Assignment 2

Names: Zhannur, Ayazhan.

Group:SE-2412

Link to the GitHub: https://github.com/zhannur17/Sorting1.git

1. Algorithm Overview

Selection Sort is a straightforward, comparison-based sorting algorithm. It divides the array into two regions: a sorted prefix and an unsorted suffix. In each iteration, the algorithm finds the smallest element in the unsorted region and places it at the beginning by swapping. This process continues until the entire array is sorted.

The version analyzed here includes an *early termination optimization*. After each pass, the algorithm checks whether a swap occurred. If no swaps are made, the algorithm concludes that the array is already sorted and halts early. This improves performance for fully sorted inputs but does not affect the worst-case runtime.

Selection Sort is not stable in its classical form, meaning equal elements may not preserve their relative order. It is also inefficient for large datasets due to its quadratic number of comparisons. However, it makes at most $n - 1$ swaps, which can be advantageous in systems where write operations are expensive.

2. Complexity Analysis

Time Complexity

- **Best Case ($\Omega(n)$)**:
  If the input is already sorted, early termination halts execution after a single pass. This requires $n - 1$ comparisons and zero swaps. Thus, the best case complexity is **$\Omega(n)$**.
- **Average Case ($\Theta(n^2)$)**:
  Regardless of input distribution, Selection Sort scans the unsorted portion of the array in every pass. On average, it performs about $n(n - 1)/2$ comparisons, which is quadratic. Swaps occur at most $n - 1$ times, which is $\Theta(n)$. Thus, the average time complexity is **$\Theta(n^2)$**.
- **Worst Case ($O(n^2)$)**:
  In the worst case (reverse-sorted array), the algorithm still performs $n(n - 1)/2$ comparisons. The number of swaps is bounded by $n - 1$. Therefore, the worst-case complexity is **$O(n^2)$**.

Space Complexity

Selection Sort sorts the array **in-place**, requiring only $O(1)$ auxiliary space. This makes it more memory-efficient than algorithms such as Merge Sort, which require additional storage.

Comparison with Insertion Sort

- **Insertion Sort** benefits from input order: nearly-sorted arrays can be sorted in linear time ($\Omega(n)$), while Selection Sort always performs $\Theta(n^2)$ comparisons unless the array is fully sorted.
- **Selection Sort** performs fewer swaps ($O(n)$) than Insertion Sort, which may shift many elements per insertion.

- In practice, Insertion Sort tends to outperform Selection Sort except in scenarios where minimizing swaps is critical.

3. Code Review

Inefficient Sections

1. The inner loop always scans the unsorted portion fully, even if the smallest element is found early.
2. Early termination only benefits fully sorted arrays but not nearly-sorted ones.
3. The algorithm is not stable; duplicate values can be reordered arbitrarily.

Suggested Optimizations

1. **Bi-directional Selection**: Select both the minimum and maximum element in each pass, reducing the number of passes by half.
2. **Enhanced Early Termination**: Track whether the current minimum is already in place before swapping, saving unnecessary operations.
3. **Stability Enhancement**: Modify the swap operation into an insertion-style placement to preserve relative order of equal elements.

Proposed Improvements

These optimizations lower constant factors but do not reduce asymptotic complexity, which remains $O(n^2)$. For large-scale sorting tasks, more efficient algorithms (Heap Sort, Merge Sort, Quick Sort) should be used.

4. Empirical Results

Benchmarks were conducted on arrays of sizes *n = 100, 1,000, 10,000,* and *100,000*. Input distributions included random, sorted, reverse-sorted, and nearly-sorted arrays.

Observations

- **Random Input**: Execution time follows quadratic growth, confirming theoretical $O(n^2)$ behavior.
- **Sorted Input**: Early termination halts after one pass, yielding linear runtime.
- **Reverse-Sorted Input**: Performs the maximum number of comparisons, with no significant advantage over random input.
- **Nearly-Sorted Input**: Performance remains quadratic, since only fully sorted inputs benefit from early termination.

Validation

- The empirical results match the theoretical analysis.
- Comparisons dominate runtime, while swaps remain relatively few.
- Selection Sort's advantage (minimal swaps) is visible in performance counters, though comparisons make it slow overall.

-

5. Conclusion

Selection Sort is conceptually simple but inefficient for large datasets, with $\Theta(n^2)$ comparisons in average and worst cases. The early termination optimization improves performance only when the input is already sorted.

Code review identified areas for improvement, such as bi-directional selection and stability fixes, but these changes only affect constant factors, not asymptotic complexity.

Ultimately, Selection Sort is mainly of educational value. For real-world applications, faster algorithms like Heap Sort, Merge Sort, or Quick Sort should be preferred. However, Selection Sort's limited number of swaps may make it useful in specialized environments where write operations are more costly than reads.