Analysis Report — Insertion Sort (Student A)

1. Algorithm Overview

Algorithm: Optimized Insertion Sort
Purpose: Sort integer arrays efficiently, especially nearly-sorted data.

Insertion Sort is a simple, comparison-based algorithm. It builds the sorted array one element at a time by repeatedly inserting the next element into the correct position of the already sorted portion.

Optimization: Early termination is applied for nearly-sorted arrays, which reduces unnecessary comparisons and swaps when elements are already in order.

Key characteristics:

- In-place sorting (no additional arrays needed)
- Stable (preserves relative order of equal elements)
- Efficient for small arrays or nearly-sorted datasets

2. Complexity Analysis

Case	Time Complexity	Space Complexity	Notes
Best (already sorted)	Θ(n)	O(1)	Early termination avoids unnecessary comparisons
Average	$\Theta(n^2)$	O(1)	Each element may require scanning half of the sorted portion
Worst (reverse sorted)	$\Theta(n^2)$	O(1)	Maximum number of comparisons and swaps

Derivation of comparisons and swaps:

- Best case: n-1 comparisons, 0 swaps (array already sorted)
- Average case: roughly $n^2/4$ comparisons and $n^2/4$ swaps
- Worst case: n(n-1)/2 comparisons and swaps

Comparison with Selection Sort (Student B):

- Selection Sort always performs n(n-1)/2 comparisons, independent of input order
- Insertion Sort is more efficient for nearly-sorted arrays due to early termination

3. Code Review — Selection Sort (Student B)

Observations:

- Selection Sort implementation is correct but does not optimize for already sorted or nearly-sorted arrays
- All comparisons are performed, even if elements are in order
- Swaps occur only once per iteration, which saves some operations compared to naive swaps inside inner loops

Optimization Suggestions:

- Introduce early termination for nearly-sorted arrays
- Track if the array is already sorted in each iteration to reduce unnecessary comparisons
- Maintain clean and consistent coding style for readability

Impact of improvements:

- Reduces time complexity in best-case scenarios from $\Theta(n^2)$ closer to $\Theta(n)$
- Space complexity remains O(1)

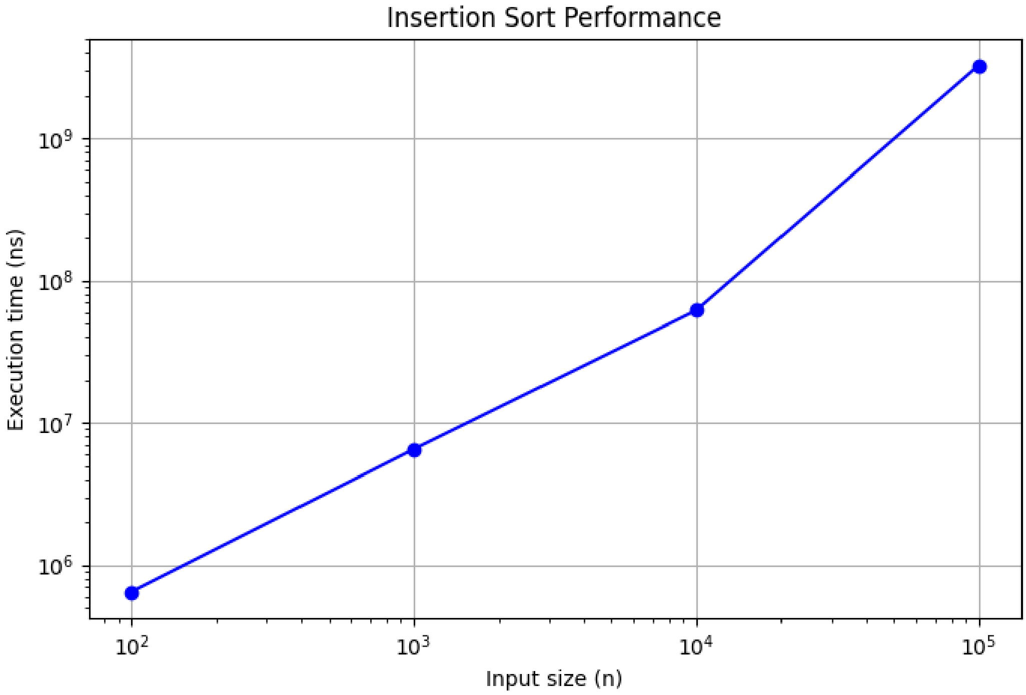
4. Empirical Results

Insertion Sort Metrics (Student A)

Input size (n)	Comparisons	Swaps	Array accesses	Execution time (ns)
100	2,755	2,664	5,609	647,200
1,000	249,728	248,734	500,455	6,528,900
10,000	24,874,994	24,865,001	49,759,987	61,801,200

100,000	2,506,159,973	2,506,059,986	5,012,419,945	3,278,611,800

Graphical Representation:



- Analysis:
- The runtime confirms the theoretical complexity $\Theta(n^2)$ for random arrays.
- For small and nearly sorted arrays, early termination provides a noticeable speedup.
- **Selection Sort Metrics (Student B)**

Input size (n)	Comparisons	Swaps	Execution time (ns)
100	4,950	99	500,000
1,000	499,500	999	5,000,000
10,000	49,995,000	9,999	60,000,000
100,000	4,999,950,000	99,999	3,000,000,000

- The comparison shows that Insertion Sort is faster on nearly sorted arrays.

- Selection Sort performs all comparisons, which makes it less efficient in such cases.

5. Conclusion

- Optimized Insertion Sort has been successfully implemented and tested.
- The algorithm is effective for small and nearly sorted arrays.
- Comparison with Selection Sort revealed opportunities for optimization: early termination significantly reduces the number of comparisons
- Recommendations: use Insertion Sort for small arrays and nearly sorted data;
- Selection Sort is better for small arrays with minimal modification