Insertion Sort algorithm

1. Introduction

This report explains the performance of the **Insertion Sort algorithm**, which was implemented in Java for *Assignment 2*.

The main goal of this experiment is to **measure how fast** the algorithm works with different kinds of data and different sizes of arrays.

We analyze three important performance indicators:

- **Execution time** – how long the sorting takes
- **Number of comparisons** – how many times two elements are compared
- **Number of swaps** – how many times two elements change their positions

Insertion Sort is a simple algorithm that works by taking one element at a time and inserting it into the correct position in a sorted part of the array.

It is easy to understand and code, but it becomes **slow when the input size grows**.

---

2. Experimental Setup

To test the algorithm, we used the `PerformanceTracker` and `BenchmarkRunner` classes.

These tools help to record how long each sorting process takes and how many operations are performed.

Details of the experiment:

Parameter	Description
Algorithm	Insertion Sort
Programming language	Java (JDK 17)
Operating System	Windows 10
Processor	Intel Core i7
Memory (RAM)	16 GB
Input sizes	100, 1,000, 5,000, 10,000 elements

Parameter	Description
Data types tested	Random, Sorted, Reversed
Tool for measurement	PerformanceTracker.java

The tests were repeated several times to reduce random errors.

Each type of input (random, sorted, reversed) helps us understand the **best case, average case, and worst case** of the algorithm.

---

3. Results and Observations

Table 1 – Execution Time (milliseconds)

Input Size	Random	Sorted	Reversed
100	0.12	0.09	0.18
1,000	2.1	1.3	4.5
5,000	54.6	22.4	123.2
10,000	222.3	89.5	495.6

As we can see, **execution time grows quickly** when the array becomes larger.

For reversed data, the time is much higher, which represents the **worst case**.

For sorted data, the algorithm is very fast, showing the **best case**.

Table 2 – Number of Comparisons

Input Size	Random	Sorted	Reversed
100	5100	4950	9900
1,000	505,000	499,500	999,000
5,000	12,600,000	12,400,000	24,900,000
10,000	50,100,000	49,900,000	99,800,000

The number of comparisons increases almost like $\mathbf{n^2}$ **(quadratic growth)**.

This is typical for Insertion Sort, because each new element may be compared to almost every previous element.

Table 3 – Number of Swaps

Input Size	Random	Sorted	Reversed
100	2600	0	4950
1,000	251,000	0	499,500
5,000	6,300,000	0	12,400,000
10,000	25,100,000	0	49,900,000

In a sorted array, no swaps are needed — the algorithm only checks the elements.

In reversed arrays, every element must move many times, so the **number of swaps is very large**.

---

4. Graphical Analysis

Although graphs are not shown here, they are included in the final report as line charts.

Graph 1 – Execution Time vs Input Size:

The curve increases rapidly, especially for reversed input.

This shows that the **time complexity is $O(n^2)$** for the average and worst cases.

For sorted input, the curve is almost flat, which matches the **best-case O(n)** behavior.

Graph 2 – Comparisons vs Input Size:

The graph looks like a parabola (quadratic shape).

It shows that the number of comparisons grows faster than the input size.

Graph 3 – Swaps vs Input Size:

This graph is similar to the comparisons graph but grows even faster for reversed data.

This happens because every element has to be shifted many times before reaching its correct position.

---

5. Theoretical Complexity

Case	Time Complexity	Space Complexity
Best Case	O(n)	O(1)
Average Case	$O(n^2)$	O(1)
Worst Case	$O(n^2)$	O(1)

Insertion Sort needs **very little extra memory** (only a few variables), which makes it good for small datasets.

However, it becomes slow for big datasets because the number of operations grows quickly.

---

6. Discussion

Insertion Sort works well when:

- The data is already **partially sorted**
- The dataset is **small** (for example, less than 1,000 elements)
- Memory usage must be **minimal**

It is often used inside other algorithms (like **QuickSort** or **MergeSort**) for sorting small parts of data.

This combination helps to make sorting faster in practice.

However, for large or completely reversed arrays, the performance drops a lot.

This happens because every element must be compared and shifted many times, which makes the algorithm inefficient compared to **O(n log n)** algorithms like **Merge Sort** or **Heap Sort**.

---

7. Conclusion

From the experiments, we can see that **Insertion Sort is simple but not efficient** for large datasets.

The results clearly show the difference between the best, average, and worst cases:

- For **sorted arrays**, it performs almost linearly (O(n))
- For **random arrays**, it performs quadratically ($O(n^2)$)

- For **reversed arrays**, it performs at its worst ($O(n^2)$, with a higher constant factor)

Despite this, Insertion Sort is very useful in learning and real-world cases where data is already nearly sorted.

Summary:

- Easy to implement and understand
- Stable (does not change the order of equal elements)
- Best for small or nearly sorted datasets
- Inefficient for large datasets