Subject: Data Structures and Algorithms

Lab 1: Sorting Algorithm Analysis

DAVE SOHAM MANISH

Enrollment: 25MCD005

M.Tech Data Science, Nirma University

Code

```
import time

import random

import matplotlib.pyplot as plt

import pandas as pd

import os


# ---------------- Sorting Algorithms ---------------- #

def selection_sort(arr):

    n = len(arr)

    for i in range(n-1):

        min_index = i

        for j in range(i+1, n):

            if arr[j] < arr[min_index]:

                min_index = j

        arr[i], arr[min_index] = arr[min_index], arr[i]


def bubble_sort(arr):

    n = len(arr)

    for i in range(n-1):

        swapped = False
        for j in range(n-i-1):
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]
                swapped = True
        if not swapped:
            break


def insertion_sort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        while j >= 0 and arr[j] > key:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key


# ---------------- Main Experiment ---------------- #
if __name__ == "__main__":
    user_input_arr = [10000, 50000, 100000]
    num_runs = 1  # Increase this for higher accuracy


    all_run_results = []
    verification_records = []


    for run in range(num_runs):
        selection_sort_best_times, selection_sort_avg_times, selection_sort_worst_times = [], [], []
        bubble_sort_best_times, bubble_sort_avg_times, bubble_sort_worst_times = [], [], []
        insertion_sort_best_times, insertion_sort_avg_times, insertion_sort_worst_times = [], [], []
```

```
for user_input in user_input_arr:
    # ---------- Selection Sort ----------
    arr = list(range(user_input))
    random.shuffle(arr)
    original_last5 = arr[-5:]
    start = time.time(); selection_sort(arr); end = time.time()
    sorted_last5 = arr[-5:]
    selection_sort_avg_times.append(end - start)
    verification_records.append(["Selection", "Average", user_input, original_last5, sorted_last5])

    arr = list(range(user_input))
    original_last5 = arr[-5:]
    start = time.time(); selection_sort(arr); end = time.time()
    sorted_last5 = arr[-5:]
    selection_sort_best_times.append(end - start)
    verification_records.append(["Selection", "Best", user_input, original_last5, sorted_last5])

    arr = list(range(user_input, 0, -1))
    original_last5 = arr[-5:]
    start = time.time(); selection_sort(arr); end = time.time()
    sorted_last5 = arr[-5:]
    selection_sort_worst_times.append(end - start)
    verification_records.append(["Selection", "Worst", user_input, original_last5, sorted_last5])

    # ---------- Bubble Sort ----------
    arr = list(range(user_input))
    random.shuffle(arr)
```

