DSA Lab 2 – Internal and External Merge Sort Algorithm Analysis

Soham Manish Dave
Enrollment: 25MCD005

ABSTRACT

In this practical, I implemented and compared Internal Merge Sort and External Merge Sort to understand how sorting works in memory versus on disk. Internal merge sort was written using the normal recursive divide-and-merge approach, while external merge sort was done by splitting large data into chunks, sorting each chunk, saving them to temporary files, and then merging them with a k-way merge using a heap. I tested the program on input sizes of 10,000, 50,000 and 100,000 elements under three conditions – best case, average case and worst case. The runtime of each algorithm was measured and graphs were generated to show the performance. This experiment helped me see how internal sorting is faster for small datasets and how external sorting is useful when the data is too big to fit into memory.

1. CODE

Listing 1. Implementation of Internal and External Merge Sort Algorithms

```
import time
import random
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import os
import csv
import heapq

# ---------------- Internal Merge Sort ---------------- #
def merge(arr, left, mid, right):
    n1 = mid - left + 1
    n2 = right - mid

    L = arr[left:mid+1]
    R = arr[mid+1:right+1]

    i = j = 0
    k = left

    while i < n1 and j < n2:
        if L[i] <= R[j]:
            arr[k] = L[i]
            i += 1
        else:
            arr[k] = R[j]
            j += 1
        k += 1

    while i < n1:
        arr[k] = L[i]
        i += 1
        k += 1
    while j < n2:
        arr[k] = R[j]
        j += 1
        k += 1

def internal_merge_sort(arr, left, right):
```

```python
    if left < right:
        mid = (left + right) // 2
        internal_merge_sort(arr, left, mid)
        internal_merge_sort(arr, mid+1, right)
        merge(arr, left, mid, right)

# ---------------- K-way Merge (File-based) ---------------- #
def k_way_merge(temp_file_paths, output_file_path):

    input_files = []
    heap = []
    readers = []

    try:
        # Open each temporary file and initialize the heap
        for i, file_path in enumerate(temp_file_paths):
            f = open(file_path, 'r')
            readers.append(csv.reader(f))
            try:
                # Read the first element and push to heap (convert to int)
                first_element = next(readers[-1])
                if first_element:
                    heapq.heappush(heap, (int(first_element[0]), i))
            except StopIteration:
                # Handle empty files
                pass
            input_files.append(f)

        # Open the output file
        with open(output_file_path, 'w', newline='') as outfile:
            writer = csv.writer(outfile)

            # Merge using the min-heap
            while heap:
                smallest_element, file_index = heapq.heappop(heap)
                writer.writerow([smallest_element])

                # Read the next element from the file the smallest came from
                try:
                    next_element = next(readers[file_index])
                    if next_element:
                        heapq.heappush(heap, (int(next_element[0]), file_index))
                except StopIteration:
                    # File is exhausted
                    pass

    except Exception as e:
        print(f"An error occurred during k-way merge: {e}")
    finally:
        # Close all input files
        for f in input_files:
            f.close()

# ---------------- External Merge Sort (File-based) ---------------- #
def external_merge_sort(arr, output_file_path, chunk_size=1000):
    temp_files = []
    for i in range(0, len(arr), chunk_size):
        chunk = arr[i:i+chunk_size]
        internal_merge_sort(chunk, 0, len(chunk)-1)

        file_path = f"temp_chunk_{i // chunk_size}.csv"
```

```
        pd.DataFrame(chunk).to_csv(file_path, index=False, header=False)
        temp_files.append(file_path)

    k_way_merge(temp_files, output_file_path)
    return output_file_path

# ---------------- Main Experiment ---------------- #
def run_experiment():
    sizes = [10000, 50000, 100000]
    chunk_size = 1000
    cases = {
        "Best": lambda n: list(range(n)),
        "Average": lambda n: random.sample(range(n), n),
        "Worst": lambda n: list(range(n, 0, -1))
    }

    results = []
    last_15_elements_data = []

    for size in sizes:
        print(f"\n{'='*20} Input Size: {size} {'='*20}")
        for case_name, case_gen in cases.items():
            arr = case_gen(size)
            arr_copy1 = arr[:]  # For internal
            arr_copy2 = arr[:] # For external

            original_last_15 = arr[-15:] if len(arr) >= 15 else arr
            print(f"\nCase: {case_name}")
            print(f"  Original last 15 elements: {original_last_15}")
            last_15_elements_data.append({
                "Algorithm": "Original",
                "Input Size": size,
                "Case": case_name,
                "Last 15 Elements": original_last_15
            })

            print(f"  Internal Merge Sort:")
            start = time.time()
            internal_merge_sort(arr_copy1, 0, len(arr_copy1)-1)
            end = time.time()
            internal_time = end - start
            print(f"    Time: {internal_time:.6f} sec")
            results.append(["Internal Merge Sort", size, case_name, None, internal_time])

            internal_sorted_last_15 = arr_copy1[-15:] if len(arr_copy1) >= 15 else arr_copy1
            print(f"    Sorted last 15 elements: {internal_sorted_last_15}")
            last_15_elements_data.append({
                "Algorithm": "Internal Merge Sort",
                "Input Size": size,
                "Case": case_name,
                "Last 15 Elements": internal_sorted_last_15
            })

            print(f"  External Merge Sort:")
            output_file = f"external_sorted_{size}_{case_name}_chunk_{chunk_size}.csv"
            start = time.time()
            external_merge_sort(arr_copy2, output_file, chunk_size=chunk_size)
            end = time.time()
            external_time = end - start
            print(f"    Time: {external_time:.6f} sec")
            results.append(["External Merge Sort", size, case_name, chunk_size, external_time])
```

```
            try:
                external_sorted_df = pd.read_csv(output_file, header=None)
                external_sorted_arr = external_sorted_df[0].tolist()
                external_sorted_last_15 = external_sorted_arr[-15:] if len(external_sorted_arr)
                    >= 15 else external_sorted_arr
                print(f"    Sorted last 15 elements from file: {external_sorted_last_15}")
                last_15_elements_data.append({
                    "Algorithm": "External Merge Sort",
                    "Input Size": size,
                    "Case": case_name,
                    "Last 15 Elements": external_sorted_last_15
                })
            except Exception as e:
                print(f"Error reading external sort output file {output_file}: {e}")

    df_runtimes = pd.DataFrame(results, columns=["Algorithm", "Input Size", "Case", "Chunk Size",
         "Runtime"])
    df_runtimes.to_csv("merge_sort_fixed_chunk_results.csv", index=False)
    print("\nRuntime results saved to merge_sort_fixed_chunk_results.csv")

    df_last_15 = pd.DataFrame(last_15_elements_data)
    df_last_15.to_csv("merge_sort_last_15_elements.csv", index=False)
    print("Last 15 elements data saved to merge_sort_last_15_elements.csv")

    plot_graphs(df_runtimes)


def plot_graphs(df):
    sizes = df["Input Size"].unique()
    cases = df["Case"].unique()
    chunk_size_series = df["Chunk Size"].dropna().unique()
    chunk_size = chunk_size_series[0] if chunk_size_series.size > 0 else 'N/A'
    case_colors = {"Best": "green", "Average": "blue", "Worst": "red"}

    for size in sizes:
        plt.figure(figsize=(12, 7))
        subset = df[df["Input Size"] == size]

        num_algorithms = 2
        num_cases = len(cases)
        group_width = num_cases * 0.8
        bar_width = group_width / num_cases
        gap_between_algorithms = 0.4

        internal_group_center = num_cases * bar_width / 2
        external_group_center = internal_group_center + group_width + gap_between_algorithms

        internal_bar_positions = np.linspace(internal_group_center - group_width / 2 + bar_width
            / 2,
                                             internal_group_center + group_width / 2 - bar_width
                                                 / 2,
                                             num_cases)
        external_bar_positions = np.linspace(external_group_center - group_width / 2 + bar_width
            / 2,
                                             external_group_center + group_width / 2 - bar_width
                                                 / 2,
                                             num_cases)
```

```python
        internal_runtimes = [subset[(subset["Algorithm"] == "Internal Merge Sort") & (subset["Case"] == case)]["Runtime"].iloc[0] for case in cases]
        external_runtimes = [subset[(subset["Algorithm"] == "External Merge Sort") & (subset["Case"] == case)]["Runtime"].iloc[0] for case in cases]

        plt.bar(internal_bar_positions, internal_runtimes, bar_width, color=[case_colors[case] for case in cases])
        plt.bar(external_bar_positions, external_runtimes, bar_width, color=[case_colors[case] for case in cases])

        plt.ylabel("Runtime (seconds)")
        plt.title(f"Merge Sort Performance - Input Size: {size}")

        all_bar_positions = np.concatenate([internal_bar_positions, external_bar_positions])
        x_labels = [f"Internal\n{case}" for case in cases] + [f"External\n{case}\n(Chunk {chunk_size})" for case in cases]
        plt.xticks(all_bar_positions, x_labels)

        case_legend_handles = [plt.Rectangle((0,0),1,1, color=case_colors[case]) for case in cases]
        plt.legend(case_legend_handles, cases, title="Case Type", bbox_to_anchor=(1.05, 1), loc='upper left')

        plt.tight_layout()
        plt.savefig(f"merge_sort_performance_size_{size}_chunk_{chunk_size}.png")
        plt.show()

if __name__ == "__main__":
    run_experiment()
```

2. OUTPUT

The terminal outputs for input sizes of 10,000, 50,000, and 100,000 elements display the performance of **Internal Merge Sort** and **External Merge Sort** 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 the correctness of both implementations. The execution time for each run is also shown on the terminal for direct performance comparison.

In addition to the terminal outputs, the verification logs are saved in the file `merge_sort_last_15_elements.csv`, which contains the last 15 elements before and after sorting for each algorithm, input size, and case. The runtimes for each algorithm and case are stored in `merge_sort_fixed_chunk_results.csv`, which records the execution time of Internal Merge Sort and External Merge Sort for all tested conditions.

The final sorted data from the External Merge Sort is saved as multiple CSV files such as `external_sorted_10000_Best_chunk_1000.csv`, `external_sorted_50000_Average_chunk_1000.csv`, and `external_sorted_100000_Worst_chunk_1000.csv`, while the intermediate chunks generated during the external sorting process are stored as `temp_chunk_X.csv` files. These chunk files are the individual sorted partitions of the dataset that were merged later to produce the final sorted output.

The runtime comparison graphs are saved as `merge_sort_performance_size_10000_chunk_1000.png`, `merge_sort_performance_size_50000_chunk_1000.png`, and `merge_sort_performance_size_100000_chunk_1000.png`, providing a visual comparison of Internal Merge Sort and External Merge Sort under different input sizes and case scenarios.

```
=================== Input Size: 10000 ===================

Case: Best
  Original last 15 elements: [9985, 9986, 9987, 9988, 9989, 9990, 9991, 9992, 9993, 9994, 9995, 9996, 9997, 9998, 9999]
  Internal Merge Sort:
    Time: 0.015644 sec
    Sorted last 15 elements: [9985, 9986, 9987, 9988, 9989, 9990, 9991, 9992, 9993, 9994, 9995, 9996, 9997, 9998, 9999]
  External Merge Sort:
    Time: 0.176156 sec
    Sorted last 15 elements from file: [9985, 9986, 9987, 9988, 9989, 9990, 9991, 9992, 9993, 9994, 9995, 9996, 9997, 9998, 9999]

Case: Average
  Original last 15 elements: [8188, 8024, 6137, 3481, 6533, 2590, 3586, 7337, 7162, 4150, 3022, 8711, 3274, 1486, 5524]
  Internal Merge Sort:
    Time: 0.019395 sec
    Sorted last 15 elements: [9985, 9986, 9987, 9988, 9989, 9990, 9991, 9992, 9993, 9994, 9995, 9996, 9997, 9998, 9999]
  External Merge Sort:
    Time: 0.126946 sec
    Sorted last 15 elements from file: [9985, 9986, 9987, 9988, 9989, 9990, 9991, 9992, 9993, 9994, 9995, 9996, 9997, 9998, 9999]

Case: Worst
  Original last 15 elements: [15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
  Internal Merge Sort:
    Time: 0.015207 sec
    Sorted last 15 elements: [9986, 9987, 9988, 9989, 9990, 9991, 9992, 9993, 9994, 9995, 9996, 9997, 9998, 9999, 10000]
  External Merge Sort:
    Time: 0.176355 sec
    Sorted last 15 elements from file: [9986, 9987, 9988, 9989, 9990, 9991, 9992, 9993, 9994, 9995, 9996, 9997, 9998, 9999, 10000]
```

Figure 1. Terminal output for 10,000 input size

```
=================== Input Size: 50000 ===================

Case: Best
  Original last 15 elements: [49985, 49986, 49987, 49988, 49989, 49990, 49991, 49992, 49993, 49994, 49995, 49996, 49997, 49998, 49999]
  Internal Merge Sort:
    Time: 0.087555 sec
    Sorted last 15 elements: [49985, 49986, 49987, 49988, 49989, 49990, 49991, 49992, 49993, 49994, 49995, 49996, 49997, 49998, 49999]
