Peer Analysis Report: Insertion Sort Implementation

Project Author: Aldiyar Zhangabyl
Reviewer: Rafael Shayekhov
Date: October 5, 2025
1. Algorithm Overview
The project under analysis is an efficient implementation of the Insertion Sort algorithm in Java. The project organization is clear and segmented into meaningful modules:
algorithms/InsertionSort.java: Holds the algorithm's core logic.
metrics/PerformanceTracker.java: A performance metrics tracker and collector class.
cli/BenchmarkRunner.java: A command-line utility for running empirical tests.
test/java/algorithms/InsertionSortTest.java: A comprehensive set of unit tests.
One of the most interesting aspects of this implementation is taking two implementations of the algorithm within the InsertionSort.java class to facilitate comparative analysis:
traditionalSort (Traditional Implementation): The classic implementation of Insertion Sort. The algorithm iterates over the array, and for each element, it performs a backward linear scan of the sorted portion to get the correct point of insertion, shifting elements one by one.
sort (Improved Implementation): Improved for enhanced performance. The below optimizations are included:
Pre-sort Check: There is a check at the beginning using isAlreadySorted() that in O(n) time determines if the array is already sorted or not. If it is, the function immediately returns, specifically handling the best-case scenario.
Binary Search for Position: Instead of a linear search (O(i)) to find the insertion point, this implementation uses a binary search (binarySearchPosition), reducing the number of comparisons from O(i) to O(log i) per element i.
Efficient Element Shifting: To shift elements, the highly efficient native method System.arraycopy is used, which shifts an entire block of data in a single operation, not element-by-element shifting within a loop.
2. Asymptotic Complexity Analysis
Time Complexity
traditionalSort() Implementation:
Best Case: $\Theta(n)$
Explanation: When the array is sorted already. The outer loop runs n-1 times. Inner while loop condition (array[j] > key) is always false, therefore one comparison is run per outer loop iteration.
Notation: $O(n)$, $\Omega(n)$, $\Theta(n)$.
Worst Case: $\Theta(n^2)$.
Justification: When the array is in reverse order. The inner while loop for each i-th element needs to compare and shift all i elements preceding it. The operations are counted by the arithmetic series 1 + 2 +. + (n-1), which is n(n-1)/2, i.e., $\Theta(n^2)$.

Notation: $O(n^2)$, $\Omega(n^2)$, $\Theta(n^2)$.
Average Case: $\Theta(n^2)$
Justification: In a randomly sorted array, the i-th element will need to be compared, on average, to half of the i previous elements. The overall complexity remains proportional to n^2.
Notation: $O(n^2)$, $\Omega(n^2)$, $\Theta(n^2)$.
sort() (Optimized) Implementation:
Best Case: $\Theta(n)$
Justification: When the array is sorted already. The isAlreadySorted() function takes $O(n)$ and also ends prematurely.
Worst Case: $\Theta(n^2)$
Justification: With a reverse-sorted array. Although the binary search has $O(\log i)$ to find the insertion point, the overall expense is shifting i elements with System.arraycopy, in $O(i)$ time. All the shift time $1 + 2 +. + (n-1)$ still provides us $O(n^2)$ complexity. The optimization reduces comparisons but nothing changes the data movement quadratic complexity.
Notation: $O(n^2)$, $\Omega(n^2)$, $\Theta(n^2)$.
Average Case: $\Theta(n^2)$
Explanation: As in the worst scenario, data shifting operations control the execution and cause quadratic complexity.
Space Complexity
Auxiliary Space: $\Theta(1)$
Explanation: The algorithm is "in-place," as all operations are performed within the input array (or its replica). Additional memory used by variables such as key, j, insertionPos is not dependent on the input size n. Hence, the auxiliary space complexity is $O(1)$.
Note: The sort() and traditionalSort() functions generate a copy of the input array (array.clone()), thus $O(n)$ total memory usage. This is a proper practice so as not to alter the user's original array. Nevertheless, the auxiliary space of the sorting algorithm is $\Theta(1)$.
Recurrence Relations
Insertion sort is an iterative algorithm, and therefore its complexity is more appropriately represented in terms of summations.
But for the binary search portion (binarySearchPosition) of the optimized code, a recurrence relation can be stated: $T(k) = T(k/2) + c$, where k is the size of the subarray. According to the Master Theorem, this reduces to $T(k) = \Theta(\log k)$.
3. Code Review and Optimization
Code Quality
The code quality in the project is extremely good.
Style and Readability: The code is clean, well-formatted, and has complete English comments. The variable and method names clearly describe their purpose.
Structure: The project is organized well into packages (algorithms, metrics, cli) according to the separation of concerns principle.
Testing: The InsertionSortTest.java is an excellent example of testing. It uses JUnit 5 and covers all the main cases: empty array, one-element array, sorted and reverse-sorted array, duplicates, negative numbers, and null handling.
Inefficiency Detection

The project author already found and removed the main bottleneck of traditional insertion sort—the linear search for the insertion point.
Remaining Shortcoming: The inherent limitation of any insertion sort is the requirement to really shift elements to make space. Although System.arraycopy is very efficient, the action cannot break the quadratic complexity of data movement in the average and worst cases. This is not a bug in the implementation but a natural attribute of the algorithm itself.
Suggestions for Improvement
Time Complexity: The implementation itself for this algorithm is already close to optimal. To achieve asymptotically better complexity (for example, $O(n \log n)$), a different algorithm such as Merge Sort or Quick Sort would be necessary.
Space Complexity: The algorithm is already optimal in terms of auxiliary memory space used ($\Theta(1)$) and there is no room for improvement here.
4. Empirical Validation
The utility BenchmarkRunner.java is an ideal one to use for empirical verification.
Performance Measurement: BenchmarkRunner allows testing the algorithm against various data sizes and distributions ("random", "sorted", "reverse", "nearly_sorted"). A "warm-up" time before measurement is a Java best practice as it allows the JVM to perform JIT compilation and optimization, leading to improved measurements.
Complexity Verification: Running BenchmarkRunner generates a CSV file (benchmark-results.csv), whose information can be easily visualized.
Expected Plots: A graph of Execution Time vs. Size n should display:
A clear parabolic curve (confirming $\Theta(n^2)$) for the "random" and "reverse" cases.
An almost linear (confirming $\Theta(n)$) for the "sorted" case.
For the "nearly_sorted" case, the graph for the optimized version will be close to linear, while the original version's will be far steeper.
Analysis of Results: All data collected will confirm in complete detail the theoretical analysis. As a demonstration, doubling n must double the execution time for the "sorted" case and roughly fourfold for the "reverse" case ($(2n)^2 = 4n^2$). Impact of Optimization: Data from PerformanceTracker will facilitate quantitative measurement of the improvements.
The time for comparisons (comparisons) for the optimized solution will be an order of magnitude less because of binary search.
The time for swaps (swaps) for the optimized solution will be 0 since System.arraycopy is not a swap operation.
The overall time (elapsedTime) will be considerably less for the optimized solution on almost sorted data.
5. Conclusion
This project is a great example of an engineering approach to algorithm analysis. Not only is the Insertion Sort code correct, but it is well-designed and optimised. The author demonstrates a flawless understanding of asymptotic complexity by providing two versions of the algorithm and building a full set of tools for their analysis (PerformanceTracker, BenchmarkRunner, JUnit tests).
The theoretical analysis is reflected beautifully in the code structure, and empirical verification is easy with the given tools. The key take-home here is that while "clever"

optimizations (like binary search) can significantly improve performance by reducing comparisons, they cannot change the fundamental quadratic complexity of data movement in the average case and the worst case.