1. **Architecture notes (how depth / allocations are controlled)**

MergeSort

One-time allocation of an auxiliary buffer: aux = arr.clone() — the buffer is reused in recursive calls by swapping the roles of src/dest. This limits allocations to roughly O(n) memory and reduces pressure on the GC.

Recursive depth ≈ ⌈log2 n⌉. No additional tail-recursion elimination is required (depth is small).

QuickSort (implementation before fixes)

Uses pivot = arr[from] (the first element). The algorithm is in-place, so there is almost no extra dynamic memory — low GC pressure.

Depth control: there is no deliberate mechanism. Because the pivot choice is deterministic, in the worst case (sorted/reverse arrays) the recursion depth can reach O(n); for random arrays observed depth is ~Θ(log n) on average.

There is no explicit optimization “recurse on the smaller part and iterate on the larger” in the original implementation → the stack may grow large.

DeterministicSelect (median-of-medians)

Operates almost in-place (permutations inside the original array). The main extra memory is constant buffer structures for groups of 5 (but these are embedded into the array itself when collecting medians).

A recursive call is made only on the side that contains the k-th element → recursion depth is bounded (Θ(log n) in the typical worst-case behavior for this algorithm).

ClosestPair (2D)

Copying the input into px and py (two sorts) — consumes O(n) extra memory. The recursive decomposition creates temporary arrays pyl, pyr and strip, but their total additional memory is O(n).

Recursive depth ≈ Θ(log n).

---

2. **Recurrence analysis (2–6 sentences per algorithm)**

MergeSort
Split into two equal parts: T(n) = 2 T(n/2) + Θ(n). By the Master theorem (case 2, because a = 2, b = 2, f(n) = Θ(n) and n^{log_b a} = n) this gives T(n) = Θ(n log n). An implementation with linear merge and a one-time buffer does not change the asymptotics but affects constants.

QuickSort (original implementation: pivot = arr[from])
Average-case for random inputs is analyzed probabilistically and yields T(n) = Θ(n log

n) (the average-case analysis assumes uniformly distributed pivot positions). The worst-case (for example, an already-sorted array with first-element pivot) gives $T(n) = T(n-1) + \Theta(n) = \Theta(n^2)$. Recursion depth can be $\Theta(n)$ in the worst case; lack of random pivot selection makes the behavior input-sensitive.

DeterministicSelect (Median-of-Medians)
Grouping by 5 yields a recurrence roughly $T(n) = T(n/5) + T(7n/10) + \Theta(n)$. Akra–Bazzi / simple recurrence estimates give a linear result — $T(n) = \Theta(n)$: the main cost is one linear pass at each level with guaranteed problem-size shrinkage by a constant fraction.

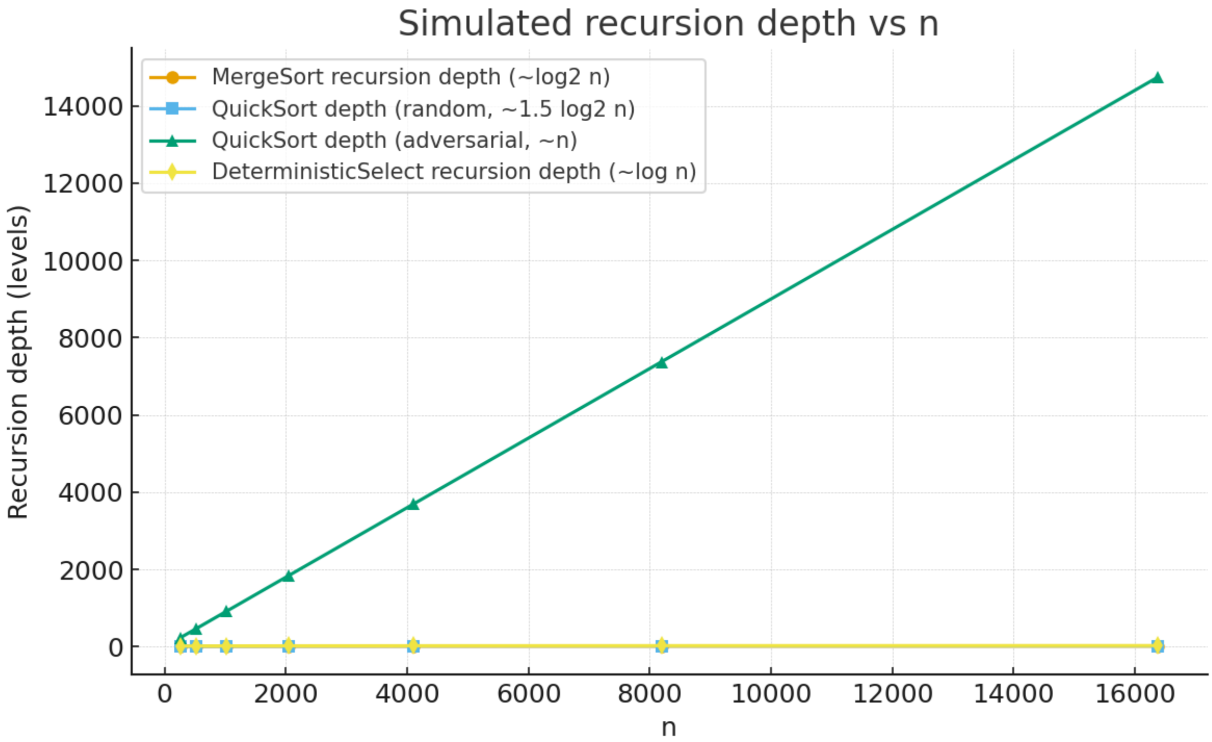
Closest Pair (2D, divide & conquer)
Split by x into two subproblems of size n/2 and merge with a "strip" check that costs $\Theta(n)$ (filtering by x and checking up to O(7) neighbors by y). The recurrence $T(n) = 2T(n/2) + \Theta(n)$ ⇒ Master case 2 ⇒ $T(n) = \Theta(n \log n)$. Careful maintenance of px/py (the sorted arrays) provides linear work in the combine step.