  External Merge Sort:
    Time: 0.568370 sec
    Sorted last 15 elements from file: [49985, 49986, 49987, 49988, 49989, 49990, 49991, 49992, 49993, 49994, 49995, 49996, 49997, 49998, 49999]

Case: Average
  Original last 15 elements: [38755, 40039, 5852, 49286, 18787, 19005, 5195, 40298, 35129, 10547, 37775, 6664, 10420, 37915, 31190]
  Internal Merge Sort:
    Time: 0.107099 sec
    Sorted last 15 elements: [49985, 49986, 49987, 49988, 49989, 49990, 49991, 49992, 49993, 49994, 49995, 49996, 49997, 49998, 49999]
  External Merge Sort:
    Time: 0.666589 sec
    Sorted last 15 elements from file: [49985, 49986, 49987, 49988, 49989, 49990, 49991, 49992, 49993, 49994, 49995, 49996, 49997, 49998, 49999]

Case: Worst
  Original last 15 elements: [15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
  Internal Merge Sort:
    Time: 0.085025 sec
    Sorted last 15 elements: [49986, 49987, 49988, 49989, 49990, 49991, 49992, 49993, 49994, 49995, 49996, 49997, 49998, 49999, 50000]
  External Merge Sort:
    Time: 0.612629 sec
    Sorted last 15 elements from file: [49986, 49987, 49988, 49989, 49990, 49991, 49992, 49993, 49994, 49995, 49996, 49997, 49998, 49999, 50000]
```

Figure 2. Terminal output for 50,000 input size

```
=================== Input Size: 100000 ===================

Case: Best
  Original last 15 elements: [99985, 99986, 99987, 99988, 99989, 99990, 99991, 99992, 99993, 99994, 99995, 99996, 99997, 99998, 99999]
  Internal Merge Sort:
    Time: 0.180093 sec
    Sorted last 15 elements: [99985, 99986, 99987, 99988, 99989, 99990, 99991, 99992, 99993, 99994, 99995, 99996, 99997, 99998, 99999]
  External Merge Sort:
    Time: 0.912438 sec
    Sorted last 15 elements from file: [99985, 99986, 99987, 99988, 99989, 99990, 99991, 99992, 99993, 99994, 99995, 99996, 99997, 99998, 99999]

Case: Average
  Original last 15 elements: [87352, 1317, 91098, 10386, 19458, 62087, 68595, 414, 52811, 1856, 68254, 74669, 32288, 2749, 80735]
  Internal Merge Sort:
    Time: 0.228871 sec
    Sorted last 15 elements: [99985, 99986, 99987, 99988, 99989, 99990, 99991, 99992, 99993, 99994, 99995, 99996, 99997, 99998, 99999]
  External Merge Sort:
    Time: 1.312030 sec
    Sorted last 15 elements from file: [99985, 99986, 99987, 99988, 99989, 99990, 99991, 99992, 99993, 99994, 99995, 99996, 99997, 99998, 99999]

Case: Worst
  Original last 15 elements: [15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
  Internal Merge Sort:
    Time: 0.178780 sec
    Sorted last 15 elements: [99986, 99987, 99988, 99989, 99990, 99991, 99992, 99993, 99994, 99995, 99996, 99997, 99998, 99999, 100000]
  External Merge Sort:
    Time: 0.920578 sec
    Sorted last 15 elements from file: [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	Input Size	Case	Last 15 Elements
Original	10000	Best	[9985, 9986, 9987, 9988, 9989, 9990, 9991, 9992, 9993, 9994, 9995, 9996, 9997, 9998, 9999]
Internal Merge Sort	10000	Best	[9985, 9986, 9987, 9988, 9989, 9990, 9991, 9992, 9993, 9994, 9995, 9996, 9997, 9998, 9999]
External Merge Sort	10000	Best	[9985, 9986, 9987, 9988, 9989, 9990, 9991, 9992, 9993, 9994, 9995, 9996, 9997, 9998, 9999]
