Solution: Minimum Number of Moves to Make Palindrome

Let’s solve the Minimum Number of Moves to Make Palindrome problem using the Two Pointers pattern.

We'll cover the following

Statement

Given a string `s`, return the minimum number of moves required to transform `s` into a palindrome. In each move, you can swap any two adjacent characters in `s`.

> **Note:** The input string is guaranteed to be convertible into a palindrome.

Constraints:

- $0 \leq$ `s.length` ≤ 2000
- `s` consists of only lowercase English letters.
- `s` is guaranteed to be converted into a palindrome in a finite number of moves.

Solution

The main strategy for solving this problem is to use a two-pointer approach to progressively match characters from the outer ends of the string toward the center, while minimizing adjacent swaps to transform the string into a palindrome. For each character on the left side, the algorithm searches for its matching counterpart on the right side and moves it into place by repeatedly swapping adjacent characters. If a match is found, the right-side pointer moves inward; if no match is found, it indicates that the character is the center of an odd-length palindrome and is positioned accordingly.

Using the above intuition, the solution can be implemented as follows:

1. Initialize a variable, `moves`, with 0 to keep track of the number of swaps required.
2. Initialize two pointers, `i` at the beginning of the string and `j` at the end of the string, to traverse the string from both ends toward the center.

   I. At each iteration, the goal is to match the character at position `i` with the corresponding character at position `j`.

3. Start an inner loop with `k` initialized to `j`, which represents the current character at the end of the string. It moves backward from `j` to `i` to find a matching character for `s[i]`.

   I. The loop checks whether `s[i] == s[k]`. If a match is found, we keep swapping `s[k]` with `s[k+1]` until `k` reaches `j`. For each swap, increment the `moves` counter.

   II. After the character is moved to position `j`, decrement `j` to continue processing the next character from the end.

4. If no match is found by the time `k` reaches `i` (i.e., `k == i`), it means that the character at `i` is the center character of an odd-length palindrome.

   I. In this case, the number of moves is incremented by `(s.size() / 2) - i`, which is the number of moves required to bring this unique character to the center of the string. In this case, we don't swap any characters; just update `moves`.

5. After processing the entire string, return the value of `moves`, which represents the minimum number of moves needed to transform the input string into a palindrome.

Let’s look at the following illustration to get a better understanding of the solution: