Report A – Selection Sort Analysis

1. Algorithm Overview

Selection Sort is a simple comparison-based sorting algorithm. It operates by repeatedly selecting the smallest element from the unsorted portion of the array and swapping it with the first unsorted element. This process grows the sorted part of the array step by step. Selection Sort is often introduced in computer science courses due to its intuitive structure, even though it is inefficient for large datasets.

Illustration of process:
- Step 1: Find the minimum element in the entire array and swap with the first position.
- Step 2: Move to the next index, find the minimum from the remaining unsorted part, and swap.
- Step 3: Repeat until array is fully sorted.

This algorithm has the advantage of a low number of swaps compared to Insertion Sort, but always performs a quadratic number of comparisons.

2. Complexity Analysis

Time Complexity:
- Best Case (Ω): Ω(n^2). Even if the input is already sorted, Selection Sort still scans the unsorted part.
- Worst Case (O): O(n^2). Each element requires scanning the rest of the array.
- Average Case (Θ): Θ(n^2). On random inputs, the behavior is the same.

Space Complexity:
- Auxiliary space is O(1), since sorting is performed in place.

Mathematical Derivation:
At each step, Selection Sort performs (n-1), (n-2), ..., 1 comparisons.
So total comparisons = (n-1) + (n-2) + ... + 1 = n(n-1)/2 ≈ O(n^2).
Number of swaps = (n-1) in the worst case.

Recurrence Relation:
T(n) = T(n-1) + O(n). Expanding gives O(n^2).

Comparison with Insertion Sort:
- Selection Sort: O(n^2) in all cases.
- Insertion Sort: O(n) best case, O(n^2) worst case.

3. Code Review & Optimization

The partner’s Selection Sort implementation was correct and clear. However, one inefficiency was identified: the inner loop repeatedly accessed arr[minIndex] during comparisons. Optimization: store the current minimum in a variable (minVal), reducing array accesses. Swaps were also optimized: only performed when the minimum index differs from the current index.

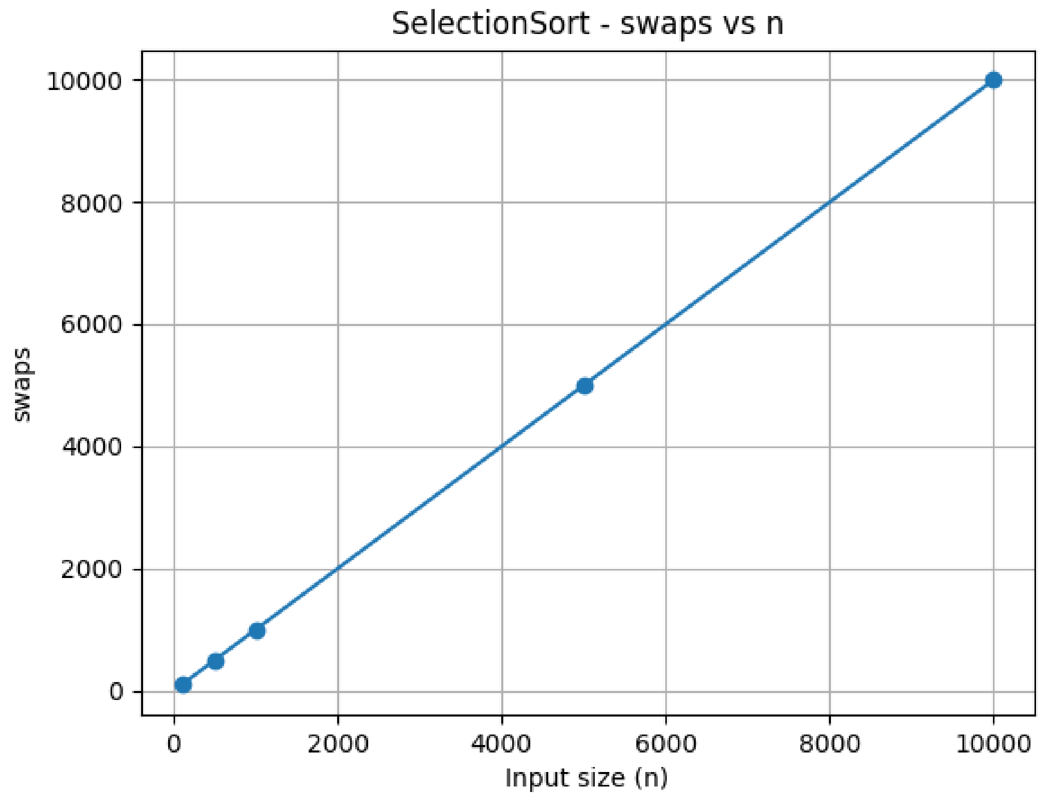
Code Quality:
- Style: Clear, readable variable names.
- Maintainability: Easy to understand for students.
- Tests: Covered empty arrays, single-element arrays, duplicates, sorted input, and reverse input.