Original	10000	Average	[8188, 8024, 6137, 3481, 6533, 2590, 3586, 7337, 7162, 4150, 3022, 8711, 3274, 1486, 5524]
Internal Merge Sort	10000	Average	[9985, 9986, 9987, 9988, 9989, 9990, 9991, 9992, 9993, 9994, 9995, 9996, 9997, 9998, 9999]
External Merge Sort	10000	Average	[9985, 9986, 9987, 9988, 9989, 9990, 9991, 9992, 9993, 9994, 9995, 9996, 9997, 9998, 9999]
Original	10000	Worst	[15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
Internal Merge Sort	10000	Worst	[9986, 9987, 9988, 9989, 9990, 9991, 9992, 9993, 9994, 9995, 9996, 9997, 9998, 9999, 10000]
External Merge Sort	10000	Worst	[9986, 9987, 9988, 9989, 9990, 9991, 9992, 9993, 9994, 9995, 9996, 9997, 9998, 9999, 10000]
Original	50000	Best	[49985, 49986, 49987, 49988, 49989, 49990, 49991, 49992, 49993, 49994, 49995, 49996, 49997, 49998, 49999]
Internal Merge Sort	50000	Best	[49985, 49986, 49987, 49988, 49989, 49990, 49991, 49992, 49993, 49994, 49995, 49996, 49997, 49998, 49999]
External Merge Sort	50000	Best	[49985, 49986, 49987, 49988, 49989, 49990, 49991, 49992, 49993, 49994, 49995, 49996, 49997, 49998, 49999]
Original	50000	Average	[38755, 40039, 5852, 49286, 18787, 19005, 5195, 40298, 35129, 10547, 37775, 6664, 10420, 37915, 31190]
Internal Merge Sort	50000	Average	[49985, 49986, 49987, 49988, 49989, 49990, 49991, 49992, 49993, 49994, 49995, 49996, 49997, 49998, 49999]
External Merge Sort	50000	Average	[49985, 49986, 49987, 49988, 49989, 49990, 49991, 49992, 49993, 49994, 49995, 49996, 49997, 49998, 49999]
Original	50000	Worst	[15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
Internal Merge Sort	50000	Worst	[49986, 49987, 49988, 49989, 49990, 49991, 49992, 49993, 49994, 49995, 49996, 49997, 49998, 49999, 50000]
External Merge Sort	50000	Worst	[49986, 49987, 49988, 49989, 49990, 49991, 49992, 49993, 49994, 49995, 49996, 49997, 49998, 49999, 50000]
Original	100000	Best	[99985, 99986, 99987, 99988, 99989, 99990, 99991, 99992, 99993, 99994, 99995, 99996, 99997, 99998, 99999]
Internal Merge Sort	100000	Best	[99985, 99986, 99987, 99988, 99989, 99990, 99991, 99992, 99993, 99994, 99995, 99996, 99997, 99998, 99999]
External Merge Sort	100000	Best	[99985, 99986, 99987, 99988, 99989, 99990, 99991, 99992, 99993, 99994, 99995, 99996, 99997, 99998, 99999]
Original	100000	Average	[87352, 1317, 91098, 10386, 19458, 62087, 68595, 414, 52811, 1856, 68254, 74669, 32288, 2749, 80735]
Internal Merge Sort	100000	Average	[99985, 99986, 99987, 99988, 99989, 99990, 99991, 99992, 99993, 99994, 99995, 99996, 99997, 99998, 99999]
External Merge Sort	100000	Average	[99985, 99986, 99987, 99988, 99989, 99990, 99991, 99992, 99993, 99994, 99995, 99996, 99997, 99998, 99999]
Original	100000	Worst	[15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
Internal Merge Sort	100000	Worst	[99986, 99987, 99988, 99989, 99990, 99991, 99992, 99993, 99994, 99995, 99996, 99997, 99998, 99999, 100000]
External Merge Sort	100000	Worst	[99986, 99987, 99988, 99989, 99990, 99991, 99992, 99993, 99994, 99995, 99996, 99997, 99998, 99999, 100000]

Figure 4. Verification Logs CSV File

Algorithm	Input Size	Case	Chunk Size	Runtime
Internal Merge Sort	10000	Best		0.015644
