1. Write all possibilites to check palindrome. Also do time and space complexity analysis

Ans:

There are 6 Possible Solutions:

**1. Reverse and Compare Method**

This method reverses the string and compares it with the original.

**Algorithm:**

1. Reverse the string.
2. Compare the reversed string with the original string.

**Time Complexity:**

* Reversing the string takes O(n), where n is the length of the string.
* Comparing two strings also takes O(n).
* **Total Time Complexity**: O(n).

**Space Complexity:**

* The reversed string requires O(n) extra space.
* **Space Complexity**: O(n).

**2.Two Pointer Approach**

**Algorithm**

1. Initialize two pointers, one at the beginning (left) and one at the end (right).
2. Compare characters at left and right.
3. If they are the same, move both pointers inward and repeat.
4. If any pair is different, the string is not a palindrome.

**Time Complexity:**

* We iterate through the string once, comparing the characters.
* **Total Time Complexity**: O(n).

**Space Complexity:**

* No additional space is required (apart from a few variables).
* **Space Complexity**: O(1).

**3. Recursion Method**

Check palindrome using recursion by comparing the first and last characters and recursively checking the substring.

**Algorithm:**

1. If the string is empty or has only one character, return true.
2. Check if the first and last characters are the same.
3. Recursively check the substring excluding the first and last characters.

**Time Complexity:**

* Each recursive call compares two characters and then calls itself on a smaller substring.
* **Total Time Complexity**: O(n).

**Space Complexity:**

* Recursion depth is O(n), so the space complexity for the call stack is O(n).
* **Space Complexity**: O(n).

**4. Using Stack**

A stack can be used to check if a string is a palindrome by pushing the first half of the characters and then popping them while checking against the second half.

**Algorithm:**

1. Push the first half of the string onto a stack.
2. For the second half, pop elements from the stack and compare them with the current characters.

**Time Complexity:**

* Pushing and popping from a stack takes O(1) per operation.
* We perform these operations for half of the string.
* **Total Time Complexity**: O(n).

**Space Complexity:**

* The stack stores half the characters of the string, which requires O(n/2) space.
* **Space Complexity**: O(n).

**5. Character-by-Character Iteration (Iterative Check)**

This method uses a loop to check each character in a forward and backward manner without using additional space.

**Algorithm:**

1. Initialize two pointers, one at the start (i) and one at the end (j).
2. Increment i and decrement j in each iteration, comparing characters at i and j.

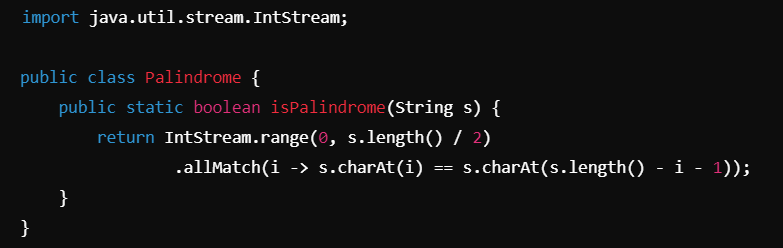
**Time Complexity:**

* We iterate through the string once, making comparisons.
* **Total Time Complexity**: O(n).

**Space Complexity:**

* No additional data structures are used.
* **Space Complexity**: O(1).

**6. Using Stream API (Java 8+)**

****

**Time Complexity:**

* The range function and comparison loop run in O(n).
* **Total Time Complexity**: O(n).

**Space Complexity:**

* Stream-based solutions don’t require additional space beyond the input.
* **Space Complexity**: O(1).

