Assignment 2

Gulnaz Nurseit

Se-2412

Githab link: https://github.com/GulnazNurseit/daa_2.git

Algorithm Overview

Insertion Sort is a comparison-based sorting algorithm that builds the sorted array one element at a time. It works by taking elements from the unsorted portion and inserting them into the correct position within the sorted portion. The given implementation uses a binary search to locate the insertion position, optimizing the number of comparisons in nearly-sorted data.

Complexity Analysis

Best Case (Ω): O(n), occurs when the input array is already sorted. Each insertion requires only one comparison. - Average Case (Θ): O(n^2), since binary search reduces comparisons to O(log n) per insertion, but shifting elements still dominates with O(n). - Worst Case (O): O(n^2), occurs when the array is in reverse order, requiring maximum shifting for each element. - Space Complexity: O(1), since sorting is done in-place with constant auxiliary memory. Compared to a basic insertion sort, this optimized version reduces comparisons but not data movement.

Code Review

The use of binary search reduces unnecessary comparisons, but element shifting still causes O(n^2) moves. - Potential inefficiency: repeatedly shifting elements one by one. This can be improved by using System.arraycopy or block moves in Java. - The code is robust (handles null arrays) but could improve readability by extracting the binary search logic into a separate function. Optimization suggestion: For very small arrays (≤ 10 elements), insertion sort is efficient, but for larger arrays, a hybrid approach (e.g., merge sort + insertion sort for small subarrays) would improve time complexity.

Empirical Results

Empirical tests should measure: - Execution time vs input size (n = 100, 500, 1000, 5000, 10000). Best, average, and worst-case scenarios (already sorted, random, reverse sorted). Expected results: - Best case grows linearly. - Average and worst cases grow quadratically. - Binary search reduces comparisons, but total runtime is still quadratic due to shifting.

Conclusion

Insertion Sort with binary search optimization is efficient for small or nearly-sorted datasets, offering improved comparison counts but not asymptotic time improvements. For larger datasets, performance degrades quadratically. The algorithm is simple, stable, and adaptive, but hybridization with divide-and-conquer algorithms is recommended for practical large-scale applications.