Assignment 2 — Selection Sort (Student B)

Technical Report (for submission & defense)

1) Project Overview

This project delivers a clean, production-ready **Java/Maven** implementation of **Selection Sort** with:

- **Two variants:** classic *standard* and *double-ended* (min & max per pass).
- **Early-exit optimization:** stops when the array is already sorted.
- **Metrics collection:** comparisons, swaps, reads, writes, iterations, early-termination flag.
- **CLI benchmark runner** and **JUnit 5 tests**.

The goal is to demonstrate algorithmic correctness, measure basic performance characteristics, and provide a reproducible benchmark harness.

---

2) Objectives & Requirements

- Implement Selection Sort (Student B role) with neat code and zero IDE errors.
- Add early-exit optimization and (optionally) a double-ended pass.
- Provide a small metrics utility to quantify operations.
- Supply unit tests and a runnable CLI for demos/experiments.

---

3) Repository Structure

```
src/
  main/
    java/
      algorithms/SelectionSort.java
      metrics/PerformanceTracker.java
      cli/BenchmarkRunner.java
  test/
    java/
      algorithms/SelectionSortTest.java
pom.xml
```

---

4) Build, Test, Run

- **Build & Tests**
- `mvn -q -DskipTests=false test`
- **Run CLI (single default run if no args)**
- `# Example manual run with arguments:`
- `mvn -q exec:java -Dexec.args="20 random standard true"`
- `# Format: <size> <mode> <algo> <earlyExit> [seed]`
- `# mode: random|sorted|reversed|duplicates`
- `# algo: standard|double`

- **IDE:** Run `cli.BenchmarkRunner.main()`; without args it performs one default random run.

---

5) Algorithm Design

5.1 Standard Selection Sort

- Outer loop index `i` scans from `0` to `n-2`.
- Find the index of the **minimum** element on `[i..n-1]`.
- Swap it with `a[i]`.
- **Loop invariant:** after the `i`-th pass, positions `0..i` hold the `i+1` smallest elements in sorted order.

5.2 Double-Ended Variant

- Maintain two pointers: `left` and `right`.
- In **one pass**, scan `[left..right]` to find both **min** and **max**.
- Place min → `left`, max → `right`, then shrink to `left+1..right-1`.
- This reduces the **number of outer passes** roughly by half, while overall time is still $\Theta(n^2)$.

5.3 Early-Exit Optimization

- After each pass, check if the array is already non-decreasing.
  If yes → **terminate** (best-case $\Theta(n)$).

---

6) Complexity & Trade-offs

Variant	Best Case	Average / Worst	Space
Standard + Early Exit	$\Theta(n)$ (already sorted)	$\Theta(n^2)$	O(1)
Double-Ended + Early	$\Theta(n)$ (already sorted)	$\Theta(n^2)$	O(1)

Notes

- Selection Sort is **not stable** (swaps can reorder equals). Stability would require extra logic (e.g., stable selection via shifting), which increases writes.
- Double-ended variant reduces *passes* but remains $O(n^2)$ because each pass still scans a linear range.

---

7) Correctness Argument (sketch)

- **Initialization:** Before any pass, the left prefix is empty; trivially sorted and minimal.
- **Maintenance:** Each pass selects the global minimum from the unsorted suffix and places it at the boundary (and, in the double-ended case, also places the global maximum at the

right boundary). The invariant is preserved: the prefix (and suffix) are sorted and contain the correct extremal elements.
- **Termination:** After `n-1` (standard) or ≈`n/2` (double-ended) passes, the unsorted range is empty; the array is fully sorted.

---

8) Implementation Details

8.1 `algorithms.SelectionSort`

- Public API:
- `SelectionSort.sort(int[] a,`
- `PerformanceTracker trackerOrNull,`
- `boolean earlyExit,`
- `boolean doubleEnded);`
- Input validation: `null` → `IllegalArgumentException`; arrays of length < 2 return immediately.
- **Metrics hooks** around comparisons, swaps, and array accesses.
- `isSorted(..)` used for early exit.

8.2 `metrics.PerformanceTracker`

- Fields: `comparisons`, `swaps`, `arrayReads`, `arrayWrites`, `iterations`, `earlyTerminated`.
- Methods: increment/add counters, `toString()`, `csvHeader()`, `toCsvRow()`.

8.3 `cli.BenchmarkRunner`

- Parses arguments and runs one benchmark; if no args, performs a default single random run.
- Prints input (truncated), sorted output (truncated), human-readable metrics, and a one-line CSV.

8.4 Tests (`SelectionSortTest`)

- Edge cases: empty and single-element arrays.
- Duplicates handling.
- Random arrays compared against `Arrays.sort`.
- Early exit triggers on already sorted input.

---

9) Metrics & Experimental Notes

What is measured

- `comparisons`: every `a[j]` vs `a[minIdx]` (and `a[maxIdx]` in double-ended).
- `swaps`: swap operations.
- `arrayReads/arrayWrites`: rough counts for memory touches (instrumented estimates).
- `iterations`: outer passes (or left/right rounds).
- `earlyTerminated`: boolean indicating early exit.

Example CLI runs (typical)

```
=== Run ===
mode=random, algo=standard, earlyExit=true, n=20, seed=...
Input:  [ .... ]
Sorted: [ .... ]
Metrics: PerformanceTracker{comparisons=190, swaps=~19, arrayReads=...,
arrayWrites=..., iterations=19, earlyTerminated=false}
CSV:
comparisons,swaps,arrayReads,arrayWrites,iterations,earlyTerminated
190,19,....,....,19,false
```

Values depend on input and variant; early-exit becomes `true` for already sorted input, significantly reducing comparisons and iterations.

Interpretation

- On random data, Selection Sort performs ~n(n−1)/2 comparisons (**$\Theta(n^2)$**) regardless of early exit.
- On sorted input, early exit reduces work to **$\Theta(n)$**, which you can show by running with `mode=sorted`.

---

10) Edge Cases & Robustness

- **Empty / single-element arrays:** no action, metrics remain near zero.
- **Large duplicates:** both variants handle equal keys; result is correct but not stable.
- **Integer range:** uses `int`; for other primitives or generics, the design can be templated.

---

11) Limitations

- Not stable in current form.
- $\Theta(n^2)$ time makes it unsuitable for very large datasets (demonstration/teaching use).
- Early exit adds a linear check per pass (negligible vs overall $\Theta(n^2)$, but worth noting).

---

12) Possible Extensions

- **Stable selection sort** (shift instead of swap).
- **Generic comparator-based** API (`Comparator<T>` with `T[]`).
- **Micro-benchmarks** with JMH (Java Microbenchmark Harness).
- Export metrics to CSV files or plot charts.

---

13) Conclusion

The project fulfills the assignment goals: correct Selection Sort implementations (standard & double-ended), early-exit optimization, measurable metrics, unit tests, and a straightforward CLI

benchmark. The code is modular, easy to run, and suitable for in-class demos and basic empirical analysis.