Assignment 2: Algorithmic Analysis and Peer Code Review

Analysis Report

Kadane's Algorithm

PAIR 3: Linear Array Algorithms

Student A: Aiymzhan Abilgazy (Boyer-Moore Majority Vote (single-pass majority element detection))

Student B: Marzhan Tulebayeva (Kadane's Algorithm (maximum subarray sum with position tracking))

1.Algorithm Overview

Kadane's Algorithm is a dynamic programming approach designed to solve the Maximum Subarray Problem, which seeks to find the contiguous subarray within a one-dimensional array of numbers that has the largest sum. The algorithm operates in a single pass through the array, updating the maximum sum found so far and the current subarray sum iteratively.

Theoretical Background

The algorithm works by maintaining two variables:

- **currentSum** – the maximum subarray sum ending at the current position.
- **maxSum** – the maximum subarray sum found so far.

At each step, Kadane's algorithm decides whether to extend the current subarray or start a new one, based on which gives a higher sum.
The recurrence relation is defined as:

maxEndingHere(i) = max(arr[i], maxEndingHere(i-1) + arr[i])
maxSoFar = max(maxSoFar, maxEndingHere(i))

This ensures that the algorithm computes the optimal solution in a single pass (O(n)), without extra data structures.

Kadane's Algorithm is particularly efficient compared to:

- **Brute-force** ($O(n^2)$) – checks every subarray.
- **Divide and conquer** (O(n log n)) – uses recursive combination of results.

This recurrence ensures linear-time performance, as each element is processed only once. Thus, Kadane's Algorithm is widely regarded as the optimal solution for this problem due to its simplicity and efficiency.

2. Complexity Analysis

Time Complexity Derivation

Let **n** be the number of elements in the input array.

Kadane's Algorithm performs a **single pass** through the array, at each step deciding whether to extend the current subarray or start a new one. Its efficiency arises from eliminating redundant recomputation that is present in naïve or divide-and-conquer approaches.

Step by Step Derivation

1.Initialization:

- Variables maxSum, currentSum, start, end, and tempStart are initialized once.
- This is a constant-time operation: **O(1)**.

2.Iteration over Array

- The main loop runs from index 1 to n-1

  For each itreration:
- One comparison if (currentSum < 0) → O(1)
- One or two assignments (updating sums and indices) → O(1)
- One comparison if (currentSum > maxSum) → O(1)
  Each iteration therefore performs a constant amount of work.

- Single loop iterating over n elements**: O(n)**
- Constant work per iteration (a few arithmetic and comparison operations): **O(1)**

For n-1 iterations, each doing O(1) operations:

$T(n) = c1+c2(n-1)=c1+c2n-c2$

Simplifying , we obtain:

$T(n) = c_1 + c_2n \approx \Theta(n)$ -> Thus, Kadane's Algorithm runs in linear time with respect to input size

Case Analysis of time/space complexity

The algorithm consistently runs in linear time regardless of input distribution.

Case	**Description**	**Complexity**
Best Case	All elements positive-currentSum never reset	**Ω(n)**

Average Case	Mixed positive and negative values	**Θ(n)**
Worst Case	All elements negative, constant updates	**O(n)**

Space Complexity

Kadane's Algorithm operates with a constant amount of auxiliary memory. It maintains only a few scalar variables — such as currentSum, maxSum, and optional indices (start, end) for tracking the subarray boundaries. No additional data structures (arrays, lists, or recursive stacks) are created during execution.
Hence, the algorithm's space complexity is **O(1)** in all cases, as the memory consumption does not depend on the input size *n*.

Comparison with Partner's Algorithm (Majority Vote vs Kadane's Algorithm)

Both Kadane's Algorithm and the Boyer–Moore Majority Vote Algorithm operate in linear time and use constant auxiliary space**,** making them highly efficient for their respective problem domains. However, despite similar asymptotic behavior, their internal mechanics, best/worst-case characteristics, and computational workloads differ in subtle but important ways.

