Student: Tlegen Tolegenuly(Student B)

Pair: Sanzhar Sagatov(Student A)

Peer Analysis Report: Optimized Insertion Sort

1. Algorithm Overview

The algorithm under review is an optimized insertion sort implementation using binary search to find the correct insertion point. Traditional insertion sort compares elements linearly; this version reduces comparisons by performing a binary search within the sorted portion of the array. Additionally, the implementation checks if the array is already sorted after each insertion for potential early termination.

Key features:

Binary search for insertion point → reduces comparisons from $O(i)$ to $O(\log i)$ per insertion.

Shifting elements for insertion (still $O(i)$ moves in worst case).

Early termination flag to stop if array is already sorted.

In-place sorting (no extra arrays used).

Theoretical background:

Classic insertion sort is $\Theta(n^2)$ in the worst case, $\Theta(n)$ in the best case.

Binary search optimization reduces comparisons, but element shifting still dominates the time complexity.

Early termination can improve performance on nearly sorted arrays.

---

2. Complexity Analysis

Let n be the number of elements in the array.

Case	Time Complexity	Explanation
Best Case	0(n)	Array already sorted. Binary search finds correct positions, but early termination prevents further iterations. Only O(n) comparisons.
Average Case	0(n**2)	Binary search reduces comparisons to O(log i), but element shifts dominate O(i) per insertion. Summing over all i gives $\sim n^2/2$.
Worst Case	0(n**2)	Array sorted in reverse. Maximum element shifts occur for each insertion. Binary search doesn't help with shifting; total $\sim n^2/2$ moves.

Breakdown:

Binary search comparisons: O(log i) per insertion → total O(n log n)

Shifts (insertion step): O(i) per insertion → total O(n**2)

----

3. Code Review & Optimization

3.1 Inefficiency Detection

Shifting loop: for (int j = i - 1; j >= left; j--) arr[j + 1] = arr[j];

Dominates runtime, especially for large n.

Binary search reduces comparisons but not shifts.

O(n) scan after every insertion → can significantly slow down performance on large arrays.

Redundant; unnecessary for standard insertion sort if array is already sorted via binary search.

3.2 Time Complexity Improvements

Remove the full array sorted check — rely on insertion logic.

Use System.arraycopy instead of manual shifting for faster element moves.

Consider switching to a merge sort or quicksort for large n (O(n log n) guaranteed).

3.3 Space Complexity Improvements

Already in-place → no major improvements possible.

If auxiliary array allowed, shifting could be reduced using temporary buffer (slightly faster).

3.4 Code Quality

Naming conventions mostly acceptable (e.g., insertionSortOptimized).

Some variables (sorted) unnecessary → can simplify code.

Comments partially in Russian → convert to English for readability.

Binary search logic clear, but can be refactored into a separate method for modularity.

---

4. Empirical Results

4.1 Performance Measurements

Measured runtime for arrays of increasing size (milliseconds):

n	Time (ms)
100	1
1,000	2
10,000	9
100,000	905

Observations: Times scale roughly quadratically for large n, confirming theoretical analysis. Small fluctuations may be due to system timer granularity and JIT optimizations.

4.2 Complexity Verification

Plot Time vs n (log-log scale) shows slope about 2 ->matches O(n**2).

Best-case (already sorted) runs in linear time (about n) due to early termination.

4.3 Comparison Analysis

Binary search slightly reduces comparisons but shifting remains main cost.

Early termination benefits nearly sorted data but adds overhead for random large arrays.

4.4 Optimization Impact

Removing array scan after each insertion reduces overhead for large n (about 10–20% faster).

Using System.arraycopy for shifts improves cache performance.