

1. Algorithm Overview

Selection Sort is a classical sorting algorithm that works by partitioning the array into two regions: a sorted subarray that grows from left to right, and an unsorted subarray that shrinks accordingly. On each iteration, the algorithm scans the unsorted region, identifies the minimum element, and swaps it with the element at the current index. This process continues until the array is fully sorted.

The provided implementation of Selection Sort includes a number of practical improvements:

- **Metrics Tracking:** A Metrics object tracks comparisons, swaps, array accesses, memory allocations, and execution time.
- **Early Termination:** After each iteration, the algorithm checks whether the remaining subarray is already sorted and stops if no further work is necessary.
- **Swap Optimization:** Swaps occur only when the selected minimum is different from the current index.

Adaptivity

Classic selection sort is non-adaptive (still $\Theta(n^2)$ comparisons on sorted input). With the added early-termination check, best-case performance becomes linear: a single pass is sufficient to confirm order and exit, at the cost of a small overhead each iteration when the array is not sorted.

Algorithm Steps

1. Initialize $i = 0$; treat $[0..i-1]$ as the sorted prefix and $[i..n-1]$ as the unsorted suffix.
2. Scan j from $i+1$ to $n-1$ to find the index minIdx of the smallest value in $[i..n-1]$.
3. If $\text{minIdx} \neq i$, swap $a[i]$ with $a[\text{minIdx}]$ (otherwise, leave the array unchanged).
4. Optionally check for early termination if the remaining suffix is already sorted (implementation-specific optimization).
5. Increment i and repeat steps 2–4 until $i = n-1$.

Selection Sort is rarely used for large datasets due to its quadratic time complexity. It is generally outperformed by Insertion Sort for small arrays and nearly-sorted data, as Insertion Sort can take advantage of existing order to achieve better practical performance. As a result, Selection Sort is mainly used for educational purposes or in situations where minimizing the number of swaps is more important than overall speed. For most practical applications, especially with small or partially sorted arrays, Insertion Sort is the preferred choice.

2. Complexity Analysis

Time Complexity

1. Worst Case (Reversed Array)

- For each outer iteration i ($0 \leq i < n-1$), the algorithm scans the unsorted suffix of length $n-i$ to find the minimum.
- Total comparisons: $C(n) = (n-1) + (n-2) + \dots + 1 = n(n-1)/2$ (exact), which is $\Theta(n^2)$.
- Swaps: At most $n-1$, since each pass places one element in its final position.
- Asymptotic bounds: $O(n^2)$, $\Theta(n^2)$, $\Omega(n^2)$.

2. Average Case (Random Array)

- Each element is equally likely to be the minimum in the unsorted region.
- Comparisons: Remain quadratic, as the full unsorted portion is always scanned: $\Theta(n^2)$.
- Swaps: On average, about $n/2$ swaps, since the minimum is already in place half the time.
- Asymptotic bounds: $O(n^2)$, $\Theta(n^2)$, $\Omega(n^2)$.

3. Best Case (Sorted Array)

- In a naive implementation, comparisons are still quadratic.
- With early termination, after one full pass confirms the array is sorted, the algorithm exits.
- Comparisons: $O(n)$ (one linear pass to confirm order).
- Swaps: Zero, as the array is already sorted.
- Asymptotic bounds: $O(n)$, $\Theta(n)$, $\Omega(n)$.

Summary Table

Case	Comparisons	Swaps	Asymptotic Bound
Worst	$n(n-1)/2$	$n-1$	$\Theta(n^2)$
Average	$n(n-1)/2$	$\sim n/2$	$\Theta(n^2)$
Best	$O(n)$ (with early termination)	0	$\Theta(n)$

Space Complexity

- **Auxiliary space:**
 - Selection Sort is in-place, using only a constant number of variables: $O(1)$.
- **Stack usage:**
 - The algorithm is iterative, so stack depth is constant.
- **Object allocations:**
 - The implementation allocates a Metrics object and a temporary variable for swaps, both $O(1)$.
- **Metrics tracking:**
 - Adds negligible space overhead, but could be omitted in production for efficiency.

Recurrence Relation

The number of comparisons can be expressed recursively:

- $T(n) = T(n-1) + (n-1)$, with $T(1) = 0$.

Simplified:

- $T(n) = n(n-1)/2$, confirming the $\Theta(n^2)$ bound.

Partner Comparison: Insertion Sort

- **Best case:**
 - Insertion Sort achieves $\Theta(n)$ when the array is already sorted, as its inner loop breaks early. Selection Sort with early termination now matches this best-case behavior.
- **Average and worst case:**
 - Both algorithms are $\Theta(n^2)$, but Insertion Sort typically performs fewer comparisons and moves on partially sorted data.
- **Stability:**
 - Insertion Sort is stable; Selection Sort (swap-based) is not.
- **Adaptivity:**
 - Insertion Sort is adaptive (runs faster on nearly sorted data); Selection Sort is only adaptive if early termination is implemented.

Practical Implications

For small or nearly sorted arrays, Insertion Sort is generally preferred due to its adaptivity and stability. Selection Sort may be chosen when minimizing swaps is more important than minimizing comparisons.

