Table of Contents

1.0 Introduction

Understanding and analyzing the efficiency of algorithms is a cornerstone of computer science, enabling developers and researchers to select and optimize solutions for real-world problems. Sorting algorithms, in particular, are among the most studied due to their wide range of applications and varying performance across different types of data. The Sorting Algorithm Comparator project was developed to provide a comprehensive and interactive tool for exploring the execution time and performance characteristics of popular sorting algorithms under various conditions.

Designed with students, educators, and professionals in mind, this project combines theoretical insights with practical application. By incorporating a user-friendly graphical user interface (GUI) developed using Python's Tkinter library, the application allows users to compare sorting algorithms through dynamic visualizations and detailed performance metrics. The GUI simplifies algorithm selection, input configuration, and data visualization, providing an accessible platform for analyzing algorithm behavior and efficiency.

Key features of the application include support for custom data input via CSV or XLSX files, generation of test data for predefined scenarios (best case, worst case, and random data), and measurement of execution time across varying input sizes. Users can analyze trends, compare results, and understand the relationship between algorithm execution time and their asymptotic growth functions through CSV outputs and graphical plots.

This report documents the design, implementation, and functionality of the application, covering the algorithms it supports, the GUI architecture, and the challenges addressed during development. It also connects the theoretical foundations of sorting algorithms and their growth functions to practical insights derived from the application's outputs, making it a valuable resource for learning and algorithm analysis.

2.0 Implementing sorting algorithms

```
n = 1000
step = 10
results = []
```

The approach of our code is to create multiple arrays of different sizes for each algorithm to test and visualize the growth of each algorithm relative to its array size, which is why we create a few global variables, n refers to the maximum size of the arrays used, step is the difference between the size of each array created, so for n equals 10 and step equals 2, we would create arrays of sizes [1, 3, 5, 7, 9] and for each array we measure its time taken for the chosen sorting algorithm. These values can be changed through our user interface.

We have a few key functions with the purpose of creating the arrays in the best case, worst case, and random case. An additional function was added specifically for the quicksort algorithm, where the worst case for the quicksort algorithm is when the array is already sorted. So, a new function tailored to quicksort was added where the best case for the quicksort algorithm is when each element in the partition routine is to be placed exactly at the middle of its corresponding subarray, we approached this issue using recursion.

```
def sortedRange(n):
    return list(range(1, n + 1))

def reverseSortedRange(n):
    return list(range(n, 0, -1))

def random_arr(n):
    arr = list(range(1, n + 1))
    random.shuffle(arr)
    return arr

def generate_best_case_array(n):

    def build_best_case(low, high):
        if low > high:
            return []
        mid = (low + high) // 2
        return build_best_case(low, mid - 1) + build_best_case(mid + 1, high) + [mid]

    return build_best_case(1, n)
```

A simple function for generating a CSV file was implemented to contain the data collected for the array size and its corresponding time taken for each algorithm.

```
def generate_csv(results, filename):
    with open(filename, mode='w', newline='') as file:
        writer = csv.writer(file)
        writer.writerow(["n", "t"])
        writer.writerows(results)
    print(f"CSV file generated successfully: {filename}")
```

Different sorting algorithms were implemented to be fully functional:

Insertion Sort is a simple algorithm that sorts an array incrementally by picking elements one by one and inserting them into the correct position, with a time complexity of $O(n^2)$.

Merge Sort is a recursive algorithm that divides the array into halves, sorts them, and merges them, running in $O(nlog(n))$ time.

Heap Sort organizes the array into a heap structure and repeatedly extracts the largest element, sorting in $O(nlog(n))$ time without extra space.

Bubble Sort repeatedly swaps adjacent elements if they are out of order, with a time complexity of $O(n^2)$ and efficiency only for small arrays.

Quick Sort partitions the array around a pivot, sorting each part recursively with an average time complexity of $O(nlog(n))$.

Counting Sort, a non-comparison algorithm, counts occurrences of elements to sort integers in $O(n+k)$, where k is the range of values.

Selection Sort finds the smallest element in the unsorted part of the array and moves it to the sorted part, operating in $O(n^2)$ time.

Shell Sort improves insertion sort by comparing and swapping elements across larger gaps, with time complexity depending on the gap sequence, typically around $O(n^1.5)$.

Radix Sort sorts numbers digit by digit from least to most significant using the stable sorting algorithm counting sort, running in $O(n(n+k))$, where k is the number of digits.

Bucket Sort distributes elements into buckets, sorts each bucket, and combines them, working best with uniformly distributed data in $O(n+k)$ time.

