Alym Baisal and Adilzhan Assanuly
SE-2437
Assignment 2

---

**Analysis Report: Sorting Algorithms Benchmark**

**1. Introduction**

This report presents a detailed comparative analysis of two classical sorting algorithms: **Selection Sort** and **Insertion Sort**. The focus is on their correctness, performance characteristics, and efficiency in handling arrays of varying sizes and distributions.

**2. Algorithm Descriptions**

**Selection Sort**

Selection Sort is a simple comparison-based algorithm that repeatedly selects the minimum element from the unsorted portion and places it at the beginning. It has a worst-case and average-case time complexity of $O(n^2)$ and a best-case of $O(n^2)$ when optimized.

**Key Features:**

- Iteratively selects the minimum element.
- Can be optimized to stop early if the array is already sorted.
- Stable when implemented carefully.

**Insertion Sort**

Insertion Sort builds the sorted array one element at a time by comparing the current element with the elements before it and inserting it at the correct position. It can be further optimized using binary search for insertion position.

**Key Features:**

- Efficient for nearly sorted arrays.
- Adaptive: performance improves when array is partially sorted.
- Time Complexity: $O(n^2)$ worst-case, $O(n)$ best-case.

**3. Testing Methodology**

Both algorithms were tested using JUnit test suites and custom benchmark runners. The tests covered:

- Empty arrays
- Single-element arrays
- Already sorted arrays

- Reverse-sorted arrays
- Random arrays with duplicates
- Large arrays up to 100,000 elements

Performance metrics tracked include **comparisons**, **swaps**, **array accesses**, and **execution time in milliseconds**.

**4. Performance Metrics**

| Algorithm | Array Type | Size | Time (ms) | Comparisons | Swaps | Array Accesses |
|---|---|---|---|---|---|---|
| Selection Sort | Random | 100 | 0 | 538 | 2694 | 6125 |
| Selection Sort | Sorted | 100 | 0 | 99 | 0 | 198 |
| Selection Sort | Reverse | 100 | 2 | 481 | 4950 | 10580 |
| Selection Sort | Nearly-sorted | 100 | 0 | 624 | 57 | 988 |
| Insertion Sort | Random | 1000 | 4 | 8571 | 252762 | 516094 |
| Insertion Sort | Sorted | 1000 | 0 | 999 | 0 | 1998 |
| Insertion Sort | Reverse | 1000 | 0 | 7988 | 499500 | 1008987 |
| Insertion Sort | Nearly-sorted | 1000 | 0 | 8595 | 6710 | 24026 |

(*Further large-scale benchmarks available in CSV results*)

**5. Comparative Analysis**

- **Efficiency**: Insertion Sort outperforms Selection Sort on nearly sorted and small datasets due to its adaptive behavior.
- **Worst-case scenarios**: Both algorithms exhibit quadratic time complexity; however, Selection Sort consistently performs a large number of swaps, impacting cache performance.
- **Memory Access**: Insertion Sort, especially with binary search, reduces unnecessary comparisons and array movements.
- **Scalability**: Both algorithms scale poorly for very large datasets compared to modern O(n log n) algorithms.

**6. Conclusion**

The analysis demonstrates that while both Selection Sort and Insertion Sort are easy to implement and suitable for educational purposes, **Insertion Sort provides superior performance in practice for small or nearly sorted datasets**. Selection Sort remains predictable in execution but suffers from higher swap overhead.

**Key Takeaways:**

- Use Insertion Sort for adaptive sorting and small datasets.
- Selection Sort is primarily useful in cases where simplicity and deterministic swapping are required.
- For large-scale sorting, more advanced algorithms are recommended.

---

**End of Report**