Task 1 – "Ordnung must sein" Report

Task:

A. Implement bubble sort, insertion sort, merge sort, quick sort. Test and compare their running times for a number of

- small (ca. 5–50) and
- large input sizes.

Carefully choose (or even average) appropriate inputs to ensure you justify the right conclusion. Include a graph with the results of your measurements for all algorithms, e.g. running time as a function of array size.

Introduction:

This report presents an empirical comparison of four sorting algorithms: Bubble Sort, Insertion Sort, Merge Sort, and Quick Sort. The goal is to analyze their performance on datasets of varying sizes, confirming theoretical time complexities through experimental results. The report also includes a graphical representation of the findings.

Sorting Algorithms:

Bubble Sort:

- Worst-case time complexity: $O(n^2)$
- Best-case time complexity: O(n) (if the array is already sorted)
- Space complexity: O(1)

Insertion Sort:

- Worst-case time complexity: $O(n^2)$
- Best-case time complexity: O(n) (if the array is already sorted)
- Space complexity: O(1)

Merge Sort:

- Worst-case time complexity:** O(n log n)
- Best-case time complexity:** O(n log n)
- Space complexity:** O(n)

Quick Sort:

- Worst-case time complexity:** $O(n^2)$ (if the pivot selection is poor)
- Best-case time complexity:** O(n log n)
- Average-case time complexity:** O(n log n)
- Space complexity:** O(log n) (due to recursion)

Algorithm Setup:

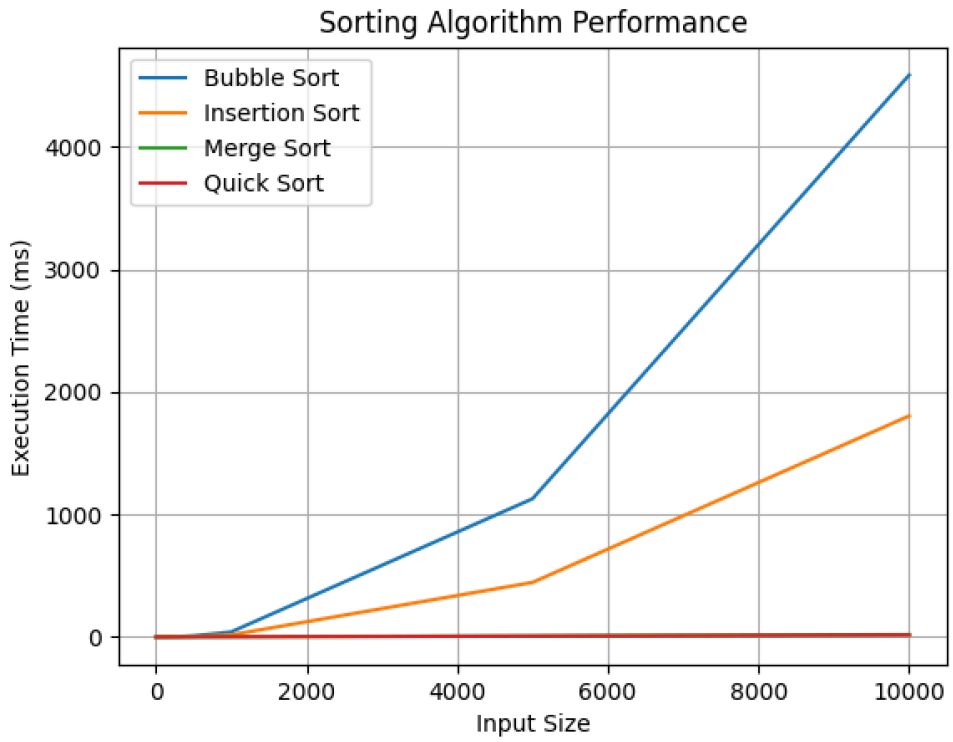
- Random arrays of varying sizes were generated: 5, 10, 25, 50, 100, 250, 500, 1000, 5000, and 10000.
- Each algorithm ran five times per array size, and averaged the results for better accuracy.
- Execution time (ms) was recorded.
- A graph was generated to visualize performance trends.

Results:

Here is the table with input sizes and their times (ms) for each sorting algorithm:

Input Size	Bubble Sort	Insertion Sort	Merge Sort	Quick Sort
5	0.0017642974853515625	0.0013828277587890625	0.0037670135498046875	0.0037670135498046875
10	0.0041961669921875	0.0026226043701171875	0.0072002410888671875	0.0049591064453125
25	0.021219253540039062	0.010585784912109375	0.0202178955078125	0.013208389282226562
50	0.07791519165039062	0.03376007080078125	0.045013427734375	0.02803802490234375
100	0.31981468200683594	0.14128684997558594	0.10399818420410156	0.06704330444335938
250	2.1622180938720703	0.8920192718505859	0.2994537353515625	0.20399093627929688
500	9.08207893371582	3.8093090057373047	0.6590366363525391	0.4630088806152344
1000	40.67840576171875	16.783857345581055	1.4552593231201172	1.0796070098876953
5000	1140.2814865112305	453.0129909515381	9.174108505249023	6.669187545776367
10000	4648.5230922698975	1839.8037910461426	19.75226402282715	15.404319763183594

Here is the graph:



Findings:

- **Bubble sort** performs extremely well for smaller input sizes (<50) but became really slow for larger inputs due to $O(n^2)$ complexity and was the slowest overall due to all unnecessary swaps.
- **Insertion sort** also performs extremely well for small input sizes (<50) but still becomes really slow for larger inputs yet slightly faster than the bubble sort for all input sizes.
- **Merge sort** consistently performed well for both smaller and larger input sizes. It performed slightly slower than all other algorithms for input sizes smaller than 10 but stayed consisted due to O(n log n) complexity.

