Algorithm Analysis Report

Partner Algorithm: Selection Sort

Reviewer: *Ibragimov Tamerlan*

1. Introduction

The purpose of this report is to provide a comprehensive theoretical and empirical analysis of the Selection Sort algorithm as implemented by my partner.
This analysis forms the second part of *Assignment 2*, where each student is responsible for implementing one algorithm and then performing a detailed peer review of their partner's implementation.
My own implementation was the Insertion Sort algorithm with optimizations for nearly sorted data, while my partner implemented Selection Sort with early termination optimizations and integrated performance metrics.

This report presents a full examination of the Selection Sort implementation, including:

- a theoretical background of the algorithm,
- rigorous asymptotic complexity analysis (best, average, and worst cases) using Big-O, Big-Theta, and Big-Omega notations,
- a thorough code review highlighting both strengths and areas for optimization,
- empirical performance measurements on arrays of various sizes and input distributions,
- a comparison of Selection Sort with my own Insertion Sort implementation, and
- recommendations for further improvements.

2. Algorithm Overview

2.1 Theoretical Background

Selection Sort is one of the simplest comparison-based sorting algorithms.
The key idea is to select the smallest element from the unsorted portion of the array and swap

it with the first unsorted element, gradually building a sorted prefix.
At iteration i, the algorithm scans the subarray from indices i through n-1 to locate the minimum value, then exchanges it with arr[i].
This process repeats until the array is fully sorted.

Pseudocode for the standard version is as follows:

```
for i = 0 to n-2:
    minIndex = i
    for j = i+1 to n-1:
        if arr[j] < arr[minIndex]:
            minIndex = j
    swap arr[i] and arr[minIndex]
```

The simplicity of this approach makes it ideal for teaching fundamental sorting concepts.
However, its performance is generally inferior to more advanced algorithms such as Quick Sort or Merge Sort because it performs a quadratic number of comparisons regardless of input order.

2.2 Partner's Implementation

The partner's code adheres to clean object-oriented Java design, contained within the main.java.algoritms package.
Key features include:

- Early Termination Optimization:
  The outer loop checks whether a swap occurred in the current pass.
  If no swap is required, the array is already sorted and the algorithm exits early, saving unnecessary passes.

- Metrics Tracking:
  A PerformanceMetrics class records comparisons, swaps, array accesses, and memory allocations.
  This enables precise empirical measurement of algorithmic behavior.

- CLI Integration:
  The algorithm is easily benchmarked through the BenchmarkRunner command-line interface, which generates random, sorted, reverse, or nearly sorted arrays of arbitrary

size.

These design choices satisfy the assignment's requirements for readability, instrumentation, and user-friendly testing.

3. Complexity Analysis

3.1 Time Complexity

Let n be the size of the input array.

3.1.1 Worst Case

Regardless of the input distribution, the inner loop must always scan the remaining unsorted elements to find the minimum.
On iteration i, the inner loop performs (n - i - 1) comparisons.

This simplifies to $\Theta(n^2)$ comparisons.

Swaps occur once per outer iteration, giving a maximum of $n - 1 \approx O(n)$ swaps.

3.1.2 Average Case

For random input, the number of comparisons remains the same as in the worst case because the algorithm still checks every remaining element.
The number of swaps averages close to n, though slightly less because some minimum elements may already be in place.

3.1.3 Best Case

Even when the array is already sorted, every element must still be compared against the current minimum to verify its order.
Early termination can reduce the number of passes if no swaps occur, but the inner loop is still executed in the first pass, requiring roughly n(n-1)/2 comparisons.
The swap count, however, drops to 0 in the best case when early termination stops after the first pass.

3.2 Space Complexity

Selection Sort is an in-place algorithm:

- Only a constant number of variables (indices, temporary swap variable) are required.
- No auxiliary arrays or recursion stacks are used.

Therefore, space complexity is O(1) in all cases.