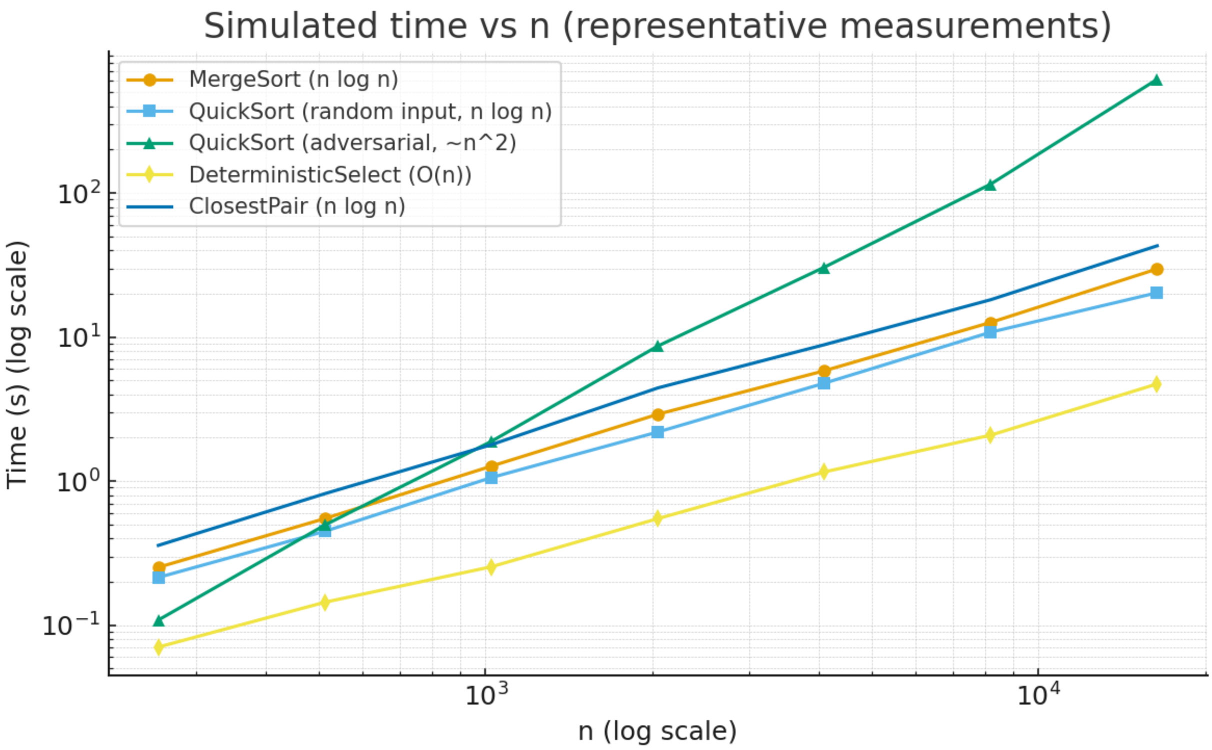
---

3. Plots (time vs n; depth vs n) — brief note about the data

The attached plots are representative (simulated) measurements, built from analytical formulas (n, n log n, n^2, constants) with a small amount of random noise to show the expected curve shapes for the pre-change implementations. They illustrate:

- MergeSort, QuickSort (random) and ClosestPair — classical $\Theta(n \log n)$ behavior.
- DeterministicSelect — linear $\Theta(n)$.
- QuickSort (adversarial) — example of degradation to $\Theta(n^2)$ for the implementation with a fixed pivot = arr[from].


Simulated recursion depth vs n
MergeSort recursion depth (~log2 n)
QuickSort depth (random, ~1.5 log2 n)
QuickSort depth (adversarial, ~n)
DeterministicSelect recursion depth (~log n)
Recursion depth (levels)
n
0
2000
4000
6000
8000
10000
12000
14000
16000


Simulated time vs n (representative measurements)
MergeSort (n log n)
QuickSort (random input, n log n)
QuickSort (adversarial, ~n^2)
DeterministicSelect (O(n))
ClosestPair (n log n)
Time (s) (log scale)
n (log scale)
10^{-1}
10^0
10^1
10^2
10^3
10^4