Assignment2 DAA
Full name: Ansar Keles
Group: SE-2427

Algorithm Overview: Selection Sort

Selection Sort is a straightforward comparison-based sorting algorithm that operates by repeatedly selecting the smallest (or largest) element from the unsorted portion of the list and placing it at the correct position in the sorted portion. The array is conceptually divided into two parts: the sorted section (initially empty) and the unsorted section (initially the entire array).

During each iteration, the algorithm searches the unsorted section to find the smallest element and swaps it with the first unsorted element, thereby expanding the sorted section by one and reducing the unsorted section. This process continues until all elements are sorted in ascending order.

The idea behind Selection Sort is similar to organizing a row of books alphabetically: repeatedly scan the shelf to find the smallest title, then move it to the front.

Formally, given an array A of size n, Selection Sort uses two nested loops:

- The outer loop iterates over each position i from 0 to n−2.
- The inner loop finds the smallest element among the remaining n−i−1 elements and records its index.

After the inner loop finishes, the smallest element is swapped with A[i].

Selection Sort is simple to understand and implement, making it ideal for teaching algorithmic fundamentals. However, it is not adaptive (performance doesn't improve for sorted data) and not stable (equal elements may change their relative order). Its main advantages are minimal memory usage and predictably consistent performance regardless of input order.

Complexity Analysis

1. Time Complexity

Selection Sort's time complexity remains consistent across all scenarios because it always scans the remaining unsorted section completely for each element, regardless of input order.

1) Best Case — $\Omega(n^2)$:

Even if the array is already sorted, the algorithm must still traverse the entire unsorted section to confirm that no smaller element exists. Therefore, the number of comparisons does not decrease.

→ Time Complexity: $\Omega(n^2)$

2) Average Case — $\Theta(n^2)$:

For random input data, Selection Sort still performs the same number of comparisons and roughly the same number of swaps as in other cases.
The total number of comparisons is determined by the nested loops:
\sum_{i=1}^{n-1} (n - i) = \frac{n(n-1)}{2}
This leads to a quadratic average-case time complexity.
→ Time Complexity: $\Theta(n^2)$

3) Worst Case — $O(n^2)$:

In the worst case (e.g., descending order), Selection Sort still performs exactly the same number of comparisons as in the best and average cases because it always scans all remaining elements.
→ Time Complexity: $O(n^2)$

Thus, Selection Sort's performance is independent of input order, making it predictable but inefficient for large datasets.

2. Space Complexity

Selection Sort is an in-place algorithm, requiring only a few extra variables to track indices and temporary values.

1)Auxiliary space: O(1)
2)In-place: Yes

This makes it memory-efficient and ideal for environments with limited memory resources.

3. Number of Comparisons

Selection Sort always performs the same number of comparisons, regardless of data order:
\frac{n(n-1)}{2}

This is because, for each iteration i, it compares the current minimum with all remaining unsorted elements.

Hence:

1)Best Case: $\Omega(n^2)$
2)Average Case: $\Theta(n^2)$
3)Worst Case: $O(n^2)$

4. Number of Swaps and Array Accesses

Swaps:

One of Selection Sort's notable advantages is that it performs at most (n−1) swaps. Unlike Bubble or Insertion Sort, which may swap elements repeatedly, Selection Sort performs only one swap per outer loop iteration.
→ Swaps: O(n)

Array Accesses:

Each comparison and swap involves element access. Since comparisons dominate, total array accesses grow roughly in proportion to $O(n^2)$.

5. Comparison with other algorithms

Algorithm	**Avg. Complexity**	**Swaps**	**Adaptivity**	**Stability**	**Space**
Selection	$\Theta(n^2)$	O(n)	No	No	O(1)
Insertion	$\Theta(n^2)$	$O(n^2)$	Yes	Yes	O(1)
Bubble	$\Theta(n^2)$	$O(n^2)$	Yes	Yes	O(1)
Merge	Θ(n log n)	—	Yes	Yes	O(n)
Quick	Θ(n log n)	—	No	No	O(log(n))

6. Summary

Case.	Time complexity.	Space complexity.	Comparisons.	Swaps.	Nature
Best	$\Omega(n^2)$	O(1)	n(n−1)/2	≤ n−1	Quad
Average	$\Theta(n^2)$	O(1)	n(n−1)/2	≤ n−1	Quad
Worst.	$O(n^2)$	O(1)	n(n−1)/2	≤ n−1	Quad

Code Review

1. Code Structure and Functionality Overview

The project consists of three main Java classes:

1)SelectionSort — Implements the sorting logic with integrated tracking of operations.

2)BenchmarkRunner — Generates random arrays, executes sorting, and logs metrics into a CSV file.

3)PerformanceTracker — Measures execution time, comparisons, swaps, and array accesses.

