Assignment 2

Design and Analysis of Algorithms

Name: Sultanbek Zhumagali

Group: SE-2420

Insertion sort

1.Algorithm Overview

Insertion Sort builds a sorted sequence incrementally by taking one element at a time and inserting it into its correct position among the already sorted elements. It is intuitive and works well on small or nearly-sorted datasets.

Theoretical Background:

1) **Best Case:** $\Omega(n)$ when the array is already sorted (only one comparison per element).
2) **Average/Worst Case:** $\Theta(n^2)$, $O(n^2)$ when many shifts are required (e.g., reverse-sorted data).
3) **Space Complexity:** $O(1)$, since sorting is done in-place.
4) **Property:** Adaptive and stable; efficient for small or nearly-sorted inputs.

2.Asymptotic Complexity Analysis

Time Complexity

- **Best Case (Nearly-Sorted Data):**
  - For each element, only one comparison is made (arr[j] <= key breaks immediately).
  - Complexity: $\mathbf{\Omega(n)}$.
- **Worst Case (Reverse-Sorted Array):**
  - Each new element is compared with all previous elements → about $n(n-1)/2$ comparisons and shifts.
  - Complexity: $\mathbf{O(n^2)}$.
- **Average Case (Random Array):**
  - On average, each element is compared with half of the sorted subarray.
  - Expected ~ $n^2/4$ operations.
  - Complexity: $\mathbf{\Theta(n^2)}$.

Space Complexity

- The algorithm is **in-place** (no extra array needed).
- Only key variable is used in addition to input array.
- Auxiliary space: **O(1)**.

Recurrence Relation

Let T(n) be the runtime for sorting n elements:

$$T(n) = T(n-1) + O(n)$$

- Base case: T(1) = O(1)
- Solution:

$$T(n) = O(n^2)$$

3.Code Review & Optimization

Inefficiency Detection

- Metrics counting (arrayAccesses, comparisons, shifts) adds small overhead, but is fine for analysis.
- The **linear search** for insertion point (while (j >= 0 && arr[j] > key)) is costly in worst/average case.

Time Complexity Improvements

- **Binary Search Insertion Sort**:
  Instead of scanning linearly, use binary search to find insertion position in **O(log n)**.
    - Overall complexity remains **O(n^2)** (due to shifting), but comparisons reduce from O(n^2) → O(n log n).
- For very large datasets, switching to **Merge Sort (O(n log n))** or **TimSort** (hybrid of merge + insertion, used in Java's Arrays.sort) is better.

Space Complexity Improvements

- Already optimal (O(1)).
- One minor tweak: avoid redundant metric increments (e.g., double counting arrayAccesses for same action).

Code Quality

- Clear variable names (key, comparisons, shifts).
- Exception handling for null.
- Separation of concerns (resetMetrics, printMetrics).
- Suggested Improvement: Document complexity assumptions directly above insertionSort for maintainability.