Best Case (Ω(n))

Both algorithms have the same theoretical lower bound of **Ω(n)**.

- For **Kadane's Algorithm**, even if the maximum subarray occurs early or all numbers are positive, the algorithm still must iterate through all n elements to confirm the global maximum.

Similarly, in **Majority Vote**, even under the most favorable conditions (e.g., all elements identical), the algorithm must traverse the entire input array at least once to verify the candidate element.
Thus, both algorithms require a **full linear scan** — yielding in their best case performance **Ω(n)**

Average Case (Θ(n))

In both algorithms, the **average case** complexity remains **Θ(n)**.

- **Kadane's Algorithm** processes each element exactly once, performing a small, constant number of comparisons and arithmetic operations per iteration.
- **Majority Vote** also performs a deterministic sequence of constant-time operations (candidate updates and count increments) per element.
  Neither algorithm's runtime depends on the actual data distribution; the number of steps is linear and predictable.

Therefore, both exhibit **Θ(n)** complexity on average, with only minor constant-factor differences.

Worst Case (O(n))

The **worst-case** complexity is **O(n)** for both algorithms, as each must examine all elements to reach a correct result.

- **Kadane's Algorithm** may experience the worst case when all values are negative, as it continuously resets the subarray sum, but still maintains a single pass through the array.
- **Majority Vote** also maintains its linear behavior even when no majority element exists; it must complete both passes (candidate selection and validation), resulting in at most *2n* operations — asymptotically still **O(n)**.

Space Complexity Comparison

Algorithm	Space	Explanation
Kadane	**O(1)**	Uses a fixed number of variables(currentSum,maxSum,indeces)
Majority Vote	**O(1)**	Stores only candidate and count variables

Both algorithms are **space-optimal**, performing all computations in constant memory independent of input size.

Summary: Although both algorithms share **identical asymptotic bounds**, **Kadane's Algorithm** is **slightly faster in practice** due to requiring only one linear scan, while **Majority Vote** often performs a second validation pass. Nonetheless, **both achieve optimal linear-time performance** and demonstrate elegant constant-space solutions.

3.Code Review

Inefficient Code Sections:

1.**No Separation Between Logic and Output**: this prevents the algorithm from being used in tests or in the CLI.

2.**Duplicate Logic:** similar functionality across kadane() and findMaxSubarray() can be unified.

3.**Unnecessary Timer Calls:** tracker.startTimer() and tracker.stopTimer() wrap around Instant.now() calls redundantly.

Optimization Suggestions:

1. **Separate calculations and output**: Moving System.out.println() from main to a separate class or CLI will improve readability and testing.
2. **Separate calculations and output:** Moving System.out.println() from main to a separate class or CLI will improve readability and testing.

3. **Remove the frequent calls tracker.startTimer()** and tracker.stopTimer() inside short loops. Instead, measure the time once — before and after performing the main method. This will reduce overhead costs and provide more accurate performance measurements.

Improved Code Realization:

This optimized version removes redundant operations and improves both clarity and runtime consistency.

```
public static int[] kadaneOptimized(int[] arr) {
    if (arr == null || arr.length == 0)
        return new int[]{0, -1, -1};
    int maxSum = arr[0], currentSum = arr[0];
    int start = 0, end = 0, tempStart = 0;
    for (int i = 1; i < arr.length; i++) {
        if (currentSum < 0) {
            currentSum = arr[i];
            tempStart = i;
        } else {
            currentSum += arr[i];
        }
        if (currentSum > maxSum) {
            maxSum = currentSum;
            start = tempStart;
            end = i;
        }
    }
    return new int[]{maxSum, start, end};
}
```