The design is modular and clear, separating algorithm logic from benchmarking, ensuring maintainability and ease of extension for additional algorithms.

2. Strengths

1)Efficient Swap Usage:
Only one swap per iteration reduces overhead and improves stability of performance.

2)Clear and Modular Design:
Each class has a single responsibility — improving readability and reusability.

3)Accurate Metrics Tracking:
The PerformanceTracker provides detailed insights into algorithmic behavior beyond time measurement alone.

4)Scalable Testing:
BenchmarkRunner executes tests on arrays of increasing sizes (100, 1,000, 10,000, 100,000), effectively demonstrating scalability trends.

3. Inefficiencies and Improvement Areas

a) Redundant Array Access Tracking:
The algorithm may increment array access counts multiple times for the same element read/write.
→ Fix: Record only unique access operations.

b) Lack of Early Stopping Optimization:
Selection Sort always scans the entire unsorted section, even when no further sorting is required.
→ Fix: Add a flag to detect if the remaining elements are already in order (minor constant-time improvement).

c) Limited Dataset Variation:
Benchmarks only use random arrays.
→ Fix: Include sorted and reversed arrays for completeness.

d) I/O Performance:
Repeatedly opening CSV files can slow down large experiments.
→ Fix: Maintain a single open file writer.

4. Optimization Recommendations

1)Hybrid Sorting: Use Selection Sort for small subarrays inside a faster algorithm like Quick Sort.

2)Parallelization: For educational purposes, selection of minimum values could be parallelized.

3)Improved Metric Collection: Optimize PerformanceTracker to minimize method call overhead.

Empirical Results

1. Experimental Setup

Tests were conducted using the same BenchmarkRunner framework on datasets of sizes 100, 1,000, 10,000, and 100,000, filled with random integers.
Each experiment recorded:

1)Comparisons
2)Swaps
3)Array accesses
4)Execution time (ms)

The results were written to a CSV file (benchmark_results_selection.csv) for visualization and analysis.

2. Observed Results and Trends

Empirical observations matched theoretical predictions:

1)Comparisons: Increased quadratically with input size.
2)Swaps: Remained very low ($\leq n-1$), consistent across all dataset sizes.
3)Execution Time: Grew proportionally to n^2, confirming the quadratic growth pattern.

When visualized on a time vs. input size plot, execution time formed a clear parabolic curve, characteristic of quadratic algorithms.

3. Validation of Complexity

The results aligned perfectly with theoretical expectations:

1)Best Case: $\Omega(n^2)$ — Still quadratic since full scans are required.
2)Average Case: $\Theta(n^2)$ — Random input results in consistent quadratic growth.
3)Worst Case: $O(n^2)$ — Reversed arrays show identical complexity.

Thus, empirical results confirm that Selection Sort's runtime is data-order independent.

4. Analysis of Constant Factors

Selection Sort benefits from a low constant factor due to fewer swaps. However, repeated comparisons dominate runtime:

- Array bounds checking in Java slightly increases per-comparison time.
- The tracker's counter increments add negligible overhead.

For small datasets, these constants dominate runtime, but for large inputs, the quadratic trend becomes apparent.

5. Practical Insights

1)Efficiency for Small Inputs: Performs reasonably well for datasets under 500 elements.

2)Predictable Runtime: Same number of operations regardless of input order.

3)Memory Efficiency: Uses O(1) space — suitable for embedded or memory-limited systems.

4)Scalability Limitation: Quickly becomes impractical beyond 10,000 elements.

6. Summary of Empirical Findings

1)Performance grows quadratically with input size.

2)Swaps remain minimal, confirming algorithm's stability in operation count.

3)The algorithm's time complexity is independent of data order.

4)While reliable and memory-efficient, Selection Sort is not suitable for large datasets.

Conclusion

The evaluation of the Selection Sort algorithm reveals that it is a simple and reliable sorting method best suited for small-scale or memory-constrained applications. Its defining characteristics—constant space usage, minimal swaps, and predictable operation count—make it useful for educational purposes and systems where consistency is more important than speed.

However, both theoretical and empirical analyses confirm that Selection Sort has $O(n^2)$ time complexity in all cases, making it inefficient for large or performance-critical applications. Even when the dataset is already sorted, the algorithm performs the same number of comparisons, resulting in no runtime improvement.

Empirical experiments demonstrated a clear quadratic trend in both execution time and number of comparisons. Despite this, the algorithm's simplicity, in-place nature, and deterministic performance ensure its continued relevance in teaching environments and as a component within hybrid sorting algorithms.

In conclusion, Selection Sort's predictability and low space requirements are outweighed by its poor scalability. Future work could focus on combining Selection Sort with faster algorithms for small subarrays or specialized applications where memory constraints are critical and dataset size remains limited.