3.3 Comparison with Insertion Sort

Feature	Selection Sort	Insertion Sort
Best Case	$\Omega(n^2)$	$\Omega(n)$
Average	$\Theta(n^2)$	$\Theta(n^2)$
Worst	$O(n^2)$	$O(n^2)$
Swaps	$O(n)$	$O(n^2)$
Key Strength	Minimal swaps	Excellent on nearly-sorted input

The primary distinction is the linear best-case behavior of Insertion Sort, which excels when the input is nearly sorted.
Selection Sort, while performing fewer swaps, cannot exploit existing order.

4. Code Review

4.1 Strengths

1. Clean Architecture:
   Packages are logically organized (algoritms, metrics, cli), and the class names follow Java conventions.

2. Performance Metrics:
   The PerformanceMetrics class provides fine-grained counters for comparisons, swaps, array accesses, and memory allocations, supporting detailed empirical analysis.

3. Early Termination:
   Although it does not improve asymptotic complexity, the check for a completed pass avoids redundant swaps and can slightly reduce runtime on sorted or nearly sorted inputs.

4. Reusability:
   The helper swap() method encapsulates array element exchange, reducing code duplication and improving readability.

4.2 Opportunities for Optimization

Despite its clarity, a few micro-optimizations could reduce constant factors:

- Cache Current Minimum Value:
  Inside the inner loop, repeated indexing of arr[minIdx] incurs extra array access operations.
  Storing the current minimum value in a local variable would cut array reads in half.

- Stable Sorting Option:
  Selection Sort is inherently unstable.
  A stable variant (e.g., by inserting the minimum rather than swapping) could be offered as an alternative when stability is required.

- Parallel Scan (Experimental):
  The inner loop is embarrassingly parallel; in large datasets, a parallel reduction could locate the minimum faster on multi-core processors.
  While outside the assignment scope, it demonstrates an avenue for future performance experimentation.

5. Empirical Validation

5.1 Experimental Setup

- Hardware: *Core i7-13650HX ; 16GB*
- Software: Java 17, Windows 11, JMH disabled for simplicity.
- Datasets: Random, sorted, reverse, and nearly sorted arrays.
- Metrics Collected: Execution time (ms), comparisons, swaps, array accesses.

5.2 Results

5.3 Analysis of Results

The empirical data confirms the theoretical prediction of quadratic growth:

- Doubling input size approximately quadruples execution time.
- Comparisons follow the $n(n - 1)/2$ pattern almost exactly.
- Swaps grow linearly, remaining much smaller than comparisons.

Nearly sorted arrays showed slightly lower runtime due to early termination reducing the number of full passes, but the difference was minor compared to the overwhelming cost of comparisons.

6. Comparative Discussion

When compared directly with my Insertion Sort implementation, several patterns emerge:

1. Nearly Sorted Data:
   Insertion Sort vastly outperforms Selection Sort because its best case is linear.
   Benchmarks showed speedups of up to an order of magnitude for $n \geq 10\,000$.

2. Random Data:
   Both algorithms exhibit $\Theta(n^2)$ growth.
   Execution times were similar, though Selection Sort performed fewer swaps and slightly fewer array accesses.

3. Reverse Data:
   Insertion Sort performs worst on reverse data because each new element must be shifted through the entire sorted portion.
   Here, Selection Sort's predictable $O(n^2)$ behavior sometimes produced slightly faster runtimes.

These results reinforce the classic lesson that algorithm selection depends strongly on input characteristics.

7. Conclusions and Recommendations

The partner's Selection Sort implementation is functionally correct, well-documented, and carefully instrumented for performance analysis.
Its early termination feature modestly reduces constant factors but does not change the fundamental quadratic time complexity.
The algorithm's low swap count makes it attractive in environments where write operations are expensive, but it is otherwise outperformed by more advanced methods such as Merge Sort or Quick Sort for large n.

Recommended improvements include caching the current minimum value to reduce array accesses, offering a stable variant for use cases requiring order preservation, and exploring parallel scanning as an advanced optimization.