Project Report of Sorting Algorithm GUI Implementation

Aman Hogan-Bailey

The University of Texas at Arlington

Table of Contents

Abstract

Sorting is a fundamental operation in computer science, and there are many sorting algorithms with different characteristics and performance. In this project, there has been several sorting algorithms implemented in Python, including merge sort, heap sort, quick sort, selection sort, insertion sort, and bubble sort. There has also been created a Graphical User Interface (GUI) that allows the user to choose an algorithm and sort an array of integers. Additionally, there is an `AlgorithmData` class to store data related to each algorithm run, such as the size of the input, start and end times, and the sorted array. Finally, there is a module to display the performance of each algorithm graphically, based on the size of the input array and the time taken to execute the algorithm. In this report, the main components of this project, including the sorting algorithms implemented in Python, the algorithm data structures, the GUI, and the graphing module will be discussed, along with their performance. Overall, this project aims to provide an easy-to-use platform for exploring different sorting algorithms and comparing their performance.

Main Data Structures

`Algo_main.py`

- A `list` used to store the input and output arrays for the sorting algorithms.
- The `class AlgorithmData` is a custom class used to store data related to each algorithm run, such as the size of the input, start and end times, and the sorted array.
- The `global variables`: `array`, `algorithm_data`, `n`, `list_of_sorted`, and `unsorted` are global variables used to store and manipulate data across functions. `array` and `unsorted` store the input array, `algorithm_data` stores information about the currently executing algorithm, `n` stores the size of the input array, and `list_of_sorted` stores a list of `AlgorithmData` objects for each algorithm that has been run.

`Sorting_algirthms.py`

- The main data structure used in the `merge_sort` algorithm is an `array`. The algorithm uses the concept of dividing the array into two halves, and then recursively sorting each half before merging the sorted halves together.
- In the `heap_sort` algorithm, a `binary heap` data structure is used to keep track of the largest element in the array. The algorithm builds a max heap from the given array and then repeatedly extracts the maximum element from the heap and places it at the end of the array.
- The `quick_sort` algorithm uses recursion to sort the `array`. The algorithm selects a pivot element and partitions the array into sub-lists containing elements less than and

greater than the pivot. The sub-lists are recursively sorted and concatenated together with the pivot element.

- The `quick_sort_median` algorithm is like `quick_sort`, but it selects the pivot element as the median of the first, middle, and last elements of the `array`.
- The `selection_sort, insertion_sort`, and `bubble_sort` algorithms all operate on an array and use simple looping constructs to sort the `array`. They do not use any complex data structures.

`Algo_gui.py`

- A `list` is used to store the unsorted array and to represent the order of the buttons on the GUI.

`Algo_graph.py`

- Uses a `list` called `graph_names` to store the names of the sorting algorithms used.
- Uses a `list` called `x` array sizes.
- Uses a `list` called `y` of runtimes of arrays.

Main Data Components

`algo_sorting.py`

- `merge_sort(arr, data)`: This function implements the merge sort algorithm. It recursively partitions the input array into two halves until it reaches a base case of an array with a single element. Then it merges the two sorted halves back into the original array. The function also takes a data parameter to log information about the algorithm.
- `heap_sort(arr, data)`: This function implements the heap sort algorithm. It first builds a max heap from the input array. Then it extracts elements one by one from the heap and places them at the end of the array. The function also takes a data parameter to log information about the algorithm.
- `heapify(arr, n, i)`: This is a helper function for the `heap_sort` function. It takes an array, its length, and an index i and recursively heapifies the sub-tree rooted at i.
- `quick_sort(arr)`: This function implements the quick sort algorithm using the first element as the pivot. It recursively partitions the array into two sub-arrays around the pivot until the base case of an array with a single element is reached. Then it concatenates the sorted sub-arrays and the pivot back into the original array.
- `quick_sort_median(arr)`: This function implements the quick sort algorithm using the median of the first, middle, and last elements as the pivot. It partitions the array into three sub-arrays around the pivot, then recursively sorts the left and right sub-arrays before concatenating them with the pivot sub-array.
- `selection_sort(arr, data)`: This function implements the selection sort algorithm. It repeatedly finds the minimum element in the unsorted part of the array and

swaps it with the first element of the unsorted part. The function also takes a data parameter to log information about the algorithm.

- `insertion_sort(arr, data):` This function implements the insertion sort algorithm. It iteratively inserts each element of the array into its correct position in the sorted part of the array. The function also takes a data parameter to log information about the algorithm.
- `bubble_sort(arr, data):` This function implements the bubble sort algorithm. It repeatedly swaps adjacent elements if they are in the wrong order until the array is sorted. The function also takes a data parameter to log information about the algorithm.

