**Individual Analysis Report**

**Algorithm:** Insertion Sort (with Binary Search Optimization)

**Group:** SE-2422  
**Student:** Nazerke Abzhamal

**Course:** Design and Analysis of Algorithms

**1. Algorithm Overview**

This report analyzes the implementation and performance of the **Insertion Sort** algorithm written in Java, instrumented with a *PerformanceTracker* for detailed metric collection.

Insertion Sort is a comparison-based, in-place sorting algorithm that builds a sorted portion of the array incrementally by inserting each element into its correct position relative to previously sorted elements.

In this implementation, an optimized binary search is used to find the insertion position within the sorted part, improving the comparison efficiency while maintaining stable sorting.

**Algorithm Steps:**

1. The algorithm first checks if the array is already sorted to avoid unnecessary work.
2. For each element (from index 1 to n−1):
   * A binary search locates the correct insertion position (left).
   * All elements greater than the key are shifted one position to the right.
   * The key is inserted at its correct position.

**Integration with PerformanceTracker**

The algorithm integrates tightly with the PerformanceTracker to record:

* **comparisons** — number of key comparisons
* **moves** — element shifts
* **reads** and **writes** — array access operations
* **allocations** — number of dynamic memory allocations (none occur in this algorithm)

This setup allows precise quantitative analysis of sorting behavior for different input distributions and array sizes.

**2. Complexity Analysis**

**2.1 Time Complexity**

| **Case** | **Description** | **Complexity** |
| --- | --- | --- |
| **Best Case (Ω(n))** | Input already sorted; early exit after linear verification. | Ω(n) |
| **Average Case (Θ(n²))** | Each element on average shifts through half of the sorted portion. | Θ(n²) |
| **Worst Case (O(n²))** | Reversed input; every element is inserted at the beginning. | O(n²) |

Insertion Sort builds the sorted portion of an array step by step. At each iteration, it inserts the current element into the correct position within the already sorted part. In this implementation, a **binary search** is used to locate the insertion position instead of a linear scan. This improves the number of **comparisons**, but **does not reduce data movement**, which remains the dominant cost.

#### ****Best Case — Ω(n)****

If the array is already sorted, the algorithm performs a quick pre-check and exits early.  
Only a single pass is made, performing one comparison per element:

* Comparisons ≈ n
* Moves = 0
* Time complexity: **Ω(n)**

#### ****Average Case — Θ(n²)****

For random input, every new element is inserted somewhere in the middle of the sorted portion.  
On average, half of the elements must be shifted for each insertion:

Even though the binary search reduces comparisons to **Θ(n log n)**, the number of shifts dominates, resulting in total **Θ(n²)** time.

#### ****Worst Case — O(n²)****

For reversed input, each element must move all the way to the start of the array:

This is the maximum possible amount of data movement, making the overall time complexity **O(n²)**.

#### ****Recurrence Relation****

Insertion Sort’s total cost can be expressed as:

Solving this gives **T(n) = O(n²)**, confirming the quadratic behavior in both average and worst cases.

**2.2 Space Complexity**

Insertion Sort is **in-place**: it requires O(1) auxiliary variables and no additional arrays.  
Total memory usage = Θ(n) (for the array) + O(1) (for local vars).

**2.3 Recurrence Relation**

Formally, the iterative process can be expressed as:

T(n)=T(n−1)+O(n)T(n) = T(n-1) + O(n)T(n)=T(n−1)+O(n)

Solving this recurrence yields:

T(n)=O(n2)T(n) = O(n^2)T(n)=O(n2)

confirming quadratic time growth in the general case.

**3. Code Review and Optimization**

**3.1 Strengths**

* **Efficient sortedness check:** The algorithm skips sorting entirely if the input is already sorted, optimizing best-case performance to linear time.
* **Binary search integration:** Reduces unnecessary comparisons compared to traditional linear insertion search.
* **Clean metric instrumentation:** The PerformanceTracker records detailed operation counts, allowing fine-grained empirical analysis.
* **Stable sorting:** The algorithm maintains input order for equal elements.
* **No recursion or extra memory:** Ensures predictable runtime and minimal overhead.

**3.2 Weaknesses**

* **Partial undercount of reads:** Some repeated reads within the binary search and shifting loops are not counted individually, slightly underestimating access metrics.
* **Redundant cmp() call before sortedness check:** Each iteration performs a comparison even if the array is already determined sorted; this can be optimized.
* **Limited scalability:** For very large datasets (n > 100,000), quadratic time growth becomes impractical compared to O(n log n) algorithms such as HeapSort or MergeSort.
* **Missing allocations tracking:** While no dynamic allocations occur, the metric allocations remains unused and could be omitted for clarity.

