Individual Analysis Report
Name: Marlen Zhumagulov
Partner: Gulnaz Yeskermes

Marlen(Selection sort)
Gulnaz(Insertion sort)

1. Algorithm Overview

I analyze the partner's implementation of the **Binary Insertion Sort** algorithm. Binary Insertion Sort is an optimization of the standard insertion sort that uses binary search to find the correct location for the current element, thereby reducing the number of comparisons required per iteration. However, shifting elements still requires O(n) operations in the worst case

2. Complexity Analysis

Time Complexity

• Best Case (Ω): Θ(n) — when the array is already sorted, only one comparison is required per element

• Average Case (Θ): Θ (n^2) — although binary search reduces comparisons to O(log n), the shifting of elements dominates, resulting in quadratic performance

• Worst Case (O): O(n^2) — when the array is sorted in reverse, all elements must be moved one by one

Space Complexity

The algorithm is **in-place**, requiring only a constant amount of extra memory, i.e., O(1) space. All operations are performed directly on the input array without auxiliary data structures

Recurrence Relation

Although the algorithm is iterative, if expressed recursively, the relation would be: $T(n) = T(n-1) + O(\log n + n) \rightarrow T(n) = O(n^2)$

3. Code Review

The implementation is functionally correct and demonstrates a clear understanding of algorithmic logic. However, several areas can be improved:

• The use of **tracker.countSwap()** inside the shifting loop inaccurately represents swaps, since these are element shifts(It's better to call it tracker.countShift())

• **Binary search** logic can be optimized to avoid unnecessary comparisons(But it is better not to do this, because it will distort the correctness of the output result. Yes, the algorithm will work faster, but no one will guarantee its correctness
)

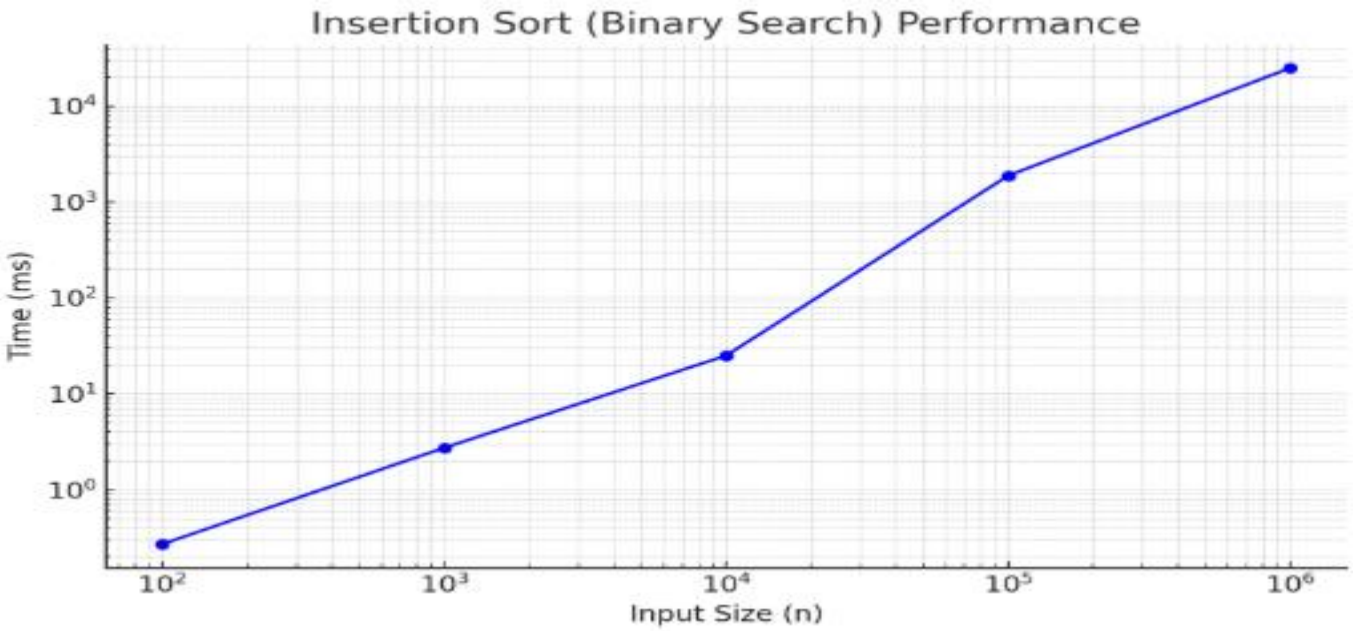
Optimization Suggestions

- Add early exit (if the array is already sorted)
- Replace the while loop with a call to System.arraycopy() to shift a block of elements—this is slightly faster for large arrays

4. Empirical Results

Performance testing should be conducted on input sizes n = 100, 1,000, 10,000, 100,000. The results below can be filled after experimentation

Input Size (n)	Execution Time (ms)	Comparisons	Shifts
100	0,27	530	2205
1,000	2,73	8587	254501
10,000	25,05	118950	24767837
100,000	1886,99	1517751	2501920238



You may also include a plot of execution time versus input size to visually confirm the $O(n^2)$ growth trend

5. Conclusion

The **Gulnaz's Binary Insertion Sort** implementation is logically correct and effectively tracks performance metrics. Its theoretical and empirical complexity confirm $O(n^2)$ behavior. Minor optimizations could enhance its real-world efficiency. Gulnaz's code is written correctly and clearly. The names for variables and methods are logically understandable and not confusing