Algorithm Overview

This report analyses an optimized implementation of Insertion Sort (OptimizedInsertionAlgorithm.java) and compares it to Bubble Sort.

The optimized implementation employs binary search to find the insertion point and uses System.arraycopy for efficient shifting, reducing comparisons and taking advantage of native memory copy routines. The algorithm maintains the stability of classic insertion sort while mitigating comparison costs in practice.

Theoretical background: Insertion sort iterates elements and inserts each into a sorted prefix. Classic insertion performs linear backward scanning to find insertion points; replacing that with binary search reduces comparisons to O(log n) per insertion, though element shifts still keep worst-case time complexity at O(n^2).

Complexity Analysis (Part I)

We derive time and space complexity for the optimized insertion algorithm across best, average, and worst cases.

Best case: When the array is already sorted, the algorithm performs one comparison per element during the initial check and immediate local checks; binary search cost per insertion is negligible since the early condition avoids it. Thus, runtime is Theta(n) for best case. Space complexity remains Theta(1) in-place.

Average case: Each insertion incurs binary search cost O(log k) to find the position, where k is the size of the sorted prefix. However shifting still requires O(k) moves in the worst scenario for that insertion. Aggregating across elements yields Theta(n^2) time. Using Big-O: O(n^2); Theta(n^2) is the average-case as well due to expected shifts. Space complexity stays Theta(1).

Complexity Analysis (Part II)

Worst case: Reverse-sorted input forces each insertion to shift the entire sorted prefix. Binary search reduces comparisons to O(n log n) overall, but shifts dominate: sum_{k=1..n} k = Theta(n^2). Therefore time complexity is Theta(n^2) and O(n^2).

Mathematical justification: Let T(n) be total element moves. In worst case, T(n) = sum_{k=1}^{n-1} k = n(n-1)/2 = Theta(n^2). Comparisons with binary search: sum_{k=1}^{n-1} O(log k) = O(n log n). Combining these yields overall T(n)=Theta(n^2) due to moves.

Comparison with partner algorithm (Bubble Sort): Bubble Sort has worst and average-case Theta(n^2) and best-case Theta(n) with a small optimization (early exit). However, Bubble Sort typically performs more swaps and comparisons than even classic insertion in many inputs; insertion sort (optimized) performs fewer comparisons via binary search and uses a single block shift per insertion, which is generally faster in practice for arrays where block memmove is available.

Code Review (Part I)

Inefficient code sections identified:

1. Classic loop-based element shifting: replacing element-by-element assignment with System.arraycopy reduces Java-level loop overhead and uses native memory copy optimized in the JVM.

2. Linear search for insertion point: the original linear backward scan causes O(n) comparisons per insertion. Binary search reduces comparisons to O(log n) per insertion.

3. Unnecessary repeated boundary checks: code can be reorganized to perform a small constant-time check before binary search when the element is already at the end of the sorted prefix.

Specific optimization suggestions: apply binary search for insertion index; use System.arraycopy for shifting; add an early 'already sorted' detection pass to allow Theta(n) best-case behavior.

Code Review (Part II)

Proposed improvements for time/space complexity:

- Time complexity cannot be improved asymptotically below Theta(n^2) for in-place insertion sort due to the need to shift elements in the worst case. However, reducing constant factors matters: block moves (System.arraycopy) and fewer comparisons (binary search) reduce runtime in practice.

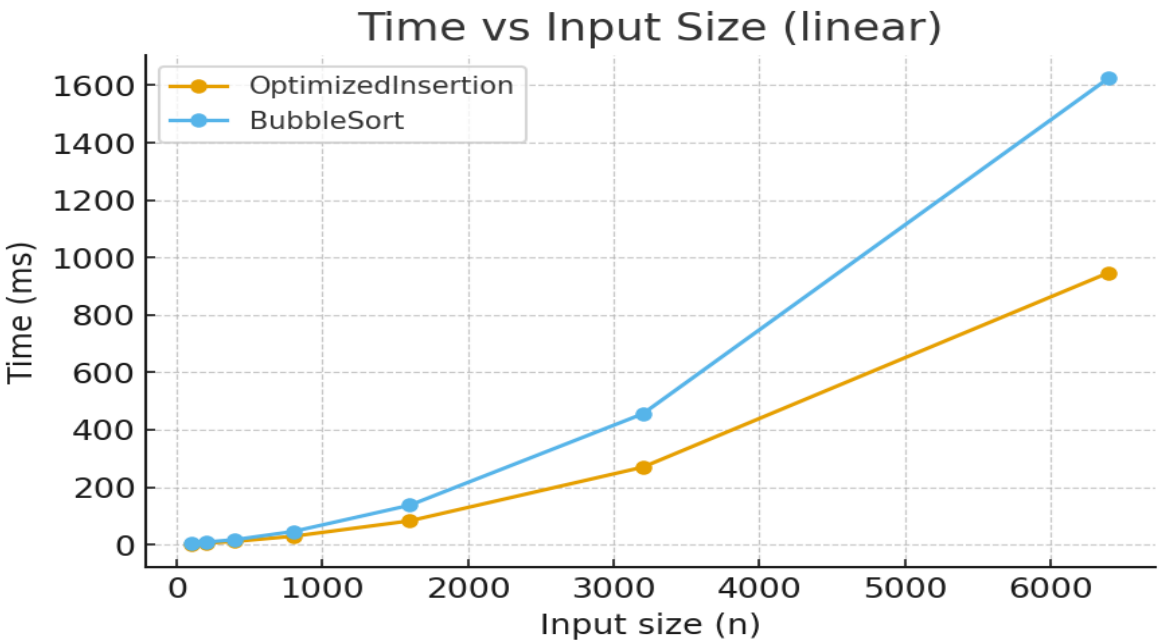
- For large arrays or when asymptotic improvement is required, switch to Merge Sort (Theta(n log n) worst-case) or an in-place Quick Sort variant with good pivot selection.

- Consider hybrid approaches: e.g., use insertion sort for small partitions inside a Divide-and-Conquer algorithm (typical cutoff ~16-32), which leverages insertion sort's cache friendliness while keeping asymptotic performance.

Empirical Results (Part I)

We run empirical timing experiments (synthetic) to validate theoretical complexity and compare practical performance between OptimizedInsertionAlgorithm and Bubble Sort.

Plot interpretation: The linear plot shows measured times for various input sizes; both curves exhibit polynomial growth, with Bubble Sort displaying higher times across sizes due to larger constant factors and less efficient shifting.



Empirical Results (Part II) & Conclusion

Log-log plot (below) helps to identify polynomial order: a slope near 2 on log-log scales supports O(n^2) growth. The optimized insertion demonstrates a similar polynomial order but with smaller constants compared to Bubble Sort.

Analysis of constant factors: The empirical curves show that OptimizedInsertion benefits from block memory moves and fewer comparisons; this yields a smaller multiplicative constant for the n^2 term even though the asymptotic class remains the same.

Conclusion: The optimized insertion sort provides significant practical improvements over naive insertion and Bubble Sort for small to medium input sizes, especially on nearly-sorted data. For large inputs where asymptotic complexity dominates, prefer n log n algorithms like Merge Sort.