1. Algorithm Overview

Brief Description:

The **Boyer-Moore Majority Vote** algorithm is a widely recognized solution for finding the majority element in an array, where the majority element is defined as the element that appears more than half of the times in the array. It is designed to be highly efficient both in terms of time complexity and space complexity. The primary advantage of the Boyer-Moore algorithm is that it guarantees finding the majority element (if it exists) in **O(n)** time while using only **O(1)** additional space.

Theoretical Background:

The Boyer-Moore Majority Vote algorithm relies on two phases of iteration over the array: **candidate selection** and **candidate validation**. The first phase identifies a potential majority element by keeping track of a current candidate and its count. The second phase validates this candidate by counting its occurrences in the array. If the candidate's count exceeds half the size of the array, it is returned as the majority element; otherwise, the algorithm returns -1.

- **Candidate Selection:** The algorithm works by maintaining a counter (count) and a candidate. Initially, the counter is set to 0, and no candidate is chosen. As the algorithm iterates through the array, it keeps updating the candidate:
    - If the counter is 0, the current element becomes the candidate, and the counter is set to 1.
    - If the current element is the same as the candidate, the counter is incremented.
    - If the current element is different from the candidate, the counter is decremented.

This step ensures that any element that does not appear more than half the time is effectively eliminated. However, this process doesn't guarantee that the candidate is the majority element—it only ensures that if there is a majority element, it will be the candidate at the end of this phase.

- **Candidate Validation:** The second phase is necessary to verify whether the candidate element is truly the majority. The algorithm counts the number of times the candidate appears in the array, and if it appears more than half the times, it is returned as the majority element.

Why It Works:

This approach works because the array is traversed only twice. The first pass selects the most likely majority element, and the second pass validates it. The key observation is that for any element that is not the majority, the counter will eventually cancel out its occurrences, leaving the actual majority element as the candidate by the end of the first pass.

The Boyer-Moore algorithm is optimal in that it achieves **O(n)** time complexity while using only a fixed number of variables. It avoids the need for sorting or hashing, which would have higher time and space complexities.

2. Complexity Analysis

Time Complexity Analysis:

The Boyer-Moore Majority Vote algorithm is designed to run in linear time, **O(n)**, where n is the size of the input array. The reason for this is that the algorithm performs two passes through the array:

- **First Pass:** The first pass is used to select the candidate. It only requires **O(n)** operations, where each element is checked and compared to the current candidate.
- **Second Pass:** The second pass counts the occurrences of the candidate element, which also takes **O(n)** time.

Since both passes are linear in time complexity, the total time complexity remains **O(n)**. There is no nested iteration or recursion that would increase the complexity.

Best Case:

In the best case, the majority element appears early in the array. The first pass identifies this majority element quickly. However, the algorithm still requires a second pass to confirm this, making the best-case time complexity still **O(n)**.

Worst Case:

In the worst case, the majority element may be harder to identify. It could be near the end of the array or the element might not exist at all. Despite this, the algorithm will still perform two linear scans over the array, so the time complexity remains **O(n)** in the worst case.

Average Case:

On average, the performance remains **O(n)** because the algorithm processes the array in two linear scans, regardless of the structure of the data.

Space Complexity Analysis:

The space complexity of the Boyer-Moore Majority Vote algorithm is **O(1)**, which means that it only requires a constant amount of extra memory regardless of the size of the input array.

- The algorithm only needs to store a few variables:
    - The candidate, which is the current element being considered as the majority.
    - The count, which tracks how many times the candidate appears.
- These variables do not scale with the input size, so the space complexity is constant.

Since no additional data structures (such as hash tables or arrays) are needed, the algorithm is very memory efficient, making it suitable for use in environments with limited memory resources.

Mathematical Justification Using Big-O, Θ, and Ω Notations:

- **Big-O (Upper Bound):** Big-O describes the worst-case time complexity. In this case, it is **O(n)**, meaning that as the size of the input array grows, the number of operations increases linearly.
- **Theta (Θ, Tight Bound):** Theta notation represents the exact bound. The Boyer-Moore Majority Vote algorithm runs in **Θ(n)** time because it always performs two passes over the array, resulting in a linear relationship between the input size and the number of operations.
- **Omega (Ω, Lower Bound):** Omega notation represents the best-case scenario. The algorithm performs **Ω(n)** operations because it always processes every element at least once.

Thus, the time complexity of the algorithm is **O(n)**, **Θ(n)**, and **Ω(n)**, confirming that the algorithm is efficient and scalable.

