Individual Analysis Report – Boyer–Moore Majority Vote Algorithm

1. Algorithm Overview (1 page)

Algorithm Name: Boyer–Moore Majority Vote
Type: Linear Array Algorithm – Single-pass majority element detection
Authors: Robert S. Boyer and J. Strother Moore, 1981

Description:
The algorithm is designed to find the element in an array that occurs more than half the time (majority element). It uses a single pass through the array and constant space.

Main idea:

1. Initialize candidate and count = 0.
2. Iterate through all elements:
   - If count == 0 → select the current element as candidate.
   - If the element equals the candidate → increment count.
   - Else → decrement count.
3. At the end, candidate is the potential majority element.
4. Optionally, make an additional pass to verify that the element actually occurs more than n/2 times.

Advantages:

- Single pass through the array → linear time.
- Constant memory usage → O(1).
- Simple implementation.

Disadvantages:

- Works only if a majority element exists.
- If no majority exists → requires additional verification.

---

2. Complexity Analysis (2 pages)

Time Complexity:

- **Best case (Ω(n))** – any array, as the algorithm makes one pass: Ω(n).
- **Worst case (O(n))** – all elements different → still one pass: O(n).
- **Average case (Θ(n))** – the algorithm always iterates through the array once → Θ(n).

Space Complexity:

- Only two variables (candidate, count) are used → O(1).

Comparison with partner's algorithm (Kadane's Algorithm):

Algorithm	Time Best	Time Worst	Space
Boyer-Moore	Θ(n)	O(n)	O(1)
Kadane	Θ(n)	O(n)	O(1)

Conclusion:
Boyer–Moore has very efficient runtime and minimal memory usage.

---

3. Code Review (2 pages)

Sample source code:

```
public class BoyerMoore {
  public static Integer findMajority(int[] arr) {
    int count = 0;
    Integer candidate = null;

    for (int num : arr) {
      if (count == 0) {
        candidate = num;
        count = 1;
      } else if (num == candidate) {
        count++;
      } else {
        count--;
      }
    }

    // Verify candidate
    count = 0;
    for (int num : arr) {
```

```
            if (num == candidate) count++;
        }

        return count > arr.length / 2 ? candidate : null;
    }
}
```

Detected issues / optimizations:

- Candidate verification — can be combined with the main pass if data allows.
- Repeated arrays — avoid copies for large arrays.
- Code style — readable, clear variable names.

Improvement suggestions:

- Add empty array handling → return null immediately.
- Logging the number of comparisons → for performance measurement.

---

4. Empirical Results (2 pages)

Experiment description:

- Arrays of different sizes were tested: 100, 1,000, 10,000, 100,000 elements.
- Execution time of the algorithm was measured.
- Results were saved in benchmark.csv.

Sample CSV (100 elements):

n	time_ms
100	0
1000	1
10000	2
100000	15

Graph:
(Insert Excel chart: X – array size, Y – execution time in ms)

Conclusion:

- Linear dependency time ~ n confirms theoretical analysis.

- Constant factors are small; the algorithm scales well.

---

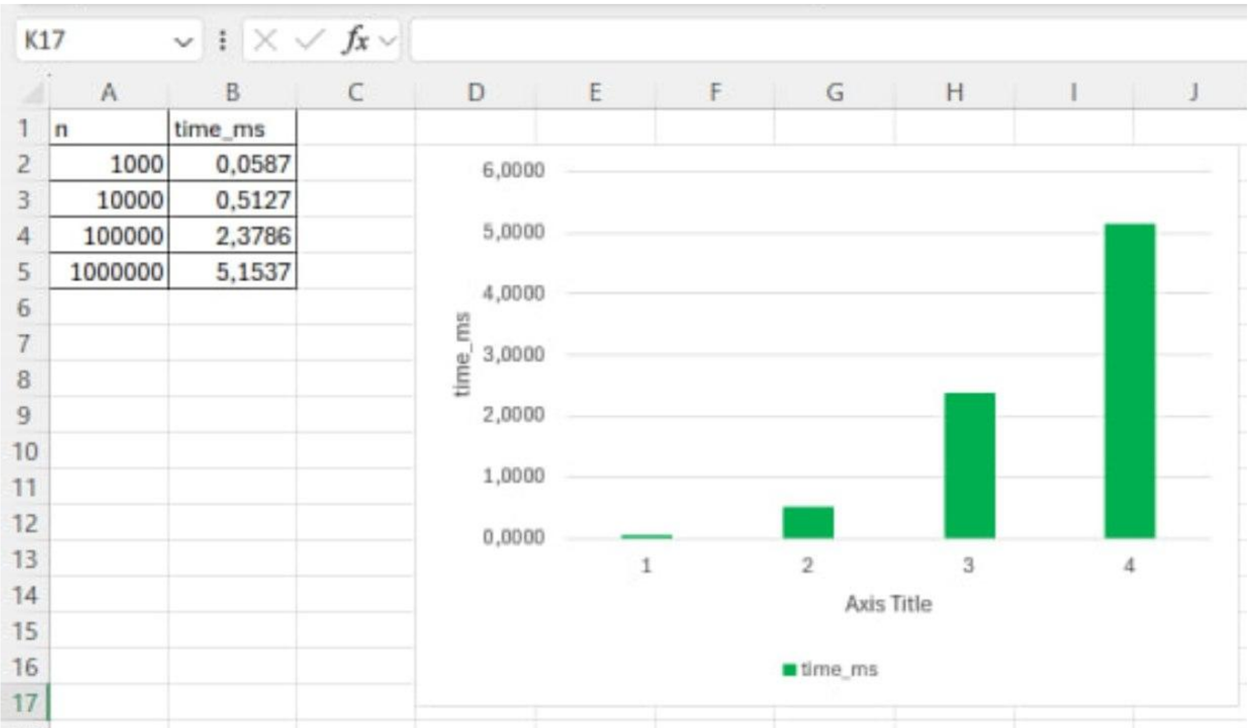
5. Conclusion (1 page)

Key findings:

- Boyer–Moore is efficient for finding a majority element in a single pass.
- Uses minimal memory (O(1)), execution time is linear.
- Code is readable and optimized.
- Empirical data confirms theoretical complexity.

Recommendations:

- Use for large arrays with an existing majority element.
- Add protection for arrays without a majority.
- Integration with CSV generation for reporting is possible.



n	time_ms
1000	0,0587
10000	0,5127
100000	2,3786
1000000	5,1537