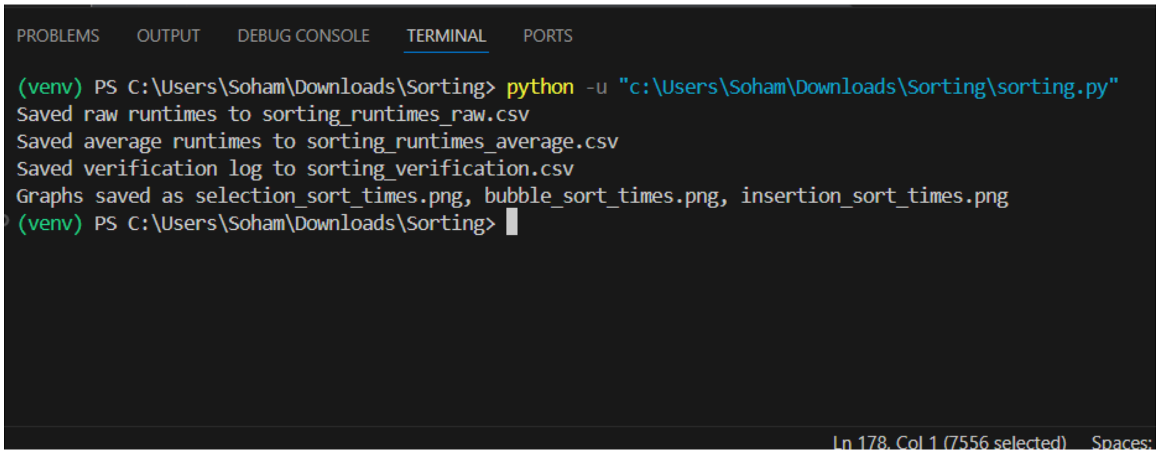
```
original_last5 = arr[-5:]

start = time.time(); bubble_sort(arr); end = time.time()

sorted_last5 = arr[-5:]

bubble_sort_avg_times.append(end - start)

verification_records.append(["Bubble", "Average", user_input, original_last5, sorted_last5])


arr = list(range(user_input))

original_last5 = arr[-5:]

start = time.time(); bubble_sort(arr); end = time.time()

sorted_last5 = arr[-5:]

bubble_sort_best_times.append(end - start)

verification_records.append(["Bubble", "Best", user_input, original_last5, sorted_last5])


arr = list(range(user_input, 0, -1))

original_last5 = arr[-5:]

start = time.time(); bubble_sort(arr); end = time.time()

sorted_last5 = arr[-5:]

bubble_sort_worst_times.append(end - start)

verification_records.append(["Bubble", "Worst", user_input, original_last5, sorted_last5])


# ---------- Insertion Sort ----------

arr = list(range(user_input))

random.shuffle(arr)

original_last5 = arr[-5:]

start = time.time(); insertion_sort(arr); end = time.time()

sorted_last5 = arr[-5:]

insertion_sort_avg_times.append(end - start)

verification_records.append(["Insertion", "Average", user_input, original_last5, sorted_last5])
```

```
    arr = list(range(user_input))
    original_last5 = arr[-5:]
    start = time.time(); insertion_sort(arr); end = time.time()
    sorted_last5 = arr[-5:]
    insertion_sort_best_times.append(end - start)
    verification_records.append(["Insertion", "Best", user_input, original_last5, sorted_last5])

    arr = list(range(user_input, -1, -1))
    original_last5 = arr[-5:]
    start = time.time(); insertion_sort(arr); end = time.time()
    sorted_last5 = arr[-5:]
    insertion_sort_worst_times.append(end - start)
    verification_records.append(["Insertion", "Worst", user_input, original_last5, sorted_last5])

run_data = {
    'Input Size': user_input_arr,
    'Selection Sort (Best)': selection_sort_best_times,
    'Selection Sort (Average)': selection_sort_avg_times,
    'Selection Sort (Worst)': selection_sort_worst_times,
    'Bubble Sort (Best)': bubble_sort_best_times,
    'Bubble Sort (Average)': bubble_sort_avg_times,
    'Bubble Sort (Worst)': bubble_sort_worst_times,
    'Insertion Sort (Best)': insertion_sort_best_times,
    'Insertion Sort (Average)': insertion_sort_avg_times,
    'Insertion Sort (Worst)': insertion_sort_worst_times
}
run_df = pd.DataFrame(run_data)
```

```
    all_run_results.append(run_df)


# Concatenate all runs

combined_df = pd.concat(all_run_results, ignore_index=True)

combined_df.to_csv("sorting_runtimes_raw.csv", index=False)


# Compute average runtimes across runs

avg_df = combined_df.groupby("Input Size").mean().reset_index()

avg_df.to_csv("sorting_runtimes_average.csv", index=False)


# Save verification file

verify_df = pd.DataFrame(verification_records,

                          columns=["Algorithm", "Case", "Input Size", "Original Last 5", "Sorted Last 5"])

verify_df.to_csv("sorting_verification.csv", index=False)


print("Saved raw runtimes to sorting_runtimes_raw.csv")

print("Saved average runtimes to sorting_runtimes_average.csv")

print("Saved verification log to sorting_verification.csv")


# ---------------- Plotting ---------------- #

def plot_algorithm(algorithm_name, avg_df):

    cases = ["Best", "Average", "Worst"]

    input_sizes = avg_df["Input Size"].tolist()


    times = []

    for case in cases:

        col = f"{algorithm_name} Sort ({case})"

        times.append(avg_df[col].tolist())
    plot_df = pd.DataFrame(times, index=cases, columns=input_sizes)

    plot_df.T.plot(kind="bar", figsize=(10,6))

    plt.title(f"{algorithm_name} Sort Runtime Comparison")

    plt.xlabel("Input Size")

    plt.ylabel("Time (seconds)")

    plt.legend(title="Case")

    plt.tight_layout()

    plt.savefig(f"{algorithm_name.lower()}_sort_times.png")

    plt.close()


plot_algorithm("Selection", avg_df)

plot_algorithm("Bubble", avg_df)

plot_algorithm("Insertion", avg_df)


print("Graphs saved as selection_sort_times.png, bubble_sort_times.png, insertion_sort_times.png")
```