External Merge Sort	10000	Best	1000	0.176156
Internal Merge Sort	10000	Average		0.019395
External Merge Sort	10000	Average	1000	0.126946
Internal Merge Sort	10000	Worst		0.015207
External Merge Sort	10000	Worst	1000	0.176355
Internal Merge Sort	50000	Best		0.087555
External Merge Sort	50000	Best	1000	0.56837
Internal Merge Sort	50000	Average		0.107099
External Merge Sort	50000	Average	1000	0.666589
Internal Merge Sort	50000	Worst		0.085025
External Merge Sort	50000	Worst	1000	0.612629
Internal Merge Sort	100000	Best		0.180093
External Merge Sort	100000	Best	1000	0.912438
Internal Merge Sort	100000	Average		0.228871
External Merge Sort	100000	Average	1000	1.31203
Internal Merge Sort	100000	Worst		0.17878
External Merge Sort	100000	Worst	1000	0.920578

Figure 5. Run-times stored in a CSV File

3. ANALYSIS

In this practical, we implemented and compared two sorting techniques: Internal Merge Sort and External Merge Sort. The objective was to analyze their performance in best, average, and worst-case scenarios on large input sizes and observe their relative efficiency, especially when data exceeds main memory limits.

3.1. Summary of Results

1. **Internal Merge Sort:** Achieved the lowest runtimes for all input sizes. For example, at 100,000 elements its runtime was about 0.18 seconds compared to more than 0.9 seconds for External Merge Sort. The differences between best, average, and worst cases were very small, showing the algorithm's stability.
2. **External Merge Sort:** Consistently slower due to file I/O overhead. At 100,000 elements it took roughly 0.9–1.3 seconds across the three cases. It still handled the large datasets correctly, demonstrating its scalability, but was outperformed by Internal Merge Sort for all input sizes tested (since the data still fit into memory).
3. **Best vs Average vs Worst Cases:** For both algorithms, the runtime differences across best, average,

and worst cases were minimal compared to quadratic-time algorithms. The major difference came from the size of the input and the disk operations of External Merge Sort.

3.2. Graphical Observations

The runtime comparison graphs for input sizes of **10,000**, **50,000**, and **100,000** elements clearly illustrate the differences between Internal and External Merge Sort:

- **Internal Merge Sort** remained consistently the fastest across all input sizes, completing even 100,000 elements in under 0.23 seconds.
- **External Merge Sort** scaled to large datasets but was 5–8 times slower because of disk read/write operations.
- Both algorithms maintained stable runtimes across best, average, and worst cases compared to traditional $O(n^2)$ algorithms.
- The graphs demonstrate that External Merge Sort is best suited for extremely large datasets that cannot fit into memory, while Internal Merge Sort is ideal for in-memory operations.

3.3. Complexity Comparison Table

Algorithm	Best Case	Average Case	Worst Case
Internal Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
External Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$

Table 1. Complexity comparison of Internal and External Merge Sort.

