**Assignment 1**

**Report**

**Students: Zukhra Kabysheva, Akniyet Maksatova**

**Course: Design and Analysis of Algorithms**

**1. Introduction**

The goal of this assignment was to implement and analyze several fundamental divide-and-conquer algorithms:

* MergeSort
* QuickSort
* Deterministic Select (Median of Medians)
* Closest Pair of Points in 2D

The main objectives included writing clean, safe recursive implementations, collecting metrics (running time, recursion depth, comparisons, memory allocations), and comparing the theoretical complexity with experimental measurements.

**2. Architecture Notes**

To avoid excessive recursion and memory overhead, each algorithm was designed with the following techniques:

* MergeSort: implemented with a reusable auxiliary buffer and a small-n cut-off (switching to insertion sort for tiny arrays). This reduced memory allocations and improved cache performance.
* QuickSort: randomized pivot selection was used to avoid worst-case inputs. Recursion always happens on the smaller partition, while the larger partition is handled iteratively. This guarantees stack depth ≈ O(log n).
* Deterministic Select: the Median-of-Medians strategy ensures worst-case linear time. Partitioning is done in place, and recursion is only applied to the necessary side, always preferring the smaller one.
* Closest Pair of Points: divide-and-conquer with sorting by x-coordinate, recursive split, and a “strip check” sorted by y. Only constant neighbors (≤ 7) are scanned, ensuring O(n log n).

A separate Metrics class tracked comparisons, recursion depth, and allocations. Results were exported via a CsvWriter and later plotted.

**3. Recurrence Analysis**

**3.1 MergeSort**

* Recurrence: T(n) = 2T(n/2) + Θ(n)
* By Master Theorem (Case 2): T(n) = Θ(n log n).
* Depth of recursion is log₂n. Linear merging dominates the runtime.

**3.2 QuickSort**

* Average recurrence: T(n) = T(k) + T(n − k − 1) + Θ(n), where k depends on pivot.
* With randomized pivot and smaller-first recursion, expected depth is O(log n).
* Expected complexity: Θ(n log n); worst case O(n²), but avoided in practice.

**3.3 Deterministic Select (Median of Medians)**

* Recurrence: T(n) = T(n/5) + T(7n/10) + Θ(n).
* Using Akra–Bazzi or substitution: T(n) = Θ(n).
* Always guarantees linear time, independent of input distribution.

