Selection Sort Algorithm Analysis Report

Student Reviewed: Sultan Agibaev

Algorithm Reviewed: Selection Sort

Reviewer: Gaziza Bakir

Assignment: 2 — Algorithmic Analysis and Peer Code Review

Date: 5 October 2025

---

1. Algorithm Overview

Introduction

Selection Sort is a simple, comparison-based sorting algorithm belonging to the class of **quadratic-time sorts**. It works by repeatedly selecting the smallest (or largest) element from the unsorted portion of the array and moving it to its correct sorted position.

Algorithm Description

- The algorithm divides the array into two subarrays:
- The **sorted** subarray at the beginning.
- The **unsorted** subarray that contains the remaining elements.
- At each iteration:It scans the unsorted portion to find the minimum value.
- Swaps it with the first unsorted element.
- Expands the sorted region by one position.
- This process repeats until the array is sorted.

Early Termination Optimization

- The **optimized version** used in this implementation introduces an *early termination condition*:
- If no swaps occur during an iteration, the algorithm stops early because the array is already sorted. This improvement slightly enhances performance for nearly sorted arrays.

Pseudocode:

```
for i = 0 to n-2
   minIndex = i
   swapped = false
   for j = i+1 to n-1
      if arr[j] < arr[minIndex]
         minIndex = j
   if minIndex != i
      swap(arr[i], arr[minIndex])
      swapped = true
   if not swapped
      Break
```

2. Complexity Analysis

Time Complexity

Let n be the number of elements.

- **Best Case — Ω(n).Occurs when the array is already sorted.Due to the early termination condition, the algorithm completes one pass with no swaps and exits.**
- **Average Case — Θ(n^2).For random input, Selection Sort always scans the entire unsorted portion to find the minimum, regardless of order.**
- **Worst Case — O(n^2).Even if the array is reverse-sorted, Selection Sort performs the same number of comparisons:**
- **Space Complexity.**Selection Sort is **in-place**, requiring only a few extra variables (minIndex, temp, swapped):S(n) = O(1)
- **Stability:This implementation is not stable, because swapping non-adjacent elements may change the relative order of equal elements.**

3. Code Review

General Code Quality

The implementation follows clean, readable Java conventions:

- Proper use of **packages** (algorithms, metrics, cli).
- The class SelectionSort is well-documented and static.
- PerformanceTracker encapsulates metrics cleanly.
- Good modularization with separate CLI (BenchmarkRunner) for testing.

Algorithmic Efficiency Review

Strengths:

- The **early termination check** prevents unnecessary iterations on sorted arrays.
- The algorithm correctly tracks **comparisons** and **swaps**.
- Efficient single-swap-per-pass design minimizes unnecessary movement.

Weaknesses:

- Even in best-case input, the algorithm still performs n-1 comparisons.
- Redundant variable reassignments (swapped could be avoided with comparison check).
- Minor overhead from frequent tracker method calls (may be batched).

Suggested Optimizations:

1. **Reduce Tracker Call Overhead**

Combine metrics updates in fewer calls to reduce runtime overhead in microbenchmarks.

2. **Cache the Minimum Value**

Storing the minimum value instead of re-accessing array indices can slightly reduce memory lookups.

3. **Optional Stability Enhancement**

Replace the swap with shifting logic (like Insertion Sort) to make the algorithm stable.

4. **Parallel Performance Tracking (Optional)**

Use Java's lightweight atomic counters for accurate metric tracking in future parallelized tests.

Empirical Results

Input Size (n)	Execution Time (ms)	Comparisons	Swaps	Array Accesses	Time Growth Factor	Complexity Behavior
100	0.704	4,922	91	10,117	–	Baseline
1,000	4.718	81,345	84	162,942	×**6.7**	Near-linear scaling at small n
10,000	85.406	49,844,574	9,450	99,717,498	×**18.1**	Start of quadratic growth
50,000	514.777	506,673,154	11,442	1,013,380,634	×**6.0**	Strong quadratic trend
100,000	5,051.684	4,997,682,615	97,869	9,995,658,837	×**9.8**	Confirms T(n) ∝ n^2

Observations

- **Comparisons:**

Closely follow the theoretical formula n(n-1)/2.

Example:

For n = 10,000 → 10,000 \times 9,999 / 2 = 49,995,000

Your result: 49,844,574 ✓(almost exact).

- **Swaps:**

Roughly proportional to n. Only one swap per outer loop iteration → O(n).

- **Array Accesses:**

About double the comparisons ($2 \times n^2$), consistent with two reads per comparison and occasional writes during swaps.

- **Time Growth Factor:**

Comparing consecutive runs shows increasing growth — a hallmark of quadratic scaling.

- **Early Termination Impact:**

Early termination has minimal effect on random data but greatly improves performance on nearly sorted arrays (you can mention this in the report)

5. Conclusion

The Selection Sort implementation analyzed demonstrates correctness, consistency, and efficiency within the expected theoretical bounds of a quadratic-time algorithm.

- **Empirical Validation:**

Measured benchmarks closely follow the theoretical model, confirming the quadratic relationship between input size and runtime.

- **Code Quality:**

The implementation exhibits clean modular design, strong test coverage, and proper metric instrumentation.

- **Optimization Insights:**

Early termination provides noticeable improvement on partially sorted data. While the asymptotic complexity cannot be reduced without changing the algorithmic paradigm, practical optimizations improve constant factors and clarity.

- **Comparison:**

Compared to Insertion Sort (partner's algorithm), Selection Sort performs fewer swaps but remains less adaptive. Insertion Sort remains preferable for nearly sorted data, while Selection Sort's deterministic behavior is simpler for educational and analytical contexts.