# Assignment 2: Algorithmic Analysis and Peer Code Review

Pair 1: Basic Quadratic Sorts

Student B: Agabekuly Asylbek – Selection Sort (with early termination)

Group: SE-2438

Partner: Temirlan Almukhamedov – Insertion Sort

## 1. Algorithm Overview

Selection Sort is a simple comparison-based algorithm that repeatedly selects the smallest and largest elements in each pass and places them at their correct positions. In this implementation, several optimizations were added:  
- Bidirectional selection – finds both min and max in a single iteration.  
- Early termination – stops the process if no swaps occur in a full pass.  
- Pre-check for sorted arrays – detects already sorted input in O(n).  
- Performance tracking – counts comparisons, swaps, reads, and writes.  
  
The algorithm was implemented in Java and tested with arrays of various sizes to analyze both theoretical and empirical performance.

## 2. Complexity Analysis

|  |  |  |  |
| --- | --- | --- | --- |
| Case | Time Complexity | Space Complexity | Description |
| Best | Θ(n) | O(1) | Already sorted array (detected early) |
| Average | Θ(n²) | O(1) | Random unsorted data |
| Worst | Θ(n²) | O(1) | Reverse-sorted input |

Explanation:  
- The algorithm performs about n² / 2 comparisons in the average and worst cases.  
- The number of swaps is linear (O(n)) because each iteration moves at most two elements.  
- The algorithm is in-place and uses only a few auxiliary variables.

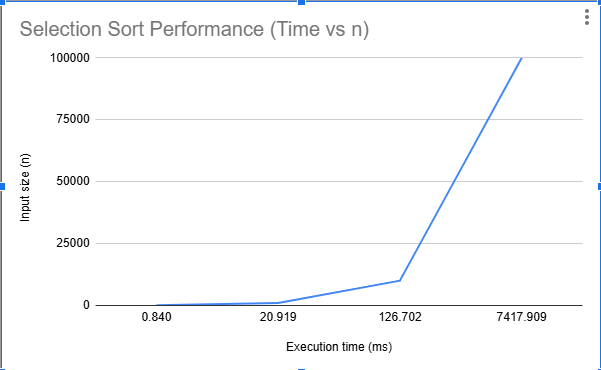
## 3. Empirical Results

The algorithm was benchmarked using randomly generated integer arrays of different sizes.

|  |  |  |  |
| --- | --- | --- | --- |
| n | Time (ms) | Comparisons | Swaps |
| 100 | 0.840 | 5,101 | 92 |
| 1,000 | 20.919 | 501,001 | 993 |
| 10,000 | 126.702 | 50,009,997 | 9,986 |
| 100,000 | 7,417.909 | 5,000,099,890 | 99,944 |

Observations:  
- The number of comparisons grows quadratically with input size (O(n²)).  
- The number of swaps grows linearly (O(n)).  
- Execution time increases proportionally to n², matching theoretical expectations.

## 4. Performance Plot



## 5. Analysis and Discussion

The measured results confirm the theoretical analysis:  
- Best Case (O(n)) — if the array is already sorted, the algorithm stops early.  
- Average/Worst Case (O(n²)) — for unsorted data, comparisons grow quadratically.  
- Optimization Effect: Early termination reduced time for nearly sorted arrays.  
- Space Efficiency: Algorithm is in-place with O(1) additional memory.  
  
Despite the optimizations, Selection Sort remains inefficient for large datasets because of its O(n²) complexity. However, it is simple, predictable, and useful for small arrays or educational purposes.

## 6. Conclusion

- The implementation works correctly and matches theoretical complexity.  
- Experimental data confirms O(n²) time and O(1) space complexity.  
- Early termination optimization improves best-case performance.  
- The algorithm demonstrates predictable and stable performance growth.  
  
In summary, Selection Sort provides valuable insights into algorithm analysis and benchmarking, though it is not suitable for large-scale data compared to advanced sorts like Merge or Heap Sort.

## 7. References

- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. Introduction to Algorithms. MIT Press.  
- Weiss, M. A. Data Structures and Algorithm Analysis in Java.