**3.3 Optimization Recommendations**

* **Cache repeated reads:** Use local variables for arr[mid], arr[j], etc., to reduce redundant read operations and improve metric precision.
* **Combine sortedness detection and insertion phase:** Integrate early-stop logic into the first few insertion iterations to minimize overhead.
* **Parallel benchmarking:** Compare results with built-in Arrays.sort() to visualize empirical performance differences.
* **Add JMH microbenchmarks:** To measure per-operation latency and constant factors under JVM optimization.

| **Aspect** | **Recommendation** | **Expected Impact** |
| --- | --- | --- |
| **Time** | Cache values of arr[mid] and arr[j] to reduce redundant reads. | ↓ reads by 10–15 % |
| **Time** | Combine early-exit logic into insertion loop to skip redundant sortedness pass. | ↓ comparisons slightly |
| **Space** | Maintain in-place approach; no extra arrays required. | — |
| **Maintainability** | Modularize binary-search section into helper method. | ↑ readability |
| **Benchmarking** | Add JMH microbenchmarks for per-operation cost validation. | ↑ empirical granularity |

These optimizations preserve algorithmic complexity but improve execution efficiency and metric accuracy.

### ****4. Empirical Validation****

Benchmarks were executed for n = 100, 1 000, 10 000, 100 000 under four input patterns.  
Each experiment was repeated 3 times; averages are reported below.

#### ****4.1 Aggregated Results****

| **n** | **Distribution** | **Avg Time (ns)** | **Avg Comparisons** | **Avg Moves** | **Avg Reads** | **Avg Writes** |
| --- | --- | --- | --- | --- | --- | --- |
| 100 | **Sorted** | 767 | 99 | 0 | 198 | 0 |
| 100 | **Random** | ~182 k | ~555 | ~2480 | ~630 | ~2580 |
| 100 | **Reversed** | ~5 100 | 481 | 4 950 | 580 | 5 050 |
| 100 | **Nearly** | ~3 000 | 550 | 627 | 650 | 730 |
| 1 000 | **Sorted** | 1 000 | 999 | 0 | 1 998 | 0 |
| 1 000 | **Random** | ~1 040 000 | 8 590 | 246 000 | 9 590 | 248 000 |
| 1 000 | **Reversed** | ~370 000 | 8 000 | 499 500 | 9 000 | 500 000 |
| 1 000 | **Nearly** | ~133 000 | 8 600 | 62 000 | 9 600 | 64 000 |
| 10 000 | **Sorted** | ~5 000 | 9 999 | 0 | 19 998 | 0 |
| 10 000 | **Random** | ~21 000 000 | 118 970 | 2.50 × 10⁷ | 128 970 | 2.50 × 10⁷ |
| 10 000 | **Reversed** | ~31 000 000 | 113 630 | 5.00 × 10⁷ | 123 630 | 5.00 × 10⁷ |
| 10 000 | **Nearly** | ~4 300 000 | 119 050 | 5.90 × 10⁶ | 129 050 | 5.91 × 10⁶ |
| 100 000 | **Sorted** | ~38 000 | 99 999 | 0 | 199 998 | 0 |
| 100 000 | **Random** | ~1.55 × 10⁹ | 1.52 × 10⁶ | 2.49 × 10⁹ | 1.62 × 10⁶ | 2.49 × 10⁹ |
| 100 000 | **Reversed** | ~3.2 × 10⁹ | 1.47 × 10⁶ | 4.99 × 10⁹ | 1.56 × 10⁶ | 5.00 × 10⁹ |
| 100 000 | **Nearly** | ~3.8 × 10⁸ | 1.52 × 10⁶ | 5.79 × 10⁸ | 1.62 × 10⁶ | 5.79 × 10⁸ |

(Values are averaged across trials.)

#### ****4.2 Trend Analysis****

* **Time Growth:** Roughly proportional to n² for unsorted inputs, confirming theoretical O(n²) behavior.
* **Input Impact:**
  + Sorted — fastest; linear performance.
  + Reversed — slowest; maximum shifts.
  + Nearly-sorted — significantly faster than random, benefiting from early stability.
* **Metric Correlations:** moves ≈ writes ≈ dominant factor; time correlates strongly with them.
* **No allocations** detected → memory-efficient in-place execution.

#### ****4.3 Complexity Verification****

Plotting **time vs n²** yields nearly linear curves for random and reversed inputs, confirming O(n²) complexity.  
Plotting **time vs n** for sorted inputs produces linear scaling, verifying Ω(n) best-case behavior.

#### ****4.4 Optimization Impact**** The optimized build (binary insertion + early-exit + read-caching) produces consistent, measurable speedups across most input distributions while keeping the algorithm in-place and preserving its asymptotic behavior. The measured before/after numbers below are taken from your benchmark CSV and summarized from the uploaded document.

