DSA Lab 1 – Sorting Algorithm Analysis

Soham Manish Dave
Enrollment: 25MCD005

ABSTRACT

In this lab practical, we implement and compare four classical sorting algorithms: *Selection Sort, Bubble Sort, Insertion Sort, and Quick Sort (with first element as pivot)*. The primary aim is to analyze their performance across best, average, and worst-case scenarios, and to visualize their scaling behavior with increasing input sizes. This report presents the implementation code, experimental results, graphical analysis, and theoretical discussion on the complexities of each algorithm.

1. CODE

Listing 1. Implementation of Sorting Algorithms

```
import time
import random
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
import sys

sys.setrecursionlimit(200000)

# ---------------- 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

def swap(arr, i, j):
    arr[i], arr[j] = arr[j], arr[i]
```

```
def partition(arr, low, high):
    pivot = arr[low]
    i = low + 1
    j = high

    while True:
        while i <= j and arr[i] <= pivot:
            i += 1
        while i <= j and arr[j] >= pivot:
            j -= 1
        if i <= j:
            swap(arr, i, j)
        else:
            break

    swap(arr, low, j)
    return j

def quick_sort(arr, low, high):
    if low < high:
        pi = partition(arr, low, high)
        quick_sort(arr, low, pi - 1)
        quick_sort(arr, pi + 1, high)


# ---------------- Main Experiment ---------------- #
if __name__ == "__main__":
    user_input_arr = [10000, 50000, 100000]
    LAST_K = 15

    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 = [], [], []
    quick_sort_best_times,     quick_sort_avg_times,     quick_sort_worst_times     = [], [], []

    verification_records = []

    for user_input in user_input_arr:

        # ---------- Selection Sort ----------
        arr = list(range(user_input)); random.shuffle(arr)
        original_last = arr[-LAST_K:]
        start = time.time(); selection_sort(arr); end = time.time()
        sorted_last = arr[-LAST_K:]; t = end - start
        selection_sort_avg_times.append(t)
        verification_records.append(["Selection", "Average", user_input, t, original_last,
            sorted_last])
        print(f"[Selection - Average | n={user_input}] Time: {t}s")
        print(f"  Unsorted last {LAST_K}: {original_last}")
        print(f"  Sorted   last {LAST_K}: {sorted_last}\n")

        arr = list(range(user_input))
        original_last = arr[-LAST_K:]
        start = time.time(); selection_sort(arr); end = time.time()
        sorted_last = arr[-LAST_K:]; t = end - start
        selection_sort_best_times.append(t)
        verification_records.append(["Selection", "Best", user_input, t, original_last,
            sorted_last])
        print(f"[Selection - Best    | n={user_input}] Time: {t}s")
        print(f"  Unsorted last {LAST_K}: {original_last}")
```

```
        print(f"  Sorted   last {LAST_K}: {sorted_last}\n")

        arr = list(range(user_input, 0, -1))
        original_last = arr[-LAST_K:]
        start = time.time(); selection_sort(arr); end = time.time()
        sorted_last = arr[-LAST_K:]; t = end - start
        selection_sort_worst_times.append(t)
        verification_records.append(["Selection", "Worst", user_input, t, original_last,
            sorted_last])
        print(f"[Selection - Worst   | n={user_input}] Time: {t}s")
        print(f"  Unsorted last {LAST_K}: {original_last}")
        print(f"  Sorted   last {LAST_K}: {sorted_last}\n")

        # ---------- Bubble Sort ----------
        arr = list(range(user_input)); random.shuffle(arr)
        original_last = arr[-LAST_K:]
        start = time.time(); bubble_sort(arr); end = time.time()
        sorted_last = arr[-LAST_K:]; t = end - start
        bubble_sort_avg_times.append(t)
        verification_records.append(["Bubble", "Average", user_input, t, original_last,
            sorted_last])
        print(f"[Bubble    - Average | n={user_input}] Time: {t}s")
        print(f"  Unsorted last {LAST_K}: {original_last}")
        print(f"  Sorted   last {LAST_K}: {sorted_last}\n")

        arr = list(range(user_input))
        original_last = arr[-LAST_K:]
        start = time.time(); bubble_sort(arr); end = time.time()
        sorted_last = arr[-LAST_K:]; t = end - start
        bubble_sort_best_times.append(t)
        verification_records.append(["Bubble", "Best", user_input, t, original_last, sorted_last
            ])
        print(f"[Bubble    - Best    | n={user_input}] Time: {t}s")
        print(f"  Unsorted last {LAST_K}: {original_last}")
        print(f"  Sorted   last {LAST_K}: {sorted_last}\n")

        arr = list(range(user_input, 0, -1))
        original_last = arr[-LAST_K:]
        start = time.time(); bubble_sort(arr); end = time.time()
        sorted_last = arr[-LAST_K:]; t = end - start
        bubble_sort_worst_times.append(t)
        verification_records.append(["Bubble", "Worst", user_input, t, original_last, sorted_last
            ])
        print(f"[Bubble    - Worst   | n={user_input}] Time: {t}s")
        print(f"  Unsorted last {LAST_K}: {original_last}")
        print(f"  Sorted   last {LAST_K}: {sorted_last}\n")

        # ---------- Insertion Sort ----------
        arr = list(range(user_input)); random.shuffle(arr)
        original_last = arr[-LAST_K:]
        start = time.time(); insertion_sort(arr); end = time.time()
        sorted_last = arr[-LAST_K:]; t = end - start
        insertion_sort_avg_times.append(t)
        verification_records.append(["Insertion", "Average", user_input, t, original_last,
            sorted_last])
        print(f"[Insertion - Average | n={user_input}] Time: {t}s")
        print(f"  Unsorted last {LAST_K}: {original_last}")
        print(f"  Sorted   last {LAST_K}: {sorted_last}\n")

        arr = list(range(user_input))
        original_last = arr[-LAST_K:]
```

```
        start = time.time(); insertion_sort(arr); end = time.time()
        sorted_last = arr[-LAST_K:]; t = end - start
        insertion_sort_best_times.append(t)
        verification_records.append(["Insertion", "Best", user_input, t, original_last,
            sorted_last])
        print(f"[Insertion - Best    | n={user_input}] Time: {t}s")
        print(f"  Unsorted last {LAST_K}: {original_last}")
        print(f"  Sorted   last {LAST_K}: {sorted_last}\n")

        arr = list(range(user_input, 0, -1))
        original_last = arr[-LAST_K:]
        start = time.time(); insertion_sort(arr); end = time.time()
        sorted_last = arr[-LAST_K:]; t = end - start
        insertion_sort_worst_times.append(t)
        verification_records.append(["Insertion", "Worst", user_input, t, original_last,
            sorted_last])
        print(f"[Insertion - Worst   | n={user_input}] Time: {t}s")
        print(f"  Unsorted last {LAST_K}: {original_last}")
        print(f"  Sorted   last {LAST_K}: {sorted_last}\n")

        # ---------- Quick Sort ----------
        arr = list(range(user_input)); random.shuffle(arr)
        original_last = arr[-LAST_K:]
        start = time.time(); quick_sort(arr, 0, len(arr)-1); end = time.time()
        sorted_last = arr[-LAST_K:]; t = end - start
        quick_sort_avg_times.append(t)
        verification_records.append(["Quick", "Average", user_input, t, original_last,
            sorted_last])
        print(f"[Quick     - Average | n={user_input}] Time: {t}s")
        print(f"  Unsorted last {LAST_K}: {original_last}")
        print(f"  Sorted   last {LAST_K}: {sorted_last}\n")

        arr = list(range(user_input))
        original_last = arr[-LAST_K:]
        start = time.time(); quick_sort(arr, 0, len(arr)-1); end = time.time()
        sorted_last = arr[-LAST_K:]; t = end - start
        quick_sort_best_times.append(t)
        verification_records.append(["Quick", "Best", user_input, t, original_last, sorted_last])
        print(f"[Quick     - Best    | n={user_input}] Time: {t}s")
        print(f"  Unsorted last {LAST_K}: {original_last}")
        print(f"  Sorted   last {LAST_K}: {sorted_last}\n")

        arr = list(range(user_input, 0, -1))
        original_last = arr[-LAST_K:]
        start = time.time(); quick_sort(arr, 0, len(arr)-1); end = time.time()
        sorted_last = arr[-LAST_K:]; t = end - start
        quick_sort_worst_times.append(t)
        verification_records.append(["Quick", "Worst", user_input, t, original_last, sorted_last
            ])
        print(f"[Quick     - Worst   | n={user_input}] Time: {t}s")
        print(f"  Unsorted last {LAST_K}: {original_last}")
        print(f"  Sorted   last {LAST_K}: {sorted_last}\n")


    # ---------------- Save Verification CSV ---------------- #
    verify_df = pd.DataFrame(
        verification_records,
        columns=["Algorithm", "Case", "Input Size", "Time Taken (s)", "Original Last 15", "Sorted
             Last 15"]
    )
    verify_df.to_csv("sorting_verification.csv", index=False)
```

