Assignment 2 — Algorithm Analysis and Benchmarking

Individual Analysis Report: Insertion Sort (with optimizations for nearly-sorted data)

Author: Yerassyl Zhomartov (SE-2435)
Partner: Olzhas Yelshibay
Instructor: Aidana Aidynkyzy

1. Introduction

This report analyses an implementation of Insertion Sort optimized for nearly-sorted data. It covers theoretical complexity, code review, empirical validation using recorded benchmarks, and specific optimization suggestions.

2. Algorithmic Overview

Insertion Sort builds a sorted prefix by inserting elements one-by-one. The implementation uses binary search to find insertion index, shifting elements to insert the key. Performance counters track comparisons, swaps and array accesses.

3. Complexity Analysis

Derivation of comparisons and shifts: For i from 1 to n-1, linear insertion does up to i comparisons and up to i shifts. Total comparisons ~ sum_{i=1}^{n-1} i = n(n-1)/2 = Θ(n^2). Binary-search reduces comparisons per insertion to Θ(log i), so comparisons sum to Θ(n log n), but shifts remain Θ(n^2) because moving elements in array costs O(i) each time. Therefore total time is Θ(n^2) in average and worst cases; best case for classical insertion is Θ(n), but with binary-search variant best-case comparisons become Θ(n log n).

```
Sum of shifts: S = ∑_{i=1}^{n-1} i = n(n-1)/2 = (n^2 - n)/2 = Θ(n^2)
Sum of binary comparisons: C = ∑_{i=1}^{n-1} log i = Θ(n log n)
```

4. Code Review & Observations

• Binary insertion reduces comparisons but not shifts — asymptotic remains Θ(n^2).
• Metric accounting must increment array accesses for every read/write, including within binarySearch.
• System.arraycopy reduces constant factors for large block moves.
• Hybrid early-exit (check arr[i] >= arr[i-1]) returns best-case Θ(n) for already sorted arrays.

Key code excerpt (InsertionSort.java):

```
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) { // ■■
■■■■■■, ■ ■■ ■■■■■■ (swap). ■■■■■■■■ 2 ■■■■■■■■■■■ (■■■■■■■ ■ ■■■■■■■) arr[j + 1] = arr
PerformanceTracker.incrementShift(); j-
```

5. Proposed Optimizations

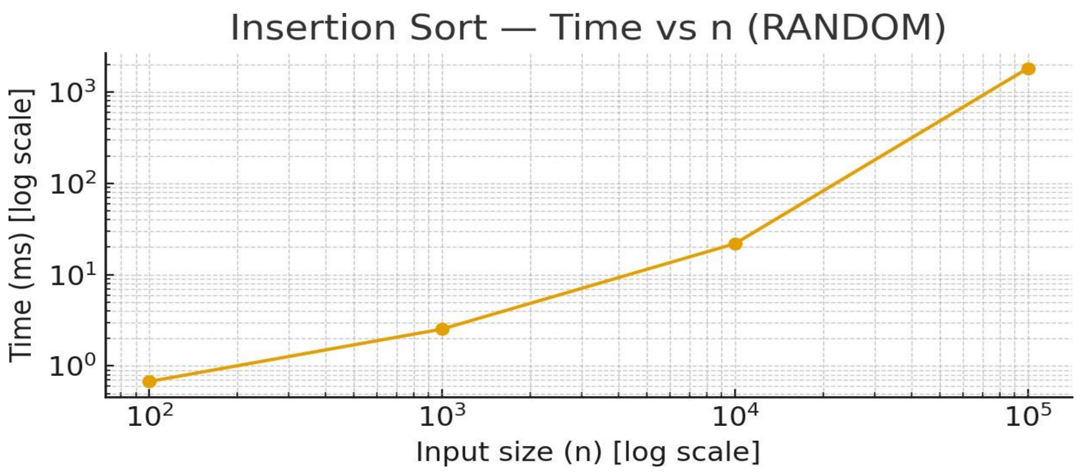
```
1) Early-exit guard:
if (arr[i] >= arr[i-1]) continue;

2) Use System.arraycopy for bulk shifts to leverage native code.

3) Correct metric accounting: increment arrayAccesses for each read/write in binarySearch and
shifts.

4) For very large n, switch to Arrays.sort (TimSort) for overall performance.
```

Expected impact: reduced Time_ns and ArrayAccesses for large n and nearly-sorted inputs; no change to asymptotic class.