## Detailed results (selected points)

### Random input (average case)

| **n** | **Baseline (ns)** | **Optimized (ns)** | **Improvement** |
| --- | --- | --- | --- |
| 100 | ~990,700 → 236,700 | **≈ 4.2× faster** (≈ **76.1%** reduction) | comparisons 597 → 533 (−10%). |
| 1,000 | ~5,171,800 → 990,000 | **≈ 5× faster** (≈ **80.9%** reduction) | comparisons 9,537 → 8,593 (−10%). |
| 10,000 | ~31,317,400 → 20,656,200 | **≈ 1.5× faster** (≈ **34.0%** reduction) | comparisons 128,852 → 118,981 (−8%). |
| 100,000 | ~1.86×10⁹ → 1.49×10⁹ | **≈ 20% faster** (≈ **19.9%** reduction) | comparisons 1,622,311 → 1,522,620 (−6%). |

**Interpretation.** biggest relative wins appear for small-to-moderate n (where micro-optimizations and cache effects dominate); improvements remain clearly measurable even at n=100k. The comparisons count declines by ~5–10% across random inputs, confirming the binary search is reducing comparison work.

### Sorted input (best case)

* The optimization does **not** hurt the best-case behavior. Comparisons remain n−1 and moves = 0. For larger n the optimized run is often noticeably faster (early-exit avoids unnecessary checks). Example observations from the CSV: for n up to 100,000 the optimized times are equal or much faster, while comparisons/moves are ideal (comparisons = n−1, moves = 0).

### Reversed input (worst case)

| **n** | **Baseline (ns) → Optimized (ns)** | **Change** |
| --- | --- | --- |
| 100 | ~5,400 → 4,900 | ~10% faster; comparisons 579 → 481. |
| 1,000 | ~440,000 → ~445,000 | ≈ same (no meaningful improvement). |
| 10,000 | ~35,000,000 → ~31,000,000 | ~10–15% faster; comparisons 123,630 → 113,632. |
| 100,000 | ~4.7×10⁹ → ~3.5×10⁹ | ~25% faster; comparisons 1,568,945 → 1,468,947. |

**Interpretation.** worst-case gains are moderate but present; binary search slightly reduces comparisons even when many shifts are required.

### Nearly-sorted (optimization case)

| **n** | **Baseline (ns) → Optimized (ns)** | **Change** |
| --- | --- | --- |
| 100 | ~6,000 → ~2,300 | ~3× faster; comparisons 561 → 533 (−5%). |
| 1,000 | ~63,000 → ~60,000 | roughly same (stable). |
| 10,000 | ~6,703,200 → ~4,248,700 | ≈1.6× faster; comparisons 128,975 → 119,046 (−8%). |
| 100,000 | ~4.16×10⁸ → ~3.65×10⁸ | ~12% faster; comparisons 1,622,568 → 1,522,260 (−6%). |

**Interpretation.** nearly-sorted inputs are where early-exit + binary search shine: the optimizer avoids work on already-ordered runs and finds insertion positions with fewer comparisons.

## Metrics summary & cause analysis

* **Time (time\_ns):** decreased substantially in most cases (document reports typical reductions ranging from ~20% up to 80% in many measurements).
* **Comparisons:** decreased by roughly **5–10%** on average (binary search replaces linear search for insertion position).
* **Moves:** essentially **unchanged** (insertion sort still requires the same element shifts).
* **Reads/Writes:** small decreases due to removing redundant reads (read-caching improves locality).
* **Allocs:** = 0 — algorithm remains fully in-place.

**5. Conclusion**

The **Insertion Sort (Binary Search version)** implementation accurately represents the theoretical properties of the algorithm, while integrating detailed performance instrumentation for empirical study.

**Key Findings**

* The algorithm demonstrates expected time complexities:
  + Best case: Ω(n)
  + Average/Worst case: Θ(n²)
* Binary search optimization successfully reduces comparison count, though element shifting dominates total cost.
* The implementation is stable, in-place, and memory-efficient (O(1) extra space).
* PerformanceTracker integration provides reliable data for comparative analysis.

**JVM-Level Performance Expectations**

Under JMH or CLI benchmarks, per-operation cost remains constant for small n, but overall execution time grows quadratically with input size.  
JIT (Just-In-Time) compilation stabilizes runtime behavior quickly, producing consistent average times per element insertion.

**Recommendations for Future Work**

1. Improve read/write metric precision.
2. Extend benchmarking to include adaptive versions (e.g., binary insertion + early exit detection).
3. Compare empirically against more efficient algorithms such as MergeSort and HeapSort.
4. Visualize time vs n² plots to clearly demonstrate quadratic growth.

**Overall Conclusion**

The implemented Insertion Sort algorithm is correct, efficient within its theoretical limits, and instrumented with fine-grained performance metrics. Despite inherent quadratic complexity, its optimized binary search and early-sorted check make it ideal for small or nearly-sorted datasets.  
Empirical behavior matches theoretical predictions, validating both algorithmic design and metric integration.