`algo_main.py`

- The `choose_algorithm` function takes a string parameter choice that determines which sorting algorithm to run on the global array variable, and updates the global `algorithm_data` variable with the execution time and sorted array. The sorted data is also printed to the console using the `algo_gui.array_to_screen` function.
- The `create_array` function creates a random array of integers of length size, and updates the global array variable with the new array.
- The `compare_algorithms` function compares the results of the two most recent sorting algorithm runs stored in the global `list_of_sorted` list, and prints the results to the console using the `algo_gui.comparison_screen` function.
- The `algo_graphs` function displays a graph of the most recent sorting algorithm run using the `algo_gui.graphing_screen` function.

- The script also imports several modules, including sys, random, time, `algo_sorting`, and `algo_gui`.

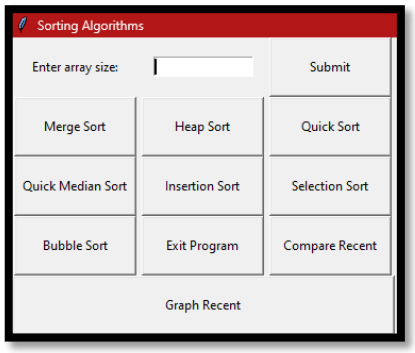
`algo_gui.py`

- The `initialize_gui()` function creates a main window for the GUI, where the user can input the desired size of the array to be sorted and select a sorting algorithm using buttons. The other buttons allow the user to exit the program, compare the performance of the most recent sorts, and graph the performance of the most recent sorts.
- The `onSubmit()` function retrieves the array size input by the user and calls the `create_array()` function from the `algo_main` module, which generates a random array of the specified size.
- The `array_to_screen()` function creates a new window to display the details of the selected sorting algorithm, including the algorithm name, array size, start time, end time, run time, bytes per microsecond, and unsorted array.
- The module imports the `tkinter` module and uses its functions and widgets to create the GUI, as well as several functions from the `algo_main` module and `algo_sorting` module to perform the actual sorting. The time module is also imported to measure the run time of the algorithms.

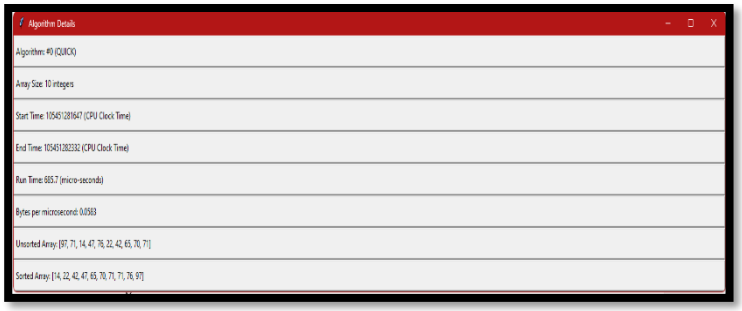
`algo_graph.py`

- The function `graphing (x, y, graph_name)` adds the `graph_name` to the `graph_names` list and prints a message indicating that it is graphing the specified algorithm. Then it uses the `matplotlib` library to create a scatter plot of the input data. The x-axis is labeled with "`Array Size (x) [#]`" and the y-axis is labeled with "`Runtime (y) [microseconds]`". The title of the graph includes the specified algorithm name. Finally, the graph is saved as a file with the algorithm name and displayed on the screen.

GUI Design

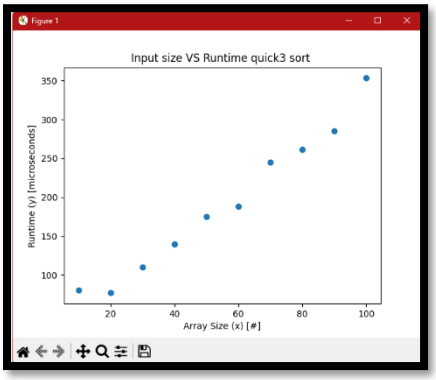


(Figure 1) Sorting Algorithms Window – Consists of 11 buttons and 1 Entry field. Entry field takes in the input size, and the submit creates an array using the size in the input; 7 sorting algorithm buttons, which perform the necessary sorting algorithms; a `Compare Recent` Button; `Exit Program` button; a `Graph Recent` button.



(Figure 2) Algorithm Details Window – Pop up window that logs details of the algorithm that just ran, immediately after sorting has finished.

(Figure 3) Comparison Window – Pop up window that shows the difference in runtimes between two algorithms.