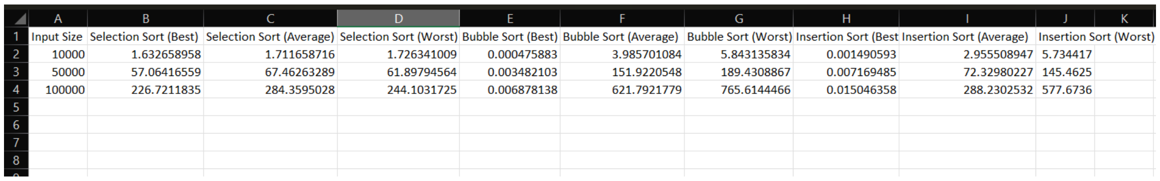
Output Screenshots

VS Code Terminal :



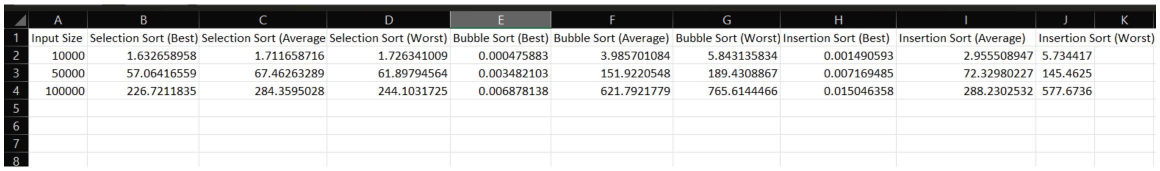
```
(venv) PS C:\Users\Soham\Downloads\Sorting> python -u "c:\Users\Soham\Downloads\Sorting\sorting.py"
Saved raw runtimes to sorting_runtimes_raw.csv
Saved average runtimes to sorting_runtimes_average.csv
Saved verification log to sorting_verification.csv
Graphs saved as selection_sort_times.png, bubble_sort_times.png, insertion_sort_times.png
(venv) PS C:\Users\Soham\Downloads\Sorting>
```

sorting_runtimes_raw.csv :



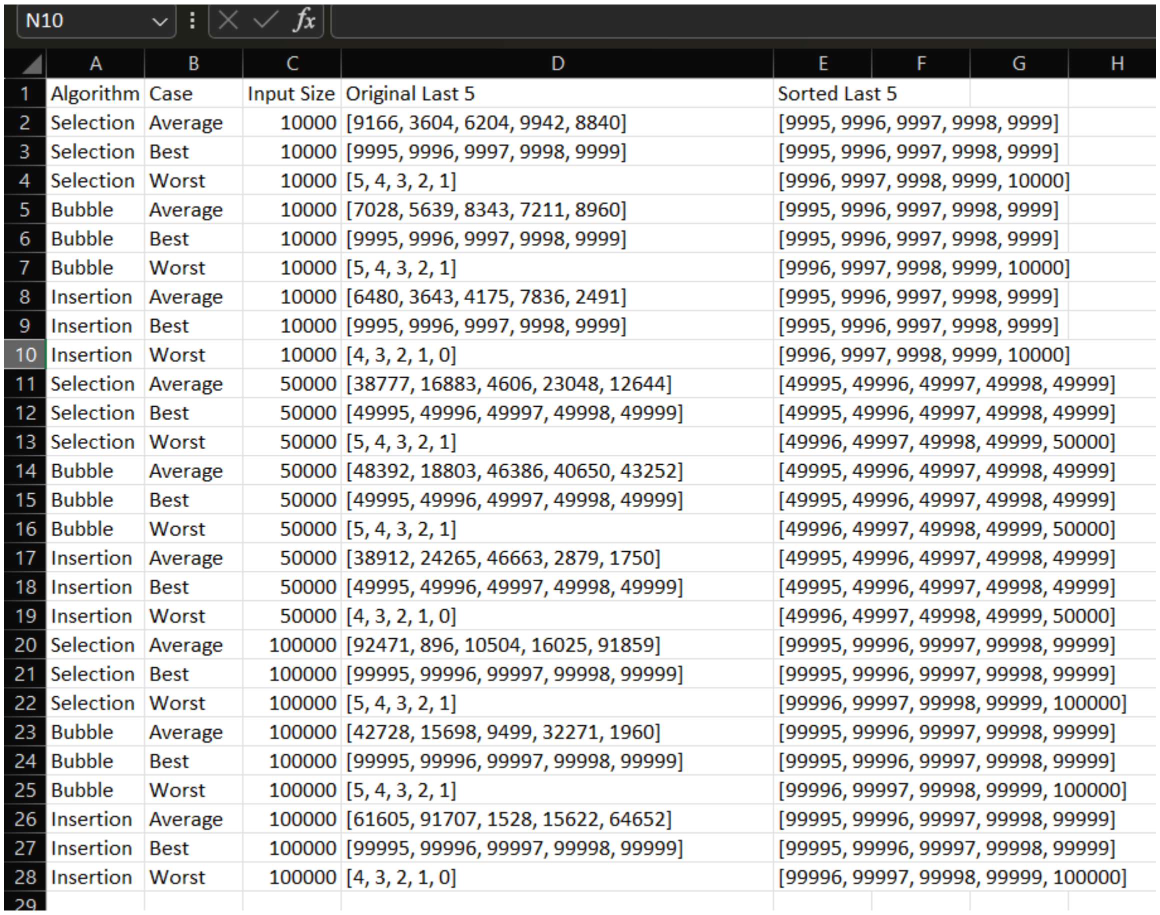
Input Size	Selection Sort (Best)	Selection Sort (Average)	Selection Sort (Worst)	Bubble Sort (Best)	Bubble Sort (Average)	Bubble Sort (Worst)	Insertion Sort (Best	Insertion Sort (Average)	Insertion Sort (Worst)
10000	1.632658958	1.711658716	1.726341009	0.000475883	3.985701084	5.843135834	0.001490593	2.955508947	5.734417
50000	57.06416559	67.46263289	61.89794564	0.003482103	151.9220548	189.4308867	0.007169485	72.32980227	145.4625
100000	226.7211835	284.3595028	244.1031725	0.006878138	621.7921779	765.6144466	0.015046358	288.2302532	577.6736

