1. Algorithm Overview

The analyzed algorithm implemented by my partner is **Selection Sort**. It is a simple, comparison-based sorting algorithm that repeatedly selects the smallest element from the unsorted part of the array and places it at the beginning. The algorithm divides the array into two parts — a sorted and an unsorted segment — and gradually extends the sorted portion by finding the minimum element in each iteration. The Selection Sort algorithm is based on the idea of iteratively selecting the minimum value, requiring *$O(n^2)$* comparisons and *$O(n)$* swaps in total. While easy to implement and predictable, it is inefficient for large datasets due to its quadratic time complexity and limited adaptability to partially sorted input. My own algorithm, **Binary Insertion Sort**, uses a binary search to find the correct position for each element in the sorted portion. This reduces the number of comparisons to *$O(n \log n)$* in the best case, while maintaining *$O(n^2)$* swaps in the worst case due to shifting elements.

2. Complexity Analysis

Time Complexity Analysis
For Selection Sort:
• Best Case: $\Omega(n^2)$ — all elements must still be compared to find the minimum.
• Average Case: $\Theta(n^2)$.
• Worst Case: $O(n^2)$.
• Space Complexity: $O(1)$, as sorting occurs in place.

For Binary Insertion Sort:
• Best Case: $\Omega(n \log n)$ — when the array is nearly sorted, binary search minimizes comparisons.
• Average Case: $\Theta(n^2)$.
• Worst Case: $O(n^2)$ — shifting elements dominates.
• Space Complexity: $O(1)$.

Comparison
Although both algorithms share the same asymptotic worst-case complexity, Binary Insertion Sort performs fewer comparisons thanks to binary search. Selection Sort, on the other hand, performs a fixed number of comparisons regardless of array order, making it less adaptive.

3. Code Review

The partner's Selection Sort implementation is clear and well-structured. However, it can be optimized for performance by minimizing redundant swaps and improving cache efficiency. Currently, the algorithm always scans the remaining array for the smallest element, even if the tail is already sorted. **Inefficiencies:**
• Inner loop always performs n-i-1 comparisons even when unnecessary.
• The swap operation increments a counter even if the array is already sorted.

Proposed Optimizations:
1. Introduce a flag to check if the array is already sorted — if no swaps occur, break early.

2. Use a local variable for temporary values to reduce memory access overhead.
3. Replace array-based implementation with an iterator-based version for flexibility.

These optimizations will not change asymptotic complexity but will reduce constant factors and improve runtime on partially sorted inputs.

4. Empirical Results

Empirical analysis confirms theoretical expectations. Both algorithms were tested on datasets ranging from 100 to 10,000 elements. Selection Sort exhibited a consistent quadratic growth in execution time, while Binary Insertion Sort showed better performance for small and partially sorted datasets due to fewer comparisons achieved through binary search. However, the cost of shifting elements keeps its overall complexity quadratic. In practice, Selection Sort maintains predictable performance, whereas Binary Insertion Sort offers practical speedup for sorted or nearly sorted data, despite the same asymptotic complexity.

5. Conclusion

In conclusion, the Selection Sort algorithm is simple and memory-efficient but not suitable for large datasets due to its $O(n^2)$ time complexity. Binary Insertion Sort demonstrates theoretical improvement in comparison operations and better empirical behavior for nearly sorted data. Overall, for small or educational purposes, Selection Sort remains a good choice for understanding sorting logic, but Binary Insertion Sort provides more practical efficiency without increasing space usage.