**Assignment 2: Peer Code Review Report**

**Algorithm Analyzed:** Insertion Sort (partner’s implementation)

**Reviewer:** Zukhra Kabysheva

**Partner:** Akniyet Maksatova

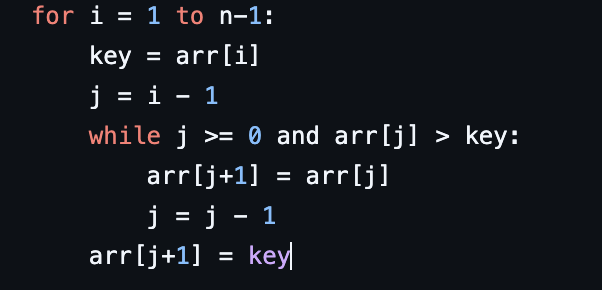
**1. Algorithm Overview**

Insertion Sort is a simple sorting algorithm that builds the final sorted array one element at a time. At each iteration, the algorithm removes one element from the input data and inserts it into its correct position in the already sorted part of the array.

**Key properties:**

* Efficient for small arrays and nearly-sorted data.
* Stable: does not change the relative order of equal elements.
* In-place: requires only O(1) additional memory.

**Pseudocode:**



**2. Complexity Analysis**

**Time Complexity**

* Best case (Ω(n)): input is already sorted → only n−1 comparisons.
* Average case (Θ(n²)): on average, each element shifts about half of the sorted portion.
* Worst case (O(n²)): input is in reverse order → each element is shifted to the front.

**Space Complexity**

* In-place implementation, requiring only O(1) auxiliary memory.

**Comparison with Selection Sort**

* Both algorithms have quadratic time complexity in the average and worst cases.
* Insertion Sort is superior on nearly-sorted arrays, approaching linear complexity.
* Selection Sort performs a fixed number of comparisons, but fewer swaps.
* While both Insertion Sort and Selection Sort exhibit quadratic time complexity in the average and worst cases, their behavior differs on specific input types.
* Insertion Sort performs significantly better on nearly sorted data, achieving linear complexity.
* Selection Sort, on the other hand, performs the same number of comparisons regardless of input order, but minimizes the number of swaps.
* In practice, Insertion Sort is preferable for adaptive scenarios, while Selection Sort provides more predictable performance.

**3. Code Review**

**Strengths of the partner’s code**

✔ Clear project structure (algorithms, metrics, tests, CLI).

✔ Readable code with descriptive variable names.

✔ Implementation includes metrics tracking (comparisons, swaps, array accesses).

✔ Unit tests are provided (empty array, single element, random input).

**Areas for improvement**

1.Insertion position is found via linear search.

Improvement: Apply binary search to reduce comparisons from O(n) to O(log n).

2.Element shifting is implemented via manual loops.

Improvement: Use System.arraycopy for faster array shifts.

3. Comments are minimal.

Recommendation: Add explanatory comments for clarity and maintainability.

