Individual Analysis Report: Selection Sort Algorithm

Project Author: Rafael Shayekhov

Reviewer: Aldiyar Zhangabyl

1. Algorithm Overview

Theoretical Background

Selection Sort is a simple comparison-based sorting algorithm. It divides the array into two parts: the sorted and unsorted segments. During each iteration, it finds the smallest element in the unsorted part and places it at the beginning.

Main Characteristics:

- In-place algorithm using O(1) additional memory.
- Unstable: does not preserve the order of equal elements.
- Has $O(n^2)$ time complexity in all cases.
- Performs exactly n − 1 swaps in the worst case.

Algorithm Mechanics:

1. The algorithm iterates through the array.
2. For each position *i*, it searches for the minimum element in the remaining unsorted part.
3. It swaps the found element with the element at position *i*.

This process continues until the entire array is sorted.

2. Complexity Analysis

Best Case ($\Theta(n^2)$)

Even for an already sorted array, Selection Sort still checks every element to find the minimum.

- Comparisons: $n(n-1)/2$
- Swaps: $n-1$

Worst Case ($\Theta(n^2)$)

A reversed array leads to the same number of comparisons as the best case.

- Comparisons: $n(n-1)/2$
- Swaps: $n-1$

Average Case ($\Theta(n^2)$)

For random input, the average number of comparisons and swaps remains quadratic.

Space Complexity:

- Auxiliary memory: **O(1)**
- No recursive calls or extra arrays required.

Mathematical Summary:

- **$O(n^2)$** — upper bound
- **$\Omega(n^2)$** — lower bound
- **$\Theta(n^2)$** — tight bound

Comparison with Insertion Sort:

Metric	Selection Sort	Insertion Sort
Best Case	$\Theta(n^2)$	$\Theta(n)$
Worst Case	$\Theta(n^2)$	$\Theta(n^2)$
Average Case	$\Theta(n^2)$	$\Theta(n^2)$
Swaps	$\Theta(n)$	$\Theta(n^2)$
Adaptive	No	Yes
Stable	No	Yes

3. Code Review

General Code Quality:

The implementation of Selection Sort is clean, readable, and logically organized. Each class performs a distinct role, following good object-oriented design principles.

Strengths:

- Clear structure and good formatting.
- Proper encapsulation of logic and variables.
- Full tracking of performance metrics (time, comparisons, swaps).
- Effective error handling and input validation.
- Comprehensive test coverage, including boundary cases.
- Well-documented with JavaDoc comments.

SelectionSort.java:

- Logical and simple structure — each method performs one task.
- Proper use of access modifiers for encapsulation.
- Clear variable naming (minIndex, currentMin).
- Correct handling of edge cases (empty and single-element arrays).
- Efficient use of System.arraycopy for internal operations.

PerformanceTracker.java:

- Tracks comparisons, swaps, accesses, and runtime accurately.
- Simple and clear interface (startTimer(), stopTimer(), reset()).
- Exports benchmark data to CSV for analysis.

Testing:

- Covers all possible input types (sorted, reversed, random).
- Includes verification using Arrays.sort().
- Measures both correctness and performance metrics.

4. Empirical Results

Observed Behavior:

- Time increases quadratically with input size — matches theoretical $\Theta(n^2)$.
- Performance remains consistent across data types.
- Swap count is minimal, confirming efficiency in swap operations.

Benchmark Observations:

1. Consistent time for all input distributions.
2. Fewer swaps than Insertion Sort for large datasets.
3. Good cache performance due to sequential array access.

Comparison Summary:

- **Selection Sort:** Consistent $\Theta(n^2)$ performance.
- **Insertion Sort:** Performance depends on data order ($\Theta(n)$–$\Theta(n^2)$).

5. Conclusion

The analyzed Selection Sort implementation correctly follows theoretical principles and demonstrates solid coding practices. It is clean, efficient, and well-tested.

While its $\Theta(n^2)$ time complexity limits scalability, the algorithm performs predictably and efficiently in terms of swaps and memory.

Optimization Suggestions:

- Add bidirectional selection to reduce comparisons.
- Implement early termination for already sorted arrays.
- Use cached values to reduce constant factors.
- Introduce configurable parameters for benchmarks.

Final Assessment:

The project shows strong technical understanding, excellent structure, and clean code. Selection Sort remains a valuable reference algorithm for learning sorting principles and performance measurement.