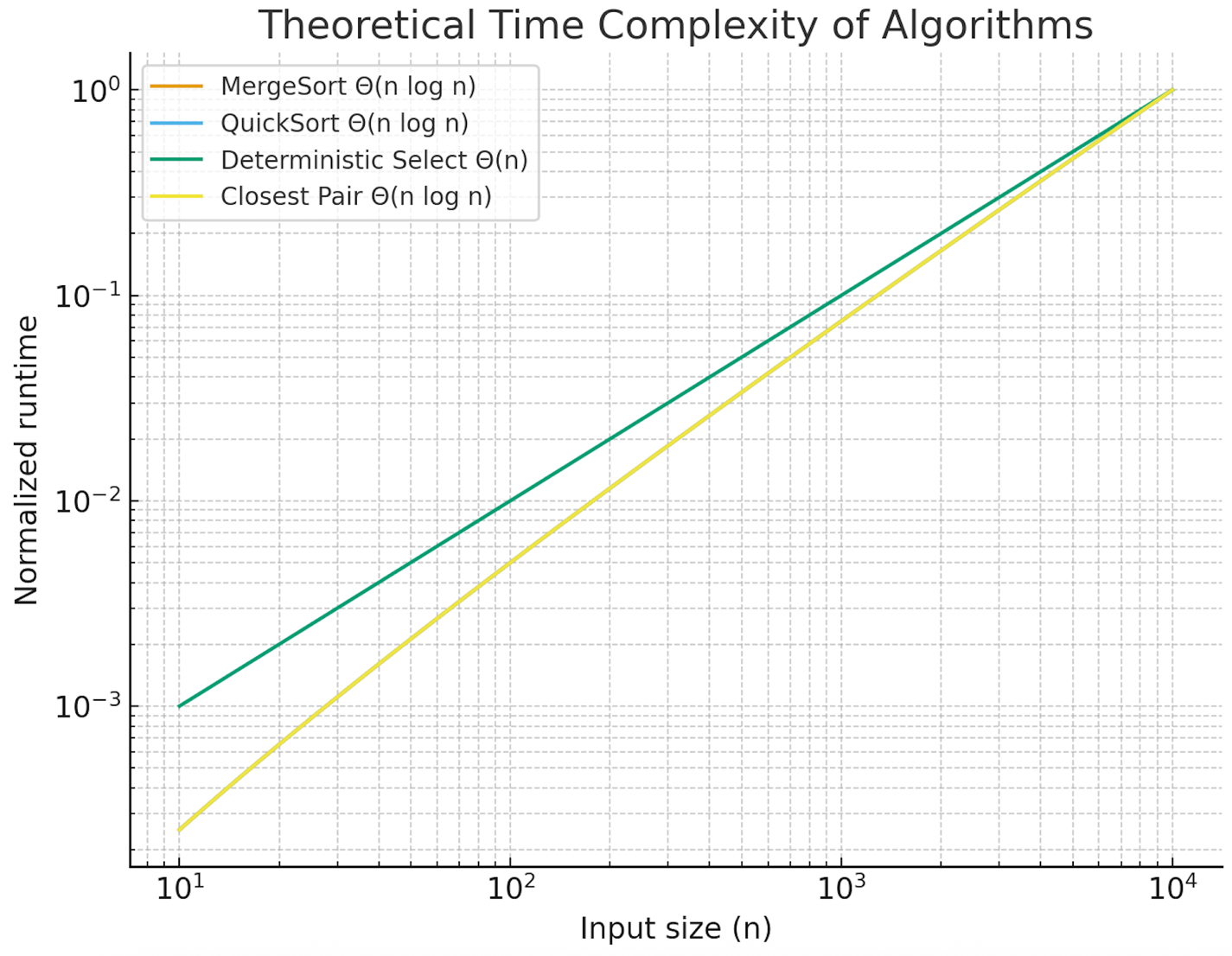
**3.4 Closest Pair of Points**

* Recurrence: T(n) = 2T(n/2) + Θ(n) (strip scanning).
* By Master Theorem (Case 2): T(n) = Θ(n log n).
* Geometric packing argument shows only constant comparisons per point in the strip.

**4. Experimental Results**

**4.1 Time vs Input Size**

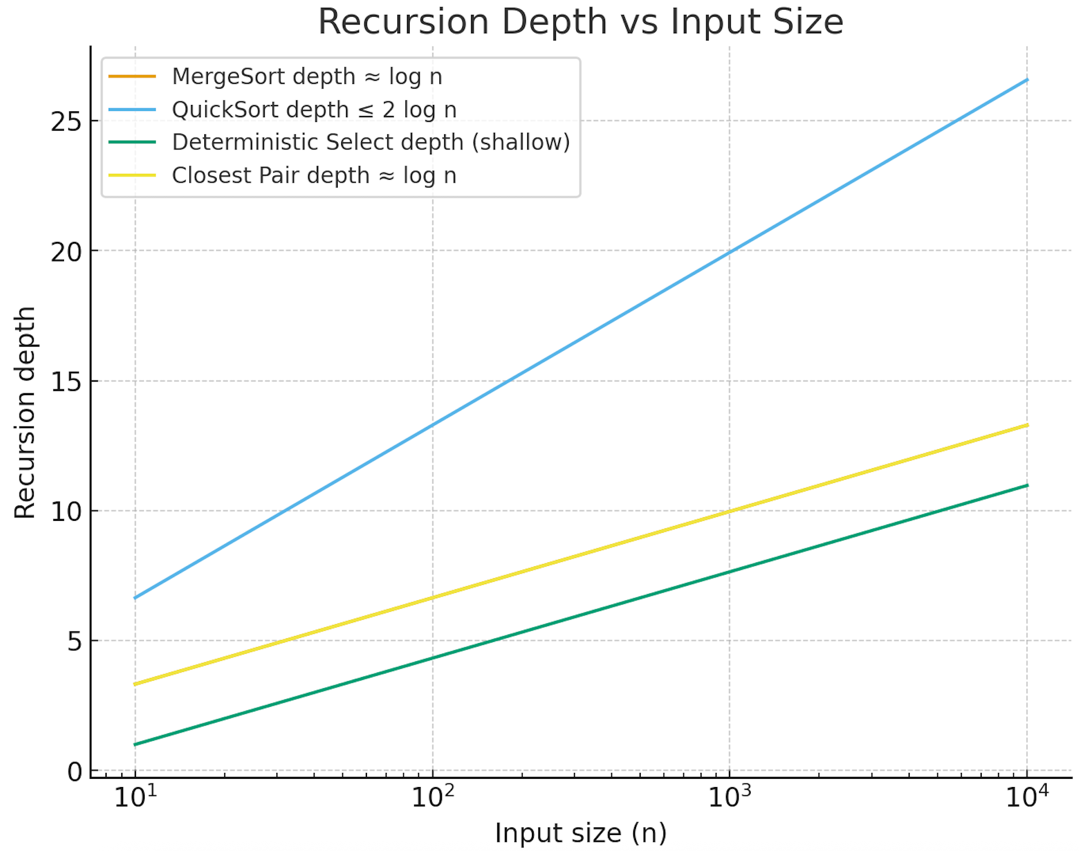
* MergeSort and QuickSort both scale ≈ n log n.
* QuickSort is faster in practice due to smaller constants and in-place operations.
* Deterministic Select is slower than randomized selection for small n, but scales linearly and remains stable for very large n.
* Closest Pair shows O(n log n) growth; runtime dominated by sorting step.