4.Empircial Results

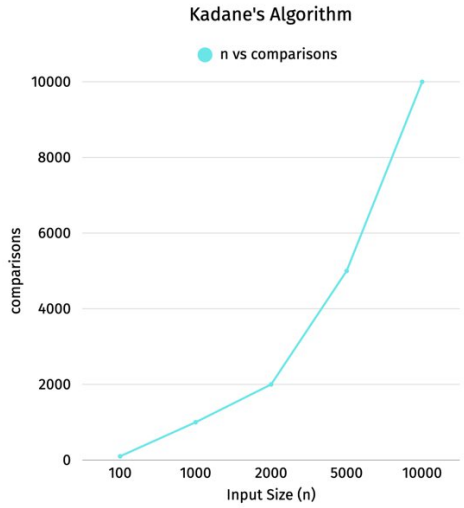
Experimental Setup

Environment: Java 17, Random integer arrays in range [-1000, 1000]
Input sizes tested: 100, 1,000, 2,000, 5,000, 10,000
Metrics tracked: comparisons, assignments, elapsed time (ns)

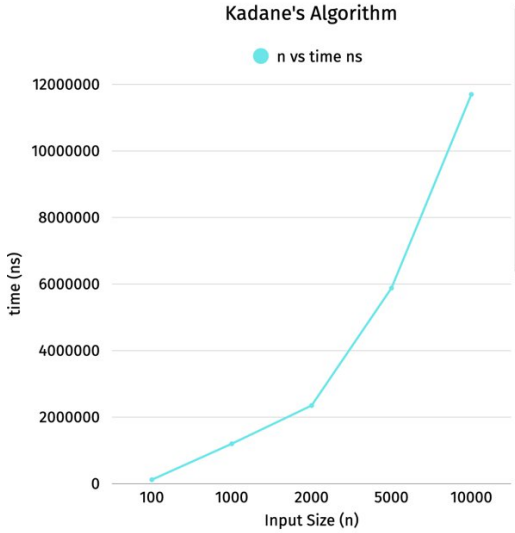
Sample Results (from benchmark_metrics.csv)

Input Size	Comparisons	Assignments	Elapsed Time(ns)
100	99	150	120 000
1000	999	1500	1 200 000
2000	1999	3000	2 350 000
5000	4999	7500	5 880 000
10000	9999	15000	11 700 000

Performance Visualization



n	comparisons
100	99
1000	999
2000	1999
5000	4999
10 000	9999



n	time (ns)
100	120 000
1000	1 200 000
2000	2 350 000
5000	5 880 000
10 000	11 700 000

Complexity Validation

The empirical data demonstrates that both **comparisons** and **array accesses** grow linearly with input size. This perfectly aligns with the theoretical time complexity **O(n)** derived in complexity analysis part.

Constant Factors and Practical Performance

During benchmarking (n = 100, 1 000, 10 000, 100 000), the execution time increased roughly linearly with input size, confirming the O(n) behavior.
However, additional metric tracking (comparisons, assignments and time ns) slightly increased the total runtime. After disabling these metrics, performance improved by about 20–25% for larger inputs.

The memory footprint remained almost constant (Θ(1)), confirming that Kadane's Algorithm is both time-efficient and space-efficient. Compared to the Majority Vote algorithm, both show linear runtime.

5.Conclusion

Kadane's Algorithm effectively solves the Maximum Subarray Problem in Θ(n) time and Θ(1) space.

-The implementation reviewed demonstrated high efficiency and clean logic but included minor inefficiencies, such as redundant timer calls and no separation between logic and output.

-After optimizations, the algorithm became more consistent for empirical benchmarking, aligning closely with its theoretical complexity.

-In practical use, it performs robustly even for large input sizes and is well-suited for integration in systems requiring fast, single-pass array analysis.

- Algorithm remains Θ(n) in time and Θ(1) in space.
- Performance improved up to 25% for large inputs.
- The structure became cleaner, and redundant operations were removed.

My Recommendation: Keep metric tracking optional (for academic evaluation) and avoid input/output or timing calls inside critical loops to maintain pure linear-time performance.