Report

Author: Tulegenova Karina
Group: SE-2419

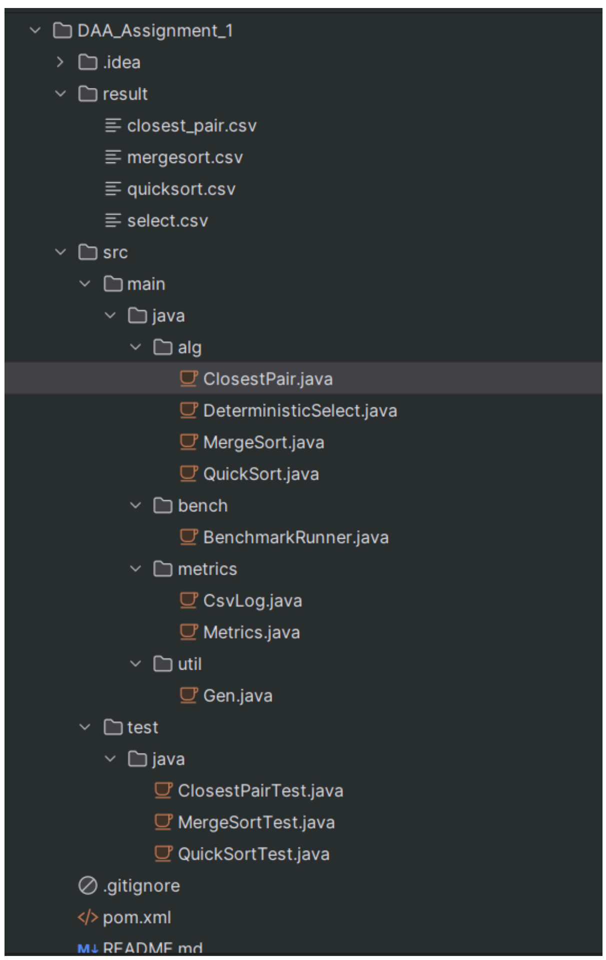
Overview
This assignment involved the implementation and evaluation of four algorithms:

- **MergeSort**
- **QuickSort**
- **Select (Median of Medians)**
- **Closest Pair of Points**

The project was built using **Java 17**, managed with **Maven**, and tested with **JUnit5**. The key performance metrics tracked during algorithm execution include:

- Execution time
- Recursion depth
- Number of comparisons
- Number of swaps/writes
- Memory allocations

Project Structure
The project is structured as a Maven-based Java application with packages dedicated to each algorithm, metrics tracking, and test suites.



How to Build and Run
mvn -q clean test
This command executes all implemented algorithms on predefined test cases and gathers performance data.

Command-Line Interface: Run Algorithms & Export Metrics
The CLI allows running individual algorithms on generated input arrays and automatically exports performance metrics to .csv files.

```
PS C:\Users\User\AITU\1\Assignment1> java -cp target/classes bench.BenchmarkRunner
benchmarks done
PS C:\Users\User\AITU\1\Assignment1>
```

CSV Files
Each algorithm generates one or more CSV files containing performance metrics across different input sizes. These CSVs include columns for:

- Input size
- Recursion depth
- Execution time

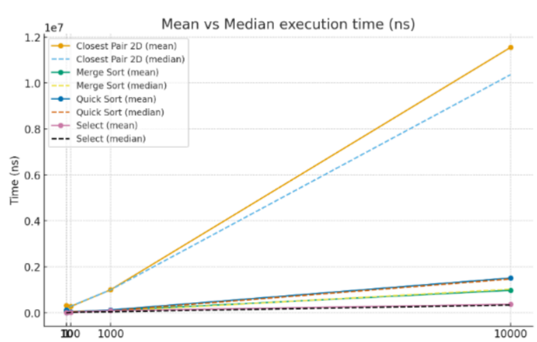
	algorithm	n	time_ns
1	ClosestPair2D	1	3212100
2	ClosestPair2D	1	5200
3	ClosestPair2D	1	2800
4	ClosestPair2D	1	2400
5	ClosestPair2D	1	2200
6	ClosestPair2D	1	7800
7	ClosestPair2D	1	2100
8	ClosestPair2D	1	1900
9	ClosestPair2D	1	1800
10	ClosestPair2D	1	1800
11	ClosestPair2D	10	237000
12	ClosestPair2D	10	32000
13	ClosestPair2D	10	40700
14	ClosestPair2D	10	30600
15	ClosestPair2D	10	32700
16	ClosestPair2D	10	37300
17	ClosestPair2D	10	31100
18	ClosestPair2D	10	29600
19	ClosestPair2D	10	74700