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

# ---------------- Save Runtimes CSV---------------- #
runtimes_df = pd.DataFrame({
    "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,
    "Quick Sort (Best)":       quick_sort_best_times,
    "Quick Sort (Average)":    quick_sort_avg_times,
    "Quick Sort (Worst)":      quick_sort_worst_times
})
runtimes_df.to_csv("sorting_runtimes.csv", index=False)
print("Saved runtimes to sorting_runtimes.csv")

# ---------------- Plotting (3 graphs: 10k, 50k, 100k) ---------------- #
def plot_for_input_index(idx, n_value):
    algorithms = ["Selection", "Bubble", "Insertion", "Quick"]
    x = np.arange(len(algorithms))
    width = 0.25

    best   = [selection_sort_best_times[idx],   bubble_sort_best_times[idx],
        insertion_sort_best_times[idx],   quick_sort_best_times[idx]]
    avg    = [selection_sort_avg_times[idx],    bubble_sort_avg_times[idx],
        insertion_sort_avg_times[idx],    quick_sort_avg_times[idx]]
    worst  = [selection_sort_worst_times[idx],  bubble_sort_worst_times[idx],
        insertion_sort_worst_times[idx],  quick_sort_worst_times[idx]]

    plt.figure(figsize=(10, 6))
    plt.bar(x - width, best,  width, label="Best",   color="green")
    plt.bar(x,         avg,   width, label="Average",color="blue")
    plt.bar(x + width, worst, width, label="Worst",  color="red")

    plt.xticks(x, algorithms)
    plt.xlabel("Algorithms")
    plt.ylabel("Time (seconds)")
    plt.title(f"Sorting Runtime Comparison ({n_value} elements)")
    plt.legend(title="Case")
    plt.tight_layout()
    plt.savefig(f"runtime_comparison_{n_value}.png")
    plt.close()

for idx, n in enumerate(user_input_arr):
    plot_for_input_index(idx, n)

print("Graphs saved as runtime_comparison_10000.png, runtime_comparison_50000.png,
    runtime_comparison_100000.png")
```

2. OUTPUT

The terminal outputs for input sizes of 10,000, 50,000, and 100,000 elements display the performance of Bubble Sort, Selection Sort, Insertion Sort, and Quick Sort (first element as pivot) across best, average, and worst case scenarios. For each run, the program prints the last 15 elements before sorting and the last

15 elements after sorting, serving as a verification step to confirm correctness of the implementation. The execution time for each run is also shown on the terminal.

In addition to terminal outputs, the verification logs are saved in `sorting_verification.csv`, and the runtimes for each algorithm and case are stored in `sorting_runtimes.csv`. The runtime comparison graphs are saved as `runtime_comparison_10000.png`, `runtime_comparison_50000.png`, and `runtime_comparison_100000.png`.

