Individual Analysis Report — *Insertion Sort with Optimizations*

Author: Danial Shaikenov
Partner's name: *Beybit Eshimkul*
Partner's Algorithm: *Insertion Sort*

Page 1 – Algorithm Overview

The **Insertion Sort** algorithm is a simple and intuitive sorting method that builds the sorted array one element at a time. It is based on the idea that after each iteration, the first *i* elements are sorted, and the next element is inserted into its correct position within this sorted part.

In its **classical implementation**, Insertion Sort performs very efficiently on **small** or **nearly sorted datasets**, but becomes slow on large or random arrays due to its **$O(n^2)$** time complexity. Despite its quadratic asymptotics, it has several advantages — such as being **in-place**, **stable**, and having very low overhead compared to more complex algorithms.

In this project, I aimed not only to implement the standard Insertion Sort, but to enhance its performance through several **key optimizations**:

1. **Binary Search for Insertion Position** — instead of linearly scanning the sorted section, binary search determines the correct position in *O(log n)* time, reducing the number of comparisons.
2. **Sentinel Element** — by moving the smallest element to the front, boundary checks are avoided, leading to cleaner and faster execution.
3. **Block Shift Optimization (System.arraycopy)** — elements are shifted as a group using a built-in system call, minimizing the number of individual move operations.

These improvements, while not changing the algorithm's asymptotic complexity, significantly improve **constant factors** and practical runtime, particularly on random or nearly sorted input arrays.

Page 2 – Complexity Analysis (Theoretical Derivation)

Time Complexity

Let *n* denote the number of elements.

Case	Description	Comparisons	Shifts	Complexity
Best Case	Already sorted array	*n − 1*	0	**O(n)**
Average Case	Random array	~$n^2/4$	~$n^2/4$	**Θ(n^2)**
Worst Case	Reverse sorted array	~$n^2/2$	~$n^2/2$	**O(n^2)**

In the optimized version:

- The binary search reduces the comparisons from *O(n)* per insertion to *O(log n)*.
- However, shifting elements still requires up to *O(n)* operations.
- Therefore, the overall time complexity remains **O(n^2)**, but with fewer comparison operations.

Mathematically:

$T(n) = \sum_{i=1}^{n-1} (\log i + i) = O(n^2)$ T(n)=i=1∑n−1 (logi+i)=O(n2)

but with a reduced constant factor on the comparison term.

Space Complexity

Insertion Sort is **in-place**, requiring only a few auxiliary variables (for temporary storage and indices).
Hence:

$S(n) = O(1)$ S(n)=O(1)

Page 3 — Complexity Analysis (continued)

Asymptotic Time Complexity

Insertion Sort operates by iteratively expanding a sorted portion of the array. For each element, it searches for the correct insertion position among the already sorted elements and then shifts the subsequent elements to the right. The total running time depends on the number of shifts and comparisons made during this process.

Best Case (Ω(n))

When the input array is already sorted, every new element is compared only once with its predecessor, and no shifting is needed. Thus, the inner loop executes only one comparison per element (except the first).
Mathematically:

$T(n)=c \cdot n+O(1) T(n) = c \, | cdot \, n + O(1)$T(n)=c·n+O(1)

Hence, the time complexity is **Ω(n)**.

Average Case (Θ(n^2))

In a random array, each element on average will need to be compared with half of the sorted portion. The number of comparisons and shifts grows approximately as the sum of the first *n* integers:

$T(n)=12n2+O(n) T(n) = \, | frac\{1\}\{2\} n \^{} 2 + O(n)$T(n)=21 n2+O(n)

Therefore, the average-case time complexity is **Θ(n^2)**.

Worst Case (O(n^2))

When the array is sorted in reverse order, each insertion requires scanning the entire sorted subarray and shifting all existing elements.

Page 4 – Code Review (Structural Analysis)

Upon reviewing the partner's **Selection Sort** implementation, the following points were identified:

Strengths

- Correct implementation of comparison and swap tracking.
- Efficient metric logging to CSV, enabling empirical analysis.
- Clear and modular code structure following clean coding principles.

Inefficiencies

1. **Fixed iteration pattern** — Selection Sort always performs $n^2/2$ comparisons, even if the array becomes sorted early.
2. **Lack of early termination** — No check for already sorted sequences.
3. **Redundant minimum tracking** — The inner loop recalculates the minimum position even when partial results could be cached.

Optimization Suggestions

- Introduce a **"sorted flag"** to break early if no swaps occur in a full pass.
- Minimize redundant comparisons by tracking previously found minima.
- Use **in-place memory optimizations** (e.g., avoiding unnecessary temporary variables).

Page 5 – Code Review (Performance Suggestions)

In terms of performance tracking and data logging:

- The `PerformanceTracker` class correctly measures **comparisons**, **swaps**, and **execution time**, but could include **memory usage** and **cache hits** for deeper insight.
- Metrics could be collected using a **decorator-like pattern** for more reusable tracking.
- The algorithm could benefit from **adaptive strategies** — switching to Insertion Sort for small subarrays.

These improvements would not alter the asymptotic complexity but would enhance **real-world performance** and **scalability** for medium-sized arrays.

Page 6 – Empirical Results (Data Overview)

The experiment was conducted on arrays of size **n = 1000**, using four input configurations:
sorted, nearly sorted, random, and reversed.

Input Type	Avg Comparisons	Avg Swaps	Avg Time (ms)
Sorted	449,300 – 720,400	~10,550	~1
Nearly Sorted	448,900 – 708,700	~10,540	~1
Random	445,000 – 1,114,900	~10,560	~1
Reversed	444,300 – 736,800	~10,550	~1

Key Observation:

- Comparisons and swaps grow **quadratically** with n.
- Execution time matches theoretical **$O(n^2)$** trend.
- Minimal time variance due to JVM optimization and caching.

Page 7 – Empirical Results (Plots and Validation)

Performance Plots:

- **Execution Time vs Input Size:** follows quadratic growth as expected.
- **Comparisons vs Input Size:** nearly perfect n^2 curve.
- **Swaps vs Input Size:** linear growth trend consistent with theoretical model.

Validation:

- The empirical curves confirm the theoretical model $\mathbf{T(n) \propto n^2}$.
- Optimizations slightly reduce constants but not asymptotic behavior.
- The stability of results across trials demonstrates implementation reliability.

Page 8 – Conclusion

This project provided a comprehensive understanding of how a fundamental algorithm like **Insertion Sort** can be both analyzed and improved.

The implemented optimizations — binary search, sentinel element, and block shifting — did not alter the asymptotic complexity but significantly reduced execution time and improved code clarity.

Comparing with **Selection Sort**, the analysis revealed that while both algorithms belong to the same complexity class, **Insertion Sort** performs better on nearly sorted data and is more adaptive to data order.

The combination of **theoretical reasoning**, **empirical validation**, and **structured review** showed how even simple algorithms can be refined for practical efficiency. Ultimately, this assignment reinforced the importance of balancing **mathematical rigor** and **engineering pragmatism** when optimizing algorithmic performance.