Analysis Report – Selection Sort

Alym Baisal SE-2437■Partner: Adilzhan Assanuly SE-2437

Course: Design Analysis of Algorithms

Introduction

This report provides a comprehensive analysis of the Selection Sort algorithm implemented by my partner, Adilzhan Assanuly. The purpose of this analysis is to evaluate the correctness, complexity, performance, and overall quality of the algorithm, as well as to suggest potential improvements.

Algorithm Overview & Pseudocode

Selection Sort is a simple comparison-based sorting algorithm that divides the array into a sorted and an unsorted region. It repeatedly selects the minimum element from the unsorted region and places it at the end of the sorted region.

Pseudocode:

```
for i from 0 to n-2:
    minIndex = i
    for j from i+1 to n-1:
        if arr[j] < arr[minIndex]:
            minIndex = j
    if minIndex != i:
        swap(arr[i], arr[minIndex])
```

Implementation Details

The provided implementation follows the standard Selection Sort logic with some additional optimizations. It creates a cloned copy of the input array to ensure immutability, and includes a check to detect if the remaining portion of the array is already sorted to terminate early. A PerformanceTracker class is used to record comparisons, swaps, array accesses, execution time, and memory usage.

Testing and Correctness

The algorithm has been thoroughly tested using JUnit tests, which cover the following cases:
- Null input handling (throws IllegalArgumentException)
- Empty arrays
- Single-element arrays
- Already sorted arrays
- Reverse-sorted arrays
- Random arrays with duplicates
- Arrays with all elements equal
- Large arrays (e.g., 1000 elements)
- Comparison of optimized vs. traditional Selection Sort

These tests demonstrate that the algorithm consistently produces correct results across a wide range of scenarios.

Complexity Analysis

Time Complexity:
- Best case: O(n^2)
- Average case: O(n^2)
- Worst case: O(n^2)

Although an early termination optimization is implemented, the algorithm’s overall complexity remains quadratic.

Space Complexity:
- O(n) if counting the cloned array created during sorting.
- O(1) additional space for an in-place version.

Performance Tracking and Benchmarking

The PerformanceTracker component provides a detailed empirical evaluation of the algorithm's behavior. It measures key performance metrics, including the number of comparisons, swaps, array accesses, execution time, and memory usage.

Benchmarking with arrays of different sizes (e.g., 100, 1000, 5000 elements) shows quadratic growth in execution time, consistent with the theoretical complexity. The tracker also confirms that the number of comparisons approaches n*(n-1)/2, while the number of swaps remains relatively low (~n).

Strengths of the Implementation

• The code is clearly structured and readable.
• Comprehensive unit testing ensures correctness.
• Performance metrics are detailed and insightful.
• Non-destructive sorting (via cloning) prevents side effects.
• Early termination optimization can improve performance on nearly sorted arrays.

Suggestions for Improvement

1. Consider offering an in-place sorting option for lower memory usage.
2. Simplify the early-termination logic for better readability.
3. Include automated export of benchmark results for analysis.
4. Compare Selection Sort performance with other algorithms (e.g., Insertion Sort, Merge Sort) for context.

Conclusion

The Selection Sort algorithm implementation analyzed in this report is correct, efficient within the limitations of its algorithmic complexity, and well-instrumented for performance evaluation. With minor refinements, it can serve as an excellent educational example and benchmarking tool.