📄 Algorithm Documentation for Dominator

1. Problem Description

Given an array `A` of integers, the goal is to find an index where the **dominator** of the array occurs. A **dominator** is defined as an element that appears in more than half of the elements of the array. If such an element exists, return any index where it occurs. Otherwise, return `-1`.

---

2. Pseudocode

Recursive Approach

```
FUNCTION findDominator(A)
    IF A is empty THEN
        RETURN -1
    END IF
    RETURN findDominatorRecursive(A, 0, length(A) - 1)
END FUNCTION

FUNCTION findDominatorRecursive(A, left, right)
    IF left == right THEN
        RETURN left
    END IF

    mid ← (left + right) / 2
    leftIndex ← findDominatorRecursive(A, left, mid)
    rightIndex ← findDominatorRecursive(A, mid + 1, right)

    leftValue ← A[leftIndex] IF leftIndex != -1 ELSE MIN_VALUE
    rightValue ← A[rightIndex] IF rightIndex != -1 ELSE MIN_VALUE

    IF leftValue == MIN_VALUE THEN
        RETURN rightIndex
    IF rightValue == MIN_VALUE THEN
        RETURN leftIndex

    leftCount ← 0
    rightCount ← 0
    FOR i FROM left TO right DO
        IF A[i] == leftValue THEN
            leftCount++
        IF A[i] == rightValue THEN
            rightCount++
    END FOR

    threshold ← (right - left + 1) / 2
    IF leftCount > threshold THEN
        RETURN leftIndex
    IF rightCount > threshold THEN
        RETURN rightIndex
```

```
    RETURN -1
END FUNCTION
```

Non-Recursive Approach

```
FUNCTION findDominator(A)
    IF A is empty THEN
        RETURN -1
    END IF

    candidate ← A[0]
    count ← 1
    candidateIndex ← 0

    FOR i FROM 1 TO length(A) - 1 DO
        IF A[i] == candidate THEN
            count++
        ELSE
            count--
            IF count == 0 THEN
                candidate ← A[i]
                candidateIndex ← i
                count ← 1
            END IF
    END FOR

    count ← 0
    FOR each value IN A DO
        IF value == candidate THEN
            count++
    END FOR

    IF count > length(A) / 2 THEN
        RETURN candidateIndex
    ELSE
        RETURN -1
    END IF
END FUNCTION
```

---

3. Time and Space Complexity Analysis

Metric	Recursive Approach	Non-Recursive Approach
Time Complexity	O(n)	O(n)

Explanation:

- **Recursive Approach**: The time complexity is O(n) because of the additional loop used to count the occurrences of the candidate element. The recursion depth is O(log n), leading to a space complexity of O(log n).

- **Non-Recursive Approach**: The time complexity remains O(n) due to the need to iterate over the entire array twice: once for candidate selection and once for verification. The space complexity is O(1) as it only requires a few extra variables.

---

4. Strengths and Weaknesses

Recursive Approach

Strengths:

- The algorithm follows a divide-and-conquer approach, which is conceptually elegant.
- It clearly demonstrates the recursive problem-solving paradigm.

Weaknesses:

- The recursion depth can lead to stack overflow for very large arrays.
- It may not be as efficient as the iterative approach in practice due to recursion overhead.

Non-Recursive Approach

Strengths:

- More efficient in terms of space usage, as it uses a constant amount of extra space.
- The iterative approach is generally more performant for large arrays.

Weaknesses:

- While still efficient, the non-recursive approach might be slightly more complex to implement and understand for those unfamiliar with the technique.

---

5. When to Use

Recursive Approach

- This approach is suitable for educational purposes or when recursion is preferred due to its simplicity in expressing divide-and-conquer logic.
- It works well for small arrays or when the maximum recursion depth is not a concern.

Non-Recursive Approach

- The non-recursive approach is ideal for larger arrays, where the efficiency of space and time is crucial.
- It should be used when performance is a concern, especially for larger datasets.

---

6. Final Comparison Table

Criteria	Recursive Approach	Non-Recursive Approach
Time Complexity	O(n)	O(n)
Efficiency	Less efficient	More efficient for large arrays
Use Case	Small arrays	large arrays