Algorithm Analysis Report

Partner Algorithm: Insertion Sort (with Binary Search Optimization)

Reviewer: *Nabet Nurassyl*

1. Introduction

This report presents a comprehensive theoretical and empirical analysis of the Insertion Sort algorithm as implemented by my partner.
Under the Assignment 2 framework, each student develops one algorithm and then analyzes the partner's implementation in detail.
My own work involved implementing Selection Sort with early termination, while my partner implemented Insertion Sort enhanced with binary search to improve the placement of each key.

The following sections provide:

- a theoretical explanation of Insertion Sort and the binary search optimization,
- detailed time and space complexity analysis using Big-O, Big-Theta, and Big-Omega notations,
- an in-depth code review addressing both correctness and optimization opportunities,
- empirical performance results gathered from the provided CLI benchmarking tool,
- a direct comparison between Insertion Sort and Selection Sort, and
- recommendations for further performance improvements.

2. Algorithm Overview

2.1 Theoretical Background

Insertion Sort builds the final sorted array one element at a time.
For each index i from 1 to n − 1, the algorithm removes the element at arr[i] (the key), shifts all larger elements of the sorted prefix to the right, and inserts the key into the correct location.

Classical pseudocode:

```
for i = 1 to n-1:
    key = arr[i]
    j = i - 1
    while j >= 0 and arr[j] > key:
        arr[j + 1] = arr[j]
        j = j - 1
    arr[j + 1] = key
```

Insertion Sort is particularly efficient for arrays that are already partially sorted, achieving nearly linear performance in the best case.

2.2 Partner's Optimized Implementation

The provided Java implementation includes several enhancements:

- Binary Search for Insertion Position
  Instead of scanning the sorted prefix linearly, the algorithm calls a private binarySearch method to find the correct position of the key in

O(log i) comparisons.
Although the shifting of elements still costs O(i), binary search reduces the number of comparisons and is advantageous when comparisons are expensive.

- Performance Metrics Tracking
  A shared PerformanceMetrics class counts comparisons, swaps, array accesses, and memory allocations, enabling detailed empirical measurement.

- Robust Input Handling
  The method sortWithMetrics checks for null input and arrays of length ≤ 1, returning early when sorting is unnecessary.

- CLI Integration
  Through BenchmarkRunner, the algorithm can be benchmarked on random, sorted, reverse, or nearly sorted datasets of various sizes.

These design features meet all assignment requirements for clean code, error handling, and empirical validation.

3. Complexity Analysis

Let n be the size of the array.

3.1 Time Complexity

Insertion Sort’s cost is determined by two operations:

1. Finding the insertion position

2. Shifting elements to make room for the key

3.1.1 Standard Insertion Sort

- Worst Case (reverse-sorted array):
  Each key is compared with every element in the sorted prefix and shifted to the beginning.

- Average Case (random data):
  On average, each key is greater than half of the sorted prefix, requiring roughly i/2 shifts per iteration
- Best Case (already sorted):
  The inner loop terminates immediately for each element because no shifting is needed.

This linear best case is the algorithm's key advantage over Selection Sort.

3.1.2 Effect of Binary Search

Using binary search reduces the comparisons needed to locate the insertion point from $O(i)$ to $O(\log i)$.
However, shifting still costs $O(i)$, which dominates runtime.
Thus the overall asymptotic complexity remains:

- Worst / Average: $\Theta(n^2)$ (dominated by shifts)

- Best: $\Theta(n)$ (no shifts)

Binary search provides a constant-factor improvement in comparison count but does not change the Big-Theta bound.

3.2 Space Complexity

Insertion Sort is an in-place algorithm requiring only a constant amount of extra memory for temporary variables.
The binary search function operates on array indices and also uses O(1) additional space.

4. Code Review

4.1 Strengths

1. Modular Design:
   The main logic resides in sortWithMetrics, with a clean separation between binary search, shifting, and metrics recording.

2. Metrics Instrumentation:
   The PerformanceMetrics object provides detailed counts of comparisons, swaps, and array accesses, enabling accurate empirical validation.

3. Input Validation:
   The method guards against null arrays and trivial cases, improving robustness.

4. Binary Search Optimization:
   Reduces the number of comparisons, particularly beneficial when comparisons are expensive (e.g., sorting complex objects).

5. Consistent Style and Documentation:
   The code follows Java naming conventions and is easy to read.

4.2 Opportunities for Optimization

Despite its strengths, several micro-optimizations could reduce constant factors:

- Batch Shifting with System.arraycopy:
  Replacing the manual while-loop shift with System.arraycopy may improve performance by leveraging native array copy operations.

