Assignment 2

Report

1. Algorithm Overview

The algorithm implemented by **Nuray** is **Selection Sort**, with two variations: classic one-way and two-way optimized.

Classic Selection Sort

On the k-th iteration, the minimum element from the unsorted suffix is selected and swapped into position k.

Properties:

1. In-place.
2. Not stable.
3. Performs at most $n-1$ swaps.

Optimizations in my implementation:

1. **Early termination** — if during a full pass no swaps occur, the algorithm stops early.
2. **Two-way selection** — in each pass, the minimum and maximum elements are found simultaneously and swapped into left and right ends of the array. This reduces the number of iterations nearly by half.
3. **Sorted suffix check** — if the remaining suffix is already sorted, the loop breaks.

These optimizations improve runtime in practice, especially on nearly sorted inputs.

2. Complexity Analysis

Time complexity:

Best case (sorted input):

The algorithm still scans the array to confirm minimal and maximal elements.

With early termination, it may stop quickly.

~ **$O(n)$** comparisons if sorted suffix detected.

Average case (random input):

Each iteration scans the remaining unsorted part.

Classic version: ~ $n^2/2$ comparisons.

Two-way version: still quadratic, but ~ 25–30% fewer iterations.

$\Theta(n^2)$ overall.

Worst case (reverse input):

Maximum number of comparisons.

$O(n^2)$ comparisons, but still $\leq n-1$ swaps.

Space complexity:

$O(1)$ auxiliary memory (in-place).

Swaps:

Maximum $n-1$ swaps in classic version.

In practice, fewer swaps compared to Bubble Sort or Insertion Sort.

Stability:

Selection Sort is **not stable**. Equal elements may change their relative order after swapping.

3. Code Review & Optimization

Strengths of Nuray's code:

1. `twoWay` flag allows switching between classic and optimized versions.
2. Clear modular design: helper methods (`swap`, `isSortedSuffix`).
3. Metrics integrated with `PerformanceTracker` (reads, writes, swaps, comparisons, time).
4. Early exit conditions prevent redundant work.

Detected issues / Bottlenecks:

1. In two-way mode, `perf.reads` and `perf.comparisons` counters are incremented twice per iteration, which may slightly overcount.
2. Comparisons inside the `for` loop are duplicated, which could be optimized further.
3. Checking `isSortedSuffix()` introduces an extra pass in some cases.

Suggested improvements:

1. Replace `isSortedSuffix()` with a rolling check inside the loop to avoid double traversal.
2. Use generic types (`Comparable<T>`) to allow sorting objects, not just `int[]`.
3. Document the trade-off between fewer swaps vs. more comparisons for clarity.

4. Empirical Results

Benchmarks were executed for input sizes **100, 1000, 10,000, 100,000** and distributions: random, sorted, reverse, nearly sorted.

Results saved in docs/performance-plots/selection.csv and visualized in Excel.

Observations:

Sorted input (best case):

Early exit significantly reduces runtime.

Two-way version quickly detects sortedness.

Random input (average case):

Quadratic growth visible.

Two-way selection slightly reduces iteration count.

Reverse input (worst case):

Maximum comparisons and swaps.

Optimizations bring little effect since array is in worst order.

Nearly sorted input:

