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.

Approach

↳ Initialize a variable, moves, with ∅ to keep track of the number of swaps required.

↳ 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 towards the center.

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

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

- The loop checks s[i] == s[k]. If so, swap s[k] with s[k+1] until k reaches j. For each swap, increment the moves counter.

- When the character is moved to j, decrement j to continue processing the next character from the end.

↳ 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.

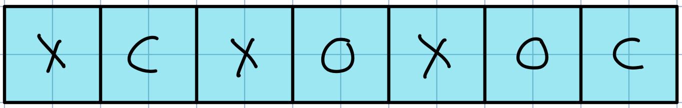
- In this case, the number of moves is incremented by (s.size() / 2) − 1, which is the number of moves required to bring the unique character to the center of the string.

- No character swapping required, just increment moves

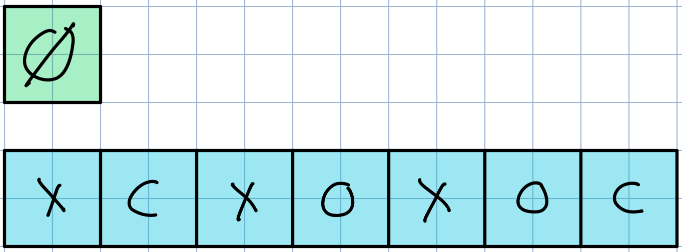
↳ After processing the string, return value of moves

Visualization

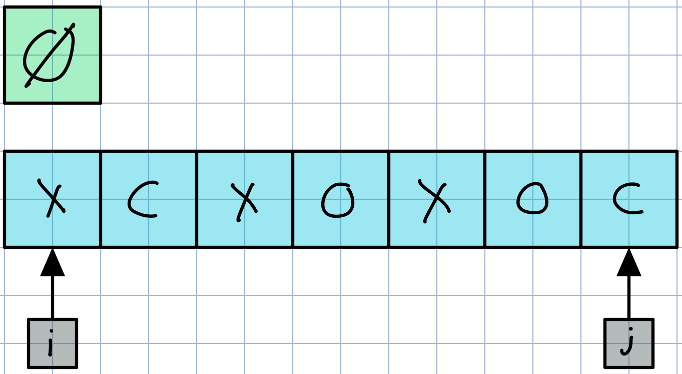
i) Given the input string, find minimum number of moves to transform into palindrome.



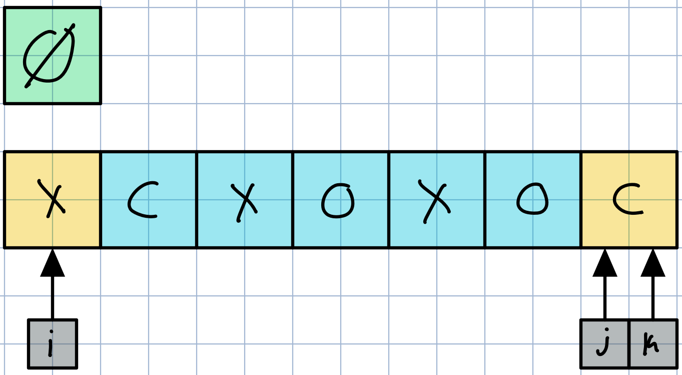
ii) Initialize a variable, moves, with Ø to keep track of the number of swaps made.



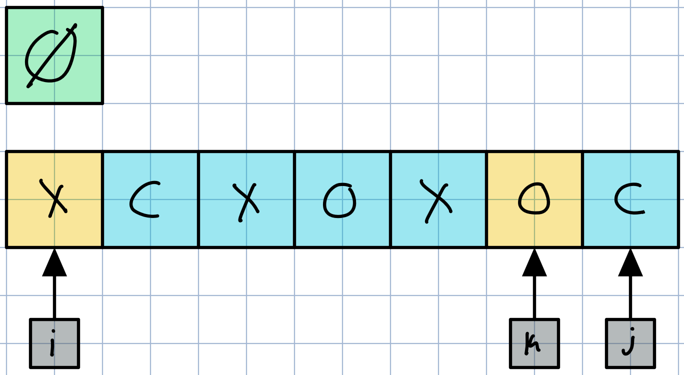
iii) Initialize two pointers, i and j, at the start and end of the string



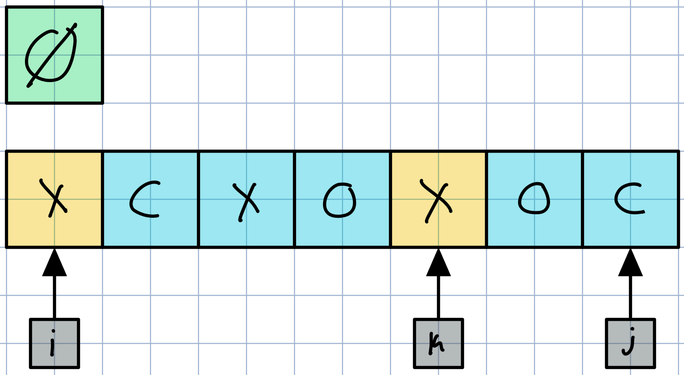
iv) Start an inner loop with k, representing the current character at the end of the string. It works backwards from j to i to find a matching character for s[i].



