Comparative Report on Kadane's Algorithm and Boyer–Moore Majority Vote Algorithm

SE-2435(Pair3)
Mirasbek Idiatulla and Abylay Dossymbek

Introduction

This report presents a comparative analysis of two important algorithms in computer science: **Kadane's Algorithm**, which solves the Maximum Subarray Problem, and the **Boyer–Moore Majority Vote Algorithm**, which determines whether a majority element exists in an array. Both algorithms are known for their efficiency, achieving **linear time complexity** while using minimal memory. This document consolidates the findings from the two individual reports, covering their theoretical principles, complexity analysis, implementation review, empirical performance, and suggested improvements.

Algorithm Overviews

Kadane's Algorithm

Kadane's Algorithm addresses the **Maximum Subarray Problem**, which seeks the contiguous subarray within a one-dimensional array that has the largest sum. The algorithm maintains two variables:

- **Current Sum**: the sum of the current subarray.
- **Maximum Sum**: the best sum encountered so far.

If the current sum becomes negative, the algorithm discards it and starts a new subarray. This guarantees efficiency and correctness in a single pass.

Key characteristics:

- Time complexity: **O(n)** (single pass).
- Space complexity: **O(1)** (constant variables only).
- Well-suited for large datasets and real-time applications.

Boyer–Moore Majority Vote Algorithm

The Boyer–Moore Majority Vote Algorithm determines whether a **majority element** exists (an element that appears more than ⌊n/2⌋ times). The algorithm proceeds in two phases:

1. **Candidate Selection:** Iterating through the array while maintaining a candidate and a counter.
2. **Verification:** Counting how many times the candidate occurs to confirm majority status.

The algorithm relies on a **cancellation principle**: pairs of different elements cancel each other, leaving the true majority element (if it exists).

Key characteristics:

- Time complexity: **O(n)** (two passes at most).
- Space complexity: **O(1)** (candidate and counter only).
- Extremely memory-efficient.

Complexity Analysis

Both algorithms achieve **linear time complexity**:

Algorithm	Best Case	Average Case	Worst Case	Space Complexity
Kadane's Algorithm	Θ(n)	Θ(n)	O(n)	O(1)
Boyer–Moore Majority Vote	O(1) (tiny arrays)	Θ(n)	O(n)	O(1)

- **Kadane's Algorithm** requires a single pass, consistently running in linear time regardless of input.
- **Boyer–Moore** may perform two passes but is still linear, with lower constant factors due to simpler state management.

Code Review and Implementation Quality

Kadane's Algorithm Implementation:

- Clean and readable code with descriptive variable names (`maxSum`, `currentSum`, `startIndex`, etc.).
- Correct handling of subarray tracking.
- Potential improvement: explicitly handle **empty arrays** to prevent runtime issues.
- Suggested refinements: reduce redundant assignments and add more inline documentation.

Boyer–Moore Implementation:

- Organized into well-structured Java classes with static methods.
- Proper handling of edge cases (empty arrays, single elements, no-majority cases).
- Improvements suggested:
  - Early stopping during verification (if majority confirmed before scanning entire array).
  - More detailed performance tracking.
  - Enhanced benchmarking output (e.g., CSV export with auto-plotting).

Empirical Results

Kadane's Algorithm

- Benchmarks conducted on arrays of sizes 100, 1,000, 10,000, and 100,000 elements.
- Execution time increased **linearly with input size**, confirming theoretical complexity.
- Minimal memory overhead, scaling well even for very large arrays.

Boyer–Moore Algorithm

- Benchmarks tested similar input sizes with random arrays containing duplicates.
- Runtime scaled linearly with input size.
- Very low constant factors, since the algorithm only maintains a candidate and a counter.
- Fewer operations per element compared to Kadane's Algorithm.

Comparative Insights

- Both algorithms run in **Θ(n)** time and **O(1)** space, making them highly efficient.
- **Kadane's Algorithm** is optimal for problems involving subarray sums and large datasets.
- **Boyer–Moore** is superior in scenarios requiring detection of majority elements, with even lighter memory use.
- Both projects demonstrate strong coding skills, correctness, and effective benchmarking, though small improvements could increase robustness and maintainability.

Conclusion

The joint study of Kadane's Algorithm and Boyer–Moore Majority Vote Algorithm highlights the elegance and efficiency of linear-time solutions.

- **Kadane's Algorithm** proves optimal for maximum subarray problems, balancing simplicity, performance, and practical applicability.
- **Boyer–Moore Majority Vote Algorithm** excels in detecting majority elements with minimal memory use.

Both implementations were correct, well-structured, and supported by empirical validation. Suggested enhancements include better edge-case handling, early termination strategies, improved documentation, and more sophisticated performance reporting.