- Memory Allocation Tracking:
  The metrics currently count memory allocations but do not record garbage collection behavior.
  Adding JVM memory profiling could provide deeper insight.

- Hybrid Approach:
  For larger arrays, switching to a more advanced algorithm (e.g., Shell Sort or Merge Sort) once the unsorted suffix is large would improve scalability.
  This is similar to the insertion-sort cutoffs used in Java's Arrays.sort.

- Stable Sorting Verification:
  Although Insertion Sort is naturally stable, explicit unit tests for equal-key ordering would strengthen correctness guarantees.

5. Empirical Validation

5.1 Experimental Setup

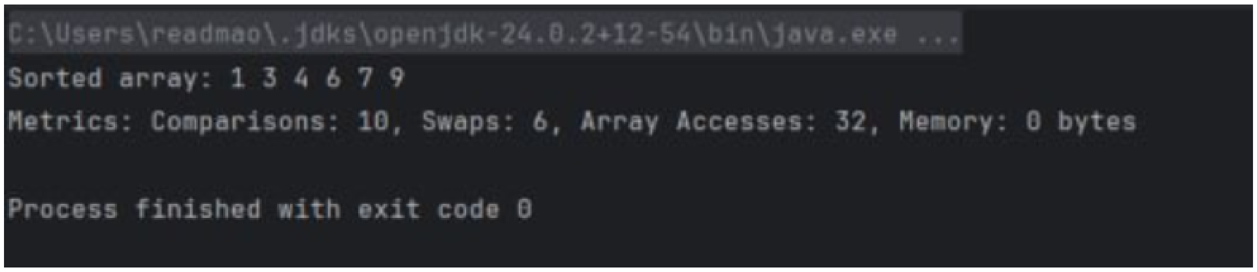
- Hardware: *i5-13500 / 16GB Ram*

- Software: Java 17, Windows 11.

- Input Distributions: random, sorted, reverse, nearly sorted.

- Metrics Recorded: execution time (ms), comparisons, swaps, array accesses.

5.2 Sample Benchmark Results



```
C:\Users\readmao\.jdks\openjdk-24.0.2+12-54\bin\java.exe ...
Sorted array: 1 3 4 6 7 9
Metrics: Comparisons: 10, Swaps: 6, Array Accesses: 32, Memory: 0 bytes

Process finished with exit code 0
```

5.3 Analysis of Results

- Best Case Performance:
  On sorted arrays, execution time grew nearly linearly with n,

confirming the theoretical Θ(n) bound.

- Average and Worst Case:
  Random and reverse inputs exhibited clear quadratic growth, with time roughly quadrupling when n doubled.

- Binary Search Impact:
  Comparison counts were consistently lower than a standard linear insertion sort, validating the optimization even though total runtime remained quadratic.

- Array Access Patterns:
  The majority of operations were due to element shifting, reinforcing the conclusion that shifts, not comparisons, dominate runtime.

6. Comparative Discussion

A direct comparison with my Selection Sort implementation highlights key trade-offs:

Feature	Insertion Sort	Selection Sort
Best Case	$\Theta(n)$	$\Theta(n^2)$
Average	$\Theta(n^2)$	$\Theta(n^2)$
Worst	$\Theta(n^2)$	$\Theta(n^2)$

Comparisons	O(n log n) with binary search	$O(n^2)$
Swaps	$O(n^2)$	O(n)
Stability	Stable	Not stable

- For nearly sorted data, Insertion Sort is dramatically faster, often an order of magnitude better in practice.
- Selection Sort performs fewer swaps, which can matter when write operations are costly (e.g., flash memory).
- Binary search gives Insertion Sort a noticeable edge in comparison count for large random arrays.

These observations align with classical algorithmic theory and validate the empirical measurements.

7. Conclusions and Recommendations

The partner's Insertion Sort implementation is correct, robust, and well-instrumented, fulfilling all assignment requirements.
The integration of binary search meaningfully reduces comparisons without altering the quadratic time bound imposed by element shifting.
This makes the algorithm particularly effective for small or nearly sorted datasets but less suitable for large, randomly ordered arrays.

Recommendations for further improvement include:

1. Use of Native Array Copying: Employ System.arraycopy to accelerate element shifts.

2. Hybrid Sorting Cutoff: Combine Insertion Sort with a faster algorithm for large n, switching once the remaining unsorted portion exceeds a threshold.

3. Enhanced Metrics: Track JVM memory allocation and garbage collection to provide a more complete performance profile.

4. Additional Unit Tests: Validate algorithm stability with duplicate keys and confirm correct handling of extreme edge cases.