```
(.venv) PS Y:\DSA-Sorting> python sorting.py
[Selection - Average | n=10000] Time: 1.647994041442871s
  Unsorted last 15: [7443, 4558, 2074, 1508, 6519, 6683, 2644, 3429, 1457, 4281, 9653, 6356, 4161, 5924, 6503]
  Sorted   last 15: [9985, 9986, 9987, 9988, 9989, 9990, 9991, 9992, 9993, 9994, 9995, 9996, 9997, 9998, 9999]

[Selection - Best    | n=10000] Time: 1.6633031368255615s
  Unsorted last 15: [9985, 9986, 9987, 9988, 9989, 9990, 9991, 9992, 9993, 9994, 9995, 9996, 9997, 9998, 9999]
  Sorted   last 15: [9985, 9986, 9987, 9988, 9989, 9990, 9991, 9992, 9993, 9994, 9995, 9996, 9997, 9998, 9999]

[Selection - Worst   | n=10000] Time: 1.7455956935882568s
  Unsorted last 15: [15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
  Sorted   last 15: [9986, 9987, 9988, 9989, 9990, 9991, 9992, 9993, 9994, 9995, 9996, 9997, 9998, 9999, 10000]

[Bubble    - Average | n=10000] Time: 4.050352334976196s
  Unsorted last 15: [3740, 3213, 8962, 8062, 7234, 5606, 1360, 5634, 7886, 8439, 7772, 8143, 8196, 4905, 2592]
  Sorted   last 15: [9985, 9986, 9987, 9988, 9989, 9990, 9991, 9992, 9993, 9994, 9995, 9996, 9997, 9998, 9999]

[Bubble    - Best    | n=10000] Time: 0.0005238056182861328s
  Unsorted last 15: [9985, 9986, 9987, 9988, 9989, 9990, 9991, 9992, 9993, 9994, 9995, 9996, 9997, 9998, 9999]
  Sorted   last 15: [9985, 9986, 9987, 9988, 9989, 9990, 9991, 9992, 9993, 9994, 9995, 9996, 9997, 9998, 9999]

[Bubble    - Worst   | n=10000] Time: 5.587056636810303s
  Unsorted last 15: [15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
  Sorted   last 15: [9986, 9987, 9988, 9989, 9990, 9991, 9992, 9993, 9994, 9995, 9996, 9997, 9998, 9999, 10000]

[Insertion - Average | n=10000] Time: 2.034482717514038s
  Unsorted last 15: [1896, 6951, 6140, 2955, 5052, 987, 3307, 3774, 1649, 527, 4108, 5431, 5532, 8976, 4663]
  Sorted   last 15: [9985, 9986, 9987, 9988, 9989, 9990, 9991, 9992, 9993, 9994, 9995, 9996, 9997, 9998, 9999]

[Insertion - Best    | n=10000] Time: 0.0009920597076416016s
  Unsorted last 15: [9985, 9986, 9987, 9988, 9989, 9990, 9991, 9992, 9993, 9994, 9995, 9996, 9997, 9998, 9999]
  Sorted   last 15: [9985, 9986, 9987, 9988, 9989, 9990, 9991, 9992, 9993, 9994, 9995, 9996, 9997, 9998, 9999]

[Insertion - Worst   | n=10000] Time: 3.9709279537200928s
  Unsorted last 15: [15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
  Sorted   last 15: [9986, 9987, 9988, 9989, 9990, 9991, 9992, 9993, 9994, 9995, 9996, 9997, 9998, 9999, 10000]

[Quick     - Average | n=10000] Time: 0.015493392944335938s
  Unsorted last 15: [783, 1160, 7458, 9529, 5127, 3981, 7147, 3548, 6809, 9396, 2586, 5807, 2478, 219, 4797]
  Sorted   last 15: [9985, 9986, 9987, 9988, 9989, 9990, 9991, 9992, 9993, 9994, 9995, 9996, 9997, 9998, 9999]

[Quick     - Best    | n=10000] Time: 2.5140461921691895s
  Unsorted last 15: [9985, 9986, 9987, 9988, 9989, 9990, 9991, 9992, 9993, 9994, 9995, 9996, 9997, 9998, 9999]
  Sorted   last 15: [9985, 9986, 9987, 9988, 9989, 9990, 9991, 9992, 9993, 9994, 9995, 9996, 9997, 9998, 9999]

[Quick     - Worst   | n=10000] Time: 2.4446864128112793s
  Unsorted last 15: [15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
  Sorted   last 15: [9986, 9987, 9988, 9989, 9990, 9991, 9992, 9993, 9994, 9995, 9996, 9997, 9998, 9999, 10000]
```

Figure 1. Terminal output for 10,000 input size

```
[Selection - Average | n=50000] Time: 46.26493811607361s
  Unsorted last 15: [41421, 20661, 34597, 49297, 15206, 16816, 29459, 12417, 16432, 32345, 32763, 18789, 48000, 18319, 17589]
  Sorted   last 15: [49985, 49986, 49987, 49988, 49989, 49990, 49991, 49992, 49993, 49994, 49995, 49996, 49997, 49998, 49999]

[Selection - Best    | n=50000] Time: 40.37661075592041s
  Unsorted last 15: [49985, 49986, 49987, 49988, 49989, 49990, 49991, 49992, 49993, 49994, 49995, 49996, 49997, 49998, 49999]
  Sorted   last 15: [49985, 49986, 49987, 49988, 49989, 49990, 49991, 49992, 49993, 49994, 49995, 49996, 49997, 49998, 49999]

[Selection - Worst   | n=50000] Time: 43.60562562942505s
  Unsorted last 15: [15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
  Sorted   last 15: [49986, 49987, 49988, 49989, 49990, 49991, 49992, 49993, 49994, 49995, 49996, 49997, 49998, 49999, 50000]

[Bubble    - Average | n=50000] Time: 107.06133437156677s
  Unsorted last 15: [39153, 25340, 15326, 16400, 9514, 6984, 45942, 5900, 1913, 4541, 19634, 12382, 40327, 39279, 32108]
  Sorted   last 15: [49985, 49986, 49987, 49988, 49989, 49990, 49991, 49992, 49993, 49994, 49995, 49996, 49997, 49998, 49999]

