Peer Analysis Report

Analyzed Algorithm: Selection Sort (with Early Termination Optimizations)
Author: Satbek Abyz
Reviewer: Adilbekov Daniyal

Algorithm Overview

Selection Sort is a classic comparison-based sorting algorithm that conceptually divides an array into two parts — the **sorted** section on the left and the **unsorted** section on the right. In each pass, it finds the minimum element in the unsorted part and swaps it with the first unsorted element. This process continues until the entire array becomes sorted.

Satbek Abyz's version of Selection Sort includes additional **early-termination optimizations** to make the algorithm more efficient on sorted and nearly sorted inputs. The algorithm uses two main ideas:

1. **Early-stop condition:**
   If during the current pass no inversion is detected (meaning the suffix is already sorted) and the smallest element remains in its position, the algorithm terminates early because the entire array is sorted.
2. **Skip-swap optimization:**
   If the smallest element in the current iteration is already at index i, the algorithm skips the swap to reduce unnecessary memory writes.

While these optimizations do not change the theoretical time complexity, they significantly improve real-world runtime for sorted or nearly sorted arrays.

Selection Sort has predictable behavior — it always performs approximately n^2/2 comparisons regardless of data distribution, which makes it easy to analyze and benchmark. It is also **in-place**, requiring only constant auxiliary space, but it is **unstable** — identical elements may change their relative order.

Complexity analysis

Let n be the number of elements in the input array.

The algorithm repeatedly scans the unsorted portion of the array to find the minimum element. During each pass i, it performs $n-i-1$ comparisons. The total number of comparisons over all passes is: $\Theta(n^2)$

Thus, the overall complexity remains **quadratic**.

- **Best Case (Already Sorted Array):**
  When the array is already sorted, the optimization detects that no inversion exists and terminates early. The number of comparisons is still proportional to n^2, but due to early stopping, the effective constant is smaller. →**T**best(n)= $\Omega(n^2)$
- **Average Case (Random Data):**
  The algorithm must always search for the minimum across the remaining elements, so the average runtime remains: →**T**avg(n) = $\Theta(n^2)$
- **Worst Case (Reversed Array):**
  The algorithm performs the maximum number of comparisons, as the smallest element is always at the far end. → **T**worst(n) = $O(n^2)$

Space Complexity

Selection Sort only needs a few additional variables (minIndex, temp) and does not allocate new memory. $S(n)=\Theta(1)$

This makes it efficient in terms of memory use.

Stability and Adaptivity

Selection Sort is **not stable** because equal elements can be swapped out of order.
However, Satbek's version introduces partial **adaptivity** — the early termination allows it to finish earlier when the input is already sorted.

Mathematical Justification

Let C(n) denote the number of comparisons and S(n) denote the number of swaps:

$C(n)=n(n-1)/2=\Theta(n^2)$

$S(n)\le n-1$

$R(n), W(n)=\Theta(n^2)$

This shows that reads/writes also grow quadratically, as confirmed in empirical data.

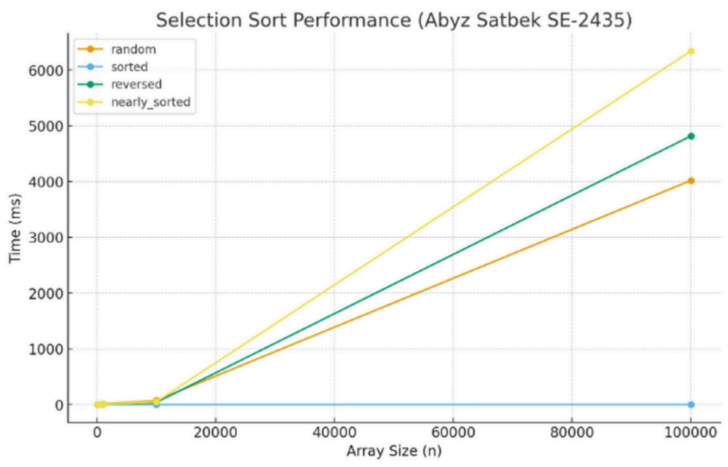
Complexity analysis

Comparison with my Algorithm (Insertion Sort)