(Figure 4) Graph Window Graph Window – Pop up window that shows a scatter plot using the size of an array and its runtime.

Experimental Results

Running a single Algorithm

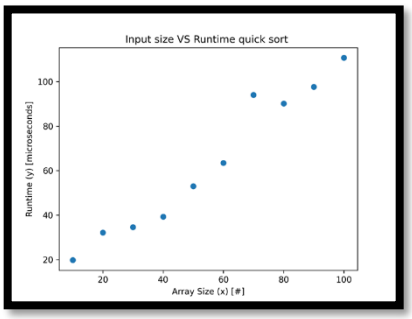
For running selection sort with an array size of `10`, the user entered the array size in the entry field, then clicked the `Submit` button. The user then clicked the `Selection Sort` button and received the following information in the GUI Window:

```
Unsorted Array: [34, 82, 44, 45, 5, 47, 70, 93, 93, 75]
SELECTION SORT
ARRAY SIZE: 10
START TIME: 187574755564.5
END TIME: 187574755781.8
RUN TIME: 217.29998779296875 MICROSECONDS
BYTES PER MICROSECOND: 0.18407732281195413
UNSORTED: [34, 82, 44, 45, 5, 47, 70, 93, 93, 75]
SORTED: [5, 34, 44, 45, 47, 70, 75, 82, 93, 93]
```

(Figure 5)

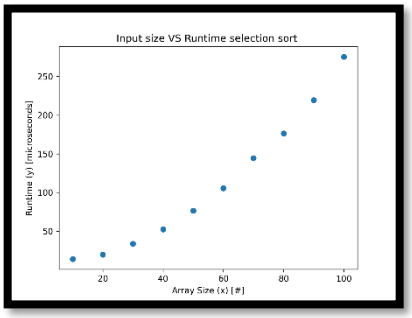
This experiment tested the Selection Sort algorithm on an unsorted array of length `10,` with the elements `[34, 82, 44, 45, 5, 47, 70, 93, 93, 75]`. The start and end times were recorded, and the runtime of the algorithm was calculated to be `217.29998779296875` microseconds. The throughput was calculated to be `0.18407732281195413` bytes per microsecond. Finally, the `unsorted` and `sorted` arrays were printed, showing the input and output of the algorithm.

Input Size VS Running Time



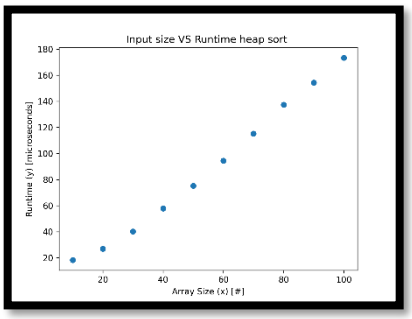
X values: [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
Y values: [19.8, 32.2, 34.6, 39.3, 53.0, 63.5, 94.0, 90.2, 97.6, 110.7]

Quick sort



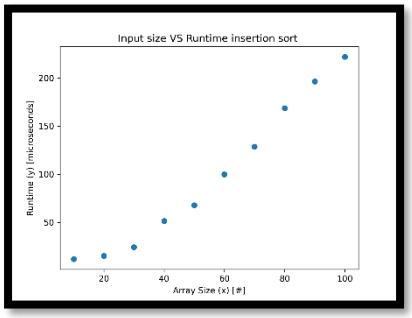
X values: [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
Y values: [13.9, 19.7, 33.8, 52.3, 76.5, 105.5, 144.4, 176.3, 219.5, 275.5]

Selection sort



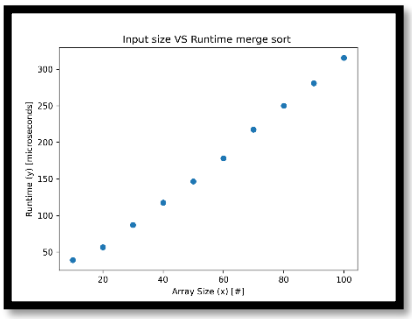
X values: [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
Y values: [18.3, 27.0, 40.2, 57.9, 75.2, 94.5, 115.2, 137.4, 154.3, 173.3]

Heap sort



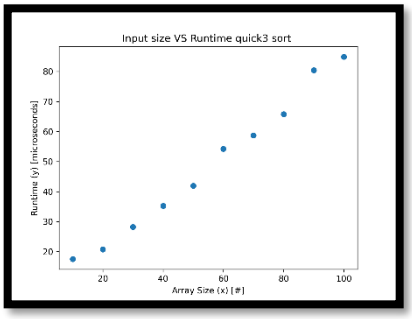
X values: [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
Y values: [11.7, 15.1, 24.1, 51.3, 67.6, 99.8, 128.6, 168.7, 196.4, 222.0]