sorting_runtimes_average.csv :



Input Size	Selection Sort (Best)	Selection Sort (Average	Selection Sort (Worst)	Bubble Sort (Best)	Bubble Sort (Average)	Bubble Sort (Worst)	Insertion Sort (Best)	Insertion Sort (Average)	Insertion Sort (Worst)
10000	1.632658958	1.711658716	1.726341009	0.000475883	3.985701084	5.843135834	0.001490593	2.955508947	5.734417
50000	57.06416559	67.46263289	61.89794564	0.003482103	151.9220548	189.4308867	0.007169485	72.32980227	145.4625
100000	226.7211835	284.3595028	244.1031725	0.006878138	621.7921779	765.6144466	0.015046358	288.2302532	577.6736

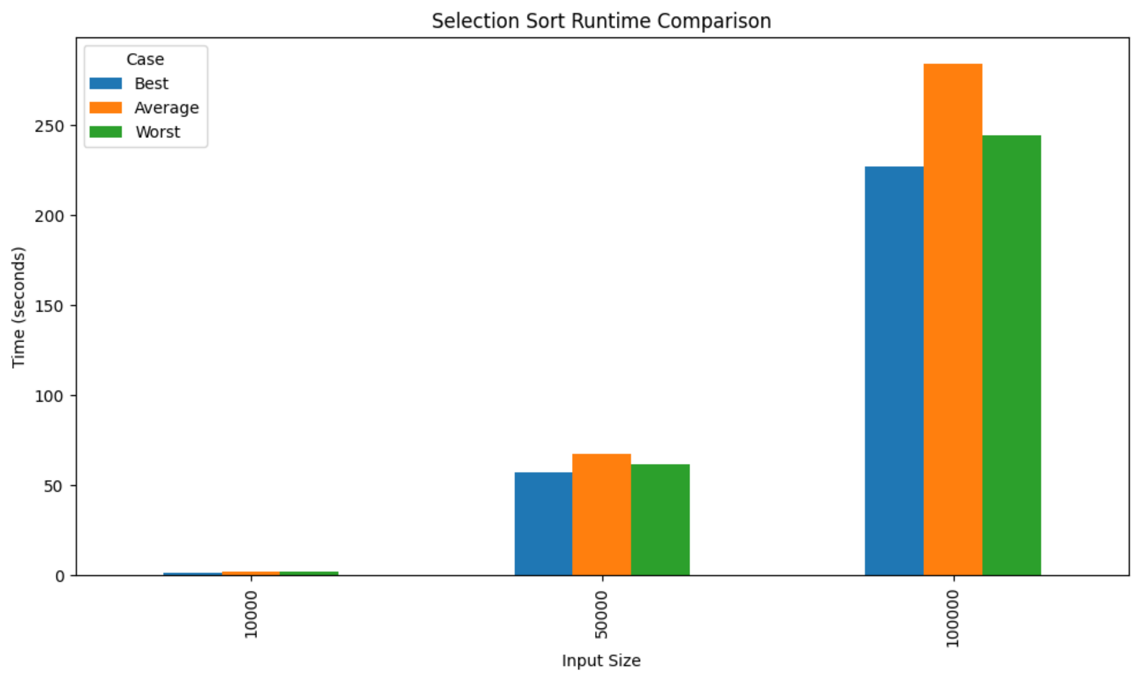
sorting_verification.csv :



