a match for `cs[i]` (which is `'b'`) starting from `cs[j]`.
9. We find the match immediately at `cs[2]` which is `'b'`.
10. Decrease `j` to `2`. Since `j` is now equal to `i`, no swap is needed.
11. Increment `i` to `2`. Now `i` is greater than `j`, the loop ends.
12. The unanswered character 'a' in `cs[1]` now has to be moved to the center.

For this example, one swap is needed to move 'a' from the end to the center, making the string `abaaa` which is a palindrome.

Please note that while these examples are simple, the approach works even with more complex strings and ensures the minimum number of moves because of the greedy two-pointer strategy used.

Python Solution

```python
class Solution:
    def min_moves_to_make_palindrome(self, s: str) -> int:
        # Convert the string into a list for easy manipulation.
        char_list = list(s)
        # Initialize the number of moves and the indexes.
        moves = 0
        n = len(s)
        left = 0
        right = n - 1

        # Iterate until we have checked all characters.
        while left < right:
            # Flag to check if we found a matching character.
            found = False

            # Iterate from right to left starting from the current right index,
            # looking for a matching character.
            for k in range(right, left, -1):
                if char_list[k] == char_list[k]:
                    # When we find a match, we set found to True.
                    found = True
                    # We then move the matched character to its correct position
                    # by swapping adjacent characters.
                    while k < right:
                        char_list[k], char_list[k + 1] = char_list[k + 1], char_list[k]
                        k += 1
                        # Increment the moves count for each swap.
                        moves += 1
                    # Once we have moved the character to the correct position,
                    # we decrement the right index.
                    right -= 1
                    break

            # If we did not find a matching character, it means
            # the character needs to be moved to the center for odd length palindrome.
            if not found:
                # For this unmatched character, how many moves will it take to reach the center?
                # It will be half the length of the string minus the current index.
                moves += (n // 2) - left

            # Move to the next character from the left.
            left += 1

        # Return the total number of moves required to form a palindrome.
        return moves
```

Java Solution

```java
class Solution {
    public int minMovesToMakePalindrome(String s) {
        // Length of the given string
        int length = s.length();
        // Initialize the count of minimum moves to 0
        int moves = 0;
        // Convert the string to character array for easy manipulation
        char[] characters = s.toCharArray();

        // We'll use two pointers, starting from the outside of the array moving inwards
        for (int leftIndex = 0, rightIndex = length - 1; leftIndex < rightIndex; ++leftIndex) {
            // A variable to check if we have found a matching pair or not
            boolean pairFound = false;

            for (int searchIndex = rightIndex; searchIndex != leftIndex; --searchIndex) {
                // Check if the characters match.
                if (characters[leftIndex] == characters[searchIndex]) {
                    // If match is found, we set the pairFound status to true
                    pairFound = true;

                    // Now we need to move the matching character to the right place
                    for (; searchIndex < rightIndex; ++searchIndex) {
                        // Swap characters
                        char temp = characters[searchIndex];
                        characters[searchIndex] = characters[searchIndex + 1];
                        characters[searchIndex + 1] = temp;

                        // Incrementing the move counter for each swap
                        ++minimumMoves;
                    }

                    // Since we found a pair and moved the character, we can move the right pointer inwards
                    --rightIndex;
                    break; // Done with moving the pair, break to outer loop for next character
                }
            }

            // If no pair was found for the current character
            if (!pairFound) {
                // This means it's the middle character in a palindrome with odd length
                // We need to move it to the center
                minimumMoves += length / 2 - leftIndex;
            }
        }
        // Return the minimum number of moves needed to make the string a palindrome
        return minimumMoves;
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    int minMovesToMakePalindrome(string str) {
        int strSize = str.size();  // Length of the string
        int moves = 0;  // Initialize the number of moves to 0

        // Use two pointers approach to move characters to form palindrome
        for (int i = 0, j = strSize - 1; i < j; ++i) {
            bool foundPair = false; // Flag to check if a character matching the left pointer is found on the right

            // Loop to find a character from the right that matches with str[i]
            for (int k = j; k < strSize - 1; ++k) {
                if (str[i] == str[k]) {
                    foundPair = true;
                    // Move the matching character to the correct position on the right
                    for (; k < j; ++k) {
                        swap(str[k], str[k + 1]); // Swap characters
                        ++moves; // Increment moves count for each swap
                    }
                    --j; // Move the right pointer inwards
                    break;
                }
            }

            // If no matching character is found, it must be moved to the center of the palindrome
            if (!foundPair) {
                moves += strSize / 2 - i; // Increment moves by the required positions to reach center
            }
        }
        return moves; // Return total number of moves
    }
};
```

Typescript Solution

```typescript
// Function to find the minimum number of moves to make a string palindrome.
function minMovesToMakePalindrome(str: string): number {
    const strSize: number = str.length;  // Length of the string
    let moves: number = 0;  // Initialize the number of moves to 0

    // Use two pointers approach to move characters to form palindrome
    for (let i = 0, j = strSize - 1; i < j; ++i) {
        let foundPair: boolean = false; // Flag to check if a character matching the left pointer is found on the right

        // Loop to find a character from the right that matches with str.charAt(i)
        for (let k = j; k < strSize - 1; ++k) {
            if (str.charAt(i) === str.charAt(k)) {
                foundPair = true;
                // Move the matching character to the correct position on the right
                for (; k < j; ++k) {
                    str = swapCharacters(str, k, k + 1); // Swap characters
                    moves++; // Increment moves count for each swap
                }
                j--; // Move the right pointer inwards
                break;
            }
        }

        // If no matching character is found, it must be moved to the center of the palindrome
        if (!foundPair) {
            moves += Math.floor(strSize / 2) - i; // Increment moves by the required positions to reach center
        }
    }
    return moves; // Return total number of moves
};

// Helper function to swap characters in a string and return the new string
function swapCharacters(str: string, i: number, j: number): string {
    const charArray: string[] = [...str]; // Convert string to array for swapping
    const temp: string = charArray[i];
    charArray[i] = charArray[j];
    charArray[j] = temp;
    return charArray.join(''); // Convert array back to string and return
}
```

Time and Space Complexity

The given Python code defines a method `minMovesToMakePalindrome` that calculates the minimum number of adjacent swaps required to make the input string a palindrome.

Time Complexity:

The time complexity of the code is governed by the two nested loops (the `while` and the inner `for` loop) that traverse the characters of the string. The outer `while` loop runs for approximately `n/2` iterations, where `n` is the length of the string (since `i` starts from `i` and `j` starts from `n - 1` and they move towards the center).

Inside the outer loop, the `for` loop searches for a matching character from the end of the substring to the current position of `i`. In the worst-case scenario, this can lead to about `n` comparisons for the first iteration of the outer loop, then `n-2` for the next, and so on. This pattern of comparisons forms a series that can be approximated to `n/2` computations on average per outer loop iteration.

Furthermore, the swapping operations within the `while` loop (inside the inner `for` if a match is found) have a potential to run up to `n` times in worst-case scenarios. This adds up to another series of operations contributing to the time complexity.

Thus, the total time complexity is the summation of these two series of operations, which can be estimated with the arithmetic series summation formula. The time complexity approximates to $O(n^2)$.

Space Complexity:

The space complexity is determined by the additional space required by the algorithm aside from the input string. Here, the input string is converted to a list `cs`, which utilizes $O(n)$ space. The rest of the variables (`ans`, `n`, `i`, `j`, `even`, and `k`) use constant space.

Therefore, the total space complexity of the algorithm is $O(n)$, as the space used by the list `cs` is the only space that grows proportionally with the input size.