Each of these algorithms were tested extensively to ensure correct functionality.

```
> def insertion_sort(arr): ···

> def merge_sort(arr): ···

> def heap_sort(arr): ···

> def bubble_sort(arr): ···

> def quick_sort(arr): ···

> def counting_sort(arr): ···

> def selection_sort(arr): ···

> def shell_sort(arr): ···

> def radix_sort(arr): ···

> def counting_sort_by_digit(arr, exp): ···

> def bucket_sort(arr): ···

# Array of sorting functions (without helper functions)
sorting_functions = [
    insertion_sort,
    merge_sort,
    heap_sort,
    bubble_sort,
    quick_sort,
    counting_sort,
    selection_sort,
    shell_sort,
    radix_sort,
    bucket_sort
]
```

The following lists include all the options the user can choose from the GUI.

data_options represents the type of data entered in the array, whether is it randomly generated data or already sorted data or others.

growth_functions is a specific functionality added where a user can, instead of comparing 2 sorting algorithms together, the user can instead choose to compare a sorting algorithm with its expected growth rate.

```
data_options = ["Custom Data", "Best Case", "Worst Case", "Random Data"]

growth_functions = ["n", "nlogn", "n^2", "2^n"]
```

3.0 Computing Execution Time

The function calculateTime is the heart of our code that ties everything together. Lets break it down into smaller components to explain what it does.

```
def calculateTime(function, index):
    global n
    global step
    n_values = list(range(1, n+1, step))  # Different array sizes
    global results
    results = []
    for n in n_values:
        data_option = data_options[index]

        if data_option == "Custom Data":
            continue  # Handle custom data later

        # Simplified data generation
        is_quicksort = (function == quick_sort)

        if data_option == "Best Case":
            arr = sortedRange(n) if not is_quicksort else generate_best_case_array(n)
        elif data_option == "Worst Case":
            arr = reverseSortedRange(n) if not is_quicksort else sortedRange(n)
        else:  # data_option == "Random Data"
            arr = random_arr(n)

        # Execute the sorting function (assumed to return the execution time)


        exectime = function(arr.copy())

        results.append((n, exectime))

    # Generate CSV file
    filename = f"{function.__name__}_{data_option.replace(' ', '_')}.csv"
    generate_csv(results, filename)
```

The first part initialized everything used, the function accepts 2 parameters, function, which is the name for the sorting algorithm chosen by the user, and an index, which refers to the type of data in the array to be sorted, whether the data is random or reversely sorted or custom data entered by the user.

Variables n and step and results are reinitialized here to accept the changes in array size and step through the GUI. Results is an empty array that stores a pair of data, n and t, those are the array size and number of operations in the sorting algorithm for its corresponding array size.

Also, a list containing all the array sizes n_values is added.

```
def calculateTime(function, index):
    global n
    global step
    n_values = list(range(1, n+1, step))  # Different array sizes
    global results
    results = []
```

The next loop iterates on each number in the array containing the array sizes n_values, and for each array size n, depending on the chosen data_option (which corresponds to the type of test data entered), an array 'arr' is created and filled with either sorted values, reverse sorted values, random values, or custom added values.

A special condition is added when choosing best and worst cases for quicksort algorithm, where an already sorted array is considered as the worst case and a special function for best case for quicksort is added.

```
for n in n_values:
    data_option = data_options[index]

    if data_option == "Custom Data":
        continue  # Handle custom data later

    # Simplified data generation
    is_quicksort = (function == quick_sort)

    if data_option == "Best Case":
        arr = sortedRange(n) if not is_quicksort else generate_best_case_array(n)
    elif data_option == "Worst Case":
        arr = reverseSortedRange(n) if not is_quicksort else sortedRange(n)
    else:  # data_option == "Random Data"
        arr = random_arr(n)

    # Execute the sorting function (assumed to return the execution time)


    exectime = function(arr.copy())

    results.append((n, exectime))
```

After the array is filled with values, the execution time 'exectime' calls the sorting algorithm function based on the variable 'function' which stores the name of the algorithm to be used.

Finally, the computed value for execution time and its corresponding array size are appended to the array 'results'.

A final step is taken to save the results into a CSV file to be later graphed for clear visualization of the chosen sorting algorithms and their different array sizes and their corresponding computed time for execution.

```
# Generate CSV file
filename = f"{function.__name__}_{data_option.replace(' ', '_')}.csv"
generate_csv(results, filename)
```

4.0 Comparison to Asymptotic Growth

This comparison is done using a function that generates growth values for a specific growth function (n, nlogn, n^2, 2^n) and uses scaling factors to determine the upper and lower bound of the sorting algorithm's time complexity.

```
def createGrowthCSV(growth_index):
    global n,step,results
    values = []
    growth_function = growth_functions[growth_index]

    # Create values based on the selected growth function
    for i in range(1, n+1, step):
        if growth_function == "n":
            values.append([i, i])
        elif growth_function == "nlogn":
            values.append([i, i * math.log(i, 2)])
        elif growth_function == "n^2":
            values.append([i, i**2])
        elif growth_function == "2^n":
            values.append([i, 2**i])

    growth_values = np.array(values)
    results_array = np.array(results)
```

The function takes the growth function as a parameter and computes a list of values.

After computing the growth values, the function compares them with experimental results stored in the `results` array. These results are transformed into NumPy arrays for numerical operations. A critical validation step ensures that the lengths of `results` and `growth_values` match, raising a `ValueError` if they do not.

The function then calculates scaling factors by dividing the experimental results by the corresponding growth values. Since the first data point is just the n, it may not represent asymptotic behavior, the scaling factors exclude it by considering only the subsequent points.