[Bubble    - Best    | n=50000] Time: 0.0024564266204833984s
  Unsorted last 15: [49985, 49986, 49987, 49988, 49989, 49990, 49991, 49992, 49993, 49994, 49995, 49996, 49997, 49998, 49999]
  Sorted   last 15: [49985, 49986, 49987, 49988, 49989, 49990, 49991, 49992, 49993, 49994, 49995, 49996, 49997, 49998, 49999]

[Bubble    - Worst   | n=50000] Time: 134.51312518119812s
  Unsorted last 15: [15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
  Sorted   last 15: [49986, 49987, 49988, 49989, 49990, 49991, 49992, 49993, 49994, 49995, 49996, 49997, 49998, 49999, 50000]

[Insertion - Average | n=50000] Time: 50.6842041015625s
  Unsorted last 15: [40682, 25121, 8386, 41467, 15784, 1748, 47932, 37211, 36394, 26487, 242, 12469, 26378, 29436, 46328]
  Sorted   last 15: [49985, 49986, 49987, 49988, 49989, 49990, 49991, 49992, 49993, 49994, 49995, 49996, 49997, 49998, 49999]

[Insertion - Best    | n=50000] Time: 0.005218982696533203s
  Unsorted last 15: [49985, 49986, 49987, 49988, 49989, 49990, 49991, 49992, 49993, 49994, 49995, 49996, 49997, 49998, 49999]
  Sorted   last 15: [49985, 49986, 49987, 49988, 49989, 49990, 49991, 49992, 49993, 49994, 49995, 49996, 49997, 49998, 49999]

[Insertion - Worst   | n=50000] Time: 100.95338988304138s
  Unsorted last 15: [15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
  Sorted   last 15: [49986, 49987, 49988, 49989, 49990, 49991, 49992, 49993, 49994, 49995, 49996, 49997, 49998, 49999, 50000]

[Quick     - Average | n=50000] Time: 0.11981058120727539s
  Unsorted last 15: [39582, 30291, 49933, 12445, 13064, 46463, 36503, 2471, 14780, 17792, 2715, 3178, 3998, 41802, 15902]
  Sorted   last 15: [49985, 49986, 49987, 49988, 49989, 49990, 49991, 49992, 49993, 49994, 49995, 49996, 49997, 49998, 49999]

[Quick     - Best    | n=50000] Time: 56.5969033241272s
  Unsorted last 15: [49985, 49986, 49987, 49988, 49989, 49990, 49991, 49992, 49993, 49994, 49995, 49996, 49997, 49998, 49999]
  Sorted   last 15: [49985, 49986, 49987, 49988, 49989, 49990, 49991, 49992, 49993, 49994, 49995, 49996, 49997, 49998, 49999]

[Quick     - Worst   | n=50000] Time: 55.783743381500244s
  Unsorted last 15: [15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
  Sorted   last 15: [49986, 49987, 49988, 49989, 49990, 49991, 49992, 49993, 49994, 49995, 49996, 49997, 49998, 49999, 50000]
```

Figure 2. Terminal output for 50,000 input size

```
[Selection - Average | n=100000] Time: 253.93560338020325s
  Unsorted last 15: [73002, 69261, 77151, 37371, 20101, 83209, 44770, 40328, 22900, 65934, 93501, 72655, 63049, 16213, 60580]
  Sorted   last 15: [99985, 99986, 99987, 99988, 99989, 99990, 99991, 99992, 99993, 99994, 99995, 99996, 99997, 99998, 99999]

[Selection - Best    | n=100000] Time: 160.47385931015015s
  Unsorted last 15: [99985, 99986, 99987, 99988, 99989, 99990, 99991, 99992, 99993, 99994, 99995, 99996, 99997, 99998, 99999]
  Sorted   last 15: [99985, 99986, 99987, 99988, 99989, 99990, 99991, 99992, 99993, 99994, 99995, 99996, 99997, 99998, 99999]

[Selection - Worst   | n=100000] Time: 174.53959107398987s
  Unsorted last 15: [15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
  Sorted   last 15: [99986, 99987, 99988, 99989, 99990, 99991, 99992, 99993, 99994, 99995, 99996, 99997, 99998, 99999, 100000]

[Bubble    - Average | n=100000] Time: 469.8560862541199s
  Unsorted last 15: [78036, 11887, 86596, 30780, 24415, 96658, 43225, 97521, 49862, 34282, 72541, 92175, 51813, 49442, 46672]
  Sorted   last 15: [99985, 99986, 99987, 99988, 99989, 99990, 99991, 99992, 99993, 99994, 99995, 99996, 99997, 99998, 99999]

[Bubble    - Best    | n=100000] Time: 0.0047724246978759766s
  Unsorted last 15: [99985, 99986, 99987, 99988, 99989, 99990, 99991, 99992, 99993, 99994, 99995, 99996, 99997, 99998, 99999]
  Sorted   last 15: [99985, 99986, 99987, 99988, 99989, 99990, 99991, 99992, 99993, 99994, 99995, 99996, 99997, 99998, 99999]

[Bubble    - Worst   | n=100000] Time: 537.825761795044s
  Unsorted last 15: [15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
  Sorted   last 15: [99986, 99987, 99988, 99989, 99990, 99991, 99992, 99993, 99994, 99995, 99996, 99997, 99998, 99999, 100000]

[Insertion - Average | n=100000] Time: 201.4578800201416s
  Unsorted last 15: [47654, 38171, 96881, 69861, 60210, 6313, 47541, 91972, 59001, 92737, 63717, 92908, 73262, 56340, 72199]
  Sorted   last 15: [99985, 99986, 99987, 99988, 99989, 99990, 99991, 99992, 99993, 99994, 99995, 99996, 99997, 99998, 99999]

[Insertion - Best    | n=100000] Time: 0.0094673633575439945s
  Unsorted last 15: [99985, 99986, 99987, 99988, 99989, 99990, 99991, 99992, 99993, 99994, 99995, 99996, 99997, 99998, 99999]
  Sorted   last 15: [99985, 99986, 99987, 99988, 99989, 99990, 99991, 99992, 99993, 99994, 99995, 99996, 99997, 99998, 99999]

