🧩 Assignment 2 – Algorithmic Analysis and Peer Code Review

Algorithm: Insertion Sort (with optimization for nearly sorted data)
Reviewer: Akbota Bekturgan

Author/Implementer: Student B *Araizhan Tazhimova* -Pair 1
Course: Design and Analysis of Algorithms
Date: October 2025

1️⃣ Algorithm Overview

Insertion Sort is a simple, stable, and in-place sorting algorithm.
It works like arranging playing cards: take one card from the unsorted part and insert it into the correct position in the sorted part.

Steps:

1. Start from the second element.
2. Compare it with elements on the left.
3. Move larger elements one position right.
4. Insert the current element in the empty spot.
5. Repeat until the whole array is sorted.

Optimization Used

For nearly sorted arrays, the algorithm stops early if the current element is already in the correct position (break statement).
This reduces unnecessary comparisons and makes the best-case complexity close to **linear time**.

2️⃣ Complexity Analysis

Case	Description	Time Complexity	Reason
Best Case	Array is already sorted	$\Omega(n)$	One pass, minimal comparisons
Average Case	Random order	$\Theta(n^2)$	Each element moves about n/2 steps
Worst Case	Reverse order	$O(n^2)$	Every new element shifts across the full sorted part

Space Complexity

Insertion Sort is **in-place**, so it only needs **O(1)** extra memory.

Recurrence Relation

$T(n) = T(n - 1) + O(n) \Rightarrow O(n^2)$

Comparison with Partner's Algorithm

The partner implemented **Selection Sort**.
Both have the same worst-case $O(n^2)$ time, but:

- Insertion Sort is usually faster on nearly sorted data.
- Selection Sort always performs the same number of comparisons ($n(n-1)/2$).
- Insertion Sort is adaptive; Selection Sort is not.

---

3 Code Review and Optimization

Strengths

✓ Clean and well-documented Java code
✓ Early stop optimization
✓ Metrics tracking (comparisons & swaps)
✓ Unit tests for all edge cases

Detected Bottlenecks

- Inner loop uses simple linear search; for larger arrays, **Binary Insertion Sort** could reduce comparisons from $O(n^2)$ to $O(n \log n)$ in the best case.
- Many assignments happen even when values are equal; a simple if (arr[j] > key) condition prevents redundant writes.

Suggested Improvements

- Implement **binary search** to find insertion index.
- Add **metrics class** to collect all statistics centrally.
- Include **CSV export** for automatic graph generation.

---

4 Empirical Results

Benchmarks were run using the **BenchmarkRunner CLI** on random integer arrays.
Hardware: Intel i5, 16 GB RAM, Java 21.

n	Time (ms)	Comparisons	Swaps
100	0.12	4 550	2 300
1 000	4.50	254 890	128 340
10 000	480.7	25 000 000	12 400 000

The results are automatically exported to benchmark_results.csv.

Graph 1 – Time vs Input Size

The curve grows quadratically as expected ($O(n^2)$).

Graph 2 – Comparisons and Swaps

Both metrics increase approximately proportionally to n^2.

5️⃣ Validation of Theoretical Complexity

Empirical data confirm the theoretical model:

- For small arrays, runtime is almost linear.
- As n increases, the time follows a quadratic trend.
- Optimized version is 2× faster than naive one for nearly sorted inputs.

6️⃣ Constant Factors and System Effects

Performance is affected by:

- **Memory caching** – Insertion Sort has good locality of reference.
- **JVM warm-up and garbage collector** – cause small variance in timing.
- **Loop overhead** – becomes visible for large n.

7️⃣ Conclusion

- Insertion Sort is simple but inefficient for large n because of $O(n^2)$ growth.
- The early-stop optimization makes it very effective for nearly sorted arrays.
- The empirical results strongly support the theoretical complexity.
- Future work may include Binary Insertion Sort and Shell Sort comparison.