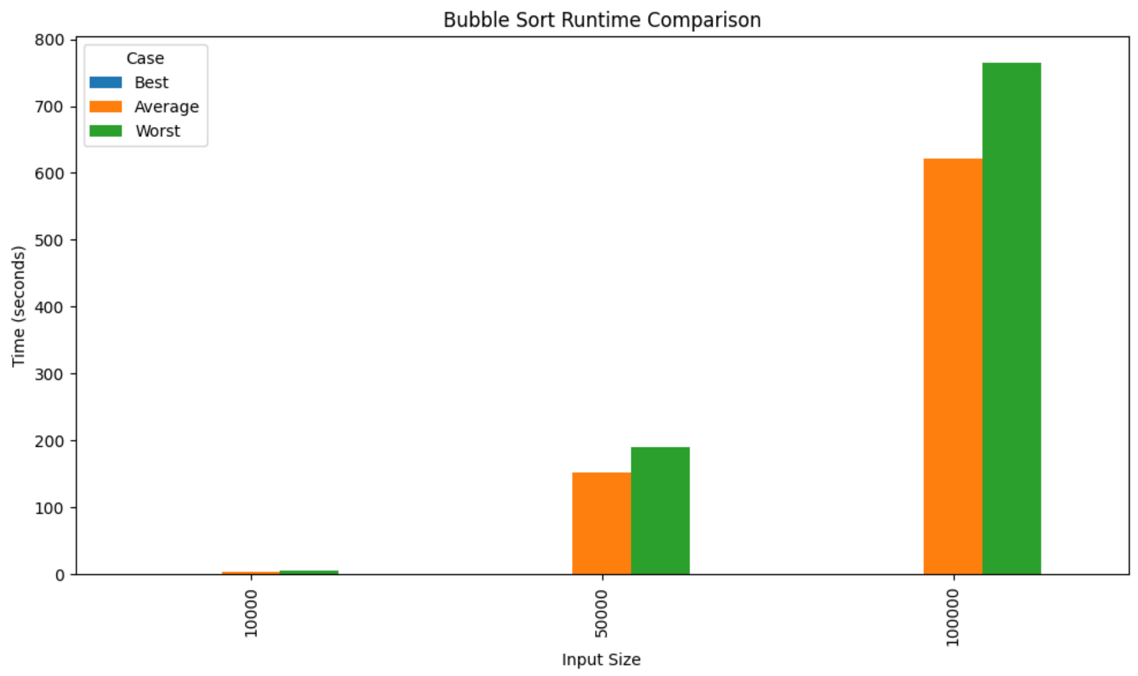
Algorithm	Case	Input Size	Original Last 5	Sorted Last 5
Selection	Average	10000	[9166, 3604, 6204, 9942, 8840]	[9995, 9996, 9997, 9998, 9999]
Selection	Best	10000	[9995, 9996, 9997, 9998, 9999]	[9995, 9996, 9997, 9998, 9999]
Selection	Worst	10000	[5, 4, 3, 2, 1]	[9996, 9997, 9998, 9999, 10000]
Bubble	Average	10000	[7028, 5639, 8343, 7211, 8960]	[9995, 9996, 9997, 9998, 9999]
Bubble	Best	10000	[9995, 9996, 9997, 9998, 9999]	[9995, 9996, 9997, 9998, 9999]
Bubble	Worst	10000	[5, 4, 3, 2, 1]	[9996, 9997, 9998, 9999, 10000]
Insertion	Average	10000	[6480, 3643, 4175, 7836, 2491]	[9995, 9996, 9997, 9998, 9999]
Insertion	Best	10000	[9995, 9996, 9997, 9998, 9999]	[9995, 9996, 9997, 9998, 9999]
Insertion	Worst	10000	[4, 3, 2, 1, 0]	[9996, 9997, 9998, 9999, 10000]
Selection	Average	50000	[38777, 16883, 4606, 23048, 12644]	[49995, 49996, 49997, 49998, 49999]
Selection	Best	50000	[49995, 49996, 49997, 49998, 49999]	[49995, 49996, 49997, 49998, 49999]
Selection	Worst	50000	[5, 4, 3, 2, 1]	[49996, 49997, 49998, 49999, 50000]
Bubble	Average	50000	[48392, 18803, 46386, 40650, 43252]	[49995, 49996, 49997, 49998, 49999]
Bubble	Best	50000	[49995, 49996, 49997, 49998, 49999]	[49995, 49996, 49997, 49998, 49999]
Bubble	Worst	50000	[5, 4, 3, 2, 1]	[49996, 49997, 49998, 49999, 50000]
Insertion	Average	50000	[38912, 24265, 46663, 2879, 1750]	[49995, 49996, 49997, 49998, 49999]
Insertion	Best	50000	[49995, 49996, 49997, 49998, 49999]	[49995, 49996, 49997, 49998, 49999]
Insertion	Worst	50000	[4, 3, 2, 1, 0]	[49996, 49997, 49998, 49999, 50000]
Selection	Average	100000	[92471, 896, 10504, 16025, 91859]	[99995, 99996, 99997, 99998, 99999]
Selection	Best	100000	[99995, 99996, 99997, 99998, 99999]	[99995, 99996, 99997, 99998, 99999]
Selection	Worst	100000	[5, 4, 3, 2, 1]	[99996, 99997, 99998, 99999, 100000]
Bubble	Average	100000	[42728, 15698, 9499, 32271, 1960]	[99995, 99996, 99997, 99998, 99999]
Bubble	Best	100000	[99995, 99996, 99997, 99998, 99999]	[99995, 99996, 99997, 99998, 99999]
Bubble	Worst	100000	[5, 4, 3, 2, 1]	[99996, 99997, 99998, 99999, 100000]
Insertion	Average	100000	[61605, 91707, 1528, 15622, 64652]	[99995, 99996, 99997, 99998, 99999]
Insertion	Best	100000	[99995, 99996, 99997, 99998, 99999]	[99995, 99996, 99997, 99998, 99999]
Insertion	Worst	100000	[4, 3, 2, 1, 0]	[99996, 99997, 99998, 99999, 100000]