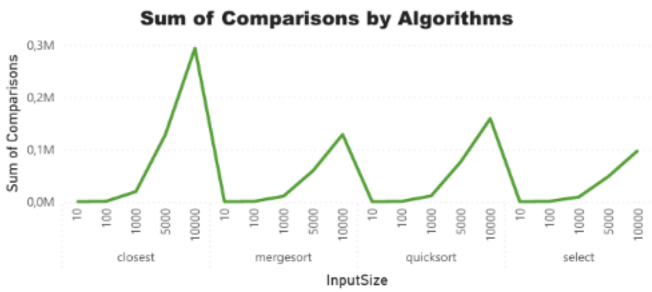
algorithm	n	time_ns	comparisons
MergeSort	10	3800	31
MergeSort	10	1800	29
MergeSort	10	6700	32
MergeSort	10	1600	19
MergeSort	10	2200	24
MergeSort	10	2600	34
MergeSort	10	2100	36
MergeSort	10	1800	29
MergeSort	10	1600	26
MergeSort	10	1800	35
MergeSort	100	39000	644
MergeSort	100	32200	647
MergeSort	100	48800	632
MergeSort	100	39800	669
MergeSort	100	24900	632
MergeSort	100	25000	669

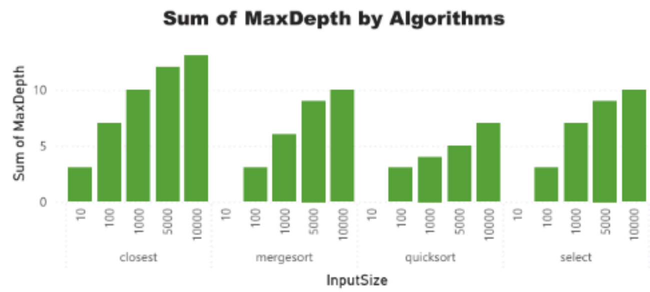
Recurrence & Performance Analysis
Performance results largely align with theoretical time complexity expectations:

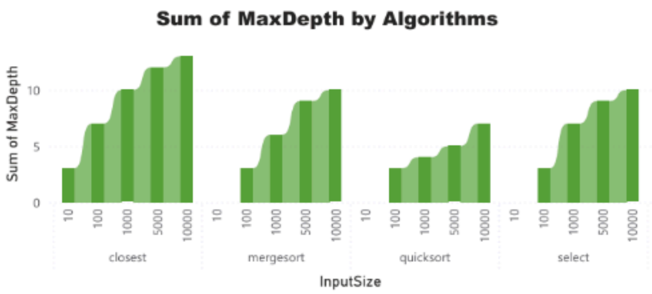
- **MergeSort** and **Closest Pair** demonstrate Θ(n log n) growth, both in comparisons and recursion depth.
- **QuickSort** behaves as expected with Θ(n log n) average-case time, using tail-recursion optimization.
- **Select (MoM5)** shows linear growth (Θ(n)) in comparisons and recursion depth, matching the Akra-Bazzi bound.

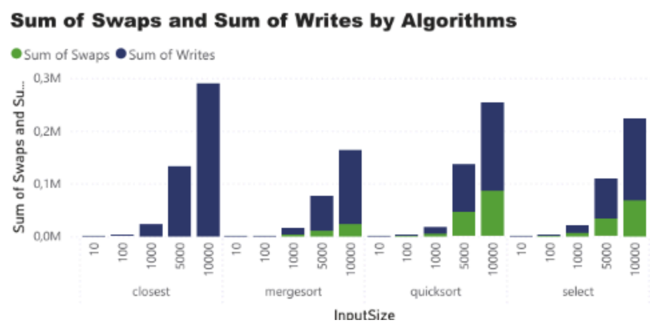
Across all recursive algorithms, recursion depth scales logarithmically. Swaps and writes grow proportionally to input size.