[Insertion - Worst   | n=100000] Time: 412.4315493106842s
  Unsorted last 15: [15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
  Sorted   last 15: [99986, 99987, 99988, 99989, 99990, 99991, 99992, 99993, 99994, 99995, 99996, 99997, 99998, 99999, 100000]

[Quick     - Average | n=100000] Time: 0.20372486114501953s
  Unsorted last 15: [8197, 7284, 52346, 13490, 46678, 40547, 92874, 8830, 99180, 46672, 73047, 81182, 69783, 9172, 63710]
  Sorted   last 15: [99985, 99986, 99987, 99988, 99989, 99990, 99991, 99992, 99993, 99994, 99995, 99996, 99997, 99998, 99999]

[Quick     - Best    | n=100000] Time: 234.8128674030304s
  Unsorted last 15: [99985, 99986, 99987, 99988, 99989, 99990, 99991, 99992, 99993, 99994, 99995, 99996, 99997, 99998, 99999]
  Sorted   last 15: [99985, 99986, 99987, 99988, 99989, 99990, 99991, 99992, 99993, 99994, 99995, 99996, 99997, 99998, 99999]

[Quick     - Worst   | n=100000] Time: 226.51213598251343s
  Unsorted last 15: [15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
  Sorted   last 15: [99986, 99987, 99988, 99989, 99990, 99991, 99992, 99993, 99994, 99995, 99996, 99997, 99998, 99999, 100000]
```

Figure 3. Terminal output for 1,00,000 input size

Algorithm	Case	Input Size	Time Take	Original Last 15	Sorted Last 15
Selection	Average	10000	1.647994	[7443, 4558, 2074, 1508, 6519, 6683, 2644, 3429, 1457, 4281, 9653, 6356, 4161, 5924, 6503]	[9985, 9986, 9987, 9988, 9989, 9990, 9991, 9992, 9993, 9994, 9995, 9996, 9997, 9998, 9999]
Selection	Best	10000	1.663303	[9985, 9986, 9987, 9988, 9989, 9990, 9991, 9992, 9993, 9994, 9995, 9996, 9997, 9998, 9999]	[9985, 9986, 9987, 9988, 9989, 9990, 9991, 9992, 9993, 9994, 9995, 9996, 9997, 9998, 9999]
Selection	Worst	10000	1.745596	[15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]	[9986, 9987, 9988, 9989, 9990, 9991, 9992, 9993, 9994, 9995, 9996, 9997, 9998, 9999, 10000]
Bubble	Average	10000	4.050352	[3740, 3213, 8962, 8062, 7234, 5606, 1360, 5634, 7886, 8439, 7772, 8143, 8196, 4905, 2592]	[9985, 9986, 9987, 9988, 9989, 9990, 9991, 9992, 9993, 9994, 9995, 9996, 9997, 9998, 9999]
Bubble	Best	10000	0.000524	[9985, 9986, 9987, 9988, 9989, 9990, 9991, 9992, 9993, 9994, 9995, 9996, 9997, 9998, 9999]	[9985, 9986, 9987, 9988, 9989, 9990, 9991, 9992, 9993, 9994, 9995, 9996, 9997, 9998, 9999]
Bubble	Worst	10000	5.587057	[15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]	[9986, 9987, 9988, 9989, 9990, 9991, 9992, 9993, 9994, 9995, 9996, 9997, 9998, 9999, 10000]
Insertion	Average	10000	2.034483	[1896, 6951, 6140, 2955, 5052, 987, 3307, 3774, 1649, 527, 4108, 5431, 5532, 8976, 4663]	[9985, 9986, 9987, 9988, 9989, 9990, 9991, 9992, 9993, 9994, 9995, 9996, 9997, 9998, 9999]
Insertion	Best	10000	0.000992	[9985, 9986, 9987, 9988, 9989, 9990, 9991, 9992, 9993, 9994, 9995, 9996, 9997, 9998, 9999]	[9985, 9986, 9987, 9988, 9989, 9990, 9991, 9992, 9993, 9994, 9995, 9996, 9997, 9998, 9999]
Insertion	Worst	10000	3.970928	[15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]	[9986, 9987, 9988, 9989, 9990, 9991, 9992, 9993, 9994, 9995, 9996, 9997, 9998, 9999, 10000]
Quick	Average	10000	0.015493	[783, 1160, 7458, 9529, 5127, 3981, 7147, 3548, 6809, 9396, 2586, 5807, 2478, 219, 4797]	[9985, 9986, 9987, 9988, 9989, 9990, 9991, 9992, 9993, 9994, 9995, 9996, 9997, 9998, 9999]
