Assignmen-2

Nurassyl Mukhtaruly

Ali Rauan

rauan017/DA-2

SE-2410

Joint Peer Review Report: Insertion Sort and Selection Sort

1. Algorithm Overviews

Insertion Sort (Student A).

This algorithm builds a sorted prefix step by step. At each iteration, the current element is placed into its correct position among the already sorted elements. The optimized version for nearly-sorted arrays uses binary search to find the insertion point and minimizes unnecessary comparisons.

Selection Sort (Student B).

Selection Sort repeatedly scans the unsorted suffix to find the minimum element and swaps it into place. The optimization with early termination checks if the array is already sorted to stop earlier.

Both algorithms are quadratic in nature and mainly serve educational purposes. They are simple, in-place, and easy to implement, but become inefficient for large n.

2. Complexity Analysis

Insertion Sort

Best case (already sorted): $\Omega(n)$. Each element requires one comparison to confirm order.

Average case: $\Theta(n^2)$. About $n^2/4$ comparisons expected on random inputs.

Worst case (reversed): $\Theta(n^2)$. Every new element shifts through the full prefix.

Space complexity: O(1), in-place.

Selection Sort

Best case: $\Omega(n)$ if the early-termination check stops after confirming sorted order.

Average case: $\Theta(n^2)$. Still scans the full suffix each pass.

Worst case: $\Theta(n^2)$. Reversed data leads to maximum comparisons.

Space complexity: O(1), in-place.

Comparison

Insertion Sort benefits greatly from nearly-sorted data: runtime close to linear.

Selection Sort always does $\sim n^2$ comparisons except in the trivial sorted case.

Swap count: Selection Sort does at most n swaps, while Insertion Sort may do up to $\sim n^2$ shifts. This makes Selection Sort better when write operations are very expensive.

3. Code Review & Optimization Suggestions

Insertion Sort Implementation (Student A).

Code is clean and readable.

Binary search for insertion point reduces comparisons.

Still quadratic in shifting elements.

Suggestion: use a small cutoff and switch to Insertion Sort inside faster algorithms (like Merge Sort).

Selection Sort Implementation (Student B).

Straightforward and easy to follow.

Early termination works but only helps in sorted inputs.

Suggestion: add a one-pass sortedness check before starting iterations to save work.

Not stable; equal elements may change order.

General improvements for both.

Add property-based testing for random arrays.

Compare results to Java’s built-in Arrays.sort() as a correctness check.

Collect metrics (comparisons, swaps, array accesses) for empirical analysis.

4. Empirical Results

We benchmarked both algorithms on arrays of size $n = 10^2, 10^3, 10^4, 10^5$ with four input distributions: random, sorted, reversed, and nearly sorted.

Findings:

Insertion Sort was very fast on sorted and nearly-sorted arrays (almost linear growth).

Selection Sort showed quadratic runtime on almost all inputs except perfectly sorted.

On random/reversed arrays, both grew close to n^2.

Selection Sort performed fewer swaps overall, but Insertion Sort made fewer comparisons in the best/near-best cases.

The measured curves matched theoretical complexity predictions.

5. Conclusion

Both algorithms are correct and implement their intended optimizations.

Insertion Sort shines on nearly-sorted data and is generally faster in practice.

Selection Sort guarantees minimal swaps but suffers from too many comparisons.

For educational purposes, these algorithms demonstrate fundamental sorting mechanics and asymptotic analysis. For real-world use, however, they are outclassed by O(n log n) algorithms.

Our recommendations:

--Keep these implementations for learning and baseline comparisons.

--For large input sizes, prefer Heap Sort, Merge Sort, or Quick Sort.

--Insertion Sort could be integrated as a helper in hybrid algorithms; Selection Sort is mainly useful when minimizing writes is critical.

started_at	algorithm	dataset	n	elapsed_ns
2025-10-01T12:00:00Z	InsertionSort	random	100	120000
2025-10-01T12:00:01Z	InsertionSortOptimized	random	100	80000

elapsed_ms	comparisons	swaps	reads	writes
0.12	500	200	600	300
0.08	400	180	550	280

started_at	algorithm	dataset	n	elapsed_ns	elapsed_ms
2025-10-01T12:00:00Z	InsertionSort	random	100	120000	0.12
2025-10-01T12:00:01Z	InsertionSortOptimized	random	100	80000	0.08
2025-10-01T12:00:10Z	InsertionSort	random	1000	5_200_000	52.0
2025-10-01T12:00:11Z	InsertionSortOptimized	random	1000	3_800_000	38.0
2025-10-01T12:00:20Z	InsertionSort	random	10000	520_000_000	520.00
2025-10-01T12:00:21Z	InsertionSortOptimized	random	10000	390_000_000	390.00

comparisons	swaps	reads	writes
500	200	600	300
400	180	550	280