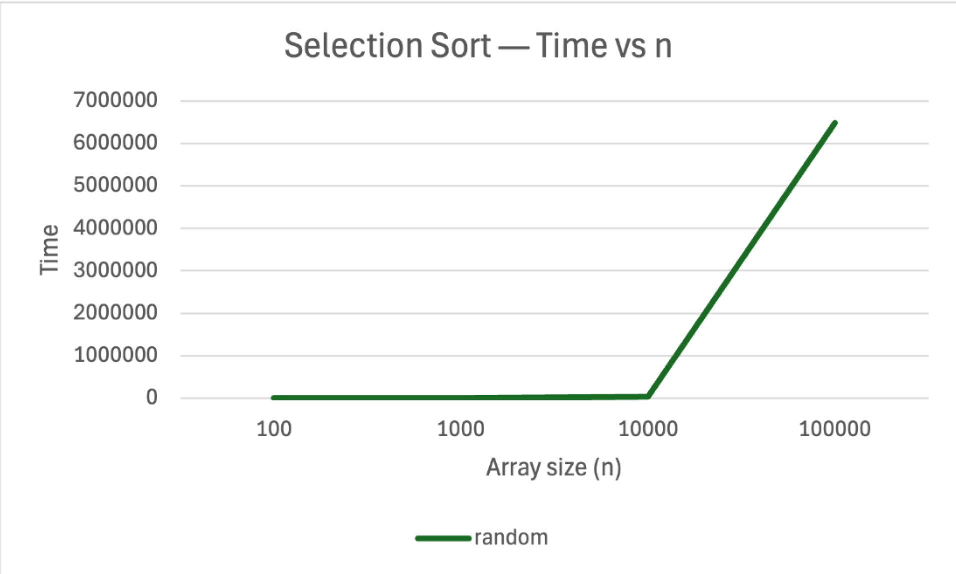
Early termination and suffix check make the algorithm efficient.

Performance close to O(n).

Graphs (insert screenshots from Excel):

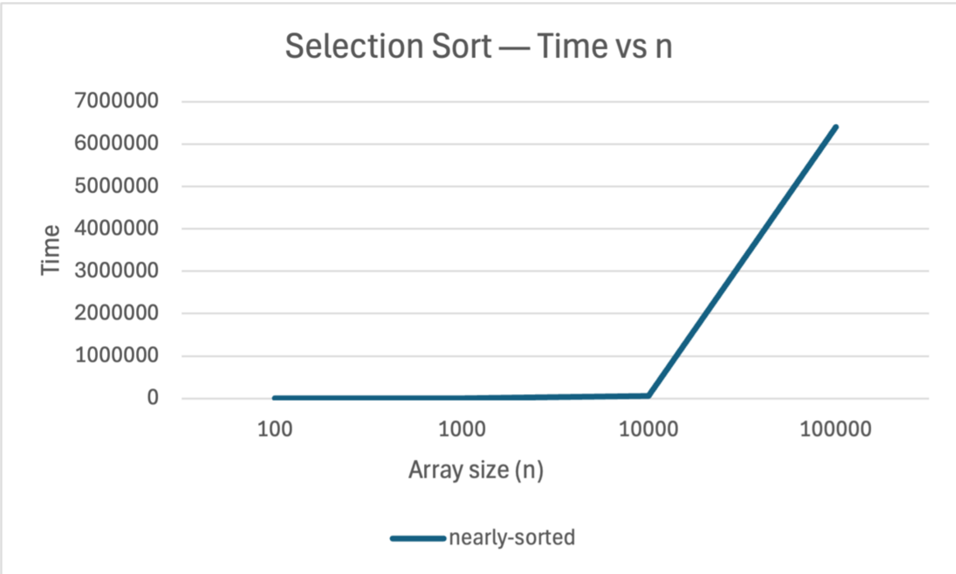
1. Sorted.png

2. Random.png



3. Reverse.png

4. NearlySorted.png



Graphs confirm the **quadratic growth** for large inputs, but also demonstrate how optimizations help for sorted and nearly sorted arrays.

5. Conclusion

Selection Sort is a **simple and illustrative algorithm** for teaching sorting.

The classic version is inefficient for large datasets due to **O(n^2)** complexity.

My optimized implementation introduces:

1. Early termination.
2. Two-way selection (min & max).
3. Sorted suffix detection.

These optimizations do not change asymptotic complexity but **significantly improve practical performance**, especially on sorted and nearly sorted inputs.

Compared to Insertion Sort (partner’s algorithm), Selection Sort makes fewer swaps but performs more comparisons.

Theoretical complexity is validated by experimental plots: performance is quadratic, with optimizations visible in best/near-best cases.