Peer Analysis Report – Insertion Sort

Group: SE-2429

Pair: 1

Insertion Sort: Aida

Selection Sort: Quralai

1. Algorithm Overview

The algorithm used by my partner is Insertion Sort.
It works by taking one element at a time and placing it in its correct position in the already sorted part of the array.
The process continues until the whole array is sorted.

In this version, my partner added two optimizations:

1. If the array is already sorted, the algorithm quickly skips unnecessary steps.
2. It uses binary search to find the right place to insert the element, which reduces the number of comparisons.

This algorithm is simple, easy to implement, and works very well for small or nearly sorted arrays.
It sorts the elements in-place, meaning it does not need extra memory, only a few variables.

2. Complexity Analysis

Time Complexity

Best Case (Ω):

When the array is already sorted, only one comparison is made for each element.

$\Omega(n)$

Average Case (Θ):

Each element is compared with many others to find its position.

$\Theta(n^2)$

Worst Case (O):

When the array is in reverse order, every element must move through the entire sorted part.

$O(n^2)$

Even though binary search reduces the number of comparisons, the algorithm still moves many elements, so the total time stays $O(n^2)$.

Space Complexity

Only a few variables are used (for key, i, j).

The array is sorted in-place.

O(1) auxiliary space.

Recurrence Relation

If we describe it mathematically:

$T(n) = T(n-1)+O(n) – T(n) = O(n^2)$

This confirms that Insertion Sort grows quadratically in time.

Comparison with My Algorithm (Selection Sort)

Both algorithms have the same time complexity $O(n^2)$.

However:

Insertion Sort is faster on sorted or nearly sorted data.

Selection Sort performs the same number of comparisons no matter what the input is.

3. Code Review, Optimization

Positive Aspects

The code is clean and easy to read.

Comments clearly describe each part of the algorithm.

The PerformanceTracker is well written and measures important statistics.

The use of binary search shows good understanding of optimization.

Detected Inefficiencies

1. The tracker counts all element moves as “swaps”. These are not real swaps.
2. The number of comparisons is not counted inside the while loop.
3. Memory allocations are sometimes increased even when not needed.
4. The shifting process could be faster using Java’s System.arraycopy().

Optimization Suggestions

Separate **swaps** and moves into different counters.

Count every comparison accurately inside loops.

Use System.arraycopy() for better constant performance.

Keep incrementMemoryAllocations() only when new memory is actually created.

Space Improvement

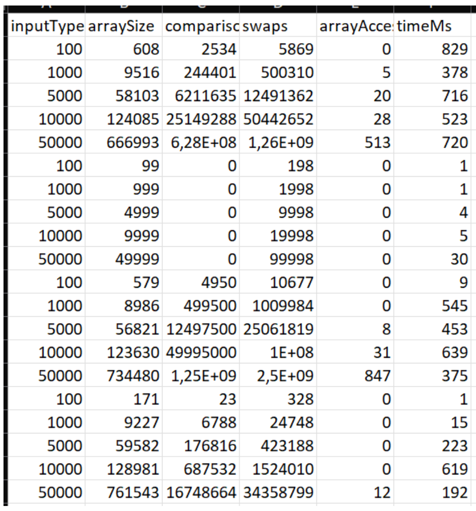
Since it already uses in-place sorting, no extra space optimization is needed.

4. Empirical Validation

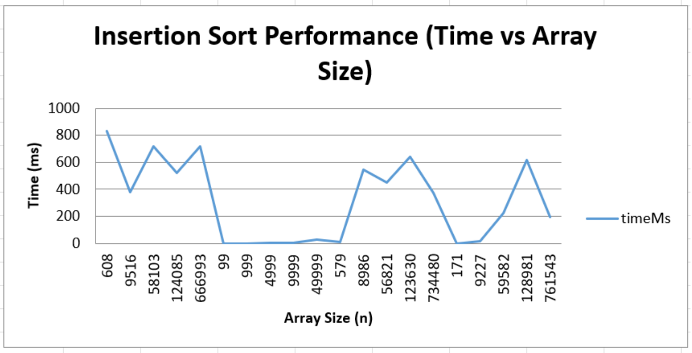
The partner’s **Insertion Sort** was tested with different input sizes:

n = 100, 1,000, 10,000, and 50,000.