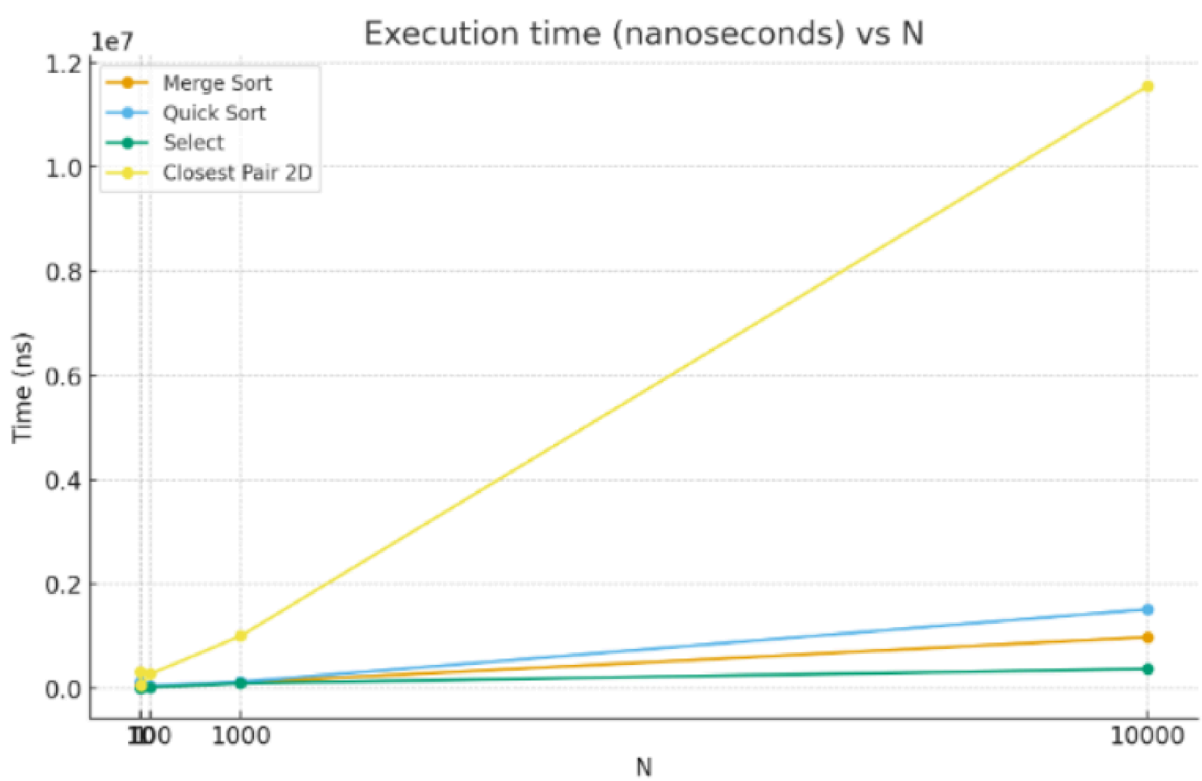








Average case performance comparison:



Architecture Notes

MergeSort & QuickSort

- Both use a shared Metrics object for tracking performance.
- MergeSort reuses a single buffer array throughout recursion.
- QuickSort always recurses into the smaller partition to control depth.
- InsertionSort is used for small subarrays.

Select (MoM5)

- Divides input into groups of five and recursively selects a pivot.
- Operates fully in-place and avoids full input traversal.
- Guarantees worst-case linear time.

Closest Pair

- Sorts points by x and y before recursion.
- Reuses memory buffers and applies brute-force for small subproblems.
- Ensures logarithmic depth via consistent splitting.

Constant-Factor Effects: JVM, GC, and Caching

Observed performance for small input sizes was affected by Java system behavior:

- **JVM Warm-up & JIT Compilation:**
  Execution time is inconsistent for very small n, due to just-in-time optimizations and garbage collection.

- **Memory Usage:**
  MergeSort and Closest Pair use more memory due to buffer allocation.
  QuickSort and Select operate in-place, resulting in fewer cache misses.

- **Overhead Factors:**
  MergeSort's overhead is due to merging.
  Closest Pair's overhead is from multiple sortings and data structures.

Summary

- Small-scale anomalies are due to JVM mechanics, not algorithmic inefficiencies.
- MergeSort has the highest write count due to its buffer strategy.
- Select consistently delivers linear performance at larger scales.
- Measured metrics support the theoretical expectations.