Sorted and Rotated Array

Give DSA notes for this program along with the code public boolean check(int[] arr) {
    // Handle edge cases: Arrays with less than 3 elements are always considered sorted and rotated
    if (arr.length < 3) return true;

    int n = arr.length;
    int breakPoint = -1;

    // Step 1: Find the first point where the array is not in ascending order
    for (int i = 1; i < n; i++) {
        if (arr[i] < arr[i - 1]) {
            breakPoint = i; // Mark the index where order breaks
            break;
        }
    }

    // If no break point is found, the array is already sorted
    if (breakPoint == -1) return true;

    // Step 2: Check if the array can wrap around correctly
    // Verify that all elements from the break point to the end are in ascending order
    for (int i = breakPoint + 1; i < n; i++) {
        if (arr[i] < arr[i - 1]) return false; // If out of order, not a rotated sorted array
    }

    // Step 3: Ensure the rotated part (end of array) connects back to the start of the array
    if (arr[n - 1] > arr[0]) return false; // Last element must not be greater than the first

    return true; // The array is sorted and rotated
}

Below is a detailed explanation and DSA notes for the provided program:

Topic: Detecting if an Array is Sorted and Rotated

Problem Statement:

Given an array `arr`, determine if it is a sorted and rotated array. An array is sorted and rotated if it can be sorted by rotating (shifting) a sorted array some number of times.

Approach:

1. **Edge Cases**:
   - Arrays with fewer than 3 elements are trivially sorted and rotated, so return `true`.
2. **Key Observations**:
   - In a sorted and rotated array, there will be at most one "break point," i.e., an index `i` where `arr[i] < arr[i - 1]`.
   - After the break point, the array should continue in ascending order till the end.
   - The last element of the array should not be greater than the first element for the array to wrap around correctly.
3. **Steps**:
   - **Step 1**: Identify the break point where the array stops being in ascending order.
   - **Step 2**: Check if all elements from the break point onward (till the end) are sorted.
   - **Step 3**: Verify the wrap-around condition where the last element connects to the first.

Code Explanation:

java

```java
public boolean check(int[] arr) {
    // Handle edge cases: Arrays with less than 3 elements are considered sorted and rotated
```

```
    if (arr.length < 3) return true;

    int n = arr.length;
    int breakPoint = -1;

    // Step 1: Find the first point where the array is not in ascending order
    for (int i = 1; i < n; i++) {
        if (arr[i] < arr[i - 1]) {
            breakPoint = i; // Mark the index where order breaks
            break;
        }
    }

    // If no break point is found, the array is already sorted
    if (breakPoint == -1) return true;

    // Step 2: Check if the array can wrap around correctly
    // Verify that all elements from the break point to the end are in ascending order
    for (int i = breakPoint + 1; i < n; i++) {
        if (arr[i] < arr[i - 1]) return false; // If out of order, not a rotated sorted array
    }

    // Step 3: Ensure the rotated part (end of array) connects back to the start of the array
    if (arr[n - 1] > arr[0]) return false; // Last element must not be greater than the first

    return true; // The array is sorted and rotated
}
```

Complexity Analysis:

1. **Time Complexity**:
    - The loop to find the break point and verify the conditions runs linearly over the array.
    - Overall time complexity: **O(n)**.

2. **Space Complexity**:
   - No additional space is used apart from a few variables.
   - Overall space complexity: **O(1)**.

Key Insights:

1. A single "break point" indicates where the rotation happens.
2. The elements after the break point and before the break point must each form an ascending sequence, with the end connecting back to the start.

Test Cases:

1. **Edge Case**:
   Input: `[1]`
   Output: `true`
   Explanation: Single-element arrays are always sorted and rotated.
2. **Sorted and Rotated**:
   Input: `[3, 4, 5, 1, 2]`
   Output: `true`
   Explanation: Rotated version of `[1, 2, 3, 4, 5]`.
3. **Not Sorted and Rotated**:
   Input: `[2, 1, 3, 4]`
   Output: `false`
   Explanation: Not a valid rotated sorted array.
4. **Already Sorted**:
   Input: `[1, 2, 3, 4, 5]`
   Output: `true`
   Explanation: The array is sorted, and no rotation is needed.

This solution efficiently checks for sorted and rotated arrays while adhering to optimal time and space complexity requirements.