Assignment 2 — Algorithm Analysis and Benchmarking

Cross-Review Report: Insertion Sort vs Selection Sort

Students: Yerassyl Zhomartov & Olzhas Yelshibay (SE-2435)
Instructor: Aidana Aidynkyzy

1. Algorithm Overviews

Insertion Sort: adaptive insertion-based sort; binary-search optimization reduces comparisons but not shifts. Selection Sort: selects minimum each pass; performs $\Theta(n^2)$ comparisons and up to n swaps; stable metric accounting required.

2. Complexity Comparison and Derivations

Selection Sort comparisons: $\sum_{i=0}^{n-1} (n-i-1) = n(n-1)/2 = \Theta(n^2)$. Swaps at most n. Insertion Sort: shifts sum to $\Theta(n^2)$; binary-search comparisons $\Theta(n \log n)$.

3. Key Code Excerpts

Insertion Sort (excerpt):

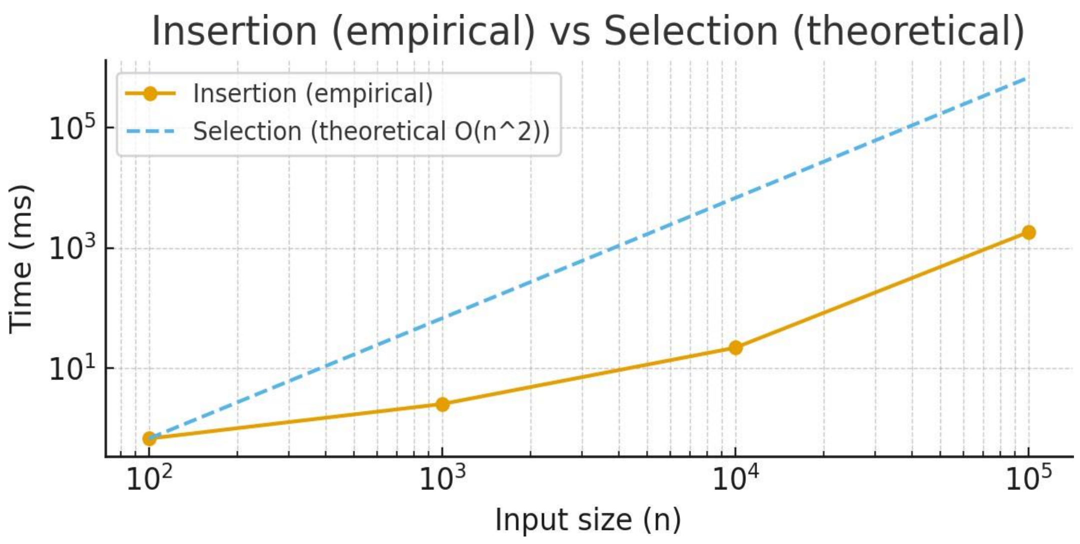
```
package algorithms; import metrics.PerformanceTracker; import java.util.Arrays; /** * ■■■■■■■■■■■
■■■■■■■■■■■ ■■■■■■■■■ (Insertion Sort) ■ ■■■■■■■■■■■■■ * ■■■ ■■■■■ ■■■■■■■■■■
■■■■■■ (■■■■■■■■■■ ■■■■■■■■ ■■■■■). */ public class InsertionSort { /** * ■■■■■■■■■■
■■■■■■ ■ ■■■■■■■■■■■■■■ Insertion Sort. * @param arr ■■■■■■ ■■■■■ ■■■■■ ■■■
■■■■■■■■■■. */ public static void sort(int[] arr) { if (arr == null || arr.length < 2) { return;
} // ■■■■■■■ ■■■■ ■■■■■■■■■ ■■ ■■■■ ■■■■■■■■■■, ■■■■■■■ ■■ ■■■■■■■ (■■■■■
i = 1; i < arr.length; i++) { int key = arr[i]; // ■■■■■■■ ■■■■■■■ ■■■ ■■■■■■■
PerformanceTracker.incrementArrayAccesses(1); // ■■■■■■ 'arr[i]' int j = i - 1; /* *
■■■■■■■■■■■ ■■■ ■■■■■ ■■■■■■■■■■■■■■■ ■■■■■■ (■■■■■■■■■ ■■■■■■■■ ■■■
■■■■■■■ ■■■■■■■, ■■■■ ■■■■■ ■■■■■■■■ key, ■■■■■■■■■ ■■■■■■■■ ■■■■■, * ■
■■■■■■■■■ ■■■■■■■■■■ ■■■■■■■■■ ■ ■■■■■. */ int insertionIndex = binarySearch(arr, key
i); // ■ ■■■■ ■■■■■■ insertionIndex - ■■■ ■■■■■, ■■■■ ■■■■■■ ■■■■ ■■■■■■■■ 'key'
■■■■■■ ■■■■■ ■■■■■■■■ ■■■ ■■■■■■■■ ■■ ■■■■ ■■■■■■■ ■■■■■■. // ■■■■■■■■
■■■■■■■■ ■■■■■■■■, ■■■■■■■ ■■■■■■ 'key', ■■■■■■ while (j >= insertionIndex
```