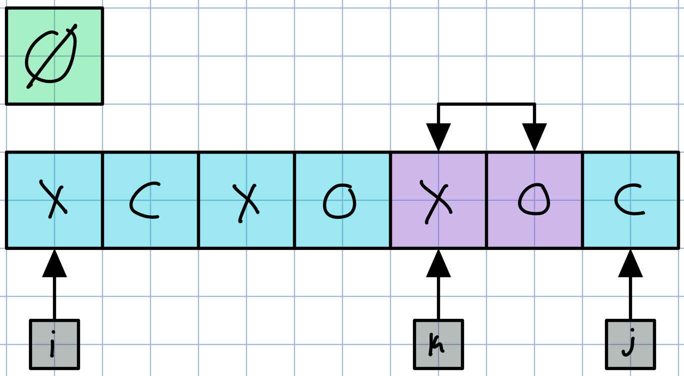
v) Because s[i] was not equal to s[k], move k inward.



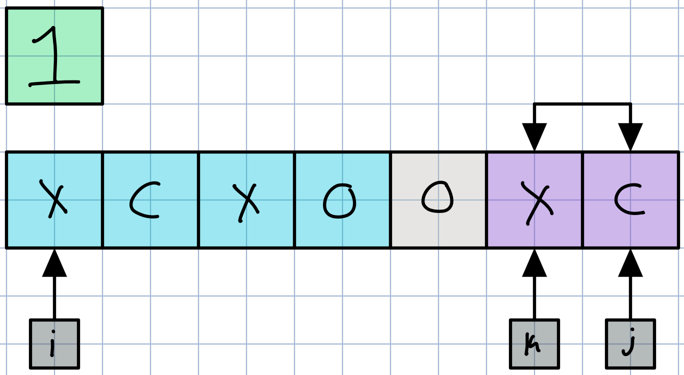
vi) Because s[i] was not equal to s[k], move k inward.



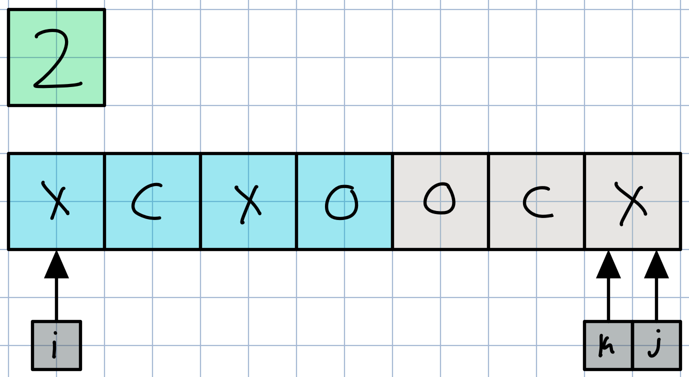
vii) s[i] == s[k], so keep swapping s[k] with s[k+1] until k reaches j.



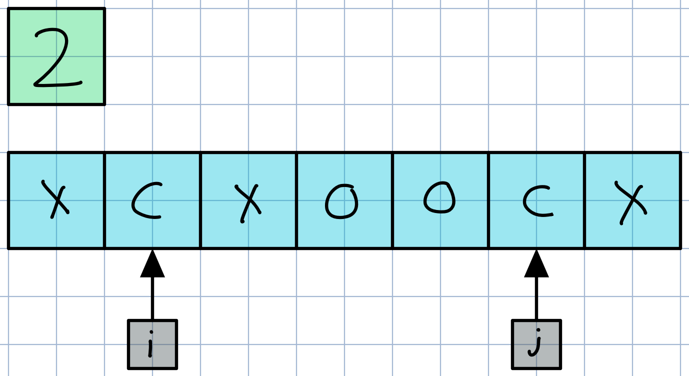
viii) Because k<j, swap s[k] with s[k+1]



