1. Introduction

Sorting algorithms are a fundamental part of computer science.

In this project, we implemented Insertion Sort with an optimization: using binary search to find the position of insertion.

The goal of this report is to:

Analyze the algorithm’s time and space complexity.

Compare results for different input sizes.

Present benchmark results using JMH.

Provide a discussion of trade-offs.

2. Algorithm Description

Classical Insertion Sort inserts elements one by one into the correct position.

Worst case: it shifts up to n elements for each insertion.

Time complexity:

Best case (sorted input): O(n)

Average case: $O(n^2)$

Worst case (reverse sorted): $O(n^2)$

Optimization: Instead of linear search, we use binary search to locate insertion index.

This reduces comparisons from $O(n)$ to $O(\log n)$, but shifting elements still costs $O(n)$.

3. Implementation

Language: Java 17

Tools: JMH for benchmarking

Additional: PerformanceTracker (CSV output for plots)

Branching strategy: Git Feature Branch workflow

Code snippets:

```
public static <T extends Comparable<? super T>> void insertionSortBinary(T[] arr) {
  for (int i = 1; i < arr.length; i++) {
    T key = arr[i];
    int insertPos = binarySearch(arr, key, 0, i - 1);
    System.arraycopy(arr, insertPos, arr, insertPos + 1, i - insertPos);
    arr[insertPos] = key;
  }
}
```

4. Benchmark Setup

Framework: JMH (Java Microbenchmark Harness)

Parameters: n = 100, 1k, 10k, 100k

Warmup: 3 iterations

Measurement: 5 iterations

Runs: average time (ms)

Command:

```
mvn clean install

java -jar target/benchmarks.jar -rf csv -rff results.csv
```

5. Results (Sample)

nAvg Time (ms)1000.021,0000.3510,00020.8100,0001980.5

Plot: log-log scale shows quadratic trend.

(Attached in report as Figure 1: Benchmark Results)

6. Analysis

Scaling: Execution time grows approximately as $O(n^2)$.

Binary search effect: Comparisons are reduced, but array shifts dominate → still quadratic.

Practicality: For small n (up to ~1k) insertion sort is competitive.

Limitation: Not suitable for large data.

7. Comparison

Compared with Merge Sort or Quick Sort:

Merge/Quick = $O(n \log n)$ for average/worst case.

Insertion Sort = $O(n^2)$.

Use cases: small arrays, partially sorted arrays, or as a subroutine in hybrid algorithms (like Timsort).

8. Conclusion

Implemented insertion sort with binary search.

Benchmarks confirm theoretical complexity.

Binary search reduces comparisons but not asymptotic runtime.

Future work: implement Merge Sort and compare.