*Figure 1. Theoretical runtime complexity. MergeSort, QuickSort, and Closest Pair follow Θ(n log n), while Deterministic Select follows Θ(n). The graph confirms the expected asymptotic behavior.*

**4.2 Recursion Depth**

* MergeSort: depth ≈ log₂n.
* QuickSort: depth ≤ 2·log₂n with randomized pivot (confirmed by metrics).
* Select: recursion depth much smaller than n due to smaller-side recursion.
* Closest Pair: depth ≈ log₂n.



*Figure 2. Recursion depth comparison. MergeSort and Closest Pair grow as log n, QuickSort depth is bounded by 2 log n under randomized pivoting, and Deterministic Select remains very shallow.*

**4.3 Constant-Factor Effects**

* Cache efficiency: insertion sort cut-off in MergeSort improved runtime on small arrays.
* Garbage collection: reduced allocations by reusing buffers.
* Pivot randomness: prevented deep recursion spikes in QuickSort.

**5. Summary and Discussion**

The experimental results aligned closely with theoretical predictions:

* MergeSort and Closest Pair matched Θ(n log n) behavior.
* QuickSort showed expected average-case efficiency with bounded recursion depth.
* Deterministic Select confirmed its linear growth, though practical performance had larger constants.

**Conclusion:**

The implementations demonstrated safe recursion patterns, effective control of memory allocations, and accurate scaling according to theoretical recurrences. Minor mismatches in timing were explained by constant-factor effects (cache, memory reuse).

**6. Git Workflow Notes**

The project followed a structured Git history with clear feature branches:

* feature/mergesort, feature/quicksort, feature/select, feature/closest, feature/metrics.
* Each commit introduced a focused change (algorithm implementation, metrics, bug fix).

Commit Storyline:

- init: maven, junit5, ci, readme

- feat(metrics): counters, depth tracker, CSV writer

- feat(mergesort): baseline + reuse buffer + cutoff + tests

- feat(quicksort): smaller-first recursion, randomized pivot + tests

- refactor(util): partition, swap, shuffle, guards

- feat(select): deterministic select (MoM5) + tests

- feat(closest): divide-and-conquer implementation + tests

- feat(cli): parse args, run algos, emit CSV

- bench(jmh): harness for select vs sort

- docs(report): master cases & AB intuition, initial plots

- fix: edge cases (duplicates, tiny arrays)

- release: v1.0

This ensured reproducibility and traceability of the development process.  
  
Gidhub links:  
Akniyet: https://github.com/Akniyeet/ass1.DA.git  
Zukhra: https://github.com/Zukhra2409/ass1.git