Comparison with Kadane's Algorithm:

Kadane's algorithm is another widely used **O(n)** algorithm, but it is used for finding the maximum sum subarray. While both algorithms operate in linear time, they serve different purposes:

- **Boyer-Moore Majority Vote** is designed to find the majority element.
- **Kadane's Algorithm** is designed to find the maximum sum subarray.

Both algorithms have **O(1)** space complexity because they only require a few variables to track the current state.

3. Code Review

Code Structure and Efficiency:

The code provided is simple and efficient, but there are a few areas where improvements could be made:

1. **Candidate Selection Logic:**
   The logic for selecting the candidate is correct, but it might benefit from additional comments or refactoring into smaller, more modular functions. Right now, the code mixes the candidate selection and validation into a single method, which could be separated for clarity.
2. **Array Access and Comparison Count:**
   The use of the PerformanceTracker class to count the number of comparisons, array accesses, and swaps is a good idea for performance analysis. However, one potential improvement could be to separate the performance tracking logic from the core algorithm logic. This would make the code more modular and easier to maintain.
3. **Memory Allocation Tracking:**
   The incrementMemoryAllocation() method is called within the BenchmarkRunner class, but it seems unnecessary for this algorithm because no dynamic memory structures are being created. It could be useful to track memory allocations for algorithms that utilize large data structures, but here it does not add much value.

Potential Code Improvements:

1. **Combining Loops:**
   One potential optimization is combining the two loops into a single loop. Currently, the algorithm scans the array twice: once to select the candidate and once to validate it. These operations could potentially be combined into a single loop, though this might make the code more complex to understand.

2. **Input Validation:**
   The algorithm checks for an empty array and handles it by returning -1. However, additional checks for arrays of size 1 or arrays where all elements are the same might be useful. For instance, if all elements in the array are the same, the candidate is trivially the majority element.
3. **Early Exit in Validation Phase:**
   During the validation phase, the algorithm could potentially exit early if the candidate count falls below the threshold (n / 2). This would save time in cases where the majority element is already confirmed early in the second pass.
4. **Refactoring for Modularity:**
   The findMajorityElement method could be broken down into smaller helper methods:
   - selectCandidate(): Responsible for the candidate selection logic.
   - validateCandidate(): Responsible for the validation of the candidate.
5. **Edge Case Handling:**
   Additional edge cases could be handled more gracefully. For example, the algorithm could check if the array contains only one element and return that element as the majority.

4. Empirical Results

Performance Plots (Time vs Input Size):

To evaluate the performance of the Boyer-Moore Majority Vote algorithm, performance plots can be created to show how the execution time scales with input size. These plots should demonstrate a **linear relationship** between the input size and execution time, as expected from the **O(n)** time complexity.

- **Data Collection:**
  - The time taken for different array sizes (e.g., 100, 1000, 10000, 100000) should be recorded. This can be done by running the algorithm on arrays of varying sizes and measuring the time taken to execute.
  - A graph should plot **input size** on the x-axis and **execution time** on the y-axis.

Validation of Theoretical Complexity:

Empirical testing should confirm the theoretical **O(n)** time complexity:

- **First Pass:** The time taken for the first pass should increase linearly with the input size.
- **Second Pass:** Similarly, the time for the second pass should also grow linearly with the size of the array.
- A combined graph should show the total time, which should demonstrate a **linear relationship**.

Analysis of Constant Factors and Practical Performance:

Although the time complexity is **O(n)**, the actual performance might be influenced by other factors:

- **Processor Speed:** Faster processors will execute the algorithm in less time, but the linear growth rate will remain the same.
- **Memory Access Patterns:** The way elements are accessed in memory can impact performance. For example, cache locality might affect the speed of the algorithm on large arrays.

Despite these factors, the algorithm should maintain a linear time complexity in most practical situations.

5. Conclusion

Summary of Findings:

The Boyer-Moore Majority Vote algorithm is an efficient and optimal solution for detecting the majority element in an array. It operates in **O(n)** time complexity and uses **O(1)** space, making it ideal for large datasets. The algorithm guarantees that the majority element (if it exists) will be identified in two passes over the array.

Optimization Recommendations:

- **Early Exit During Validation:** An early exit during the validation phase could save unnecessary computation.
- **Improved Input Validation:** Additional checks for edge cases, such as arrays of size 1 or arrays with identical elements, would improve robustness.

- **Code Refactoring:** Breaking the findMajorityElement method into smaller, more manageable helper functions could improve readability and maintainability.