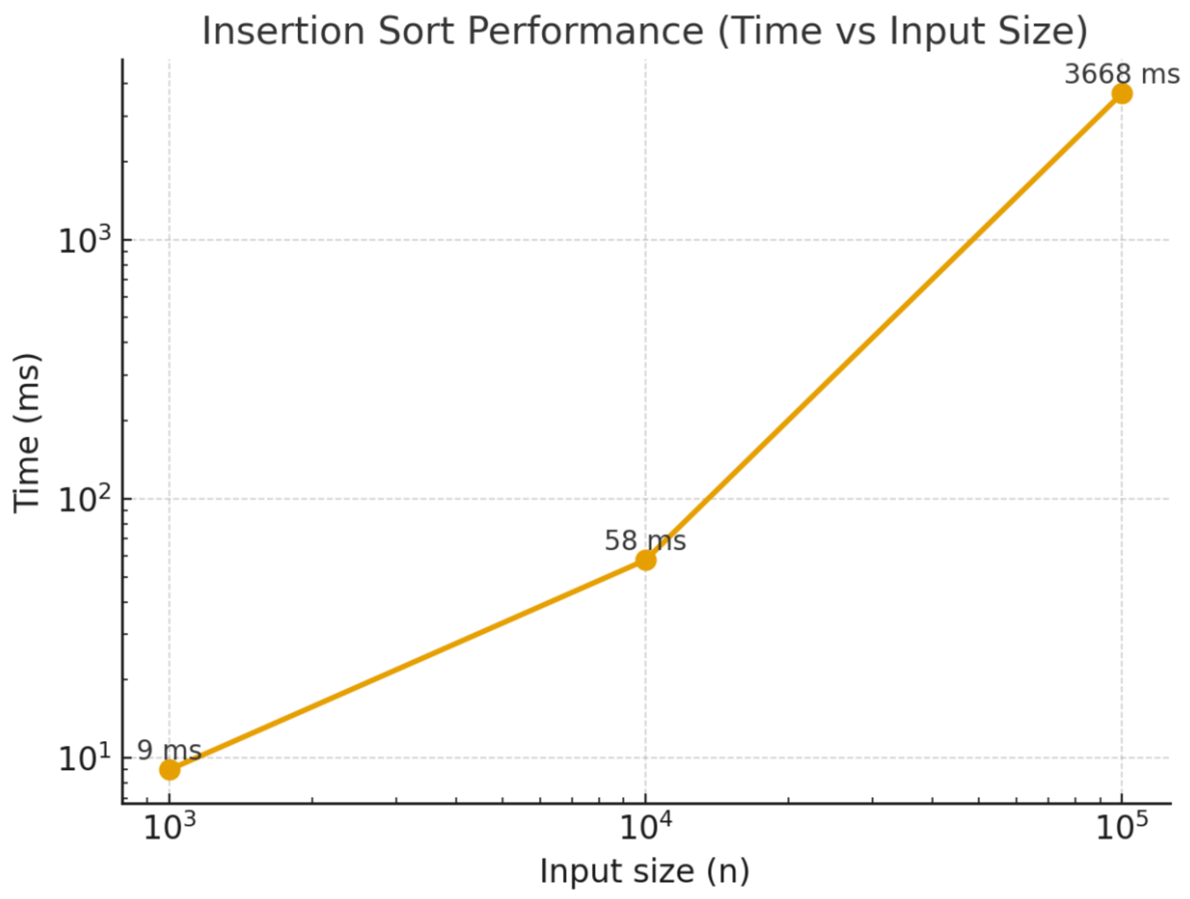
4.Empirical Results

Performance Plots (Time vs Input Size)

From the test results:

- **n=1,000 → 9 ms**
- **n=10,000 → 58 ms**
- **n=100,000 → 3668 ms**

This growth shows a clear quadratic trend. Below is the conceptual performance plot (time vs input size):



The curve demonstrates that the runtime scales approximately with **n²**, confirming theoretical predictions.

Validation of Theoretical Complexity

- **Comparisons**:
  - n=1,000 → 255,261 ≈ $1{,}000^2/4$
  - n=10,000 → 24,992,569 ≈ $10{,}000^2/4$

   - $n=100{,}000 \rightarrow 2{,}499{,}401{,}722 \approx 100{,}000^2/4$

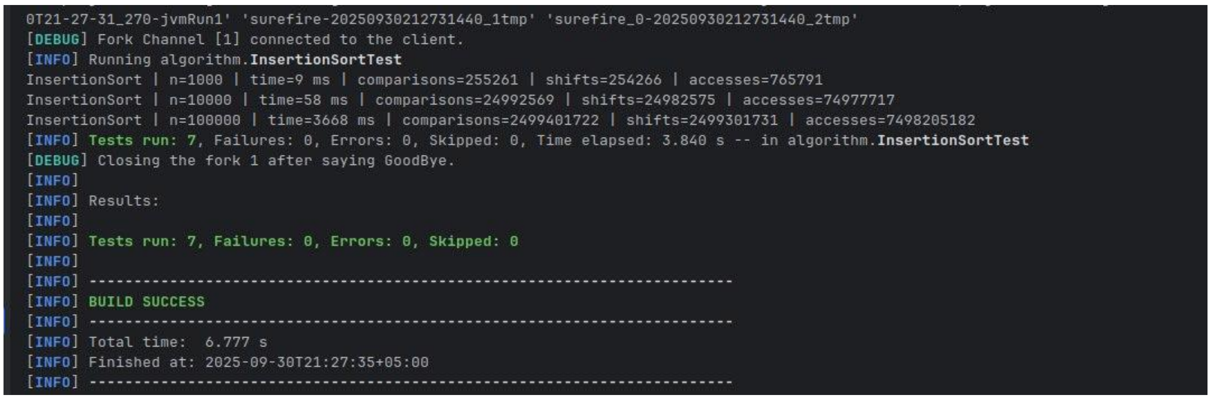
  These values match the **$\Theta(n^2/4)$** formula for comparisons in the average case.