To focus on the asymptotic behavior, the function uses the last 80% of the scaling factors (starting at 20% of the total length). We found that starting at 20% gives the most optimal results as taking the whole list will result in a really wide difference between the upper and lower bound. If it was set to more than 20%, the values before the percent may lie outside the bounds and so more values will lie outside the bounds if the percentage is increased. It determines the minimum (`c1`) and maximum (`c2`) scaling factors within this range. These constants represent the lower and upper bounds of the growth function for the given data.

```
453     # Calculate scaling factors
454     scaling_factors = results_array[1:] / growth_values[1:]
455     scaling_factors = scaling_factors[:, 1]
456
457     # Focus on large n values to determine the scaling factor (asymptotic behavior)
458     critical_n_start = int(len(scaling_factors) * 0.2)  # Start from the middle of the values
459     large_n_scaling_factors = scaling_factors[critical_n_start:]
```

The next step involves using `c1` and `c2` to calculate the lower and upper bounds for each input size. These bounds are saved in CSV files named according to the selected growth function, suffixed with `_lower.csv` and `_upper.csv`.

Finally, the function prints the values of `c1` and `c2` for verification and returns these constants to the caller.

```
457     # Focus on large n values to determine the scaling factor (asymptotic behavior)
458     critical_n_start = int(len(scaling_factors) * 0.2)  # Start from the middle of the values
459     large_n_scaling_factors = scaling_factors[critical_n_start:]
460
461     # Find the minimum and maximum scaling factors (c1 and c2) over large n range
462     c1 = np.min(large_n_scaling_factors)  # Lower bound constant
463     c2 = np.max(large_n_scaling_factors)  # Upper bound constant
464
465     # Calculate lower and upper bounds for f(n) using c1 and c2
466     lower_values = [[i, value * c1] for i, value in values]
467     upper_values = [[i, value * c2] for i, value in values]
468
469     # Generate CSV files for the lower and upper bounds
470     generate_csv(lower_values, f"growth_{growth_function}_lower.csv")
471     generate_csv(upper_values, f"growth_{growth_function}_upper.csv")
```

This function is useful in analyzing the performance of algorithms by comparing their empirical results to theoretical growth functions, helping to estimate their asymptotic complexity accurately.

5.0 Graphical User Interface (GUI)

5.1 Overview

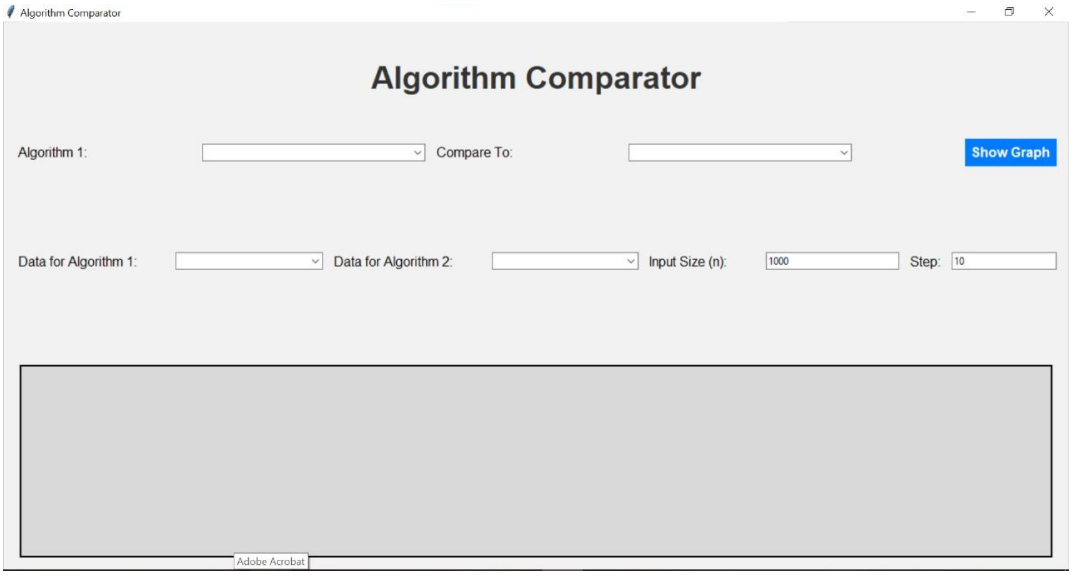
The GUI (Graphical User Interface) is a crucial part of the Sorting Algorithm Comparator project, designed to provide an intuitive and interactive way for users to

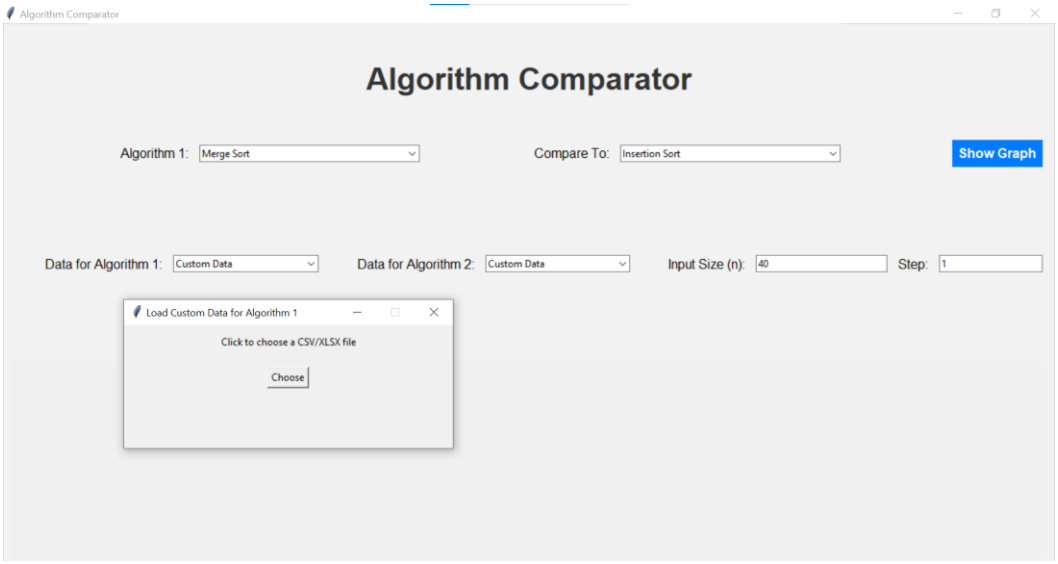
compare the performance of different sorting algorithms. The GUI simplifies the process of selecting algorithms, customizing input data, and visualizing the results, making the application accessible and user-friendly.