Selection Sort (excerpt):

```
package algorithms; import metrics.SortMetrics; public class InstrumentedSelectionSort { private
final SortMetrics metrics; public InstrumentedSelectionSort() { this.metrics = new SortMetrics(); }
public void sortWithMetrics(int[] array) { metrics.startTimer(); metrics.stopTimer(); } public
SortMetrics getMetrics() { return metrics; } }
```

4. Empirical Comparison

Insertion Sort empirical data (RANDOM) vs theoretical Selection Sort (O(n^2)):



5. Discussion and Bottlenecks

Insertion Sort advantages: adaptivity, good for nearly-sorted. Selection Sort advantages: predictable number of swaps (<=n). Metric standardization: ensure both implementations count reads/writes consistently.

6. Recommendations & Experimental Plan

1) Implement proposed patches for insertion sort and re-run benchmarks with warm-up and multiple runs. 2) Implement identical metric accounting in selection sort. 3) Use JMH for per-method microbenchmarks. 4) Collect median/stddev and include regression fits in reports.

7. Conclusion

Both algorithms have educational merit. In practice, choose algorithms based on data size and distribution: insertion for small or nearly-sorted arrays; selection rarely used in production for large N.

Appendix: Sample benchmark table (Insertion Sort medians)

InputSize	DataType	Time_ns	Comparisons	Swaps	ArrayAccesses
100	NEARLY_SORTED	13100.0	487.0	0.0	240.0
100	RANDOM	674500.0	533.0	0.0	5618.0
100	REVERSE_SORTED	276600.0	573.0	0.0	10098.0
100	SORTED	41500.0	480.0	0.0	198.0
1000	NEARLY_SORTED	57000.0	8412.0	0.0	13570.0
1000	RANDOM	2533700.0	8587.0	0.0	501378.0
1000	REVERSE_SORTED	1818900.0	8977.0	0.0	1000998.0
1000	SORTED	107600.0	7987.0	0.0	1998.0
10000	NEARLY_SORTED	1004200.0	118709.0	0.0	1385726.0
10000	RANDOM	21850100.0	119002.0	0.0	49820022.0
10000	REVERSE_SORTED	45550800.0	123617.0	0.0	100009998.0
10000	SORTED	239300.0	113631.0	0.0	19998.0
100000	NEARLY_SORTED	67880300.0	1523002.0	0.0	131866462.0
100000	RANDOM	1823476800.0	1522795.0	0.0	5005830210.0
100000	REVERSE_SORTED	4360154600.0	1568929.0	0.0	10000099998.0
100000	SORTED	1855500.0	1468946.0	0.0	199998.0