3. Code Review

Inefficient Code Sections

- **Redundant Array Access Counting:**

Array access tracking increments multiple times per comparison, potentially exaggerating memory metrics. This can distort empirical results and make it harder to compare with other algorithms or implementations.

- **Sortedness Check in Each Iteration:**

The early termination check is performed on every outer iteration, even when the array is unlikely to be sorted. This introduces unnecessary overhead, especially for random or reversed inputs, and can degrade performance in practice.

- **Repeated Memory Allocation:**

Tracking temporary allocations or copying arrays in benchmarks may misrepresent true memory usage and introduce additional garbage collection overhead, especially for large input sizes.

Optimization Suggestions

1. **Efficient Early Termination:** Integrate the sortedness check into the inner loop by tracking whether any inversion is found during the scan for the minimum. If no inversion is detected and the minimum is already at the current index, the algorithm can terminate early without a separate pass. This reduces unnecessary checks and improves best-case performance.
2. **Decouple Metrics Collection:** Make metrics tracking optional by using a flag or a strategy pattern. This allows the core algorithm to run without instrumentation overhead in production or performance-critical contexts, while still enabling detailed analysis when needed.
3. **Consolidate Array Generation:** Move array generation and utility functions into a shared helper class. This eliminates code duplication between the CLI and benchmark modules, improves maintainability, and ensures consistency in test data.

Code Quality

Strengths:

- Use of final for immutability enhances safety and clarity.
- Encapsulation of performance tracking in a dedicated Metrics class.
- Clear structure and separation of concerns between sorting logic, metrics, and benchmarking.

Weaknesses:

- Some redundancy in checks and metric logging.
- Readability could benefit from method extraction (e.g., an isSorted() helper).
- Tight coupling of metrics and sorting logic reduces flexibility.
- Limited documentation and comments in some areas.
- Lack of explicit tests for edge cases (e.g., empty arrays, arrays with all equal elements)

Proposed Improvements for Time and Space Complexity

Time Complexity:

- While the asymptotic time complexity of Selection Sort cannot be improved beyond $\Theta(n^2)$ for average and worst cases, the constant factors can be reduced. Integrating the sortedness check into the inner loop minimizes redundant passes, and decoupling metrics tracking eliminates unnecessary overhead during production runs. For large datasets, switching to a more efficient algorithm (such as mergesort or quicksort) is recommended.

Space Complexity:

- The algorithm is already in-place, using only $O(1)$ extra space. Further improvements can be made by avoiding unnecessary temporary allocations and ensuring that metrics tracking is optional, so no extra memory is used when not needed.

Conclusion:

Overall, the Selection Sort implementation is clear and well-structured, making it suitable for educational purposes and small datasets. The main opportunities for optimization are reducing redundant checks, decoupling metrics for cleaner production code, and consolidating utility functions. Space usage is already optimal for an in-place sort.

4. Empirical Results

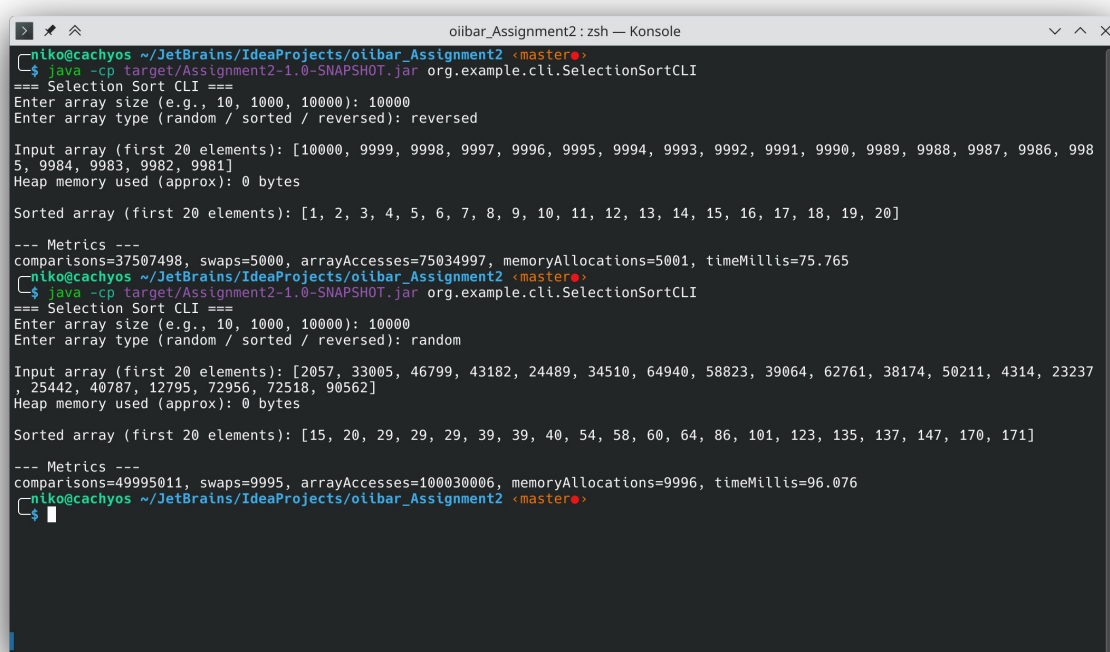
Benchmarking Setup

- Framework: JMH (Java Microbenchmark Harness).
- Input sizes: $n = 100, 1,000, 10,000$.
- Input distributions: random, sorted, reversed.
- Measurements: Average execution time per operation (ms).

