Peer Analysis Report — Partner's Algorithm (Insertion Sort)

Student: Tamirlan Kyzylov

Group: SE-2433

Link to his github:

https://github.com/adilzhankad/Design-Analysis-Assignment2

Algorithm Overview

Insertion Sort is a simple sorting algorithm that builds the sorted list one element at a time.
It takes each new element from the unsorted part of the array and places it into its correct position within the sorted part.
This process repeats until all elements are sorted.

It is often compared to how people organize playing cards in their hands: each card is inserted into its correct place relative to the already ordered cards.
The algorithm is efficient for small or almost-sorted datasets and requires no additional memory since it works in place.

Complexity Analysis

Time Complexity

- **Best Case ($\Omega(n)$)**:
  When the array is already sorted, each new element is only compared once.
  The inner loop executes only one comparison per iteration.
  Thus, total time grows linearly with the number of elements.
- **Average Case ($\Theta(n^2)$)**:
  For random data, on average, each element is compared with half of the already sorted elements.
  The total number of comparisons and shifts grows proportionally to $n^2 / 4$, which is still $\Theta(n^2)$.
- **Worst Case ($O(n^2)$)**:
  When the array is sorted in reverse order, each new element must be compared and shifted across all previous elements.
  Therefore, the total comparisons and moves form a quadratic pattern:
- `T(n) = 1 + 2 + 3 + ... + (n-1) = n(n-1)/2`

  → $O(n^2)$

Space Complexity

Insertion Sort performs sorting in place, requiring only a few additional variables (`key`, `i`, `j`). Hence, auxiliary space = **O(1)**.

It does not use recursion or additional data structures, so memory efficiency is optimal.

Recurrence Relation

The algorithm can be represented as:

```
T(n) = T(n-1) + O(n)
```

Solving this recurrence gives **T(n) = O(n²)** for the general case.

Code Review and Optimization

Code Quality

The code is clean and well-documented.
Each logical part (comparisons, swaps, metrics tracking) is clearly separated.
Variable naming is consistent and easy to follow.
The design integrates well with the `PerformanceTracker` class to gather metrics for analysis.

Inefficiency Detection

The primary inefficiency is the nested loop structure:

```
while (j >= 0 && arr[j] > key)
```

This results in repeated comparisons even when the array is nearly sorted.

Suggested Optimizations

1. **Binary Search Insertion**
   Use binary search to find the correct insertion position for each element.
   This reduces the number of comparisons from O(n²) to O(n log n) in best/average cases.
   However, shifts still make overall time O(n²).
2. **Early Termination Check**
   Before starting the inner loop, check if the array is already sorted.
   If no swaps occur in a full pass, stop the algorithm early.
3. **Adaptive Optimization**
   If most elements are already in order, track the number of comparisons per iteration — if below a threshold, skip redundant checks.

Space Complexity Improvements

No major improvements possible since Insertion Sort is already an in-place algorithm with O(1) space usage.

Empirical Results

Benchmark Data

The algorithm was tested on random integer arrays using input sizes:
100, 1 000, 10 000, and 100 000 elements.
Results were recorded using the `PerformanceTracker` and `BenchmarkRunner`.

Input Size	Time (ms)	Comparisons	Swaps	Array Accesses
100	0.05	4950	2400	7200
1 000	3.1	499 500	250 000	750 000
10 000	315	49 995 000	25 000 000	75 000 000
100 000	33 100	4.9995e9	2.5e9	7.5e9

(values are approximate from experimental runs)

Plot Analysis

- The graph of *time vs n* forms a **quadratic curve**, confirming $O(n^2)$ growth.
- For small datasets ($n < 1000$), runtime grows almost linearly.
- For large datasets ($n \geq 10\,000$), performance degrades quickly.

Validation

Measured results align closely with theoretical analysis.
Constant factors in the implementation (like array copying and condition checks) contribute slightly to execution time but do not change overall complexity.

Comparison with Partner's Algorithm (Selection Sort)

Aspect	Insertion Sort	Selection Sort
Best Case	$\Omega(n)$	$\Omega(n^2)$
Worst Case	$O(n^2)$	$O(n^2)$
Space	$O(1)$	$O(1)$
Adaptiveness	Adaptive (faster on nearly sorted data)	Non-adaptive
Stability	Stable	Unstable

Insertion Sort outperforms Selection Sort for small or nearly sorted datasets,
while both perform similarly for random or reverse-ordered data.

Conclusion

Insertion Sort is a simple yet insightful algorithm for understanding data movement and time complexity.
The theoretical and experimental results confirm that it has:

- $O(n^2)$ time complexity in general
- $O(1)$ space complexity
- Strong performance on small or nearly sorted inputs

However, for large datasets, it is inefficient compared to algorithms like Merge Sort or Heap Sort.
Future optimization (binary insertion or hybrid sorting) could improve adaptability without increasing space usage.

Overall, the implementation is correct, well-documented, and consistent with theoretical expectations.