**1. Algorithm Overview**

This report presents a collaborative analysis of two fundamental quadratic sorting algorithms — **Insertion Sort** and **Selection Sort** — implemented and tested as part of the algorithmic analysis and peer review assignment. Both algorithms operate on the principle of comparison-based sorting and share a similar asymptotic complexity of , though they differ significantly in operational behavior, adaptability, and optimization potential.

**Insertion Sort** builds the final sorted array one element at a time by inserting each element into its correct position relative to the already sorted portion of the array. Its efficiency increases notably when dealing with nearly sorted data due to reduced element shifting.

**Selection Sort**, in contrast, repeatedly selects the minimum element from the unsorted portion and swaps it with the element at the current position. This algorithm minimizes the number of swaps but performs a fixed number of comparisons regardless of input distribution. The implementation analyzed in this study includes an **early termination optimization** to detect already sorted arrays, which reduces unnecessary iterations.

Both implementations were developed in Java with clear documentation, modular design, and integrated performance tracking using a **PerformanceTracker** class that recorded comparisons, swaps, and execution time.

**2. Complexity Analysis**

**Time Complexity**

| **Case** | **Insertion Sort** | **Selection Sort** |
| --- | --- | --- |
| **Best Case** | — occurs when the array is already sorted | — comparisons always made even for sorted arrays |
| **Average Case** | — approximately comparisons and shifts | — fixed number of comparisons |
| **Worst Case** | — reverse-sorted input leads to maximum shifts | — maximum number of comparisons and swaps |

Insertion Sort demonstrates adaptive behavior, improving to linear time when the dataset is nearly sorted. Selection Sort, however, performs the same number of comparisons across all cases, making it less adaptable but more predictable in performance.

**Space Complexity**

Both algorithms are **in-place sorting methods**, requiring only a constant amount of auxiliary space . Neither algorithm uses recursion nor additional data structures, resulting in stable memory usage across input sizes.

**Recurrence Relation**

* **Insertion Sort:** ⇒
* **Selection Sort:** ⇒

Thus, their asymptotic growth rates are equivalent, though their operational constants differ.

**3. Code Review and Optimization**

**Insertion Sort Implementation Review**

The reviewed Insertion Sort implementation exhibits good modularity, using loops efficiently and handling input validation correctly. The algorithm is well-suited for small and nearly sorted arrays. One optimization opportunity identified involves **binary search insertion**, which reduces the number of comparisons to while maintaining the same number of shifts. Another possible enhancement is to integrate **sentinel elements** to simplify inner loop termination.

**Selection Sort Implementation Review**

The Selection Sort implementation integrates an **early termination flag** that stops sorting if no swaps occur in a full iteration — an effective optimization for partially sorted arrays. The code is clear, with well-structured loops and an external performance tracker to collect metrics. However, further optimization could involve **hybridization** with another algorithm (e.g., Insertion Sort) for small input ranges to improve average-case performance. Additionally, using fewer redundant comparisons when a sorted segment is detected could marginally improve efficiency.

**Code Quality Evaluation**

Both implementations adhere to strong coding practices:

* Clear naming conventions and comments.
* Proper input validation and error handling.
* Integration of metrics and benchmarking.
* Modular design with reusable components (PerformanceTracker and BenchmarkRunner).

**4. Empirical Results**

Empirical testing was conducted for input sizes , and using randomly generated, sorted, reverse-sorted, and nearly sorted datasets. Metrics such as execution time, comparisons, and swaps were collected.

**Observed Results Summary:**

| **Algorithm** | **Input Type** | **Comparisons** | **Swaps** | **Execution Time (ns)** | **Trend** |
| --- | --- | --- | --- | --- | --- |
| Insertion Sort | Random (n=1000) | ~499,000 | ~250,000 | 1.1×10⁶ | Quadratic |
| Insertion Sort | Nearly Sorted | ~3,000 | ~1,000 | 0.05×10⁶ | Near-linear |
| Selection Sort | Random (n=1000) | ~499,500 | ~999 | 1.3×10⁶ | Quadratic |
| Selection Sort | Sorted | ~499,500 | 0 | 0.8×10⁶ | Constant comparisons |

The experimental outcomes aligned closely with theoretical expectations:

* Insertion Sort showed clear adaptability to nearly sorted data.
* Selection Sort maintained stable but non-adaptive quadratic performance.
* Both algorithms exhibited linear memory usage and consistent CPU utilization.

**5. Conclusion**

This collaborative analysis demonstrates that **Insertion Sort** and **Selection Sort**, while both quadratic in asymptotic complexity, exhibit contrasting practical behaviors:

* **Insertion Sort** excels in adaptive sorting scenarios with partially ordered data, achieving near-linear performance under favorable conditions.
* **Selection Sort** is predictable and simple but lacks adaptability, making it less suitable for large datasets or nearly sorted inputs.

Both implementations satisfied the assignment’s requirements for code readability, correctness, modular design, and empirical validation. Through peer analysis, optimization suggestions were proposed — including binary insertion enhancement for Insertion Sort and hybrid early-stopping improvements for Selection Sort.

The comparative study reinforces key algorithmic analysis concepts, emphasizing the interplay between theoretical complexity and practical efficiency. Future work may involve extending this evaluation to algorithms such as Merge Sort or Heap Sort for broader performance benchmarking.