3.4. Graphical Analysis

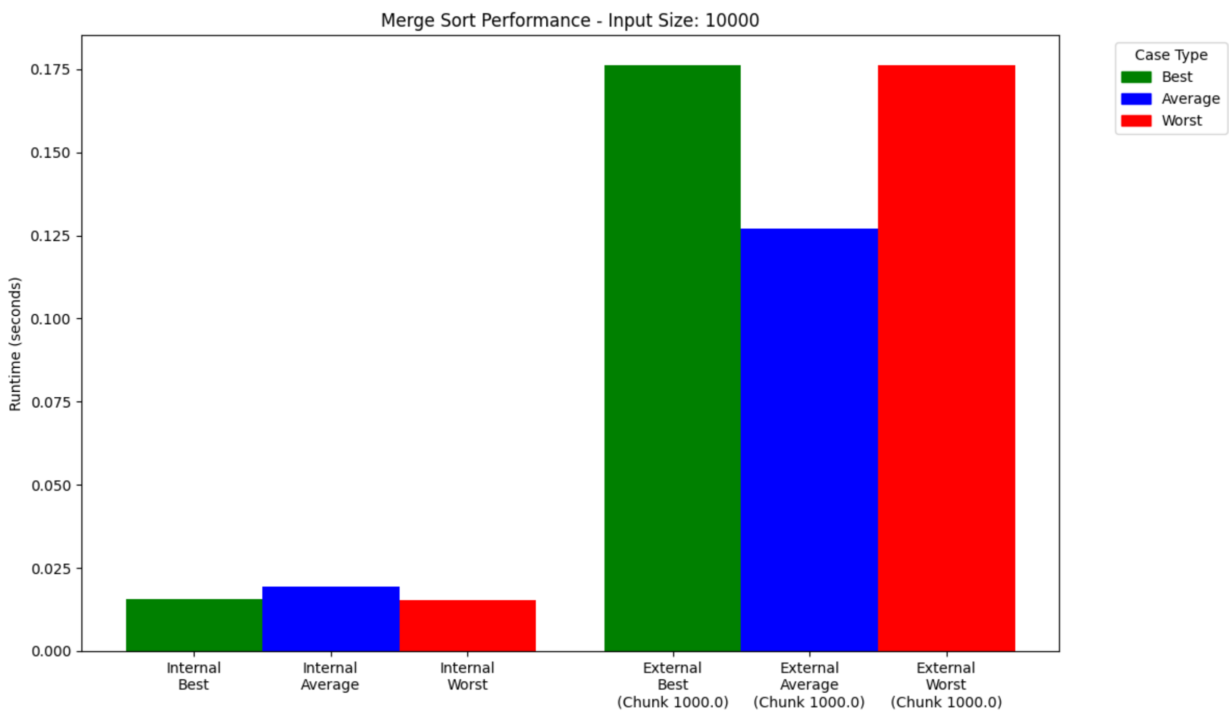

Figure 6. Performance comparison with 10,000 inputs.

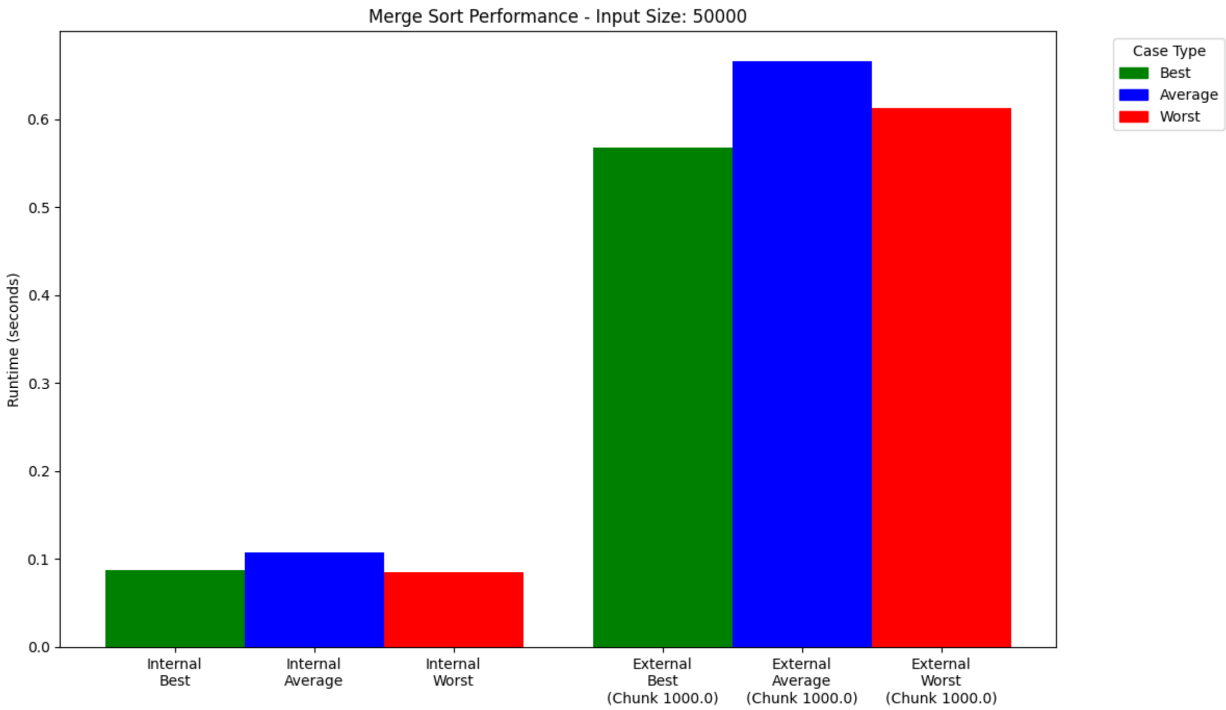

Figure 7. Performance comparison with 50,000 inputs.