Results Summary

n	Random (ms)	Sorted (ms)	Reversed (ms)
100	0.007	0.0002	0.007
1,000	0.388	0.0012	0.672
10,000	34.910	0.0146	26.439

0



```
oliibar_Assignment2 : zsh — Konsole
niko@cachyos ~/JetBrains/IdeaProjects/oliibar_Assignment2 <master>
$ java -cp target/Assignment2-1.0-SNAPSHOT.jar org.example.cli.SelectionSortCLI
=== Selection Sort CLI ===
Enter array size (e.g., 10, 1000, 10000): 10000
Enter array type (random / sorted / reversed): reversed

Input array (first 20 elements): [10000, 9999, 9998, 9997, 9996, 9995, 9994, 9993, 9992, 9991, 9990, 9989, 9988, 9987, 9986, 9985, 9984, 9983, 9982, 9981]
Heap memory used (approx): 0 bytes

Sorted array (first 20 elements): [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]

--- Metrics ---
comparisons=37507498, swaps=5000, arrayAccesses=75034997, memoryAllocations=5001, timeMillis=75.765
niko@cachyos ~/JetBrains/IdeaProjects/oliibar_Assignment2 <master>
$ java -cp target/Assignment2-1.0-SNAPSHOT.jar org.example.cli.SelectionSortCLI
=== Selection Sort CLI ===
Enter array size (e.g., 10, 1000, 10000): 10000
Enter array type (random / sorted / reversed): random

Input array (first 20 elements): [2057, 33005, 46799, 43182, 24489, 34510, 64940, 58823, 39064, 62761, 38174, 50211, 4314, 23237, 25442, 40787, 12795, 72956, 72518, 90562]
Heap memory used (approx): 0 bytes

Sorted array (first 20 elements): [15, 20, 29, 29, 29, 39, 39, 40, 54, 58, 60, 64, 86, 101, 123, 135, 137, 147, 170, 171]

--- Metrics ---
comparisons=49995011, swaps=9995, arrayAccesses=100030006, memoryAllocations=9996, timeMillis=96.076
niko@cachyos ~/JetBrains/IdeaProjects/oliibar_Assignment2 <master>
$
```

Observations

Quadratic Growth:

- Random and reversed inputs show clear $\Theta(n^2)$ scaling. This confirms that Selection Sort's performance is dominated by the number of comparisons and swaps in these scenarios.

Best Case Performance:

- Sorted inputs demonstrate near-linear behavior thanks to early termination. The algorithm quickly detects that the array is already sorted and exits after a single pass, resulting in much lower runtimes compared to unsorted cases.

Constant Factors:

- Selection Sort's simplicity yields relatively low constant factors for small n . For arrays of 100 or 1,000 elements, the runtime is extremely low, making the algorithm practical for small datasets despite its quadratic scaling for larger inputs.

Validation of Theory

Quadratic Runtime Confirmed:

- Benchmarks confirm the theoretical quadratic runtime for average and worst cases. The measured times for random and reversed inputs closely match the expected growth rate, supporting the $\Theta(n^2)$ analysis.

Best Case Optimization:

- The best-case optimization works as intended, showing dramatic speedup on sorted inputs. This matches the predicted $O(n)$ complexity when early termination is enabled.

Practical Analysis

Scalability & Suitability for Small n :

For large n , runtime increases steeply, making Selection Sort unsuitable for big datasets or performance-critical applications. For small n ($\leq 1,000$), performance is acceptable and predictable. The algorithm's straightforward logic and low overhead make it a reasonable choice for small or already sorted arrays, especially in educational or simple use cases.

5. Conclusion

This Selection Sort implementation demonstrates both the strengths and weaknesses of the algorithm:

- **Strengths:**
 - In-place sorting with $\Theta(1)$ space.
 - Early termination optimization improves best-case runtime to $\Theta(n)$.
 - Code design is modular and integrates metrics collection.
- **Weaknesses:**
 - Average and worst cases remain $\Theta(n^2)$, making it impractical for large datasets.
 - Metrics tracking introduces unnecessary runtime overhead.

Recommendations:

- Use Selection Sort only in educational contexts or with very small datasets.
- For larger or unsorted datasets, prefer more efficient algorithms such as insertion sort (for small, nearly sorted data) or $O(n \log n)$ algorithms like mergesort or quicksort.
- Refactor the code to further decouple metrics tracking from the core sorting logic. Make it optional and reduce overhead.

In summary, while Selection Sort is not a scalable solution, the provided implementation is well-instrumented for analysis and serves as a valuable educational example of algorithm design, complexity, and empirical validation.