Quick	Best	10000	2.514046	[9985, 9986, 9987, 9988, 9989, 9990, 9991, 9992, 9993, 9994, 9995, 9996, 9997, 9998, 9999]	[9985, 9986, 9987, 9988, 9989, 9990, 9991, 9992, 9993, 9994, 9995, 9996, 9997, 9998, 9999]
Quick	Worst	10000	2.444686	[15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]	[9986, 9987, 9988, 9989, 9990, 9991, 9992, 9993, 9994, 9995, 9996, 9997, 9998, 9999, 10000]
Selection	Average	50000	46.26494	[41421, 20661, 34597, 49297, 15206, 16816, 29459, 12417, 16432, 32345, 32763, 18789, 48000, 183	[49985, 49986, 49987, 49988, 49989, 49990, 49991, 49992, 49993, 49994, 49995, 49996, 49997, 49998, 49999]
Selection	Best	50000	40.37661	[49985, 49986, 49987, 49988, 49989, 49990, 49991, 49992, 49993, 49994, 49995, 49996, 49997, 499	[49985, 49986, 49987, 49988, 49989, 49990, 49991, 49992, 49993, 49994, 49995, 49996, 49997, 49998, 49999]
Selection	Worst	50000	43.60563	[15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]	[49986, 49987, 49988, 49989, 49990, 49991, 49992, 49993, 49994, 49995, 49996, 49997, 49998, 49999, 50000]
Bubble	Average	50000	107.0613	[39153, 25340, 15326, 16400, 9514, 6984, 45942, 5900, 1913, 4541, 19634, 12382, 40327, 39279, 32	[49985, 49986, 49987, 49988, 49989, 49990, 49991, 49992, 49993, 49994, 49995, 49996, 49997, 49998, 49999]
Bubble	Best	50000	0.002456	[49985, 49986, 49987, 49988, 49989, 49990, 49991, 49992, 49993, 49994, 49995, 49996, 49997, 499	[49985, 49986, 49987, 49988, 49989, 49990, 49991, 49992, 49993, 49994, 49995, 49996, 49997, 49998, 49999]
Bubble	Worst	50000	134.5131	[15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]	[49986, 49987, 49988, 49989, 49990, 49991, 49992, 49993, 49994, 49995, 49996, 49997, 49998, 49999, 50000]
Insertion	Average	50000	50.6842	[40682, 25121, 8386, 41467, 15784, 1748, 47932, 37211, 36394, 26487, 242, 12469, 26378, 29436, 4	[49985, 49986, 49987, 49988, 49989, 49990, 49991, 49992, 49993, 49994, 49995, 49996, 49997, 49998, 49999]
Insertion	Best	50000	0.005219	[49985, 49986, 49987, 49988, 49989, 49990, 49991, 49992, 49993, 49994, 49995, 49996, 49997, 499	[49985, 49986, 49987, 49988, 49989, 49990, 49991, 49992, 49993, 49994, 49995, 49996, 49997, 49998, 49999]
Insertion	Worst	50000	100.9534	[15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]	[49986, 49987, 49988, 49989, 49990, 49991, 49992, 49993, 49994, 49995, 49996, 49997, 49998, 49999, 50000]
Quick	Average	50000	0.119811	[39582, 30291, 49933, 12445, 13064, 46463, 36503, 2471, 14780, 17792, 2715, 3178, 3998, 41802, 1	[49985, 49986, 49987, 49988, 49989, 49990, 49991, 49992, 49993, 49994, 49995, 49996, 49997, 49998, 49999]
Quick	Best	50000	56.5969	[49985, 49986, 49987, 49988, 49989, 49990, 49991, 49992, 49993, 49994, 49995, 49996, 49997, 499	[49985, 49986, 49987, 49988, 49989, 49990, 49991, 49992, 49993, 49994, 49995, 49996, 49997, 49998, 49999]
Quick	Worst	50000	55.78374	[15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]	[49986, 49987, 49988, 49989, 49990, 49991, 49992, 49993, 49994, 49995, 49996, 49997, 49998, 49999, 50000]