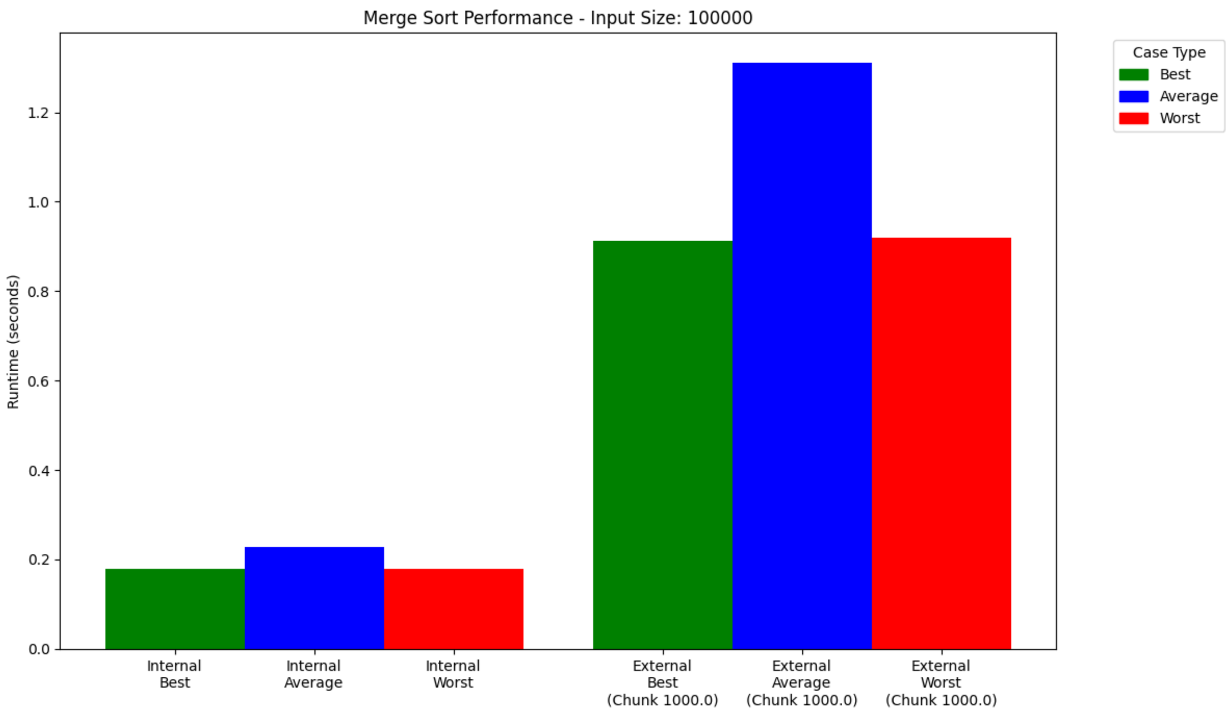

Figure 8. Performance comparison with 100,000 inputs.

4. CONCLUSION

From this practical, we conclude:

- Internal Merge Sort shows very consistent runtimes across best, average, and worst cases. This confirms its $O(n \log n)$ performance regardless of input order.
- External Merge Sort (chunk size 1000) also maintains $O(n \log n)$ complexity, but its runtimes are higher than internal merge sort due to disk I/O and chunk merging overhead.

- As the input size increases from 10,000 to 100,000, both internal and external merge sort runtimes increase proportionally, confirming the scalability trend.
- External Merge Sort shows a small variation between best, average, and worst cases, but the difference is minimal compared to the dramatic changes seen in Quick Sort or Bubble Sort.
- Internal Merge Sort is preferable for in-memory datasets, while External Merge Sort becomes essential for large datasets that cannot fit entirely in memory.

This experiment shows how Merge Sort maintains predictable $O(n \log n)$ behavior across different cases, and how external memory operations influence performance at larger scales.