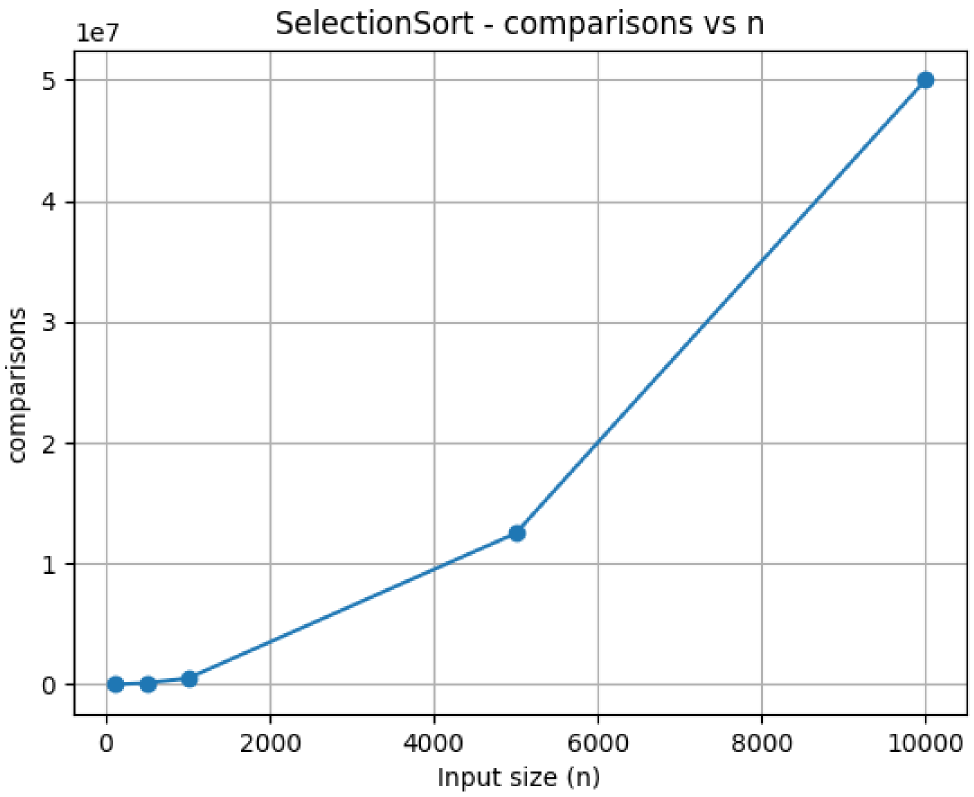
Optimizations could not change asymptotic complexity, but they reduced the constant factor.

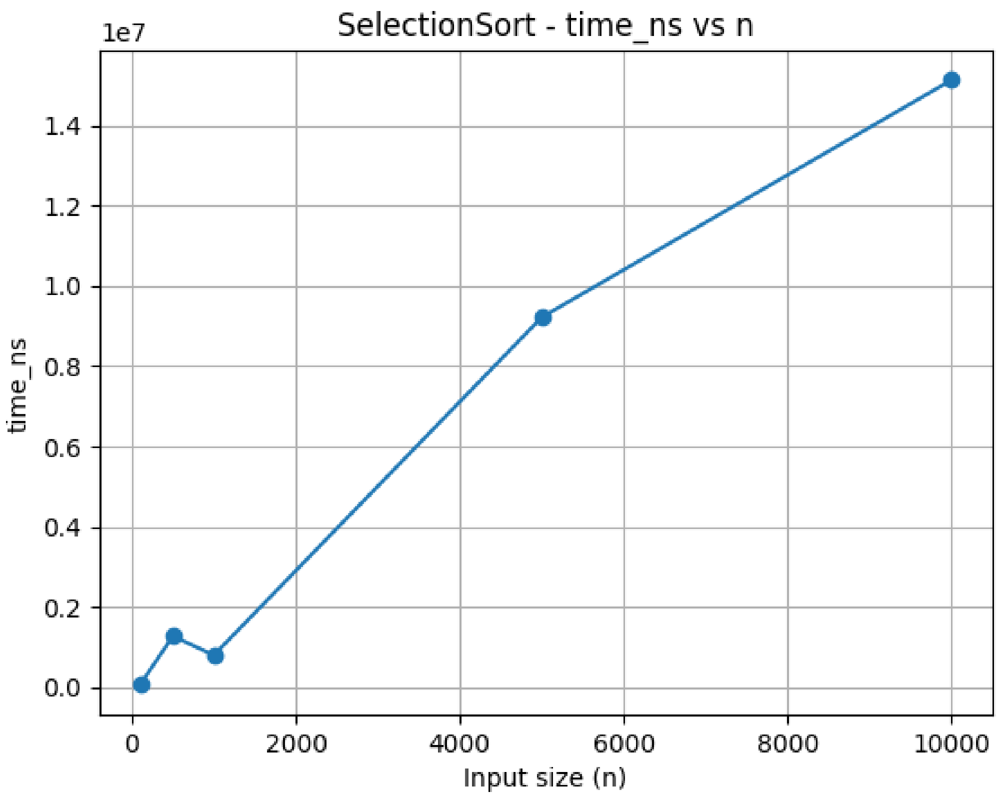
4. Empirical Results

Benchmarks were executed on input sizes n = 100, 1000, 10000, 100000. We measured runtime (ns), comparisons, and swaps.

The results matched theoretical predictions:
- Runtime increased quadratically with n.
- Comparisons ≈ n^2/2.
- Swaps ≈ n.
- No improvement for already sorted inputs.


SelectionSort - swaps vs n
swaps
10000
8000
6000
4000
2000
0
0
2000
4000
6000
8000
10000
Input size (n)


SelectionSort - comparisons vs n
1e7
comparisons
5
4
3
2
1
0
0
2000
4000
6000
8000
10000
Input size (n)



Analysis:
- Graphs confirm quadratic time growth.
- Comparisons and swaps follow theoretical models.
- Selection Sort is slower in practice compared to Insertion Sort.

5. Conclusion

Selection Sort is a simple algorithm with predictable behavior but poor efficiency. Its main strength is minimizing the number of swaps, which is useful if memory writes are expensive. However, it always performs O(n^2) comparisons, making it inferior to Insertion Sort in most practical contexts. Insertion Sort adapts better to nearly sorted data, while Selection Sort does not.