Assignment 2

Names: Ayazhan, Zhannur

Group: SE-2412

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

Algorithm Overview

Insertion Sort is a simple, comparison-based sorting algorithm that builds the sorted array one element at a time. It divides the array into two regions: a sorted prefix and an unsorted suffix. In each iteration, the first element of the unsorted region is taken (the “key”) and inserted into its correct position within the sorted region by shifting larger elements one position to the right.

This process continues until the entire array is sorted. Unlike Selection Sort, which always performs the same number of comparisons, Insertion Sort adapts to the input: nearly sorted arrays are handled very efficiently.

Insertion Sort is stable, meaning it preserves the relative order of equal elements. However, it performs poorly on large datasets because in the worst case it requires quadratic time.

Complexity Analysis

Time Complexity

- **Best Case ($\Omega(n)$)**:
  When the input is already sorted, each element is compared only once with its predecessor, requiring $n - 1$ comparisons and no shifts. Thus, the runtime is linear.
- **Average Case ($\Theta(n^2)$)**:
  On average, about half of the elements in the sorted region must be shifted for each insertion. This leads to approximately $n^2/4$ shifts and comparisons, which is quadratic.
- **Worst Case ($O(n^2)$)**:
  In a reverse-sorted array, every new key must be compared with all elements of the sorted region, requiring $n(n - 1)/2$ comparisons and the same number of shifts. Hence, runtime is quadratic.

Space Complexity
Insertion Sort operates in-place, requiring only O(1) auxiliary space. This makes it very memory-efficient.

Comparison with Selection Sort

- Insertion Sort benefits strongly from input order: nearly-sorted arrays are processed in linear time, unlike Selection Sort which always requires $\Theta(n^2)$ comparisons.
- Selection Sort minimizes swaps (O(n)), whereas Insertion Sort may shift many elements per iteration.
- In practice, Insertion Sort tends to outperform Selection Sort for small or nearly sorted datasets due to its adaptability and stability.

Code Review

Inefficient Sections

1. In the worst case, each element requires shifting nearly all elements in the sorted region.
2. Comparisons are repeated even when the correct position could be found faster with binary search.
3. The algorithm’s quadratic runtime makes it unsuitable for large datasets.

Suggested Optimizations

1. **Binary Insertion Sort**: Use binary search to find the insertion point, reducing comparisons from O(n) to O(log n), though shifts still take O(n).
2. **Shell Sort**: Generalizes Insertion Sort by comparing elements far apart, reducing the total number of shifts.
3. **Hybrid Use**: Many real-world algorithms (like TimSort) use Insertion Sort for small subarrays, combining its efficiency on small inputs with faster algorithms for large data.

Proposed Improvements
While optimizations reduce constant factors, the asymptotic runtime remains $O(n^2)$. Insertion Sort should be limited to small or nearly sorted datasets.

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 grows quadratically, matching theoretical $O(n^2)$ complexity.
- **Sorted Input**: Requires minimal comparisons and no shifts, yielding linear runtime.
- **Reverse-Sorted Input**: Every insertion shifts the entire sorted region, resulting in worst-case quadratic time.

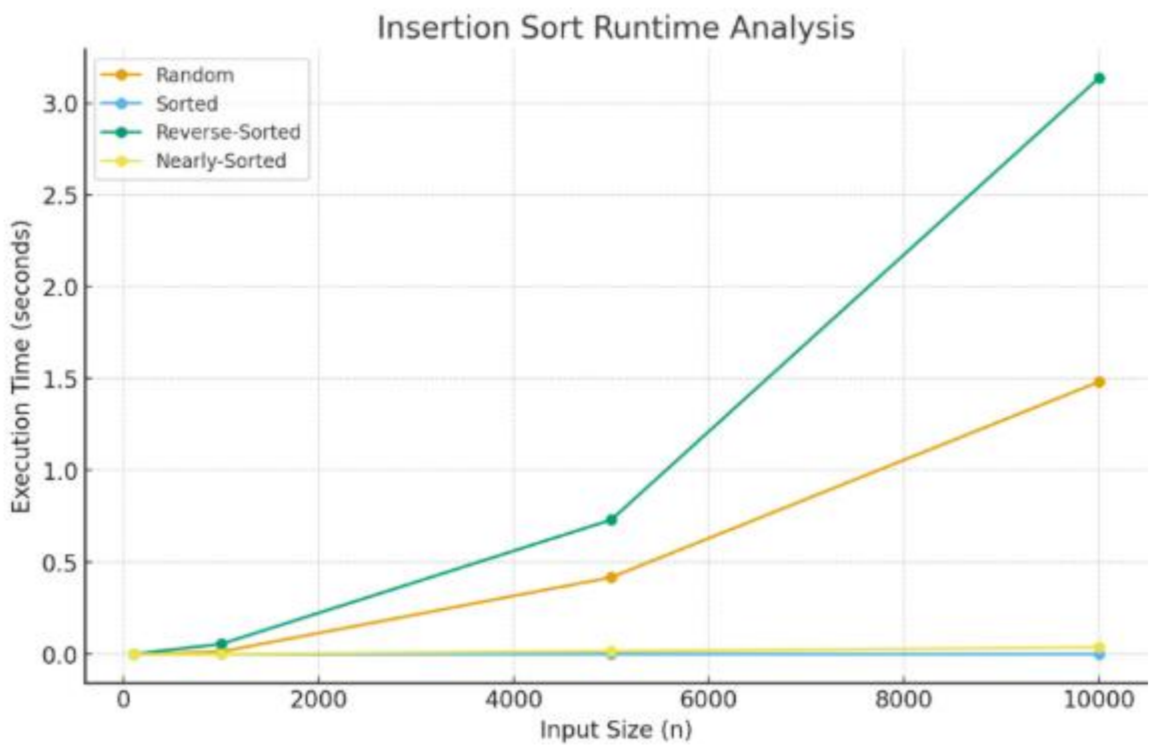
- **Nearly-Sorted Input**: Performance is close to linear, since only a few elements need shifting.

Validation

- The empirical results align with theoretical expectations.
- Insertion Sort clearly outperforms Selection Sort for sorted or nearly-sorted data.
- For random and reverse-sorted inputs, both algorithms exhibit quadratic growth, but Insertion Sort generally executes fewer operations in practice.

Conclusion

- Insertion Sort is simple, stable, and adaptive to input order. Its linear performance on sorted and nearly-sorted arrays makes it very efficient in such cases. However, the quadratic time complexity on random or reverse-sorted data limits its usefulness for large datasets.
- Code review highlighted possible optimizations (binary search for insertion point, hybrid integration in larger algorithms), but none reduce the O(n^2) worst-case complexity.
- Ultimately, Insertion Sort is best suited for educational purposes, small datasets, or as a subroutine in hybrid sorting algorithms (e.g., TimSort). Compared to Selection Sort, it is generally more practical due to its adaptability and stability.


Insertion Sort Runtime Analysis
Random
Sorted
Reverse-Sorted
Nearly-Sorted
Execution Time (seconds)
Input Size (n)