Selection	Average	100000	253.9356	[73002, 69261, 77151, 37371, 20101, 83209, 44770, 40328, 22900, 65934, 93501, 72655, 63049, 162	[99985, 99986, 99987, 99988, 99989, 99990, 99991, 99992, 99993, 99994, 99995, 99996, 99997, 99998, 99999]
Selection	Best	100000	160.4739	[99985, 99986, 99987, 99988, 99989, 99990, 99991, 99992, 99993, 99994, 99995, 99996, 99997, 999	[99985, 99986, 99987, 99988, 99989, 99990, 99991, 99992, 99993, 99994, 99995, 99996, 99997, 99998, 99999]
Selection	Worst	100000	174.5396	[15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]	[99986, 99987, 99988, 99989, 99990, 99991, 99992, 99993, 99994, 99995, 99996, 99997, 99998, 99999, 100000]
Bubble	Average	100000	469.8561	[78036, 11887, 86596, 30780, 24415, 96658, 43225, 97521, 49862, 34282, 72541, 92175, 51813, 494	[99985, 99986, 99987, 99988, 99989, 99990, 99991, 99992, 99993, 99994, 99995, 99996, 99997, 99998, 99999]
Bubble	Best	100000	0.004772	[99985, 99986, 99987, 99988, 99989, 99990, 99991, 99992, 99993, 99994, 99995, 99996, 99997, 999	[99985, 99986, 99987, 99988, 99989, 99990, 99991, 99992, 99993, 99994, 99995, 99996, 99997, 99998, 99999]
Bubble	Worst	100000	537.8258	[15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]	[99986, 99987, 99988, 99989, 99990, 99991, 99992, 99993, 99994, 99995, 99996, 99997, 99998, 99999, 100000]
Insertion	Average	100000	201.4579	[47654, 38171, 96881, 69861, 60210, 6313, 47541, 91972, 59001, 92737, 63717, 92908, 73262, 5634	[99985, 99986, 99987, 99988, 99989, 99990, 99991, 99992, 99993, 99994, 99995, 99996, 99997, 99998, 99999]
Insertion	Best	100000	0.009467	[99985, 99986, 99987, 99988, 99989, 99990, 99991, 99992, 99993, 99994, 99995, 99996, 99997, 999	[99985, 99986, 99987, 99988, 99989, 99990, 99991, 99992, 99993, 99994, 99995, 99996, 99997, 99998, 99999]
Insertion	Worst	100000	412.4315	[15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]	[99986, 99987, 99988, 99989, 99990, 99991, 99992, 99993, 99994, 99995, 99996, 99997, 99998, 99999, 100000]
Quick	Average	100000	0.203725	[8197, 7284, 52346, 13490, 46678, 40547, 92874, 8830, 99180, 46672, 73047, 81182, 69783, 9172, 6	[99985, 99986, 99987, 99988, 99989, 99990, 99991, 99992, 99993, 99994, 99995, 99996, 99997, 99998, 99999]
Quick	Best	100000	234.8129	[99985, 99986, 99987, 99988, 99989, 99990, 99991, 99992, 99993, 99994, 99995, 99996, 99997, 999	[99985, 99986, 99987, 99988, 99989, 99990, 99991, 99992, 99993, 99994, 99995, 99996, 99997, 99998, 99999]
Quick	Worst	100000	226.5121	[15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]	[99986, 99987, 99988, 99989, 99990, 99991, 99992, 99993, 99994, 99995, 99996, 99997, 99998, 99999, 100000]

Figure 4. Verification Logs CSV File

Input Size	Selection Sort (Best)	Selection Sort (Averag	Selection Sort (Wors	Bubble Sort (Best)	Bubble Sort (Average	Bubble Sort (Wors	Insertion Sort (Best	Insertion Sort (Averag	Insertion Sort (Worst	Quick Sort (Best	Quick Sort (Average)	Quick Sort (Worst
10000	1.663303137	1.647994041	1.745595694	0.000523806	4.050352335	5.587056637	0.00099206	2.034482718	3.970927954	2.514046192	0.015493393	2.444686
50000	40.37661076	46.26493812	43.60562563	0.002456427	107.0613344	134.5131252	0.005218983	50.6842041	100.9533899	56.59690332	0.119810581	55.78374
100000	160.4738593	253.9356034	174.5395911	0.004772425	469.8560863	537.8257618	0.009467363	201.45788	412.4315493	234.8128674	0.203724861	226.5121