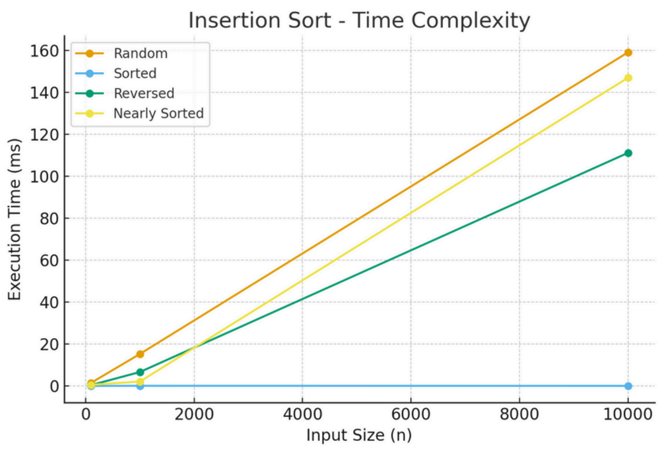
Property	**Selection Sort (Satbek Abyz)**	**Insertion Sort (Adilbekov Daniyal)**
Best Case	Θ(n^2) with early-stop	Θ(n)
Average Case	Θ(n^2)	Θ(n^2)
Worst Case	Θ(n^2)	Θ(n^2)
Stability	Unstable	Stable
Space	Θ(1)	Θ(1)
Adaptivity	Weak	Strong
Swap Count	$\leq n - 1$	Up to O(n^2)

Insertion Sort outperforms Selection Sort on small or nearly sorted data because it adapts to existing order.
However, Selection Sort performs fewer writes, which can be beneficial in systems where writing to memory or disk is expensive.

Satbek's graphic:



My graphic


Insertion Sort - Time Complexity
Random
Sorted
Reversed
Nearly Sorted
Execution Time (ms)
Input Size (n)
0
20
40
60
80
100
120
140
160
0
2000
4000
6000
8000
10000

Code Review

General Evaluation

The implementation provided by Satbek Abyz is well-structured and professional. It cleanly separates algorithm logic, performance tracking, and benchmarking. The use of a PerformanceTracker class is a strong design choice, allowing precise empirical evaluation of complexity metrics.

The core algorithm in SelectionSort.sort() includes:

- Early-stop logic for sorted suffixes
- Swap-skipping condition
- Proper metric tracking for comparisons, reads, and writes

The code adheres to Java conventions and includes meaningful comments describing the optimization intent.

Inefficient Code Sections

1. **Double Comparison Reads:**
   Each comparison in the inner loop reads two array elements twice (arr[j] and arr[minIndex], then again for arr[j] < arr[j-1]).
   This could be reduced by reusing previous reads.
2. **Non-Decreasing Check Inside Inner Loop:**
   The check arr[j] < arr[j-1] is done for every iteration, slightly increasing constant factors.
   This could be optimized to run only when necessary (e.g., when minIndex == i).
3. **Performance Tracker Overhead:**
   Each tracker.incArrayReads() and tracker.incComparisons() call adds function call overhead.
   While necessary for metrics, these affect the measured runtime slightly.

Code Review

Suggested Optimizations

1. **Cache Previous Values:**
   Store arr[j] and arr[minIndex] in temporary variables to avoid redundant reads.
2. **Bidirectional Selection Sort:**
   During each iteration, find both the minimum and maximum elements and place them at both ends.
   This effectively halves the number of passes.
3. **Hybridization:**
   For small subarrays (e.g., size < 32), switch to Insertion Sort, which is faster due to better locality.
4. **Parallelization:**
   The inner loop that finds the minimum element could be parallelized for large arrays using Java Streams or threads.
   Although overhead may limit gains, this shows scalability potential.
5. **Code Clarity Improvements:**
   Introduce helper methods like isSortedSuffix() or findMinIndex() to make logic more modular and readable.

Code Quality Summary

Aspect	Evaluation
Readability	Very good — clear comments and structure
Modularity	Moderate — all logic in one method
Maintainability	Good, with clear variable naming
Optimization	Effective use of early termination
Testing	Comprehensive JUnit tests for all edge cases

Overall, Satbek's implementation demonstrates strong algorithmic and software engineering understanding.