Together, these two projects provide valuable insights into the design of efficient algorithms and the importance of combining **theoretical rigor with practical testing**.

Introduction

This report presents a comparative analysis of two important algorithms in computer science: **Kadane's Algorithm**, which solves the Maximum Subarray Problem, and the **Boyer–Moore Majority Vote Algorithm**, which determines whether a majority element exists in an array. Both algorithms are known for their efficiency, achieving **linear time complexity** while using minimal memory. This document consolidates the findings from the two individual reports, covering their theoretical principles, complexity analysis, implementation review, empirical performance, and suggested improvements.

Algorithm Overviews

Kadane's Algorithm

Kadane's Algorithm addresses the **Maximum Subarray Problem**, which seeks the contiguous subarray within a one-dimensional array that has the largest sum. The algorithm maintains two variables:

- **Current Sum**: the sum of the current subarray.
- **Maximum Sum**: the best sum encountered so far.

If the current sum becomes negative, the algorithm discards it and starts a new subarray. This guarantees efficiency and correctness in a single pass.

Key characteristics:

- Time complexity: **O(n)** (single pass).
- Space complexity: **O(1)** (constant variables only).
- Well-suited for large datasets and real-time applications.

Boyer–Moore Majority Vote Algorithm

The Boyer–Moore Majority Vote Algorithm determines whether a **majority element** exists (an element that appears more than ⌊n/2⌋ times). The algorithm proceeds in two phases:

1. **Candidate Selection:** Iterating through the array while maintaining a candidate and a counter.
2. **Verification:** Counting how many times the candidate occurs to confirm majority status.

The algorithm relies on a **cancellation principle**: pairs of different elements cancel each other, leaving the true majority element (if it exists).

Key characteristics:

- Time complexity: **O(n)** (two passes at most).
- Space complexity: **O(1)** (candidate and counter only).
- Extremely memory-efficient.

Complexity Analysis

Both algorithms achieve **linear time complexity**:

Algorithm	Best Case	Average Case	Worst Case	Space Complexity
Kadane's Algorithm	Θ(n)	Θ(n)	O(n)	O(1)
Boyer–Moore Majority Vote	O(1) (tiny arrays)	Θ(n)	O(n)	O(1)

- **Kadane's Algorithm** requires a single pass, consistently running in linear time regardless of input.
- **Boyer–Moore** may perform two passes but is still linear, with lower constant factors due to simpler state management.

Code Review and Implementation Quality

Kadane's Algorithm Implementation:

- Clean and readable code with descriptive variable names (`maxSum`, `currentSum`, `startIndex`, etc.).
- Correct handling of subarray tracking.
- Potential improvement: explicitly handle **empty arrays** to prevent runtime issues.
- Suggested refinements: reduce redundant assignments and add more inline documentation.

Boyer–Moore Implementation:

- Organized into well-structured Java classes with static methods.
- Proper handling of edge cases (empty arrays, single elements, no-majority cases).
- Improvements suggested:
  - Early stopping during verification (if majority confirmed before scanning entire array).
  - More detailed performance tracking.
  - Enhanced benchmarking output (e.g., CSV export with auto-plotting).

Empirical Results

Kadane's Algorithm

- Benchmarks conducted on arrays of sizes 100, 1,000, 10,000, and 100,000 elements.
- Execution time increased **linearly with input size**, confirming theoretical complexity.
- Minimal memory overhead, scaling well even for very large arrays.

Boyer–Moore Algorithm

- Benchmarks tested similar input sizes with random arrays containing duplicates.
- Runtime scaled linearly with input size.

- Very low constant factors, since the algorithm only maintains a candidate and a counter.
- Fewer operations per element compared to Kadane's Algorithm.

Comparative Insights

- Both algorithms run in **Θ(n)** time and **O(1)** space, making them highly efficient.
- **Kadane's Algorithm** is optimal for problems involving subarray sums and large datasets.
- **Boyer–Moore** is superior in scenarios requiring detection of majority elements, with even lighter memory use.
- Both projects demonstrate strong coding skills, correctness, and effective benchmarking, though small improvements could increase robustness and maintainability.

Conclusion

The joint study of Kadane's Algorithm and Boyer–Moore Majority Vote Algorithm highlights the elegance and efficiency of linear-time solutions.

- **Kadane's Algorithm** proves optimal for maximum subarray problems, balancing simplicity, performance, and practical applicability.
- **Boyer–Moore Majority Vote Algorithm** excels in detecting majority elements with minimal memory use.

Both implementations were correct, well-structured, and supported by empirical validation. Suggested enhancements include better edge-case handling, early termination strategies, improved documentation, and more sophisticated performance reporting.

Together, these two projects provide valuable insights into the design of efficient algorithms and the importance of combining **theoretical rigor with practical testing**.