In this project, the GUI allows users to:

- **Select two sorting algorithms to compare.**
- **Define the type of input data (e.g., random, sorted, reversed).**
- **Set parameters such as input size and step size.**
- **Display performance metrics as graphical plots.**

This interface ensures seamless interaction with the backend logic while maintaining clarity and usability for users.





5.2 Design and Layout

Main Window

The main window is divided into the following sections:

- **Algorithm Selection:**
  - **Dropdown menus for selecting the primary and comparison sorting algorithms.**
- **Data Configuration:**
  - **Options to choose the type of data (e.g., random, sorted).**
  - **Input fields for specifying input size (n) and step size.**
- **Control Panel:**
  - **A button to generate the comparison graph.**
- **Graph Display Area:**
  - **A plot area where the performance of the selected algorithms is visualized as a line graph.**
- **Custom Data File Path Chooser**
  - **A window that prompts the user to choose the csv file for custom data input, pops up when the user's chosen data option is Custom Data.**

5.3 Features

1. **Algorithm Comparison:**
   a. **Users can compare the performance of two algorithms based on metrics like the number of steps or execution time.**
2. **Customizable Input:**
   a. **Provides flexibility to test algorithms under different conditions by varying generated input data type, size, and step intervals, and by allowing the user to choose their own data input to be sorted.**
3. **Visualization:**
   a. **Graphical representation of algorithm performance to highlight differences in efficiency.**

4. **Responsive Design:**
    a. **Ensures that all controls and visualizations are neatly arranged and easy to navigate.**

5.4 Technical Implementation

The GUI is implemented using Python's `Tkinter` library, with `Matplotlib` integrated for data visualization. Below is an outline of the main components:

5.4.1 Main Components

- **Algorithm Selection Panel:**
    - **Contains `OptionMenu` widgets to allow users to select algorithms from a predefined list.**
- **Input Configuration Panel:**
    - **Includes input fields (`Entry` widgets) for the user to specify the size of the input and the step size.**
- **Graph Display:**
    - **Uses `Matplotlib`'s `FigureCanvasTkAgg` to embed dynamic graphs within the GUI.**
- **Custom Data File Path Chooser**
    - **Popup window to allow user to choose a custom dataset.**

5.4.2 Workflow

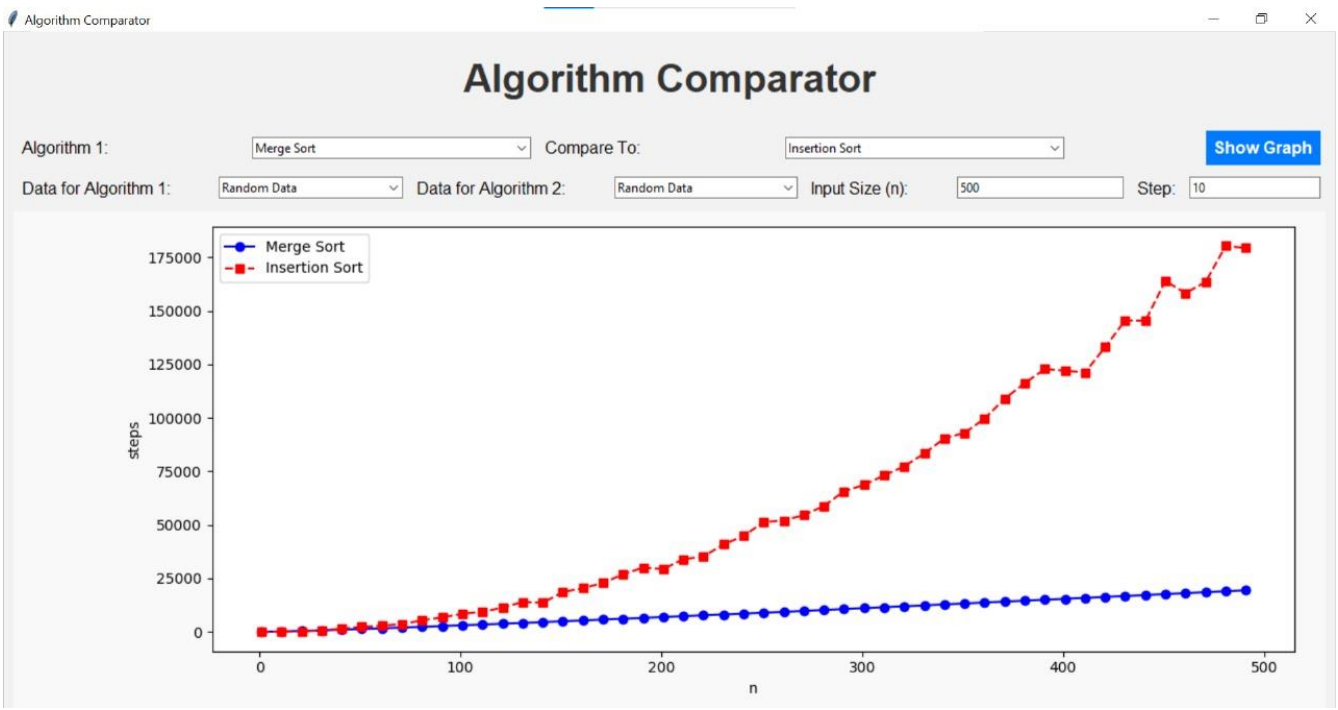
1. **The user selects the sorting algorithms and input parameters.**
2. **Upon clicking "Show Graph," the backend processes the inputs and generates performance data.**
    a. **If user chooses custom data, they are then prompted to input the custom data file path and presses Confirm to submit it.**
3. **The GUI retrieves the data and updates the graph dynamically to display the results.**

