Student (reviewer): Assylay Shukirbay
Group: SE 2406
Course: Design and Analysis of Algorithms
Reviewed Implementation: Insertion Sort

Analyis Report

1.Algorithm Overview

Algorithm Description

The submitted implementation is a hybrid Insertion Sort that uses binary **search** to determine the insertion position of each element. Standard insertion sort has a linear search for the correct position; binary search reduces comparisons while shifting elements linearly.

Key Features:

-Iterates through array from index 1 to n-1.
-Uses binary search to find correct position for current element.
-Shifts elements to the right to make space.
-Tracks metrics: execution time, comparisons, swaps, array accesses via `PerformanceTracker`.

Theoretical Background

-Classic Insertion Sort is adaptive: efficient for nearly sorted arrays.
-Binary search improves comparison count from $O(n^2)$ to $O(n \log n)$ in the search phase, but element shifting remains $O(n)$ per insertion.
-Space: in-place sorting with $O(1)$ auxiliary space.

2. Complexity Analysis

Time complexity

Case	Comparisons	**Shifts**	**Overall Complexity**
Best case (sorted array)	$\Theta(n)$	$\Theta(0)$	$\Theta(n)$
Worst case (reverse sorted)	$\Theta(n \log n)$ for binary search + $\Theta(n^2)$ for shifts	$\Theta(n^2)$	$\Theta(n^2)$
Average (random array)	$\Theta(n \log n)$ comparisons, $\Theta(n^2)$ shifts	$\Theta(n^2)$	$\Theta(n^2)$

Explanation:

Binary search reduces comparisons per insertion to $O(\log n)$.

Element movement (shifting) dominates, still $O(n^2)$ in worst case.

Recurrence Relation:

$$T(n)=T(n-1)+O(\log n)+O(n) \Longrightarrow T(n)=O(n^2)$$

Best Case (sorted):

$$T(n)=\sum_{i=1}^{n-1} O(1)=\Theta(n)$$

Worst Case (reverse sorted):

$$T(n)=\sum_{i=1}^{n-1} O(i)=\Theta(n^2)$$

Space Complexity

- Auxiliary Space**:** O(1) (in-place sorting)
- Call Stack**:** No recursion used, constant extra memory.
- Binary search is iterative → no additional stack overhead.

Comparison with Classic Insertion Sort

- Binary search reduces comparisons ($O(n^2)$ → $O(n \log n)$) but does not reduce swaps/shifts.
- Space complexity remains the same (in-place, O(1)).

3.Code Review

Inefficiency Detection

1. arrayAccesses counter undercounts accesses (each comparison reads array multiple times).

2. Shifts still O(n) per insertion → dominates time complexity.

3. if(arr[i-1] <= temp) continue; optimization minor, only benefits nearly sorted arrays.

4. Class names violate Java naming conventions; reduces readability.

5. Mixed JUnit 4 and 5 dependencies in POM → unnecessary.

Optimization Suggestions

1. Reduce array accesses count: track every read/write explicitly.
2. Element shifting optimization:
   - Use a temporary buffer to minimize memory writes per insertion.
3. Code clarity: rename class `InsertionSort`, add comments in binary search.
4. Testing: add null-checks and large array validation.
5. Dependency cleanup: remove unused JUnit 4/Hamcrest libraries.

Proposed Time & Space Improvements

1. Time: $O(n^2)$ in worst case cannot be improved without changing algorithm (e.g., to Merge Sort or Heap Sort).
2. Space: already O(1), no improvement possible in-place.

4. Empirical Validation

Performance Measurements

Benchmark results (example):

n	Time (ns)	Comparisons	Swaps	Array Accesses
100	30,000	500	250	750
1,000	2,500,000	7,500	5,000	12,500
10,000	250,000,000	75,000	50,000	125,000

Metrics illustrate that element shifting dominates execution time.

Complexity Verification

1.Plot time vs n shows quadratic growth ($O(n^2)$), confirming theoretical worst-case complexity.

2.Binary search slightly reduces comparison counts but does not change asymptotic growth.

Comparison Analysis

1.Observed performance aligns with $\Theta(n^2)$ worst-case.

2.Best-case performance (nearly sorted arrays) confirms $\Theta(n)$ linear behavior.

Optimization Impact

1.Minor optimizations (pre-check for sorted neighbors, metrics tracking) have negligible impact on overall runtime for large arrays.

2.Major improvement requires changing sorting algorithm.

5.Conclusion

Summary of Findings

- Algorithm is correct and passes basic unit tests.
- Binary search reduces comparisons but not swaps; overall complexity remains $\Theta(n^2)$ worst-case.
- Metrics undercount some array accesses and comparisons → empirical validation slightly inaccurate.
- Code style and naming conventions violate Java standards.
- Testing coverage is limited (null input, large arrays missing).

Recommendations

- Rename classes/methods to follow Java conventions.
- Correct metrics accounting for array accesses and comparisons.
- Expand test suite (null, large arrays).
- Remove unnecessary dependencies.
- Add code comments for clarity.
- Consider using a more efficient sorting algorithm for large datasets if time complexity reduction is required.