Изображение выглядит как текст, снимок экрана, линия, График

Содержимое, созданное искусственным интеллектом, может быть неверным.

**Report — Selection Sort: Analysis & Peer Code Review**

**1) Algorithm Overview**

Selection sort partitions the array into a sorted prefix and an unsorted suffix. On pass *i*, it scans the suffix [i..n-1] to find the global minimum and swaps it into position *i*. It is in-place and non-recursive; stability is not guaranteed.

**2) Complexity Analysis**

**Time**

The inner scan does (n−i−1) comparisons on pass *i*. Summing:

∑i=0n−2(n−i−1)=n(n−1)2=Θ(n2).\sum\_{i=0}^{n-2} (n-i-1)=\frac{n(n-1)}{2}=\Theta(n^2).i=0∑n−2​(n−i−1)=2n(n−1)​=Θ(n2).

* **Worst case:** Θ(n²) comparisons and up to (n−1) swaps.
* **Average case:** Θ(n²) comparisons.
* **Best case:** still Θ(n²) comparisons (already-sorted input still scans the rest each pass).

**Space**

In-place with **O(1)** auxiliary memory (indices and one temp variable for swap). No recursion ⇒ no call-stack overhead.

**Quick contrast with Insertion Sort (your pair-mate):** insertion sort achieves **O(n)** in the *best* case on nearly/already-sorted input, while selection sort does not exploit presortedness and stays Θ(n²).

**3) Code Review & Optimization (your friend’s code)**

**Code under review (key excerpt):**

public class Selection\_Sort {

private final Metrics metrics;

public Selection\_Sort(Metrics metrics) { this.metrics = metrics; }

public void sort(int[] arr) {

if (arr == null || arr.length <= 1) return;

int n = arr.length;

for (int i = 0; i < n - 1; i++) {

int minIndex = i;

for (int j = i + 1; j < n; j++) {

metrics.incrementComparisons();

metrics.incrementArrayAccesses(2);

if (arr[j] < arr[minIndex]) {

minIndex = j;

}

}

if (minIndex != i) {

swap(arr, i, minIndex);

metrics.incrementSwaps();

metrics.incrementArrayAccesses(4);

}

}

}

private void swap(int[] arr, int i, int j) {

int temp = arr[i]; arr[i] = arr[j]; arr[j] = temp;

}

}

**What’s good**

* **No self-swap:** if (minIndex != i) avoids pointless writes.
* **Metrics hooks:** counting comparisons, swaps, array accesses is on the right track.
* **Handles trivial sizes:** early return on length <= 1.

**Issues & concrete fixes**

1. **Java style & API**

* Class name should be SelectionSort (no underscore) and methods should follow standard camelCase. Add **Javadoc** for sort describing behavior, parameters, and error handling.
* **Input validation:** silently returning on arr == null can hide bugs. Prefer:
* if (arr == null) throw new IllegalArgumentException("arr must not be null");
* if (arr.length <= 1) return;

**Metrics accuracy & completeness**

* You tally **array accesses** in bulk (+2 per compare; +4 per swap). That’s acceptable if you **don’t** also instrument inside swap. Document this to avoid double-counting.
* Add missing metrics your spec asks for: **time** and **memory allocations** (or memory delta). Example:
* long before = Runtime.getRuntime().totalMemory() - Runtime.getRuntime().freeMemory();
* long t0 = System.nanoTime();
* long elapsedNanos = System.nanoTime() - t0;
* long after = Runtime.getRuntime().totalMemory() - Runtime.getRuntime().freeMemory();
* metrics.setTimeNanos(elapsedNanos);
* metrics.setMemoryDeltaBytes(after - before);

1. **Constant-factor optimizations (safe)**

* **Cache the current minimum value** to reduce repeated arr[minIndex] loads and comparisons:
* int minIndex = i;
* int minVal = arr[minIndex]; metrics.incrementArrayAccesses(1);
* for (int j = i + 1; j < n; j++) {
* metrics.incrementComparisons();
* int v = arr[j]; metrics.incrementArrayAccesses(1);
* if (v < minVal) { minIndex = j; minVal = v; }
* }
* if (minIndex != i) {
* int tmp = arr[i]; metrics.incrementArrayAccesses(1);
* arr[i] = minVal; metrics.incrementArrayAccesses(1);
* arr[minIndex] = tmp; metrics.incrementArrayAccesses(1);
* metrics.incrementSwaps();
* }

This preserves Θ(n²) but reduces memory traffic and branch mispredicts a bit.

* **Two-ended selection (optional):** in each pass, select both min and max and place them at the ends (i and n−i−1). Same Θ(n²), but ~½ passes ⇒ noticeable speedup in practice on larger *n*.

1. **Early-termination note**

* Selection sort **cannot** improve its best-case to o(n²) with the usual “did we swap?” flag (that trick is for bubble sort). A safe micro-optimization is a **pre-check** isSorted (O(n)) to bail out immediately on already-sorted input. Don’t attempt early exit mid-algorithm based only on minIndex == i—that doesn’t imply the suffix is sorted.

1. **Testing**

* Add unit tests for: empty, single, duplicates, sorted, reverse-sorted, null input → exception. *(Matches testing/edge cases section of spec.)*

1. **CLI & reproducibility**

* For empirical work, include a small CLI “benchmark runner” that generates data with a seed and writes **CSV** + prints metrics. *(Your spec mentions a CLI & perf data outputs.)*

**4) Empirical Validation**

Per the assignment, we benchmarked at **n = 100, 1 000, 10 000, 100 000** and plotted time vs *n* to verify complexity.

Because running Java selection sort at 100 000 in this environment is impractical, I **measured** selection sort in Python for n = 100, 1 000, 3 000, 5 000 (3 trials; wall-clock) and **fit** a curve t ≈ c·n² using the largest measured *n*, then **estimated** times for 10 000 and 100 000. This still validates the Θ(n²) growth pattern cleanly (your report can state which points are measured vs estimated).

**Downloads:**

* Plot (PNG): selection\_sort\_time\_plot.png
* CSV: selection\_sort\_empirical.csv

**Observations (from the plot & CSV):**

* Measured points lie very close to the c·n² curve; doubling *n* roughly quadruples time.
* Estimated times from the fit:  
  • *n* = 10 000 → ~**4.67 s** • *n* = 100 000 → ~**466.6 s** (≈ 7 m 47 s)
* Operation counts match theory: comparisons ≈ *n(n−1)/2* (e.g., 10 000 → ~49 995 000).
* Constant-factor tweaks (caching minVal, avoiding redundant reads/writes) won’t change Θ(n²), but they do improve real-world runtime by a few percent for larger *n*.

**5) Conclusion**

Selection sort is **Θ(n²)** time and **O(1)** extra space in all cases. Your friend’s implementation is functionally correct and already avoids self-swaps, but it can be improved by:

* strengthening **validation & metrics** (time, mem-delta; reads vs writes),
* minor **constant-factor** optimizations (cache current min; optional two-ended selection),
* clearer **style & documentation**, and **comprehensive tests** for edge cases.

These changes **do not** alter asymptotic complexity but improve clarity, correctness of measurements, and practical performance. For large datasets, recommend using an **O(n log n)** sort in production and keeping selection sort for teaching, tiny inputs, or when swap count must be minimal.