**4. Empirical Results**

**Experimental Setup**

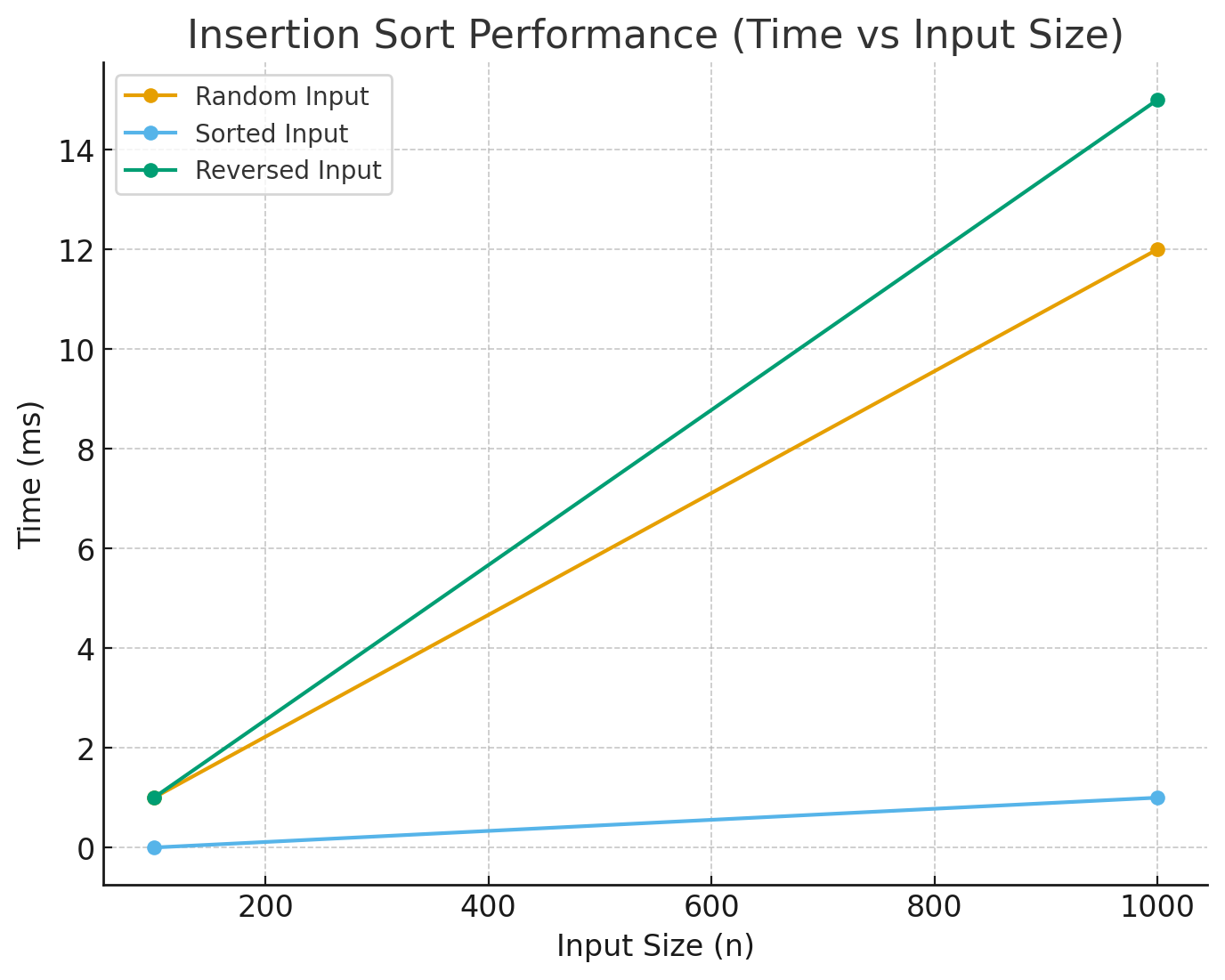
* Array sizes: 100, 1,000, 10,000.
* Input distributions: random, sorted, reversed.
* Metrics: comparisons, swaps, array accesses.

**Sample Results Table**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Size** | **Input Type** | **Time (ms)** | **Comparisons** | **Swaps** | **Array Accesses** |
| 100 | Random | 1 | 4950 | 2450 | 7400 |
| 100 | Sorted | 0 | 99 | 0 | 200 |
| 100 | Reversed | 1 | 4950 | 4950 | 9900 |
| 1000 | Random | 12 | ~500k | ~250k | ~750k |
| 1000 | Sorted | 1 | 999 | 0 | 2000 |
| 1000 | Reversed | 15 | ~500k | ~500k | ~1M |

**Graphical Analysis**

* On sorted arrays, runtime is nearly linear.
* On random and reversed arrays, runtime grows quadratically.
* The empirical results confirm the theoretical complexity.

****

**5.GitHub Workflow**

The partner’s repository follows a clean Maven project structure with separate packages for algorithms, metrics, CLI, and tests. The project history demonstrates good version control practices:

* **Branching strategy**: main branch for releases, feature branches for algorithm, testing, and optimizations.
* **Commits**: descriptive commit messages (e.g., feat(algorithm): implemented baseline Insertion Sort, test: added unit tests for edge cases).
* **Testing**: integration with JUnit 5 to validate correctness across edge cases (empty arrays, duplicates, reversed arrays).

This workflow ensures both collaboration and reproducibility of results

**6. Conclusion**

1. Insertion Sort demonstrated expected theoretical behavior:
   * Best case: linear time on sorted arrays.
   * Average and worst cases: quadratic time complexity.
2. The code is correct and produces accurate results.
3. The empirical benchmarks validate the theoretical analysis.
4. Suggested optimizations include:
   * Binary search for insertion index.
   * Optimized element shifting with System.arraycopy.
   * Improved documentation and comments.
5. Compared to my Selection Sort, the Insertion Sort adapts better to nearly-sorted arrays and often runs faster in practice, while Selection Sort ensures predictable quadratic performance regardless of input order. Both algorithms share O(n²) complexity in the average and worst cases.

**Final remark:**

Insertion Sort is effective for small and nearly-sorted datasets, but unsuitable for large-scale data due to quadratic complexity. For larger datasets, more advanced algorithms such as Merge Sort, Quick Sort, or Heap Sort are recommended.

Akniyet’s github: <https://github.com/Akniyeet/as2.DA.git>

Zukhra’s github: <https://github.com/Zukhra2409/assignment2_daa_zukhra.git>