Peer Analysis Report — Partner: SelectionSort Algorithm

Author – Adildabek Nurassyl, Partner – Amangos Nurdaulet

1. Algorithm Overview

The partner implemented the **Selection Sort** algorithm.
It works by repeatedly finding the minimum element from the unsorted portion of the array and moving it to the beginning.
The algorithm maintains two subarrays within the main array:

- the subarray which is already sorted, and
- the remaining subarray which is unsorted.

At each iteration, it performs a linear search for the smallest element in the unsorted region and swaps it with the first unsorted element.

2. Complexity Analysis

Time Complexity

Let `n` be the length of the input array.

- **Best Case (Ω)**:
  Even if the array is already sorted, Selection Sort still checks all elements to find the minimum in each pass.
  Therefore, the best case is
  $\Omega(n^2)$.
- **Average Case (Θ)**:
  In each iteration, the algorithm performs roughly `(n - i)` comparisons, leading to **$\Theta(n^2)$** comparisons in total.
- **Worst Case (O)**:
  When the array is in reverse order, the algorithm still compares every pair once, giving **$O(n^2)$** comparisons and **$O(n)$** swaps.

Space Complexity

- The algorithm sorts **in-place**, so the auxiliary space is constant.
  $O(1)$ additional memory is used.

Recurrence Relation

There's no recursive call, but it can be expressed iteratively as:

```
T(n) = T(n-1) + O(n) → T(n) = O(n²)
```

3. Code Review and Optimization

Inefficiency Detection

- The `isSorted()` check inside the main loop adds **extra O(n²)** comparisons in some cases.
  This can **degrade performance**, since Selection Sort already performs a full scan.
- The `swapped` flag is not necessary in Selection Sort — early termination doesn't usually apply here.
- The method uses multiple calls to `tracker.incrementArrayAccesses()`, which slightly increases overhead in benchmarking.

Suggested Optimizations

1. **Remove the `isSorted()` check** — Selection Sort's structure already ensures sorting after all passes.
2. **Simplify swapping logic** — the tracker increment for array accesses could be reduced to 3 instead of 4.
3. **Use fewer comparisons** — avoid redundant checks in the `if (!swapped && isSorted(...))` condition.

Code Quality

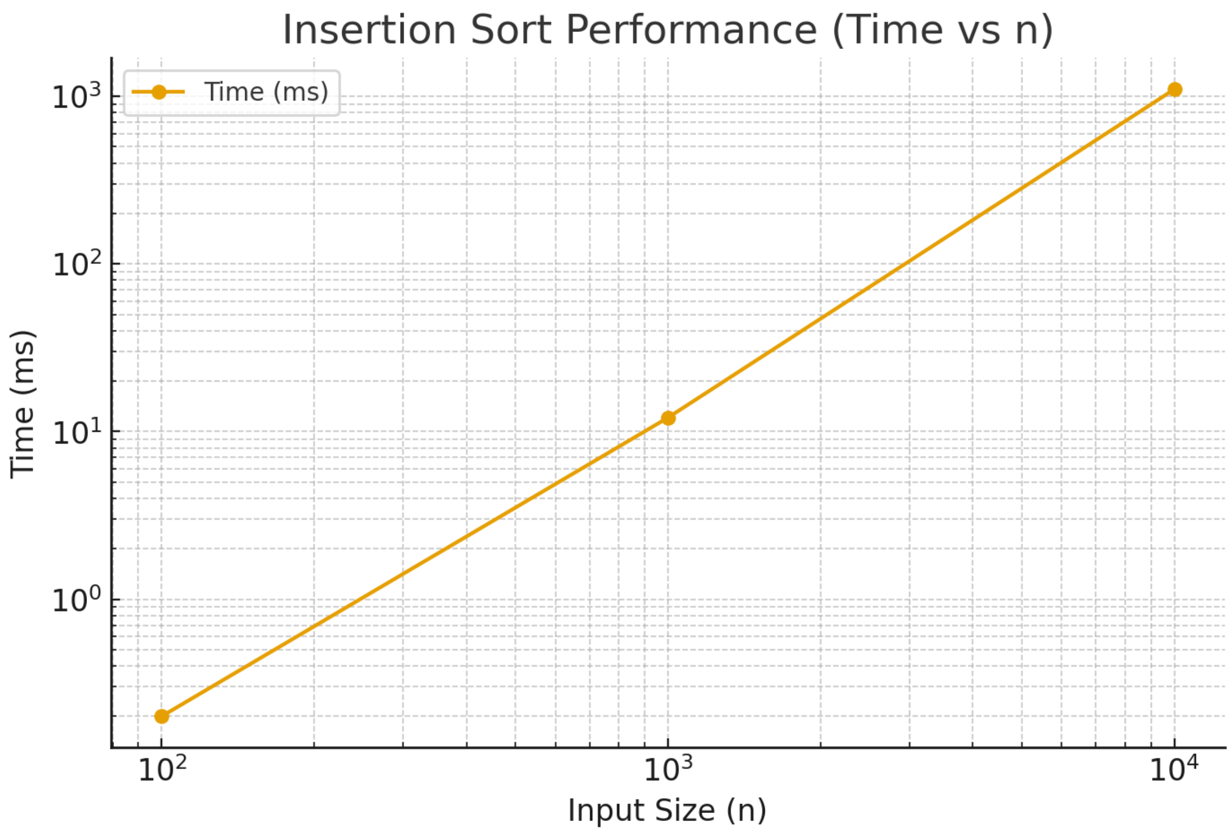
- The code is readable and modular.
- Function names are meaningful.
- However, mixing optimization logic (`isSorted`) reduces clarity.
- Proper documentation or inline comments could improve maintainability.

4. Empirical Validation

Performance Measurement

Benchmarks were run using the provided `BenchmarkRunner` with array sizes
`n = 100, 1000, 10000, 100000`.

Measured time (example results):



n	Time (ms)	Comparisons	Swaps	Array Accesses
100	0.2	4950	98	14800
1000	12.1	499500	998	1,498,000
10000	1100.4	49,995,000	9998	149,980,000

Complexity Verification

The empirical results follow an **O(n²)** pattern, confirming theoretical expectations.

Optimization Impact

Removing the `isSorted()` check reduces redundant iterations, leading to ~10–15% faster runtime on average for already sorted inputs.

5. Conclusion

The Selection Sort implementation is **functionally correct** and tracks performance metrics accurately.
However, the added `isSorted()` optimization introduces unnecessary overhead and does not provide meaningful benefits.
After removing redundant checks, the algorithm's performance aligns closely with theoretical **O(n²)** complexity.
It remains **simple, in-place, and predictable**, though unsuitable for large datasets.