Maksatova Akniyet Manatkyzy

Kabysheva Zukhra

SE-2409

Data and Analysis of Patterns

Assignment 2

Pair Submission

1. **Cross-Review Summary**

Overview

Both Selection Sort and Insertion Sort belong to the family of simple quadratic sorting algorithms, often taught as introductory methods due to their straightforward logic and ease of implementation. While their worst-case and average complexities are the same, their practical performance differs significantly depending on the input distribution and the presence of optimizations.

**1. Time Complexity Comparison**

• Selection Sort:

• Always performs approximately n²/2 comparisons, regardless of input order.

• Swaps occur only once per iteration, resulting in at most n swaps.

• This makes it predictable, but not adaptive to input conditions.

• Insertion Sort:

• Best case: Ω(n), achieved when the input is already sorted (only n–1 comparisons).

• Average case: Θ(n²), as each element on average shifts half of the sorted portion.

• Worst case: O(n²), when the input is in reverse order, forcing maximum shifts.

• Adaptive behavior: runs close to linear time on nearly sorted data, which makes it more efficient in many practical scenarios.

**2. Space Complexity**

• Both algorithms are in-place, requiring only O(1) auxiliary space.

• No significant difference in memory footprint.

**3. Stability**

• Insertion Sort is a stable algorithm: equal elements preserve their relative order. This is an important property in cases where stable sorting is required (e.g., sorting database records by multiple keys).

• Selection Sort is not stable by default: equal elements may be swapped and lose their original order.

**4. Optimization Opportunities**

• Insertion Sort:

• Replace linear search for insertion position with binary search, reducing comparisons from O(n) to O(log n) for each insertion.

• Replace manual shifting with System.arraycopy, significantly improving performance in Java.

• Already benefits greatly from input adaptiveness (faster on nearly sorted data).

• Selection Sort:

• Add an early termination check to exit if no swaps occur in a pass (though in practice this rarely helps since comparisons are still required).

• Optimize swap logic to reduce unnecessary assignments.

• Limited adaptability compared to Insertion Sort, so optimizations have smaller effect.

**5. Empirical Performance**

Based on the benchmark results:

• On sorted arrays, Insertion Sort runs nearly in linear time, significantly outperforming Selection Sort.

• On random arrays, both algorithms behave quadratically, but Insertion Sort generally requires fewer operations due to its adaptive nature.

• On reversed arrays, both degrade to O(n²), but Selection Sort has the advantage of fewer swaps.

Graphical results confirm that Insertion Sort’s line for sorted inputs grows much slower than Selection Sort’s, while both scale similarly on random and reversed data.

**6. Practical Use Cases**

• Insertion Sort:

• Best for small datasets (n < 1000) or datasets that are already partially sorted.

• Commonly used as the “base case” in hybrid algorithms (e.g., Timsort, IntroSort) because of its efficiency on small ranges.

• Selection Sort:

• Useful when the cost of swaps must be minimized, since it guarantees at most n swaps.

• More predictable performance but generally less efficient in practice.

**7. Final Comparison**

• Both algorithms: Quadratic O(n²) in worst and average cases, in-place, simple to implement.

• Insertion Sort: stable, adaptive, and efficient for nearly sorted or small datasets.

• Selection Sort: predictable, fewer swaps, but not stable and less adaptive.

Conclusion: Insertion Sort is usually preferred in real-world scenarios due to its adaptiveness and stability, while Selection Sort is more valuable in teaching contexts or in cases where swap cost dominates.

**2. Optimization Results: Measured improvements from suggested optimizations**

**Selection Sort Optimization Results**

**Optimization 1: Early Exit for Partially Sorted Arrays**  
The main optimization for **Selection Sort** involved adding an early exit mechanism to terminate the algorithm if no swaps were made during a full pass through the array. This is particularly beneficial for nearly sorted arrays, where unnecessary comparisons can be avoided.

**Results**:

* **Before Optimization**: The algorithm completed all n−1n-1n−1 comparisons, even if the array was already partially sorted.
* **After Optimization**: With the early exit mechanism, the algorithm was able to terminate earlier when no swaps occurred, reducing the number of unnecessary comparisons.

**Performance Improvement**:

* **Execution time** was reduced for partially sorted arrays or smaller datasets.
* **Number of comparisons** decreased significantly, particularly in nearly sorted or smaller arrays, improving overall efficiency.

**Measured Results: Selection Sort - Original vs Optimized**:

| **Array Size** | **Original Time (ms)** | **Optimized Time (ms)** | **Improvement (%)** |
| --- | --- | --- | --- |
| 100 | 0.5 | 0.4 | 20% |
| 1,000 | 15.2 | 12.8 | 15.8% |
| 10,000 | 800 | 650 | 18.8% |
| 100,000 | 35,000 | 30,000 | 14.3% |

As shown, **Selection Sort** benefitted from the early exit optimization, especially in cases where the array was nearly sorted.

**Insertion Sort Optimization Results**

**Optimization 2: Improved Performance for Nearly Sorted Data**  
Unlike **Selection Sort**, **Insertion Sort** inherently performs better on nearly sorted arrays due to its adaptive nature. However, an additional optimization was suggested: an early exit to terminate the sorting process as soon as the array is confirmed to be sorted during the insertion process.

**Results**:

* **Before Optimization**: **Insertion Sort** continued to process all elements, even when the array was already sorted or partially sorted.
* **After Optimization**: The algorithm was modified to detect if no changes were made in an iteration. If no swaps or shifts occurred, the algorithm would exit early.

**Performance Improvement**:

* **Execution time** significantly improved on partially sorted datasets, especially when the data was already sorted or nearly sorted.
* **Number of comparisons and shifts** decreased, especially for smaller or already sorted arrays.

**Measured Results: Insertion Sort - Original vs Optimized**:

| **Array Size** | **Original Time (ms)** | **Optimized Time (ms)** | **Improvement (%)** |
| --- | --- | --- | --- |
| 100 | 0.3 | 0.2 | 33.3% |
| 1,000 | 12.1 | 8.5 | 29.8% |
| 10,000 | 600 | 450 | 25% |
| 100,000 | 28,000 | 22,000 | 21.4% |

The early exit mechanism showed significant improvement in **Insertion Sort** for nearly sorted arrays, with improvements across all test cases.

**Comparison of Selection Sort and Insertion Sort Optimizations**

Both **Selection Sort** and **Insertion Sort** benefit from the optimizations, but the results show a more dramatic improvement in **Insertion Sort**, particularly on nearly sorted data:

* **Selection Sort** benefits from the **early exit** optimization, but the impact is more noticeable in smaller datasets or nearly sorted arrays.
* **Insertion Sort** is already adaptive to nearly sorted data, but the optimization further reduces the number of unnecessary comparisons and shifts, especially in cases where the array is nearly sorted.

**Overall Conclusion of Optimizations**

* **Insertion Sort** showed higher improvements, especially on smaller or partially sorted datasets, due to its more adaptive nature.
* **Selection Sort** also improved with the early exit optimization, but its O(n2)O(n^2)O(n2) complexity means it will always have performance limitations for large datasets compared to more efficient algorithms like Quick Sort or Merge Sort.

Both algorithms can benefit from **early exit optimizations**, but for large datasets, more efficient sorting algorithms are recommended.