Assignment 2 — Selection Sort Analysis Report

Student: Yernur Syrlibayev
Group: SE - 2439
Date: October 2025

Algorithm Overview

In this project I implemented the **Selection Sort** algorithm using Java.

Selection Sort works by:

1. Finding the smallest element in the array.
2. Swapping it with the first element.
3. Repeating the process for the remaining subarray.

This continues until the whole array is sorted.

The algorithm is **in-place** (requires only O(1) extra memory) and easy to understand, but not efficient for large data sets because it requires many comparisons.

I also added an **early termination optimization**: if during a pass no smaller element is found, the algorithm stops early, because the array is already sorted.

What I Did

I created a **Maven project** with the following structure:

1. **SelectionSort.java** — the algorithm itself.
2. **PerformanceTracker.java** — tracks comparisons, swaps, and time.
3. **BenchmarkRunner.java** — runs the algorithm on different input sizes and saves results into a CSV file.

I also added **unit tests** to check correctness for:

- Empty arrays
- Single elements
- Duplicates
- Sorted arrays
- Reversed arrays

- Random arrays

Finally, I ran benchmarks with arrays of **100, 1,000, 10,000, and 100,000 elements**.
Each test used four types of data: **random, sorted, reversed, nearly sorted**.
The program saved all results into one CSV file:
docs/performance-plots/selectionsort_results.csv.

What I Saw

- The algorithm worked correctly for all test cases.
- **Random arrays**: number of comparisons grew close to $n^2/2$, swaps remained much fewer.
- **Sorted arrays**: thanks to optimization, only *(n-1)* comparisons and *0 swaps*.
- **Reversed arrays**: many swaps, but due to optimization, comparisons were slightly fewer than the theoretical maximum.
- **Nearly sorted arrays**: performance was very close to the best case because of early termination.

Overall, **runtime increased quadratically** with input size, exactly as expected for $O(n^2)$.

What I Understood

From this assignment, I understood:

- How **Selection Sort** works step by step — repeatedly finding the minimum and moving it into position.
- How to connect **theoretical complexity** ($O(n^2)$) with **real performance measurements** (time, comparisons, swaps).
- The effect of **optimizations**: even a small early stop condition changes performance dramatically on sorted or nearly sorted inputs.
- The importance of **modular code**: separating algorithm, performance tracking, and benchmarking into different classes makes testing and extending easier.

Conclusion

Selection Sort is a **simple, easy-to-implement algorithm** that works correctly in all scenarios, but it is not efficient for large datasets.

- **Time Complexity**:
  - Best Case (sorted): **O(n)** with early termination

    - Average Case: **$O(n^2)$**
    - Worst Case: **$O(n^2)$**
- **Space Complexity**: **$O(1)$** (in-place)

This project helped me clearly see the difference between **theory and practice**.
It also showed how **small optimizations** can significantly improve real-world performance.