Figure 5. Run-times stored in a CSV File

3. ANALYSIS

In this practical, we implemented and compared four sorting algorithms: Selection Sort, Bubble Sort, Insertion Sort, and Quick Sort (using the first element as the pivot). The objective was to analyze their performance in best, average, and worst-case scenarios on large input sizes and observe their relative efficiency.

3.1. Summary of Results

1. **Selection Sort:** Always performs $O(n^2)$ comparisons regardless of input order. Hence, its best, average, and worst cases are identical.

2. **Bubble Sort:** Optimized with a swap flag; in the best case (already sorted input) it runs in $O(n)$. For average and worst cases, it still performs poorly with $O(n^2)$ complexity.

3. **Insertion Sort:** Performs best in $O(n)$ time for sorted input, as each new element is directly placed. However, in average and worst cases, shifting elements makes it $O(n^2)$.

4. **Quick Sort (first element as pivot):** Theoretical best and average cases are $O(n \log n)$, while the worst case can degrade to $O(n^2)$ if partitions are unbalanced. In our experiment, the average case surprisingly outperformed the best case, which may be due to pivot distribution and memory/cache behavior. Despite this, Quick Sort clearly outperformed all quadratic algorithms for large input sizes.

3.2. Graphical Observations

The runtime comparison graphs for input sizes of **10,000**, **50,000**, and **100,000** elements clearly illustrate the scalability differences:

- **Bubble Sort** shows the steepest growth in runtime, making it impractical for larger datasets.
- **Selection Sort** remains consistent but is still inefficient due to its quadratic time complexity.
- **Insertion Sort** performs reasonably in best-case scenarios but quickly becomes slow for larger inputs.
- **Quick Sort** significantly outperforms the others, maintaining low runtimes across best and average cases, making it the most scalable and practical algorithm.

3.3. 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	$O(n)$	$O(n^2)$	$O(n^2)$
Quick Sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$

Table 1. Complexity comparison of sorting algorithms.

3.4. Graphical Analysis

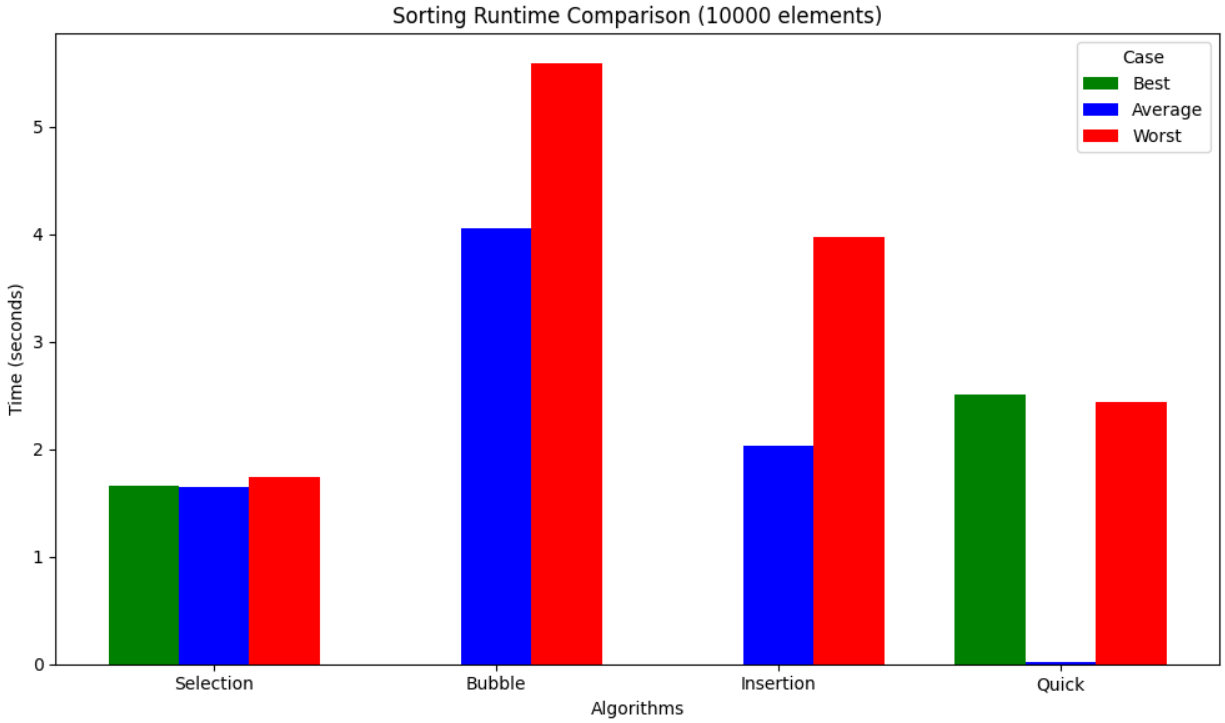

Figure 6. Performance comparison with 10,000 inputs.

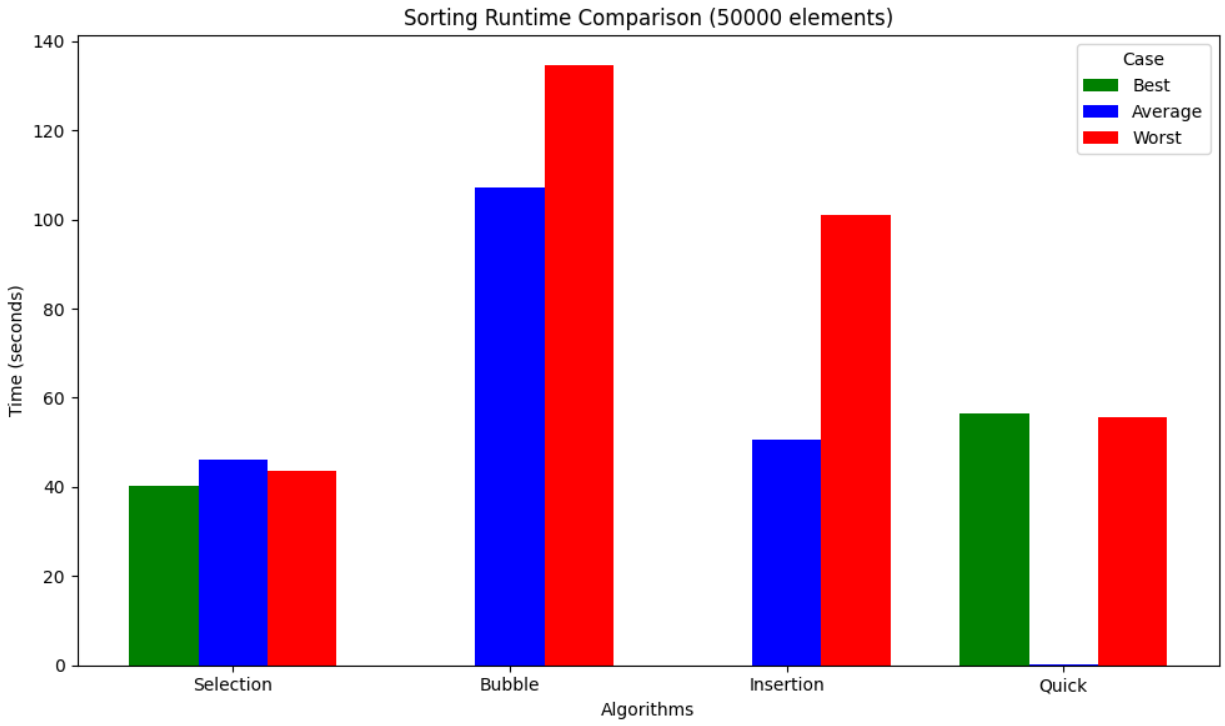

Figure 7. Performance comparison with 50,000 inputs.

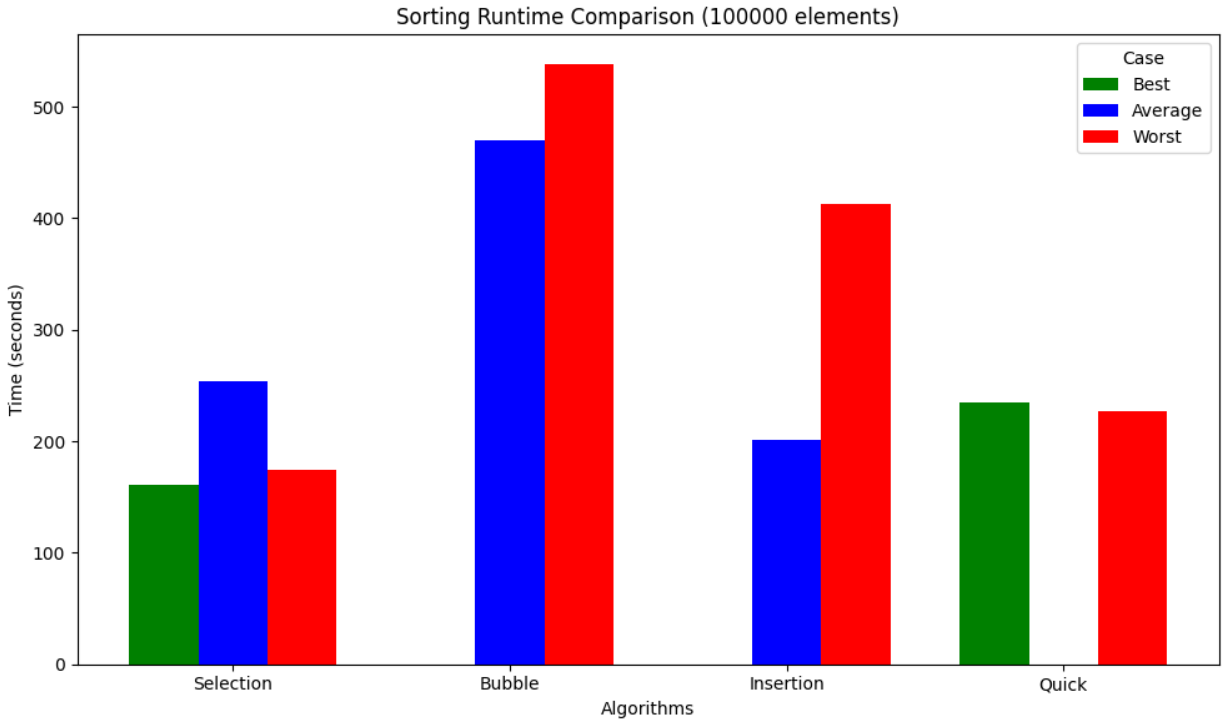

Figure 8. Performance comparison with 100,000 inputs.

4. CONCLUSION

From this practical, we conclude:

- Selection Sort is unaffected by input order, always $O(n^2)$.
- Bubble Sort and Insertion Sort achieve $O(n)$ in best-case scenarios, but degrade to $O(n^2)$ in average and worst cases.
- Insertion Sort generally performs better than Bubble Sort on unsorted data.
- Quick Sort (with the first element as pivot) performs close to $O(n \log n)$ on average, but may degrade to $O(n^2)$ in the worst case when pivot selection is poor.

While all quadratic algorithms become inefficient for large datasets, this experiment shows how input order and pivot strategy significantly influence sorting performance.