5.5 Graphical Snapshots

Merge Sort vs Insertion Sort Comparison:

Graph Setup:

- **X-axis: Input size (n).**
- **Y-axis: steps**
- **Lines:**
  - **Merge Sort: O(n log n).**
  - **Insertion Sort: O(n^2).For small inputs, Insertion Sort might perform better, but Merge Sort scales better for larger datasets.**



Quick Sort vs Merge Sort Comparison:

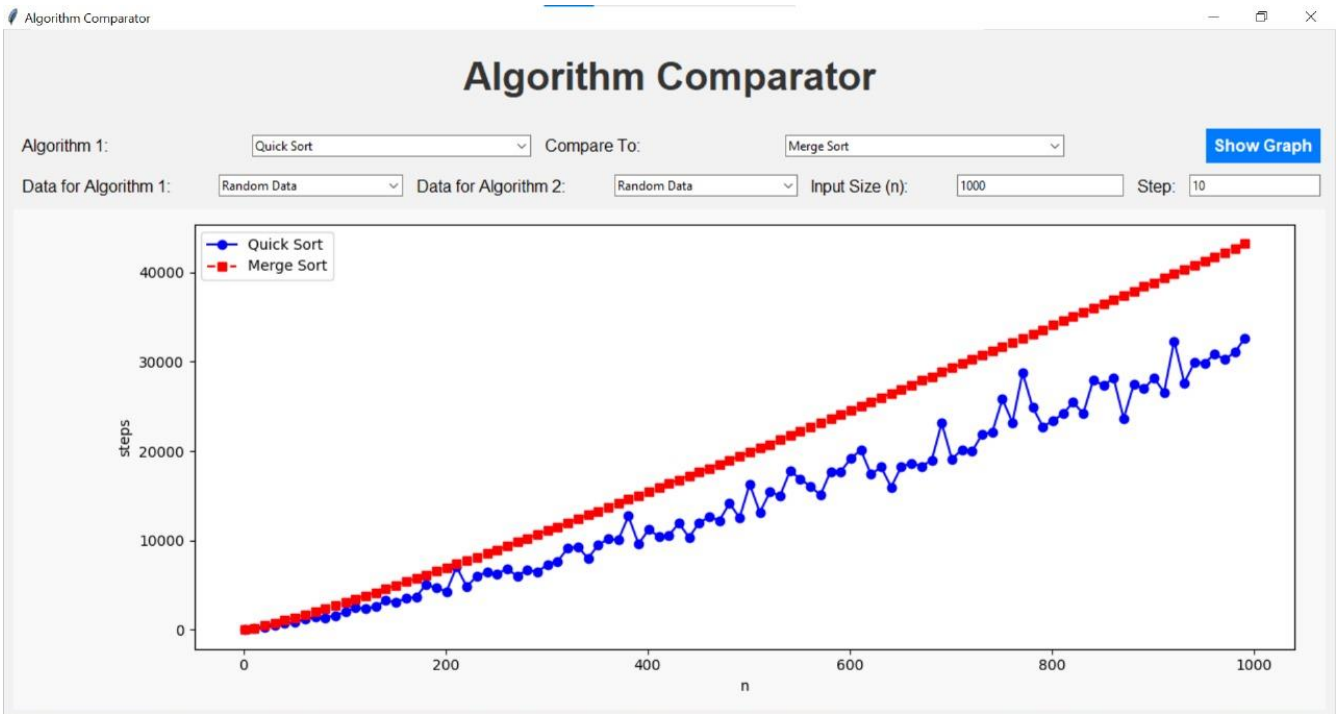
Graph Setup:

- **X-axis: Input size (n).**
- **Y-axis: Steps (number of operations).**

Lines:

- **Quick Sort: Average O(n log n), Worst O(n^2).**
- **Merge Sort: O(n log n).**

For small datasets, Quick Sort may perform better due to its in-place sorting. However, Merge Sort provides consistent O(n log n) performance and is more predictable, especially for larger datasets.



