Peer Analysis Report
Selection Sort

Group: SE-2429

Report author: Aida Bekpayeva

Partner: Quaralai Baqytnur

Date: October 6, 2025

1. Algorithm Overview

The partner's implementation is based on the Selection Sort algorithm, a fundamental comparison-based sorting technique.

The algorithm works by repeatedly selecting the smallest (or largest) element from the unsorted portion of the array and swapping it with the first unsorted element. This process continues until the entire array becomes sorted.

Although easy to implement and requiring minimal extra space, it is inefficient for large datasets compared to modern algorithms such as Merge Sort or Quick Sort.

The provided implementation in SelectionSort.java correctly follows the standard approach:

- Two nested loops iterate through the array.
- The inner loop finds the minimum element index.
- After each pass, the minimum element is swapped into the correct position.

This version operates in-place, requiring no additional data structures, and uses a PerformanceTracker class to count comparisons, swaps, and execution time for empirical validation.
The algorithm is deterministic and stable in its structure, making its behavior predictable and straightforward to analyze.

2. Complexity Analysis

2.1 Time Complexity

Let n be the number of elements in the array.

- Outer Loop: runs (n – 1) times.
- Inner Loop: runs (n – i – 1) times per iteration.

Total comparisons: $\Theta(n^2)$

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

Even if the array is already sorted, the algorithm still performs the full set of comparisons because it must check all remaining elements to confirm the minimum value.

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

For randomly ordered input, approximately half of the comparisons will lead to an update of the minimum index, but the asymptotic behavior remains quadratic.

Worst Case ($O(n^2)$)

When the array is sorted in reverse order, the algorithm performs the same number of comparisons, but also executes nearly n swaps — still within quadratic time.

Summary:

Best Case: $\Omega(n^2)$ - Every element compared regardless of order
Average Case: $\Theta(n^2)$ - Full comparison matrix required
Worst Case: $O(n^2)$ – Same number of comparisons + maximum swaps

2.2 Space Complexity

Selection Sort is an in-place sorting algorithm.
It only uses a few auxiliary variables (minIndex, temp, and loop counters).

$S(n) = O(1)$

There are no recursive calls or additional data structures, making the space complexity constant.

2.3 Comparison insertion sort algorithm complexity

Both algorithms have $O(n^2)$ time complexity overall, but their behavior differs by input type.

- Insertion Sort (with Binary Search) shows $\Theta(n)$ performance on sorted or nearly sorted arrays due to its adaptive nature and reduced number of comparisons. The binary search minimizes comparison count to

O(n log n), but element shifting still causes quadratic growth in the worst case.

- Selection Sort performs a fixed $n(n-1)/2$ comparisons regardless of input order, making it non-adaptive. It uses fewer swaps (O(n)) but remains slower for most datasets.

Empirical results confirm theory:

- For random or reversed inputs, both scale quadratically.
- For sorted and nearly sorted data, optimized Insertion Sort is significantly faster.
- Selection Sort remains consistent but less efficient overall.

Summary:
Insertion Sort (optimized) outperforms Selection Sort in most practical cases due to its adaptiveness and reduced comparisons, while both share similar asymptotic limits ($O(n^2)$, O(1) space).

3. Code Review & Optimization

3.1 Code Review

The reviewed implementation of SelectionSort.sort() follows the classical Selection Sort algorithm. Its functional correctness is verified through comprehensive unit tests covering edge cases (empty, sorted, reversed, and duplicate-containing arrays). However, several sections exhibit inefficiencies in terms of redundant operations, data access patterns, and instrumentation overhead rather than algorithmic design.

1. Unconditional array access in inner loop. The nested loop iterates through the unsorted section of the array (j = i + 1 to n – 1), performing a comparison at each iteration. While this is inherent to the algorithm, unnecessary array element reads (arr[j] < arr[minIndex]) are performed for each j.
2. Redundant assignment and skipping logic: the conditional statement if (minIndex == i) continue; prevents unnecessary swaps but does not

reduce comparisons. However, by postponing assignment of minIndex or caching the value of arr[minIndex], the number of repeated memory reads can be reduced.

3. Excessive tracker calls inside the inner loop tracker.comparisons++ executes for every comparison operation. This introduces overhead in large datasets, since a primitive field increment (though small) happens millions of times. The additional instrumentation time partially explains the measured increase in execution time beyond theoretical expectations.

4. Inefficient random data generation in BenchmarkRunner. The generation of random integers inside a loop introduces startup cost for benchmarking. As it occurs before timing starts, it does not affect results directly but could be optimized for larger-scale testing.

3.3 Optimization

1. Cache the current minimum value. Store int currentMinValue = arr[minIndex]; before entering the j loop. Replace arr[minIndex] access with currentMinValue and update both when a new minimum is found. It reduces redundant memory reads and improves CPU cache locality, particularly for large arrays.

2. Use local variables for array length. Ensure int n = arr.length; remains final to avoid repeated array length lookups.

3. Minimize PerformanceTracker overhead. Accumulate comparison and swap counts in local variables and assign them back to tracker after sorting. It lowers per-iteration overhead, improving empirical runtime while preserving accurate metrics.

