Assignment 2 – Pair 3

Student A: Arstanbek Fariza
Group: SE-2424
Algorithm implemented: Boyer–Moore Majority Vote
Algorithm analyzed: Kadane's algorithm

1. Introduction

This report analyzes my partner's implementation of Kadane's algorithm. It focuses on theoretical complexity, code quality, and empirical performance. The goal is to confirm efficiency, highlight strengths, identify possible improvements, and validate results through benchmark testing.

Kadane's algorithm offers an efficient solution to this problem by using a dynamic programming approach. Instead of checking all possible subarrays, it runs in linear time, updating the current and global maximum sums as it scans the array.

2.. Algorithm overview

Kadane's algorithm finds the contiguous subarray with the maximum sum in a one-dimensional array. The reviewed implementation provides:

- Kadane.maxSubarray(int[] a, PerformanceTracker p) — baseline result returning Result(maxSum, start, end) and updating PerformanceTracker counters (comparisons, arrayAccesses).
- Kadane.maxSubarrayOptimizedLong(int[] a) — an optimized variant using long accumulator to avoid overflow and (slightly) fewer branches; returns ResultLong.
- BenchmarkRunner that produces several datasets (random, neg-heavy) and optionally writes CSV output.
- Comprehensive JUnit tests (KadaneTest) covering canonical examples and randomized checks.

This implementation is production-quality for the assignment: correct, well-tested, instrumented and benchmarkable.

3. Implementation Details

The reviewed implementation follows the standard structure of Kadane's algorithm. The method scans the input array once and maintains two variables:

- currentSum — the best subarray sum ending at the current index.
- maxSum — the global maximum subarray sum found so far.

At each iteration, the algorithm decides whether to extend the previous subarray by adding the current element or to start a new subarray beginning at the current element. This decision is made using a Math.max comparison. By doing so, the algorithm effectively applies a dynamic programming recurrence relation:

$$currentSum(i) = \max(nums[i], currentSum(i-1) + nums[i])$$

```
if (extend < a[i]) {
        cur = a[i];
        curStart = i;
    } else {
        cur = extend;
    }
```

The maxSum is updated after each iteration:

```
maxSum = Math.max(maxSum, currentSum);
```

The implementation also performs basic input validation by throwing an IllegalArgumentException if the array is empty. This ensures that the algorithm does not attempt to access elements in an invalid input.

Overall, the code achieves linear time complexity and constant auxiliary space. Its compact design makes it efficient, but it could be improved with clearer comments, modular input validation, and extended test coverage.

4. Complexity analysis

4.1 Time complexity

For both baseline and optimized implementations:

- **Per-iteration work:** constant work (a few arithmetic ops, one comparison, occasional assignments).
- **Number of iterations:** n-1 iterations of the loop plus constant-time setup.

Therefore:

- **Θ(n)** — exact growth rate; the algorithm performs a constant number of operations per array element.
- **O(n)** — worst-case linear time.
- **Ω(n)** — best-case linear time (even if all items positive, you still need to read every element).

Conclusion: Time complexity is linear for best, average and worst cases: **Θ(n), O(n), Ω(n)**.

4.2 Space complexity

- Uses only a few scalar variables (cur, best, indices) and returns a small result object.
- **Auxiliary space:** O(1) (not counting output).
- maxSubarrayOptimizedLong also uses O(1) extra memory (uses long instead of int).

4.3 Recurrence relations

Not applicable — Kadane is iterative, not recursive.

5. Code Review

5.1 Correctness

- Implementation is correct: passes canonical tests (classic example, all-negative, single-element) and randomized consistency checks between baseline and optimized long version.
- The empty array is validated (IllegalArgumentException) — good defensive coding.

5.2 Readability & structure

- Code is concise and clear. Naming is good (maxSubarray, Result, ResultLong).
- Result and ResultLong encapsulate outputs and provide utility (toIntResultSaturated) — nice design.
- BenchmarkRunner is feature-rich (dataset kinds, deterministic RNG seed, CSV support).

5.3 Tests

- Very good test coverage: deterministic examples, edge cases, randomized validation of optimized variant, a performance sanity check (optimizedIsNotSlowerThan2xOnAverage).
- Suggest adding a property-based test or fuzzing for more confidence on extreme inputs (very large arrays, extremes close to Integer.MAX_VALUE).

6. Empirical Results

Methodology

Benchmarks were executed using BenchmarkRunner for input sizes:
n = 100, 1,000, 10,000, 100,000.
Datasets: random, neg-heavy.
Metrics collected: time (ms), comparisons, array accesses.

Results

n	dataset	time(ms)	comparisons	arrayAccesses
100	random	1.196	99	100
1000	random	0.049	999	1000
10000	random	0.520	9999	10000
100000	random	38.6	99999	100000

7. Conclusion

Kadane’s algorithm implementation is correct, efficient, and benchmark-ready. It achieves linear time and constant space complexity, confirmed both theoretically and empirically. Suggested improvements are micro-level: encapsulating performance tracker counters, caching array elements, and adding property-based tests. With these refinements and empirical validation, the implementation fully satisfies assignment requirements.