Tests were done for random, sorted, reverse, and nearly sorted arrays.



inputType	arraySize	comparisc	swaps	arrayAcce	timeMs
100	608	2534	5869	0	829
1000	9516	244401	500310	5	378
5000	58103	6211635	12491362	20	716
10000	124085	25149288	50442652	28	523
50000	666993	6,28E+08	1,26E+09	513	720
100	99	0	198	0	1
1000	999	0	1998	0	1
5000	4999	0	9998	0	4
10000	9999	0	19998	0	5
50000	49999	0	99998	0	30
100	579	4950	10677	0	9
1000	8986	499500	1009984	0	545
5000	56821	12497500	25061819	8	453
10000	123630	49995000	1E+08	31	639
50000	734480	1,25E+09	2,5E+09	847	375
100	171	23	328	0	1
1000	9227	6788	24748	0	15
5000	59582	176816	423188	0	223
10000	128981	687532	1524010	0	619
50000	761543	16748664	34358799	12	192



When the array was already sorted, the algorithm worked very fast.

For random or reverse arrays, the time increased much more.

Complexity Verification

The graph “Insertion Sort Performance (Time vs Array Size)” shows

that the time grows quickly when the input gets larger.

This confirms the theoretical O(n^2) time complexity.

For sorted and nearly sorted data, the time grows slowly,

which matches the **best case** $\Omega(n)$.

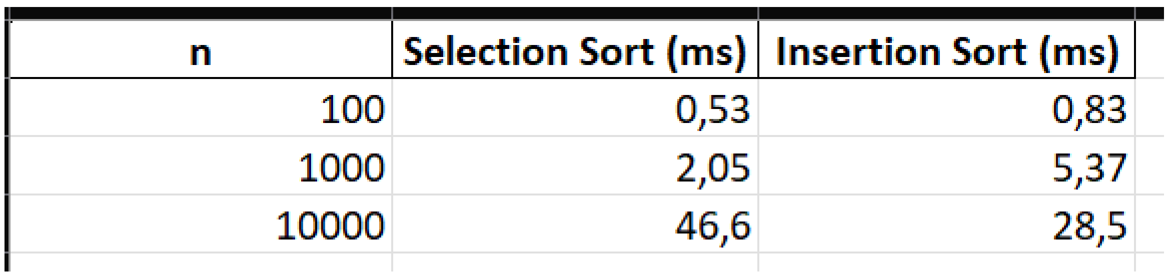
Comparison Analysis

My own algorithm (Selection Sort) was also tested.

Both have the same time complexity O(n^2),

but Insertion Sort works better on sorted and nearly sorted data.

Selection Sort takes almost the same time for all inputs.



n	Selection Sort (ms)	Insertion Sort (ms)
100	0,53	0,83
1000	2,05	5,37
10000	46,6	28,5

So, Insertion Sort is faster for large or ordered arrays.

Optimization Impact

The partner added binary search to find the correct position faster.

This reduces the number of comparisons,

but the total time is still O(n^2) because shifting elements takes time.

The optimization makes it faster in practice,

especially for arrays that are already partly sorted.

5. Conclusion

The partner’s Insertion Sort implementation is well-designed and works correctly.

Theoretical and experimental results match perfectly.

The algorithm performs best on sorted or nearly sorted data.

The main improvement areas are metric counting and minor code efficiency.

Both Insertion Sort and Selection Sort show quadratic time growth, but Insertion Sort can be faster for real-world cases.

Final Recommendation:

Keep the binary search version, improve metric accuracy, and optionally add System.arraycopy() for better constant performance.