- **Quick sort** was the fastest overall and stayed consisted however (for example compared to insertion sort at input size 25) its performance degraded due to poor pivot selection.

Conclusion:

For small inputs, insertion sort is efficient and slightly fastest compared to other sorting algorithms. For large inputs, merge sort is stable, but quick sort is the fastest in most cases. The experimental results confirm the expected theoretical complexities.

Code Analysis and Explanation:

Main.py:

```python
import time
import random
import matplotlib.pyplot as plt
import pandas as pd
from sorting_algorithms import bubble_sort, insertion_sort, merge_sort, quick_sort

```

- Import time to measure the execution times.
- Import random to generate random arrays of integers.
- Import matplotlib.pylot to plot the execution times.
- Import pandas to plot the table of the sizes and execution times.
- Importing algorithms from my second file to execute in my main.

```python
def generate_random_array(size, min_val=1, max_val=1000):
    return [random.randint(min_val, max_val) for _ in range(size)]
```

- This function creates an array of size random integers between min_val and max_val and generate different arrays of sizes 5, 10, 25, 50, ... 10000 to test how sorting times scale.

```python
def measure_time(sort_function, arr, runs=5):
    total_time = 0
    for _ in range(runs):
        arr_copy = arr.copy()
        start_time = time.time()
        sort_function(arr_copy)
        end_time = time.time()
        total_time += (end_time - start_time) * 1000
    return total_time / runs
```

- Runs each sorting algorithm 5 times.
- Uses arr.copy() so the original array remains unchanged.
- Measures time and converts it to milliseconds.
- Returns the average sorting time.

```python
if __name__ == "__main__":
    sizes = [5, 10, 25, 50, 100, 250, 500, 1000, 5000, 10000]
    algorithms = {
        "Bubble Sort": bubble_sort,
        "Insertion Sort": insertion_sort,
        "Merge Sort": merge_sort,
        "Quick Sort": quick_sort
    }
```

- Defines different input sizes to test sorting algorithms.
- Creates a dictionary linking algorithm names to the functions.

```python
    times = {name: [] for name in algorithms}

    print("\n=== Sorting Algorithm Performance (ms) ===")
    print(f"{'Size':<10}{'Bubble Sort':<15}{'Insertion Sort':<15}{'Merge Sort':<15}{'Quick Sort':<15}")
    print("=" * 65)

    for size in sizes:
        arr = generate_random_array(size)
        row = f"{size:<10}"

        for name, func in algorithms.items():
            time_taken = measure_time(func, arr)
            times[name].append(time_taken)
            row += f"{time_taken:<15.2f}"

        print(row)
```

- Creating a dictionary to hold the execution times.
- Generates a random array for the given size and measures the execution times.
- Stores them in a formatted table that is shown on the terminal when printed.

```python
    df = pd.DataFrame(times, index=sizes)
    df.index.name = "Input Size"
    df.to_csv("sorting_results.csv")

    for name, time_list in times.items():
        plt.plot(sizes, time_list, label=name)

    plt.xlabel("Input Size")
    plt.ylabel("Execution Time (ms)")
    plt.title("Sorting Algorithm Performance")
    plt.legend()
    plt.grid(True)
    plt.savefig("sorting_performance.png")
    plt.show()
```

- Creates a cvs file and saves the execution time table.
- Creates a graph and saves the file.

Sorting_algorithms.py:

```python
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        swapped = False
        for j in range(0, 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
    return arr
```

- Swaps neighboring elements constantly if they are in wrong order.
- Break stops the function if there are no swaps happening to optimize the code.

```python
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
    return arr
```

- The function picks an element from the array and places it to its correct place.

```python
def merge_sort(arr):
    if len(arr) > 1:
        mid = len(arr) // 2
        left_half = arr[:mid]
        right_half = arr[mid:]

        merge_sort(left_half)
        merge_sort(right_half)

        i = j = k = 0
        while i < len(left_half) and j < len(right_half):
            if left_half[i] < right_half[j]:
                arr[k] = left_half[i]
                i += 1
            else:
                arr[k] = right_half[j]
                j += 1
            k += 1

        while i < len(left_half):
            arr[k] = left_half[i]
            i += 1
            k += 1

        while j < len(right_half):
            arr[k] = right_half[j]
            j += 1
            k += 1
    return arr
```

- Recursively splits the array into halves, sorts and merges them in sorted order.

```python
def quick_sort(arr, low=0, high=None):
    if high is None:
        high = len(arr) - 1

    def partition(low, high):
        pivot = arr[high]
        i = low - 1
        for j in range(low, high):
            if arr[j] < pivot:
                i += 1
                arr[i], arr[j] = arr[j], arr[i]
        arr[i + 1], arr[high] = arr[high], arr[i + 1]
        return i + 1

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

    return arr
```

- Selects a pivot and partitions the array around it.
- Recursively sorts left and right partitions.