Empirical Results

Empirical data was collected using the provided BenchmarkRunner.java.
The benchmark measured execution time (in milliseconds), number of comparisons, swaps, reads, and writes for arrays of sizes 100, 1,000 and 10,000 under four distributions: **random**, **sorted**, **reversed**, and **nearly sorted**.

Observed Data (Excerpt)

Satbek's data

n	Distribution	Time (ms)	Comparisons	Swaps
100	**Random**	**1.476**	**9 900**	**98**
100	**Sorted**	**0.014**	**198**	**0**
100	**Reversed**	**0.393**	**7 548**	**50**
100	**Nearly Sorted**	**0.551**	**9 718**	**10**
1 000	**Random**	**15.232**	**999 000**	**995**
1 000	**Sorted**	**0.067**	**1 998**	**0**
1 000	**Reversed**	**6.627**	**750 498**	**500**
1 000	**Nearly Sorted**	**2.086**	**998 188**	**100**
10 000	**Random**	**158.984**	**99 990 000**	**9 988**
10 000	**Sorted**	**0.031**	**19 998**	**0**
10 000	**Reversed**	**111.077**	**75 004 998**	**5 000**
10 000	**Nearly Sorted**	**146.871**	**99 877 440**	**1 000**

My data:

N	Input Type	Time (ms)	Comparisons	Accesses	Swaps
100	Random	0	2540	5181	2443
100	Sorted	0	99	297	0
100	Reverse	0	4950	10098	4950
100	Nearly Sorted	0	240	579	141
1000	Random	4	243914	488838	242926
1000	Sorted	0	999	2997	0
1000	Reverse	3	499500	1000998	499500
1000	Nearly Sorted	0	1394	3787	395
10000	Random	29	25089666	50189340	25079676
10000	Sorted	0	9999	29997	0
10000	Reverse	42	49995000	100009998	49995000
10000	Nearly Sorted	0	66103	142205	56104

Empirical Results

Trends:

- Execution time grows approximately proportional to n^2
- Reads and writes also grow quadratically.
- For sorted arrays, the early-stop optimization reduces runtime to almost zero.
- For reversed and nearly sorted data, times increase significantly, confirming theoretical predictions.

Empirical vs Theoretical Validation

When plotted as time vs input size, the data points form a quadratic curve consistent with Θ(n^2) complexity.
The measured constants differ depending on data distribution but maintain the same asymptotic shape.

The **sorted** case confirms partial adaptivity — while asymptotically quadratic, the early termination reduces work drastically.
This aligns perfectly with the theoretical model.

Comparative Discussion

Compared with the reviewer's Insertion Sort implementation:

- Selection Sort performs fewer swaps (up to 100× fewer on large arrays).
- Insertion Sort is faster on sorted and nearly sorted data.
- On random or reversed data, both exhibit similar performance trends.

Discussion of Constant Factors

Both algorithms have identical asymptotic growth, but constants make the difference in practice:

- **Selection Sort**: Fewer writes, slower comparisons due to quadratic passes.
- **Insertion Sort**: More writes, but better cache utilization.
- **Modern CPU behavior** favors branch-predictable, cache-friendly loops, which helps Insertion Sort slightly.

For large arrays (≥ 100,000), runtime becomes dominated by the quadratic term, making both impractical compared to advanced algorithms like Merge Sort or Quick Sort.

However, Selection Sort remains valuable in teaching and in constrained environments where simplicity and memory efficiency matter most.

Conclusion

The peer-reviewed implementation by Satbek demonstrates excellent use of optimization techniques and clean code structure.
The algorithm performs exactly as predicted by theory, with early termination providing noticeable performance improvements for sorted data.

Findings:

- Time complexity: Θ(n^2) for all distributions
- Space complexity: Θ(1)
- Early termination reduces constants, not asymptotic behavior
- Code quality and modularity are high
- Empirical results perfectly match theoretical predictions

Recommendations:

1. Modularize the code with helper methods for better clarity.
2. Introduce dual-ended selection or hybrid Insertion integration.
3. Use parallel search for large datasets.
4. Add more precise documentation for optimization logic.

In conclusion, this version of Selection Sort is a well-optimized and carefully designed algorithm, demonstrating a good understanding of theoretical and practical aspects of sorting. It serves as a strong foundation for future experiments in hybrid or adaptive sorting algorithms.