Counting Sort vs Asymptotic Growth of n Comparison:

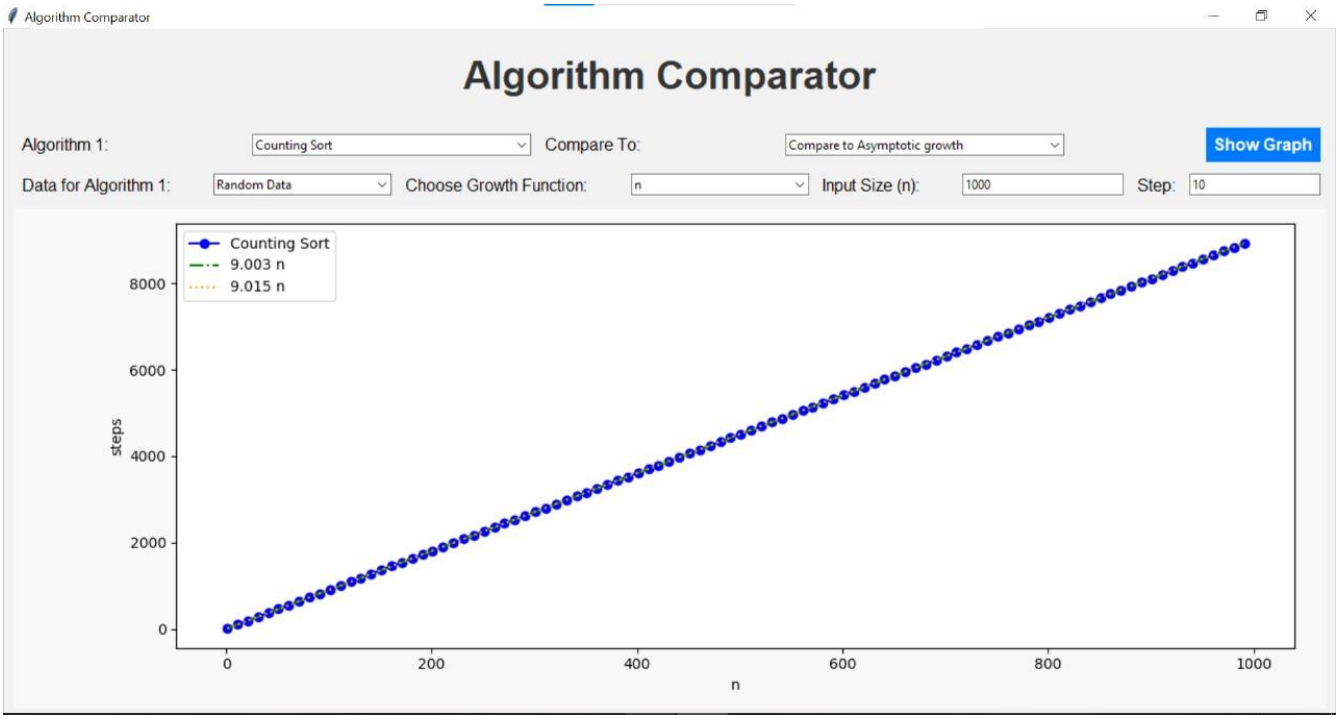
Graph Setup:

- **X-axis: Input size (n).**
- **Y-axis: Steps (number of operations).**

Lines:

- **Counting Sort: O(n + k), where k is the range of the input.**
- **Asymptotic Growth of n: O(n) (linear growth).**

Counting Sort has linear time complexity O(n) when the range of input values (k) is small, making it efficient for sorting small integers. However, its performance degrades when k is large relative to n. In contrast, O(n) asymptotic growth is used to describe algorithms with linear performance, regardless of constant factors like the range of values.



Bubble Sort vs Asymptotic O(n^2) Comparison:

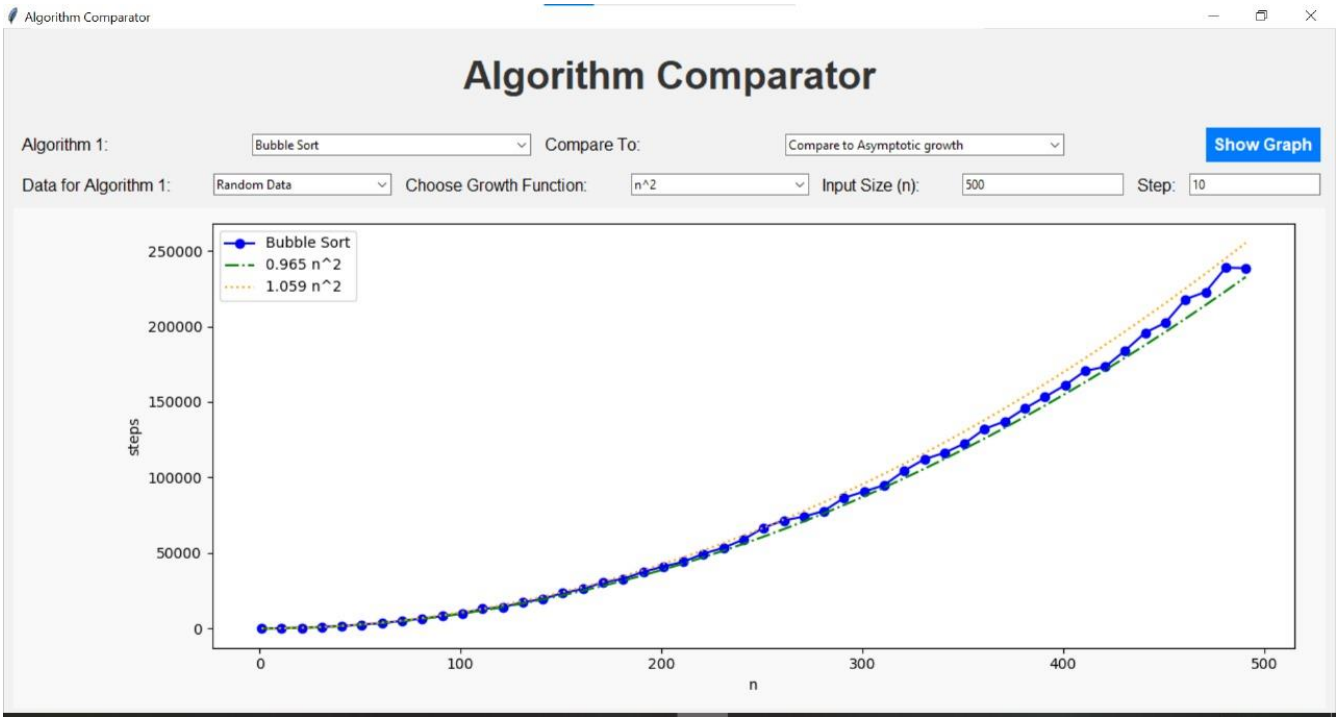
Graph Setup:

