Algorithm Overview

Selection Sort

Selection sort repeatedly selects the smallest element from the unsorted tail of the array and swaps it into the next position of the sorted prefix. On iteration i (0-based), it finds index minIndex of the minimum in array[i..n-1] and swaps array[i] and array[minIndex].

Optimized Selection Sort

sortOptimized adds an O(n) pre-check: scan once to determine if the array is already non-decreasing. If yes, return early (zero further work). This turns an otherwise O(n²) best-case into O(n) for already-sorted inputs.

Complexity Analysis

Algorithm	Selection Sort	Optimized Selection Sort	Insertion sort
Time complexity	$\Theta(n^2)$	Best: $\Theta(n)$ Average/Worst: $\Theta(n^2)$	Best: $\Theta(n)$ Average/Worst: $\Theta(n^2)$
Space complexity	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$

Time complexity justification by induction:

$T(n) = T(n - 1) + c \times n = (T(n - 2) + c \times (n - 1)) + c \times n = \ldots = T(1) + c \times \sum_{k=2}^{n} k = \frac{c}{2} \times n^2 + O(n) = \Theta(n^2)$.

Optimized Selection Sort iterates through each element and checks order, if the array is already sorted, time complexity will be Θ(n).

Code Review

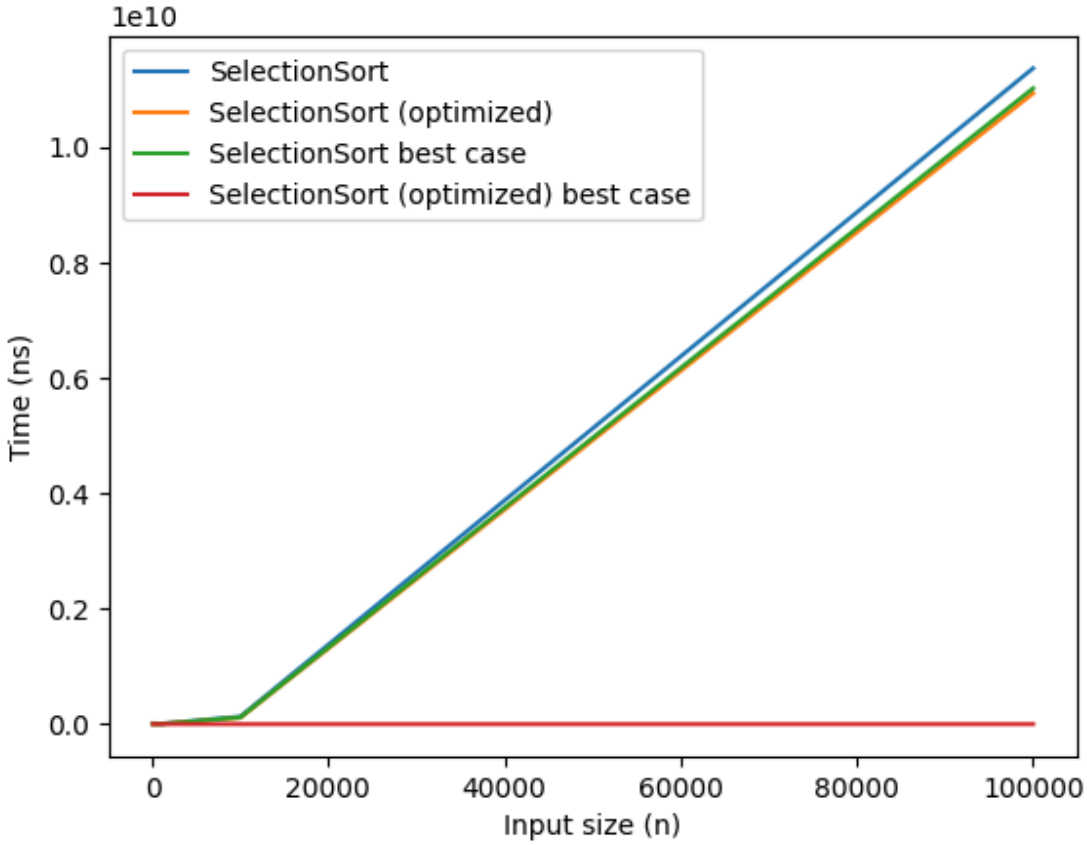
The project is well-structured: packages `algorithms`, `metrics`, `cli`, and tests separate core logic, and benchmarking
`PerformanceTracker` provides detailed instrumentation of comparisons, swaps, array accesses, and memory allocations.
Test coverage includes empty arrays, single elements, duplicates, and optimized path, which is excellent for functional correctness.

Unnecessary swaps in `SelectionSort`

```
int temp = array[minIndex];
array[minIndex] = array[i];
array[i] = temp;
```

This swap executes even when `minIndex == i`. That results in redundant reads/writes. For large arrays, this adds measurable overhead.
Solution: skip redundant swaps: `if (minIndex != i)` swap.

Empirical Results



The empirical plot validate theoretical $\Theta(n^2)$ growth for random cases; the reference quadratic curve overlays well with measured times. The optimized variant shows $\Theta(n)$ behavior in the best-case (sorted inputs), validating the claim that Optimized Selection Sort improves best-case complexity.