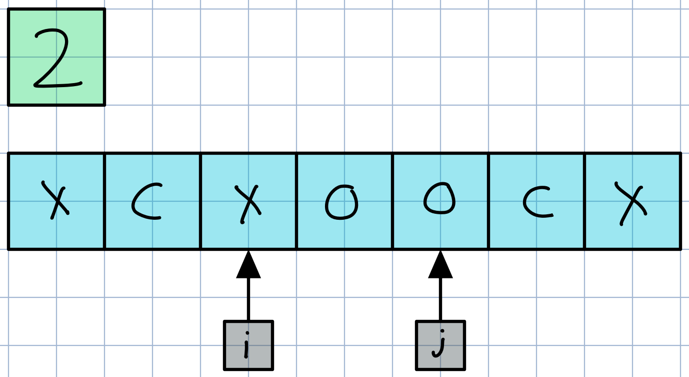
ix) The character corresponding to s[i] = 'x' has been placed at the correct position.



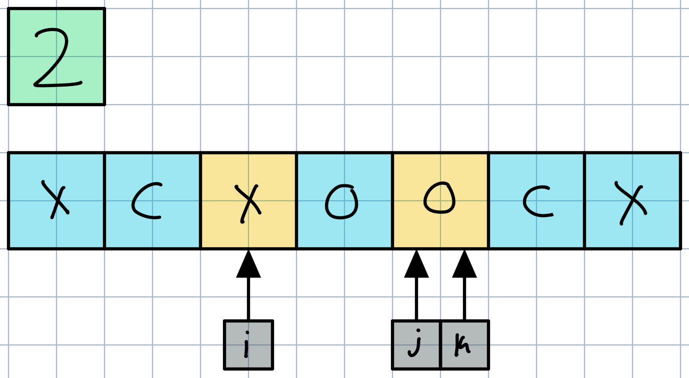
x) Move both pointers, i and j, inward.



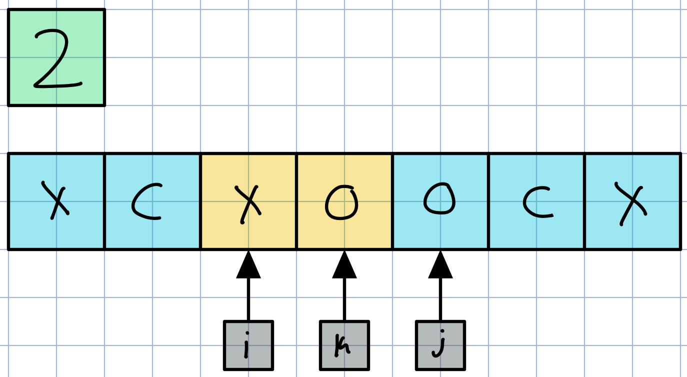
xi) Because s[i] == s[j], don't perform any swap and move both pointers inward.



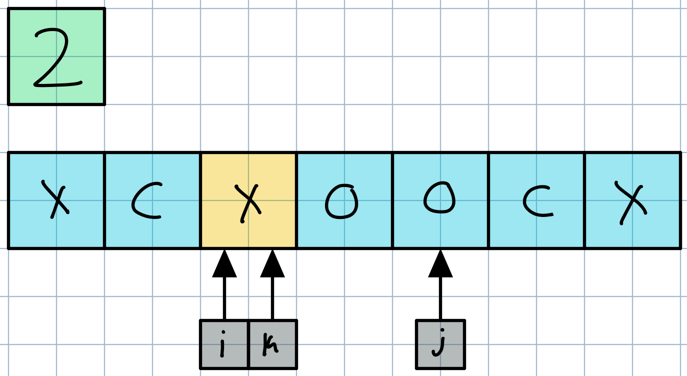
xii) Initialize k with j to start the inner loop to find the matching character for s[i] = 'x`.



xiii) Because s[i] != s[k], move k inward



xiv) Because s[i] != s[k], move k inward



xv) Because k has reached i, no matching character has been found for s[i]. This means the character at i is the center character of an odd-length palindrome. Here, moves is incremented by (s.size()/2)-i, which is the number of moves required to bring the character to the center.

Code

```
int MinMovesToMakePalindrome(string s) {
  int moves=Ø;

  for (int i=Ø, j= s.size()-1; i<j; i++) {
     int k= j;
     for ( ; k>i; k--) {
        if ( s[i]==s[k]) {
           for ( ; k<j; k++) {
              swap (s[k], s[k]+1);
              moves++;
           }
           j--;
           break;
        }
     }
     if (k==i) {
        moves += s.size()/2 - i;
     }
  }
  return moves;
}
```

Time Complexity

$O(n^2)$

Space Complexity

$O(1)$