- **X-axis: Input size (n).**
- **Y-axis: Steps (number of operations).**

Lines:

- **Bubble Sort: O(n^2) (worst and average case).**
- **Asymptotic O(n^2): Represents the growth of algorithms with quadratic time complexity.**

Bubble Sort has a time complexity of O(n^2) in both the worst and average cases due to its nested loops. It performs poorly on large datasets. The O(n^2) asymptotic growth represents algorithms that exhibit quadratic growth, where the number of steps increases rapidly as the input size grows.



Quick Sort vs Asymptotic O(n log n) Comparison:

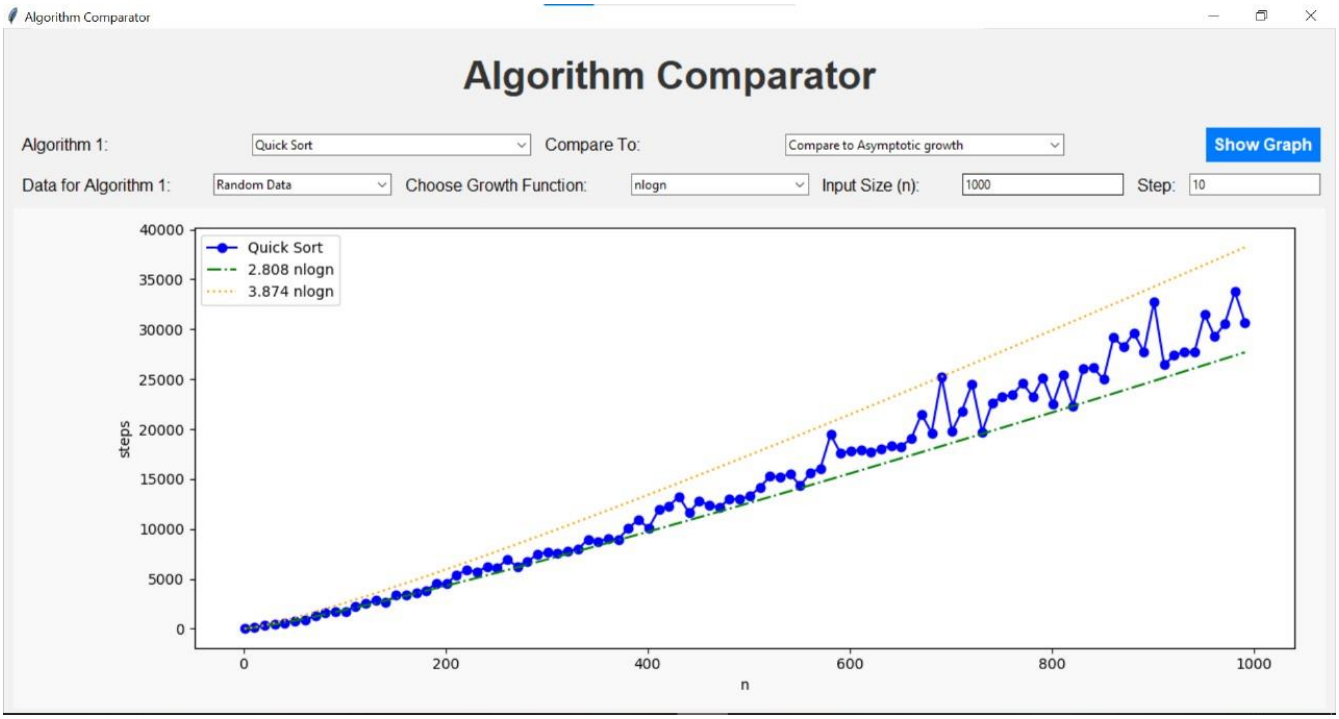
Graph Setup:

- **X-axis: Input size (n).**
- **Y-axis: Steps (number of operations).**

Lines:

- **Quick Sort: O(n log n) (average case), $O(n^2)$ (worst case).**
- **Asymptotic O(n log n): Represents the growth of algorithms with O(n log n) time complexity.**

Quick Sort has an average-case time complexity of O(n log n), similar to algorithms like Merge Sort. However, its worst-case complexity can degrade to $O(n^2)$ if a poor pivot is chosen. The O(n log n) growth represents algorithms that perform efficiently and are typically used in divide-and-conquer algorithms like Merge Sort.