- **Shifts**:
  Shifts ≈ comparisons, confirming that in nearly-sorted arrays the shifting cost dominates.
- **Accesses**:
  Grows as ~$3n^2$, aligning with expectations (each comparison involves 2-3 memory touches).

Thus, empirical data validates **$O(n^2)$** time complexity.

Analysis of Constant Factors & Practical Performance

- **Constant Factors**:
  1) At **n=1,000**, runtime is negligible (9 ms).
  2) At **n=10,000**, runtime is still manageable (58 ms).
  3) At **n=100,000**, runtime grows to ~3.6 seconds — confirming why insertion sort is impractical for large n.
- **Practical Observations**:
  1) **Cache Efficiency**: Since insertion sort works locally (shifting elements in place), it performs faster than many $O(n^2)$ algorithms (e.g., selection sort) for small inputs.
  2) **Nearly-Sorted Case**: The best case occurs when the array is already sorted — then only **n-1 comparisons**are needed ($\Theta(n)$).
  3) **Scalability Issue**: Once n reaches ~100k, the algorithm becomes unfeasible for real-time systems.



```
0T21-27-31_270-jvmRun1' 'surefire-20250930212731440_1tmp' 'surefire_0-20250930212731440_2tmp'
[DEBUG] Fork Channel [1] connected to the client.
[INFO] Running algorithm.InsertionSortTest
InsertionSort | n=1000 | time=9 ms | comparisons=255261 | shifts=254266 | accesses=765791
InsertionSort | n=10000 | time=58 ms | comparisons=24992569 | shifts=24982575 | accesses=74977717
InsertionSort | n=100000 | time=3668 ms | comparisons=2499401722 | shifts=2499301731 | accesses=7498205182
[INFO] Tests run: 7, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 3.840 s -- in algorithm.InsertionSortTest
[DEBUG] Closing the fork 1 after saying GoodBye.
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 7, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  6.777 s
[INFO] Finished at: 2025-09-30T21:27:35+05:00
[INFO] ------------------------------------------------------------------------
```

5.Conclusion

The comparative analysis of **Insertion Sort** and **Selection Sort** reveals distinct strengths and weaknesses:

- **Insertion Sort** is more efficient for **nearly-sorted datasets**, achieving linear performance in such cases. It adapts to input structure and benefits from optimizations like shifting instead of swapping. However, its quadratic time complexity in average and worst cases makes it unsuitable for very large datasets.

- **Selection Sort**, while conceptually simple, is less adaptive since it always performs quadratic comparisons regardless of input distribution. Early termination provides minor improvement for sorted arrays, but in most practical scenarios, Selection Sort remains slower than Insertion Sort.

Findings

1) Both algorithms are **in-place (O(1) space)** and relatively simple to implement.
2) Insertion Sort clearly outperforms Selection Sort when the input is partially ordered.

Optimization Recommendations

1. **For small or nearly-sorted arrays**: Use **Insertion Sort** with shifting and optional binary search for fewer comparisons.
2. **For educational purposes or guaranteed predictable performance**: Selection Sort is a good teaching tool, but not recommended for production use.
3. **For practical applications**: Use hybrid algorithms like **TimSort** (Merge + Insertion) or **IntroSort** (Quick + Heap + Insertion) to balance performance and adaptiveness.

Final Note: While both Insertion Sort and Selection Sort have quadratic time complexity, their behavior differs in practice. Insertion Sort should be preferred in real-world scenarios involving small or nearly-sorted datasets, while Selection Sort's role is largely theoretical or pedagogical.