Report

Benchmark Results (n = 1000)

Algorithm	n	Comparisons	Allocations	Depth	Time (ns)
Closest	1000	0	0	0	16,507,560
MergeSort	1000	13,479	2,508	6	2,014,008
QuickSort	1000	58,720	0	7	4,532,665
Select	1000	2,145	0	1	4,201,759

Benchmark Results (n = 100000)

Algorithm	n	Comparisons	Allocations	Depth	Time (ns)
Closest	100000	0	0	0	172,493,758
MergeSort	100000	1,864,265	601,360	13	36,385,557
QuickSort	100000	21,106,995	0	12	146,764,850
Select	100000	205,837	0	1	34,042,088

Architecture Notes

To better understand the algorithms, we tracked three metrics along with runtime:

- **Comparisons**: this counts each time two elements are compared, such as during the merge step in MergeSort, partitioning in QuickSort, or pivot checks in Select.
- **Allocations**: this records instances when a temporary buffer is created, like the auxiliary array in MergeSort.
- **Recursion depth**: this is incremented when entering recursion and decremented when exiting. We store the maximum depth reached as `maxDepth`.

These metrics provide a clearer view of performance and resource use.

Recurrence Analysis

- **MergeSort**
  Recurrence: `T(n) = 2T(n/2) + Θ(n)`.
  By the Master Theorem (Case 2), this simplifies to `T(n) = Θ(n log n)`.
  Experimental depth increases logarithmically (`= log₂n`), while allocations rise linearly with `n` due to buffer usage.

- **QuickSort**
  Recurrence: `T(n) = T(k) + T(n−k−1) + Θ(n)`.
  The average case is `T(n) = Θ(n log n)` and the worst case is `T(n) = Θ(n^2)`.
  Depth is roughly `log n` when pivots are balanced. Comparisons tend to increase faster than in MergeSort.

- **Deterministic Select (Median of Medians)**
  Recurrence: `T(n) = T(n/5) + T(7n/10) + Θ(n)`.
  Using Akra–Bazzi, the solution is `T(n) = Θ(n)`.
  Depth stays constant (equal to 1), comparisons increase linearly, but runtime has a significant constant-factor overhead.

- **Closest Pair (Divide and Conquer)**
  Recurrence: `T(n) = 2T(n/2) + Θ(n)`.
  By the Master Theorem (Case 2), the runtime is `T(n) = Θ(n log n)`.
  Our current setup does not gather data on comparisons or depth, but runtime follows the expected order and has large constants.

Plots and Discussion

Time vs n

- At `n = 1000`: MergeSort is the fastest at about 2 ms, QuickSort is slower around 4.5 ms, Select is similar at about 4.2 ms, and Closest is much slower at approximately 16.5 ms.
- At `n = 100000`: MergeSort scales well at about 36 ms, Select is close at roughly 34 ms, QuickSort runs slower around 147 ms, and Closest is the slowest at about 172 ms.

Depth vs n

- MergeSort depth grows logarithmically ($= \log_2 n$): 6 at `n=1000`, 13 at `n=100000`.
- QuickSort depth ranges from 7 to 12, depending on pivot choices.
- Select stays constant at a depth of 1.
- Closest Pair depth is not recorded.

Constant-factor effects

- **Cache locality**: MergeSort benefits from sequential memory access, while QuickSort struggles more as input size increases.
- **Memory/GC**: MergeSort's temporary buffers can cause garbage collection for large inputs.
- **Overhead**: Select's shallow recursion should be inexpensive, but pivot sampling and partitioning introduce constant overhead, making it slower than MergeSort for smaller inputs, even with linear complexity.

Summary

- **Alignment with theory**:
    - MergeSort, QuickSort, and Closest Pair exhibit `Θ(n log n)` behavior.
    - Select shows linear comparisons and depth, which matches `Θ(n)`.
- **Mismatch**:
    - Select runs slower than MergeSort for small inputs due to overhead, though it gets closer as `n` increases.
    - QuickSort performs more comparisons and runs slower despite having the same asymptotic complexity.
    - Closest Pair has the highest runtime constants, which largely determine execution time despite optimal performance bounds.

Overall, the experiments validate theoretical expectations but also highlight the strong effects of constants, memory allocation, cache behavior, and JVM overhead.