Merge Sort vs Insertion Sort Comparison on Small Custom Dataset:

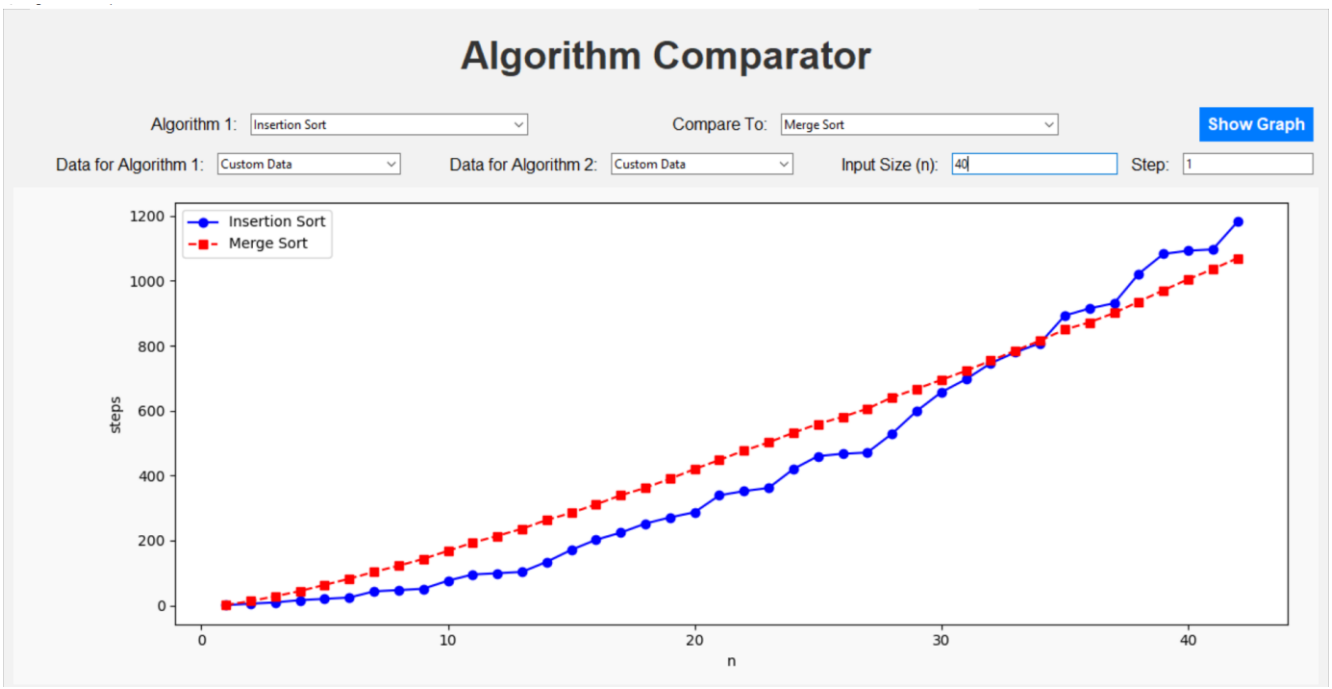
Graph Setup:

- **X-axis: Input size (n).**
- **Y-axis: Steps (number of operations).**

Lines:

- **Insertion Sort: O(n) (best case), $O(n^2)$ (worst/average case).**
- **Merge Sort: O(n log n) all cases**

For a much smaller dataset, insertion sort outperforms merge sort. Nonetheless, due to the exponentially growing insertion sort with growth $O(n^2)$, as the dataset gets larger, the time required to sort it grows exponentially, making merge sort outperform in the long run.



6.0 Conclusion:

In this project, we successfully implemented and analyzed a variety of sorting algorithms, including but not limited to insertion sort, merge sort, heap sort, quick sort, and counting sort. By utilizing a graphical user interface (GUI), we made these algorithms accessible for testing under different conditions, such as best-case, worst-case, and random data scenarios.

The integration of features like customizable array sizes, step increments, and data configurations allowed for a comprehensive evaluation of algorithmic performance. A notable addition was the capability to compare sorting algorithms against their theoretical growth rates, providing deeper insights into their asymptotic behavior.

Furthermore, the results were systematically saved into CSV files, facilitating visual representation and comparison. This approach highlighted their efficiency and scalability.

The project underscores the importance of understanding algorithmic complexities and serves as a robust framework for further exploration into advanced data structures and algorithms.

7.0 References

1. **T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. Cambridge, MA: MIT Press, 2009.**
2. **Matplotlib Documentation. Accessed: Dec. 28, 2024. [Online]. Available: https://matplotlib.org/stable/contents.html**
3. **Tkinter Documentation. Accessed: Dec. 28, 2024. [Online]. Available: https://docs.python.org/3/library/tkinter.html**