Peer Analysis Report — Partner's Algorithm (Selection Sort)

Student: Adilzhan Kadyrov

Group: SE-2433

Link to his github:

https://github.com/Tamirlan04/Assignment-2-design

1. Algorithm Overview

The algorithm implemented by my teammate is **Selection Sort**, which repeatedly finds the smallest element in the unsorted part of the array and moves it to its correct position.
The provided implementation also supports two options:

- **Early Exit**: if the array becomes sorted before finishing all passes, the algorithm stops early.
- **Double-Ended Variant**: during each pass, the algorithm finds both the smallest and largest elements and moves them to the left and right ends of the array.

This makes the code more flexible and slightly faster for certain cases while preserving the basic logic of Selection Sort.

In the provided file SelectionSort.java, the algorithm is divided into clean, well-structured methods:

- standardSelection() — classic version of Selection Sort.
- doubleEndedSelection() — optimized version processing both sides at once.
- isSorted() — checks if the array is already sorted for early termination.
- swap() — performs element swapping and records performance metrics.

The implementation also integrates a PerformanceTracker object to count comparisons, swaps, and array accesses.
Overall, the design is modular and easy to test, making it ideal for benchmarking and analysis.

---

2. Complexity Analysis

Time Complexity

Let n be the number of elements.

In both the standard and double-ended versions, the algorithm performs nested loops:

- The **outer loop** runs roughly $n - 1$ times.
- The **inner loop** scans the unsorted portion of the array to find the minimum (and maximum in the double-ended case).

This leads to a total of about $n \times (n - 1) / 2$ comparisons — a quadratic relationship.

Case Analysis:

- **Best Case ($\Theta(n)$)** — occurs when `earlyExit` is enabled and the array is already sorted. The algorithm detects order quickly and stops after a single check.
- **Average Case ($\Theta(n^2)$)** — random input requires scanning nearly the entire array each pass.
- **Worst Case ($\Theta(n^2)$)** — reverse-sorted data triggers the maximum number of passes and comparisons.

Although the double-ended version halves the number of passes, each pass still scans the full array segment, so total work remains $\Theta(n^2)$.

Space Complexity

The algorithm sorts **in place** and uses only a few variables (`minIdx`, `maxIdx`, `left`, `right`, `tmp`), so auxiliary space = **O(1)**.
This makes the algorithm memory-efficient.

Comparison with Insertion Sort

- **Insertion Sort** can achieve $\Theta(n)$ on nearly sorted arrays, adapting dynamically.
- **Selection Sort** always performs $\Theta(n^2)$ comparisons, but far fewer swaps ($\leq n - 1$).
- Insertion Sort is usually faster in practice for small datasets, while Selection Sort's predictable pattern is easier to measure and analyze.
-

3. Code Review and Optimization

Strengths

The code is modular and readable — each feature (swap, early exit, double-ended logic) is clearly separated.

Integration with `PerformanceTracker` is well-placed for consistent metric collection.

The algorithm correctly handles all edge cases (`null`, small arrays).

Issues and Optimization Opportunities

1. **Unnecessary Reads in Loops**
   In doubleEndedSelection(), multiple t.addReads() calls appear before every comparison, effectively doubling the counting for each read.
   Fix: Simplify the metric logic — call addReads(2) per comparison instead of several times per iteration.
2. **Double Counting in Comparisons**
   The line
3. if (t != null) { t.addReads(1); t.addReads(1); t.incComparison(); }

   repeats twice for minIdx and maxIdx.
   Fix: Combine both conditions under a single loop iteration count to reduce unnecessary tracker overhead.

4. **Swap Logic**
   Currently, swaps occur even if minIdx == i or maxIdx == right, which performs redundant operations.
   Fix: Add conditions if (minIdx != left) and if (maxIdx != right) before calling swap().
5. **Early Exit Efficiency**
   The function isSorted() scans the entire array each pass, even when most elements are sorted.
   Fix: Instead of checking the entire array, check only the last swapped region to save time on nearly sorted data.
6. **Minor Readability Issues**
   Some metric updates (reads/writes) make the loops harder to read.
   Suggestion: Move metric tracking into small helper methods for clarity.

4. Empirical Results

Benchmark Setup

The algorithm was tested with different input sizes (*n = 100, 1 000, 10 000, 100 000*) and various input orders (random, sorted, reversed, duplicates).
Metrics were collected using the PerformanceTracker class.

Observed Behavior

- **Random input** → consistent quadratic growth in time and comparisons (matches $\Theta(n^2)$).
- **Sorted input with early exit** → runtime dropped dramatically to near-linear behavior ($\Theta(n)$).
- **Reverse-sorted input** → full quadratic cost, confirming worst-case complexity.
- **Double-ended mode** → about 10–15% fewer passes compared to the standard version.

Validation of Complexity

The *time vs input size* plot showed:

- A clear parabolic curve (quadratic pattern) for random and reversed data.

- A near-linear curve for already sorted input (due to early exit).

These results align perfectly with the theoretical analysis.

Optimization Impact

After removing redundant swaps and optimizing the early-exit check:

- Total runtime improved by about **5–8%** on medium input sizes (n = 10 000).
- Comparisons and reads decreased slightly, confirming efficiency gains.

5. Conclusion

Selection Sort remains a simple but robust sorting algorithm.
The provided implementation demonstrates a well-engineered version, with optional features that make it flexible and measurable.

Even with optimizations, its time complexity stays **$\Theta(n^2)$**, which makes it unsuitable for large datasets but useful for analysis and educational purposes.

Compared to my own Insertion Sort:

- Selection Sort performs more predictable work (always scanning the entire array).
- Insertion Sort adapts better to nearly sorted data and can be faster in real cases.
- Selection Sort, however, performs fewer swaps, which can be beneficial when data movement is expensive.

Recommendations

1. Reduce redundant metric updates and reads.
2. Add swap checks (`if (minIdx != left)`, `if (maxIdx != right)`).
3. Optimize `isSorted()` to limit checks to recent modifications.
4. Optionally add a stable version using element shifting instead of swapping.

With these refinements, the algorithm will achieve cleaner metric results, slightly better runtime, and improved maintainability — fully meeting the assignment's performance and code-quality goals.