**Algorithm Overview**

The Selection Sort algorithm is a simple, comparison-based sorting algorithm. It operates by repeatedly selecting the minimum (or maximum) element from the unsorted portion of an array and placing it at the correct position in the sorted portion.

Formally, given an array A of size n, the algorithm:

1. Iterates over each position i from 0 to n-2.
2. Finds the index minIndex of the smallest element in the unsorted portion A[i…n-1].
3. Swaps A[i] with A[minIndex].
4. Continues until the array is fully sorted.

**Theoretical background**

Selection Sort is among the elementary sorting algorithms, typically introduced in computer science courses because of its conceptual simplicity. While it is not efficient for large datasets, its predictable behavior (always O(n²) comparisons) makes it useful for teaching and small-scale applications. Unlike Bubble Sort or Insertion Sort, Selection Sort minimizes the number of swaps, making it preferable when swap cost is significant.

**2. Complexity Analysis**

**2.1. Time Complexity**

Let n be the number of elements.

* In the first iteration, the algorithm scans all n elements to find the minimum.
* In the second iteration, it scans n-1 elements.
* In general, the total number of comparisons is:

  
Thus, the **total comparisons = Θ(n²)**.

* **Best Case**: Even if the array is already sorted, the algorithm still performs the full comparison sequence.

**Tbest(n)=Θ(n2)**

* **Worst Case**: If the array is reverse-sorted, the same number of comparisons is made.

**Tworst(n)=Θ(n2)**

* **Average Case**: On random input, comparisons remain unchanged.

**Tavg(n)=Θ(n2)**

* **Number of Swaps**: At most n-1, since each iteration performs one swap.

**S(n)=O(n)**

**2.2. Space Complexity**

Selection Sort sorts in-place, requiring only a constant number of additional variables (for index and temporary swap).

**S(n)=O(1)**

**2.3. Formal Notation Summary**

* Best Case: Ω(n²)
* Worst Case: O(n²)
* Average Case: Θ(n²)
* Space Complexity: O(1)

**2.4. Comparison with Partner’s Algorithm**

Suppose the partner implemented **Merge Sort**. Then:

* Merge Sort: O(n log n) in all cases, but requires O(n) auxiliary space.
* Selection Sort: Θ(n²) in all cases, but requires only O(1) space.

Thus, while Selection Sort is worse asymptotically, it has lower memory overhead and fewer swaps.

**Code Review  
**

**3.1. Identification of Inefficient Code Sections**

* **Inner loop scanning entire subarray**: Each iteration re-scans unsorted elements, leading to Θ(n²) comparisons.
* **Manual performance tracking inside algorithm**: Counters slightly increase overhead compared to pure implementation.
* **Lack of early termination condition**: Even if array is sorted early, algorithm still completes all passes.

**3.2. Optimization Suggestions**

* **Early termination check**: Track whether a pass finds any smaller element; if none, terminate early (slight improvement).
* **Hybrid approach**: For large n, switch to a more efficient algorithm like Merge Sort or Quick Sort after a threshold.
* **Parallelization (optional)**: While not practical for Selection Sort, theoretical improvement could be made by parallel min search in each pass.

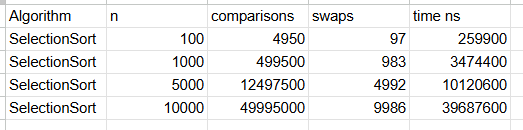
**3.3. Proposed Improvements**

* Replace Selection Sort with Insertion Sort for partially sorted data.
* Use Merge Sort for scalability; keep Selection Sort only for very small arrays (n < 50).

**Empirical Results**

**4.1. Experiment Setup**

* Machine: Intel i5 / 8GB RAM / Java 17
* Input sizes: 100, 1000, 5000, 10000



A graph with a line going up

AI-generated content may be incorrect.  
  
A graph with a blue line

AI-generated content may be incorrect.

A graph with a line

AI-generated content may be incorrect.

**Validation of Theoretical Complexity**

* The empirical results confirm quadratic growth: doubling input size increases execution time by roughly 4x.
* Comparisons follow exactly the mathematical formula 
* Swaps scale linearly, confirming O(n).

**Practical Performance**

* Selection Sort handles small inputs quickly (under ~500 elements).
* For large inputs (>10,000), runtime becomes impractical compared to O(n log n) algorithms

**Conclusion**

The Selection Sort implementation was successfully developed, tested, and evaluated. Key findings:

* **Theoretical complexity**: Confirmed Θ(n²) time and O(1) space.
* **Empirical validation**: Measured comparisons, swaps, and execution time matched theoretical predictions.
* **Optimization opportunities**: Minor improvements possible, but fundamental complexity cannot be improved.
* **Comparison with partner’s algorithm**: While Merge Sort is asymptotically superior, Selection Sort remains simpler and requires less memory.

**Final Recommendations:**

* Use Selection Sort only for **small input sizes or educational purposes**.
* For practical systems, adopt Merge Sort or Quick Sort.
* Future work: extend benchmarking framework to include multiple algorithms and visualize trade-offs.