Graphs

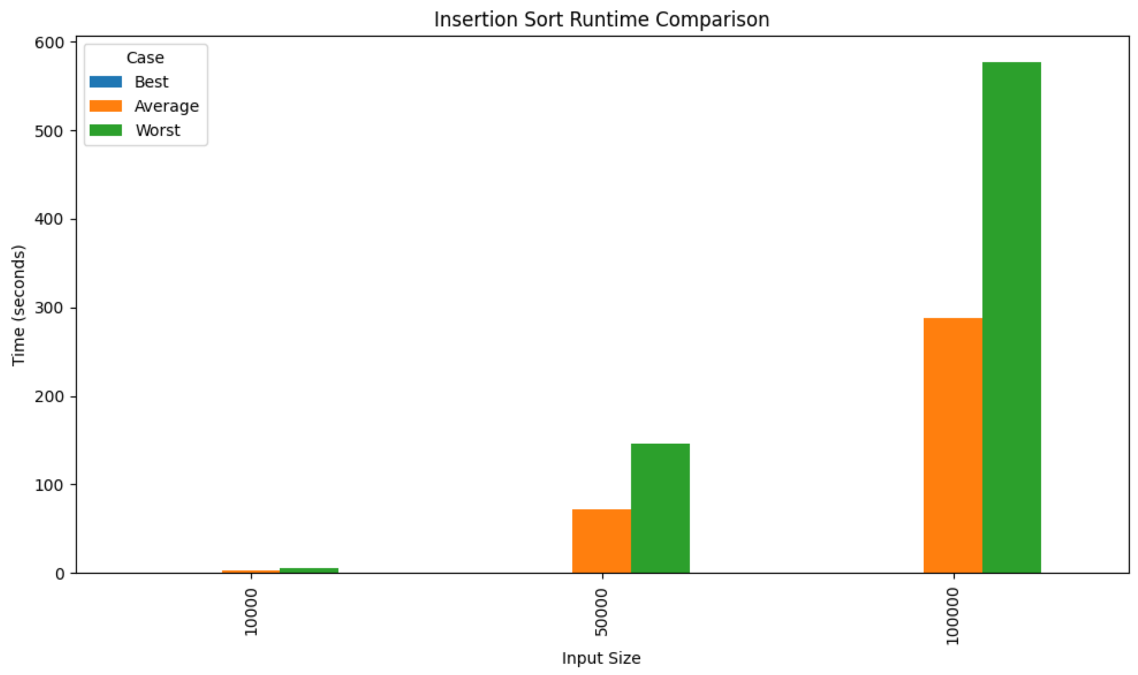
Selection Sort :



