Insertion Sort – Peer Analysis Report

Author: ***Galymzhan Turemuratov***
Partner Algorithm: Insertion Sort (by Danial Sadykov)
Course: Design and Analysis of Algorithms
Date: October 2025

1. Algorithm Overview

The Insertion Sort algorithm is a simple, comparison-based sorting method that builds the final sorted array one element at a time. It works similarly to the way humans sort playing cards — by inserting each new card into its proper position among the previously sorted cards.

Insertion Sort is particularly effective for small or nearly-sorted datasets, as it adapts quickly when elements are already close to their correct positions.
The optimized version implemented by my partner includes a check for "early termination" when the array is already sorted, and minimizes unnecessary swaps by using shifting instead of full element swapping.

2. Complexity Analysis

Time Complexity

Case	Description	Complexity
Best Case (Ω)	When the array is already sorted	$\Omega(n)$
Average Case (Θ)	Random order input	$\Theta(n^2)$
Worst Case (O)	When the array is in reverse order	$O(n^2)$

Derivation:

- Each insertion may require comparing the current element with all previous ones.
- In the best case, only one comparison is made per element → $T(n) = n - 1 \approx O(n)$
- In the worst case (reverse sorted), each element is compared with all previous → $T(n) = n(n-1)/2 \approx O(n^2)$

Space Complexity

- O(1) — The algorithm is *in-place*, requiring only a constant amount of extra space.
- No additional arrays or recursive calls are used.

3. Code Review and Optimization

Code Quality

The code is well-structured, uses clear naming conventions, and includes extensive bilingual (English/Russian) comments.
Exception handling ensures that null or empty arrays are processed safely without runtime errors.

Detected Inefficiencies

- The shifting process uses explicit loops instead of System.arraycopy, which could improve speed on large arrays.
- No threshold-based hybridization (e.g., switching to Binary Insertion Sort for large datasets).

Optimization Suggestions

1. Binary Search for Insertion Position:
   Replace the inner linear search with binary search to reduce comparisons from *O(n²)* to *O(n log n)* (though still *O(n²)* overall for shifts).
2. Adaptive Check:
   Add a pre-scan to detect if the array is already sorted — avoids unnecessary passes.
3. Memory Efficiency:
   Currently optimal (*O(1)* auxiliary), no improvements needed.

4. Empirical Results

Testing Setup

Benchmarks were run using BenchmarkRunner.java and metrics collected via PerformanceTracker.
Test arrays of sizes 100, 1,000, 10,000, and 100,000 elements were used (random, sorted, and reversed).

Input Size (n)	**Case**	**Comparisons**	**Swaps**	**Time (ms)**
100	Sorted	~99	~0	0.01
1,000	Random	~500,000	~500,000	0.25
10,000	Random	~50,000,000	~50,000,000	25
100,000	Reverse	~5×10^9	~5×10^9	2600

Performance Observations

- The runtime grows approximately quadratically with n, confirming the theoretical *O(n²)* prediction.
- Best case (sorted input) nearly linear, validating the *Ω(n)* lower bound.
- Optimizations for nearly-sorted data show significant improvements in practical performance.

5. Conclusion

Insertion Sort demonstrates clear asymptotic behavior consistent with its theoretical complexity.
While not suitable for large datasets due to quadratic growth, it remains an excellent educational example and performs efficiently on small or nearly-sorted inputs.

Key findings:

- The implementation is clean, stable, and memory-efficient.
- Time complexity measurements match theoretical expectations.
- Future optimization could include binary insertion or adaptive hybridization.

Overall, this is a well-engineered implementation that fulfills all course requirements and provides accurate metric tracking for academic analysis.