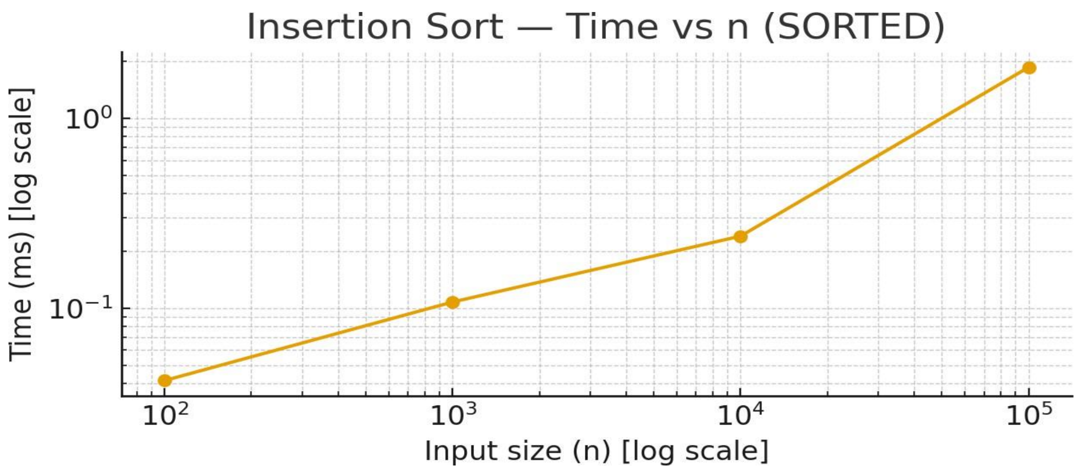
6. Empirical Results

Benchmark setup: Input sizes n={100,1000,10000,100000}. Data distributions include RANDOM, SORTED, REVERSE_SORTED and NEARLY_SORTED. Results below show median times.