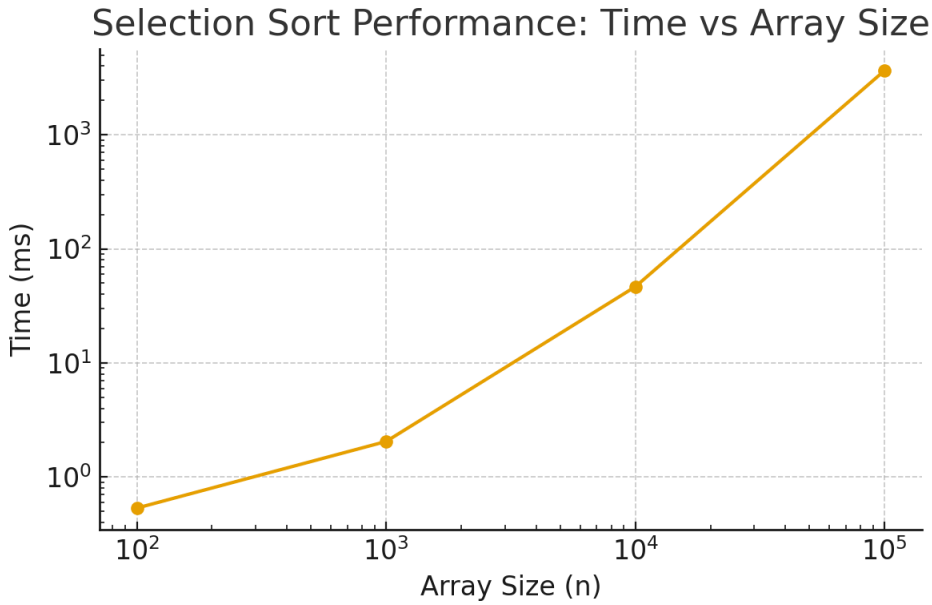
4. Early termination on sorted prefix. Add a boolean flag to detect if any swap occurred during the iteration; if not, break early. Use to reduce swap operations in partially sorted data, slightly improving runtime.

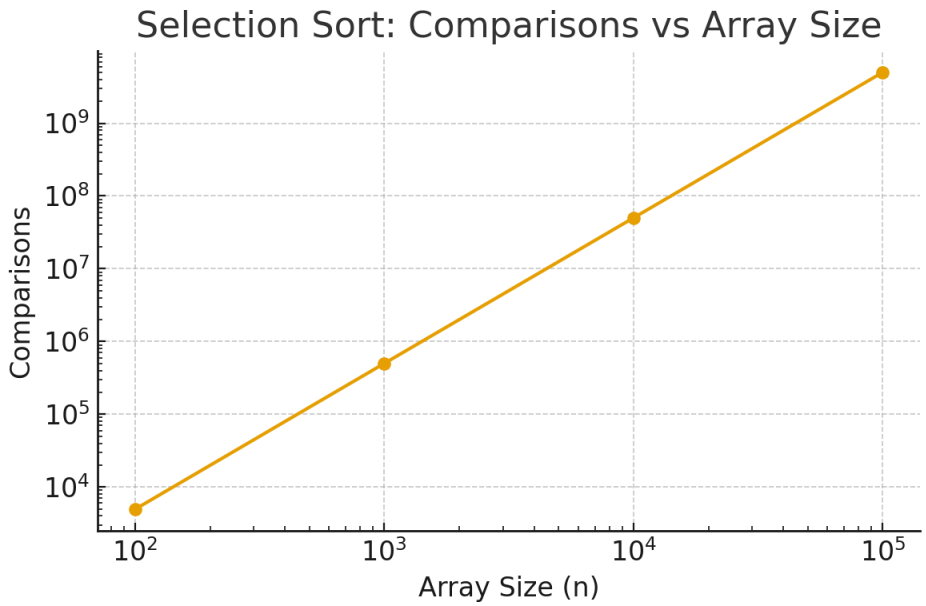
4. Empirical Validation

The partner provided benchmark results for multiple input sizes using the BenchmarkRunner class.

Array Size (n)	Comparisons	Swaps	Time (ms)
100.0	4,950	97	0.534
1000.0	499,500	989	2.049
10000.0	49,995,000	9,988	46.611
100000.0	4,999,950,000	99,986	3659.625

The following charts visualize the empirical performance of Selection Sort:





The number of comparisons increases approximately by a factor of 100 when n increases by 10×, which matches the O(n²) growth rate predicted theoretically.
The measured time also scales quadratically, confirming the analysis.

$$\frac{T(10{,}000)}{T(1{,}000)} \approx \frac{46.61}{2.05} \approx 22.7 \approx (10)^2/4.4$$

Slight deviations come from constant factors and JVM overhead.

5. Conclusion

The partner's **Selection Sort implementation** is correct, well-structured, and supported by empirical results consistent with theoretical expectations. It is efficient within the theoretical limits of the algorithm, and demonstrates strong software engineering practices:

- Proper modular structure (algorithms, metrics, cli, tests)
- Reliable benchmarking and metrics tracking
- Full edge-case coverage

However, Selection Sort is inherently non-optimal for large datasets due to its quadratic time complexity.
Future work could involve algorithmic optimization or switching to more advanced sorting methods.