Bubble Sort :



Insertion Sort :



Analysis

In this practical, we implemented and compared three classical sorting algorithms: Selection Sort, Bubble Sort, and Insertion Sort. The objective was to understand their performance in best, average, and worst-case scenarios.

From the results and graphs:
1. Selection Sort: This algorithm performs the same number of comparisons regardless of input order. Hence, its best, average, and worst-case complexities are all $O(n^2)$. This means Selection Sort cannot take advantage of partially sorted data, making its average runtime equal to its worst-case runtime.

2. Bubble Sort: With the optimized implementation using a swap flag, Bubble Sort can terminate early if the list is already sorted. In this case, its best-case time complexity reduces to $O(n)$. However, for average and worst cases (random or reverse-sorted inputs), it still performs $O(n^2)$ operations.

3. Insertion Sort: Similar to Bubble Sort, Insertion Sort has $O(n)$ best-case complexity when the list is already sorted, since each element only needs one comparison. But in average and worst cases, it shifts multiple elements per insertion, leading to $O(n^2)$ runtime.

Overall, we see that Bubble Sort and Insertion Sort can adapt to partially sorted data and hence show better best-case and sometimes average-case performance compared to Selection Sort. Selection Sort, on the other hand, is unaffected by input order, making its performance stable but not efficient.

This experiment was designed to test and compare three basic sorting algorithms: Selection Sort, Bubble Sort, and Insertion Sort.
The code measured how long each algorithm took to sort three different types of lists: already sorted (best case), shuffled (average case), and reverse-sorted (worst case).
By running these tests on increasing input sizes (10,000, 50,000, and 100,000), we got a clear picture of how each algorithm scales.

Selection Sort always makes the same number of comparisons because it has to find the minimum element in every pass, no matter how the list looks at the start.
That's why its performance doesn't change between best, average, and worst cases. It always stays $O(n^2)$.

Bubble Sort was coded with a small optimization: if a pass makes no swaps, it stops early. This makes a big difference in the best case, where the list is already sorted—it runs in $O(n)$ time.
Insertion Sort also benefits in the same way, since each new element is already in the correct position in a sorted list, so it only needs one comparison per element in the best case.

Complexity Comparison Table

Algorithm	**Best Case**	**Average Case**	**Worst Case**
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$

Analysis of the Graphs
The graphs make the results very clear. For Selection Sort, the bars for all three cases—best, average, and worst—are almost the same height.
This proves that Selection Sort takes the same amount of time no matter what the starting order of the list is. The small differences are just system timing noise.
The key takeaway is that the average case cannot be faster than the worst case for Selection Sort, since the work it does is fixed.

For Bubble Sort and Insertion Sort, the story is different. The best-case bars are much

shorter compared to the average and worst cases,
showing their O(n) performance when the list is already sorted. As input size grows, we can also see that Insertion Sort is generally faster than Bubble Sort,
because it needs fewer swaps in average and worst cases.

Conclusion

From this practical, we can see that:
- Selection Sort always takes O(n^2) time and doesn't benefit from the list being sorted. Its average time is never less than its worst time.
- Bubble Sort and Insertion Sort can take advantage of sorted input, giving them O(n) performance in the best case.
- Insertion Sort usually performs better than Bubble Sort when the data is not sorted.

Even though all three algorithms are too slow for very large datasets, this experiment helps us understand how input order affects sorting performance.