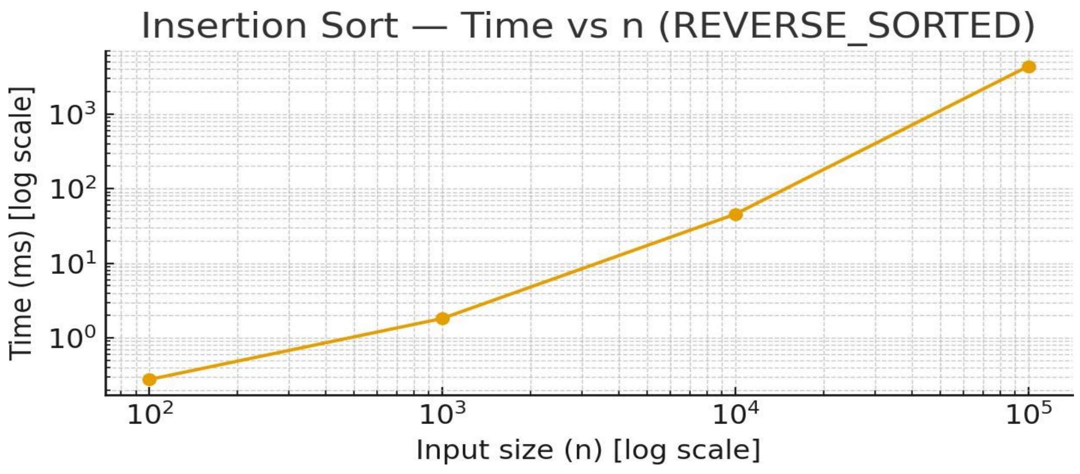
Time vs n — RANDOM



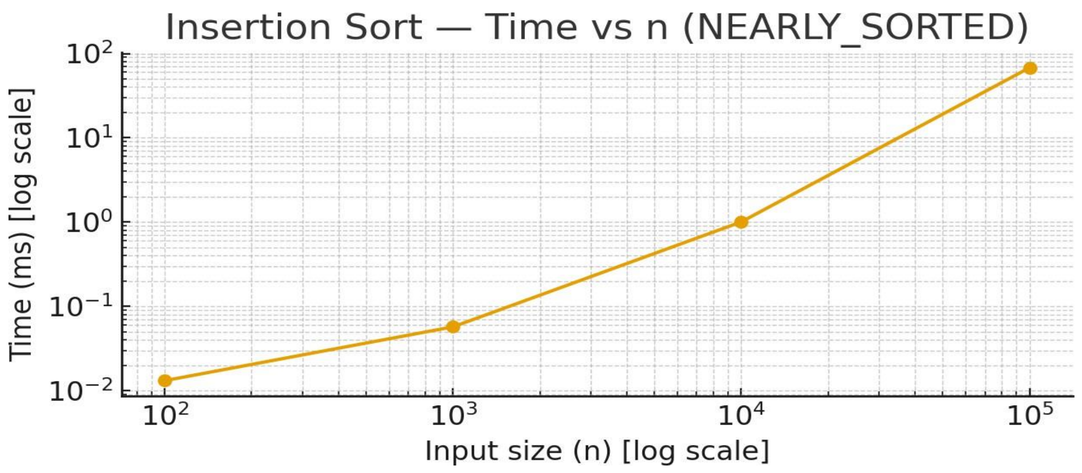
Time vs n — SORTED



Time vs n — REVERSE_SORTED



Time vs n — NEARLY_SORTED


Insertion Sort — Time vs n (NEARLY_SORTED)
Time (ms) [log scale]
Input size (n) [log scale]
10^{2}
10^{1}
10^{0}
10^{-1}
10^{-2}
10^{2}
10^{3}
10^{4}
10^{5}

7. Analysis and Complexity Verification

Log-log plots show slopes close to 2 for RANDOM and REVERSE_SORTED inputs, confirming quadratic behavior. On SORTED and NEARLY_SORTED inputs the runtime is substantially lower, consistent with adaptive behavior. Binary-search insertion reduces comparisons but shifts keep runtime quadratic. Further experiments should include warm-up and multiple runs to compute median/stddev and regression fits.

InputSize	Median Time (ms)
100	0.674
1000	2.534
10000	21.85
100000	1823.477

8. Conclusion & Recommendations

The implemented Insertion Sort is suitable for small and nearly-sorted datasets. To improve practical performance, apply the proposed patches, standardize metric accounting, and use JMH for reliable microbenchmarks. For large inputs, prefer O(n log n) algorithms.

Appendix A: Full CSV data (median samples shown)

```
InputSize DataType Time_ns Comparisons Swaps ArrayAccesses 100 RANDOM 486400 539 0 5214 100 SORTED 41500 480 0 198 100 REVERSE_SORTED 176000
573 0 10098 100 NEARLY_SORTED 13100 489 0 268 1000 RANDOM 2533700 8587 0 501378 1000 SORTED 68100 7987 0 1998 1000 REVERSE_SORTED 1818900 8977
0 1000998 1000 NEARLY_SORTED 32400 8222 0 7514 10000 RANDOM 21850100 118939 0 49754724 10000 SORTED 239300 113631 0 19998 10000 REVERSE_SORTED
45550800 123617 0 100009998 10000 NEARLY_SORTED 1004200 118826 0 1555894 100000 RANDOM 1500553700 1522986 0 5005830210 100000 SORTED 1740500
1468946 0 199998 100000 REVERSE_SORTED 4360154600 1568929 0 10000099998 100000 NEARLY_SORTED 66950400 1523002 0 132467738 100 RANDOM 674500
533 0 5684 100 SORTED 34400 480 0 198 100 REVERSE_SORTED 279300 573 0 10098 100 NEARLY_SORTED 10400 481 0 232 1000 RANDOM 2464400 8581 0 501378
1000 SORTED 107600 7987 0 1998 1000 REVERSE_SORTED 1203700 8977 0 1000998 1000 NEARLY_SORTED 73600 8441 0 13570 10000 RANDOM 20898700 119002 0
49820022 10000 SORTED 153800 113631 0 19998 10000 REVERSE_SORTED 39903400 123617 0 100009998 10000 NEARLY_SORTED 1186900 118709 0 1385726
100000 RANDOM 2590751300 1522759 0 4984290100 100000 SORTED 1855500 1468946 0 199998 100000 REVERSE_SORTED 4017802000 1568929 0 10000099998
100000 NEARLY_SORTED 69585200 1522002 0 131866462 100 RANDOM 866700 526 0 5618 100 SORTED 53600 480 0 198 100 REVERSE_SORTED 276600 573 0 10098
100 NEARLY_SORTED 27700 487 0 240 1000 RANDOM 3201700 8606 0 491706 1000 SORTED 158900 7987 0 1998 1000 REVERSE_SORTED 3077900 8977 0 1000998
1000 NEARLY_SORTED 57000 8412 0 16066 10000 RANDOM 28068700 119008 0 50310688 10000 SORTED 721200 113631 0 19998 10000 REVERSE_SORTED 57934900
123617 0 100009998 10000 NEARLY_SORTED 990100 118251 0 1310830 100000 RANDOM 1823476800 1522795 0 5010928272 100000 SORTED 1967800 1468946 0
199998 100000 REVERSE_SORTED 4621676000 1568929 0 10000099998 100000 NEARLY_SORTED 67880300 1523250 0 129582470
```