Insertion sort



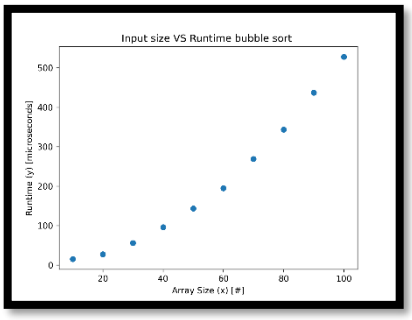

```
X values: [10, 20, 30, 40, 50, 60, 70,
80, 90, 100]
Y values: [38.9, 56.6, 87.1, 117.4,
146.5, 178.2, 217.2, 250.1, 280.8,
315.5]
```

Merge Sort




```
X values: [10, 20, 30, 40, 50, 60, 70,
80, 90, 100]
Y values: [17.5, 20.7, 28.2, 35.2,
41.9, 54.2, 58.7, 65.8, 80.4, 84.9]
```

Quick Median Sort



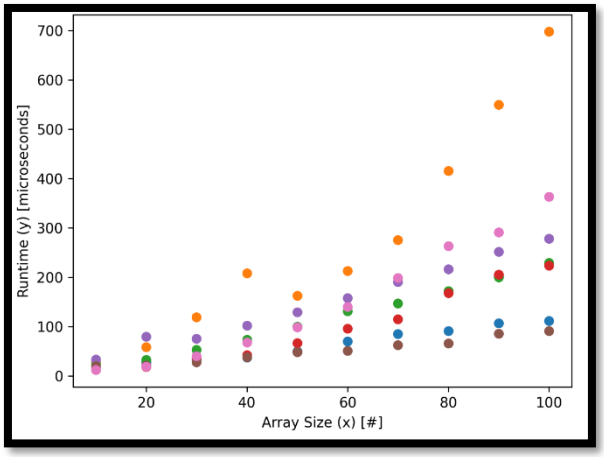

```
X values: [10, 20, 30, 40, 50, 60, 70,
80, 90, 100]
Y values: [15.0, 27.5, 55.8, 96.1,
143.1, 195.0, 269.1, 343.4, 437.0,
528.0]
```

Bubble Sort

(Figure 6) – A series of graphs that show to runtimes of algorithms from size 10 to size 100 in 10 integer increments. To receive a graph of input size Vs runtime, each algorithm must be run first, then the `Graph Recent` *Button must be pressed.*

Runtime For all Algorithms

`Selection sort` has the *worst performance* among all the algorithms with a runtime of around 200 microseconds for input size of 100. `Bubble sort` also has a poor performance with a runtime of around 500 microseconds for input size of 100. `Insertion sort` has a better performance than selection sort and bubble sort, but still has a higher runtime than other algorithms for input size of 100. `Quick sort` and `Quick Median` sort have a *good performance* and their runtimes increase slowly as input size increases. Quick Median sort is slightly faster than Quick sort for input sizes greater than 50. `Heap sort` has a good performance for small input sizes, but its runtime increases rapidly for larger input sizes. `Merge sort` has the worst performance among Quick sort, Quick Median sort, and Merge sort, with its runtime increasing rapidly for larger input sizes.



Legend
Blue: quick sort
Orange: selection sort
Green: Heap sort
Red: Insertion sort
Purple: Merge sort
Brown: Quick median sort
Pink: Bubble sort

(Figure 7) – A graph that plots the input size vs run time of all seven sorting algorithms.

Conclusions

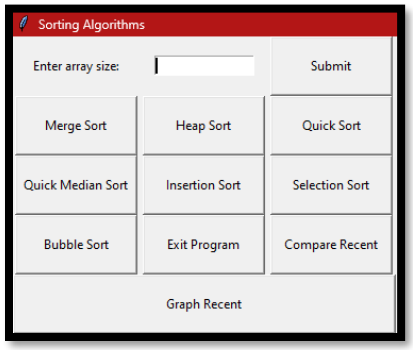
Based on the output, we can see that the running times of the different sorting algorithms vary significantly with respect to data size. For example, insertion sort and bubble sort have very high running times for larger input sizes, while quick sort and heap sort have much lower running times for larger input sizes.

In terms of comparing the speed of the different algorithms, we can see that selection sort and bubble sort are the slowest, while quick sort (regular and median), heap sort, and insertion sort are faster. Merge sort falls somewhere in the middle.

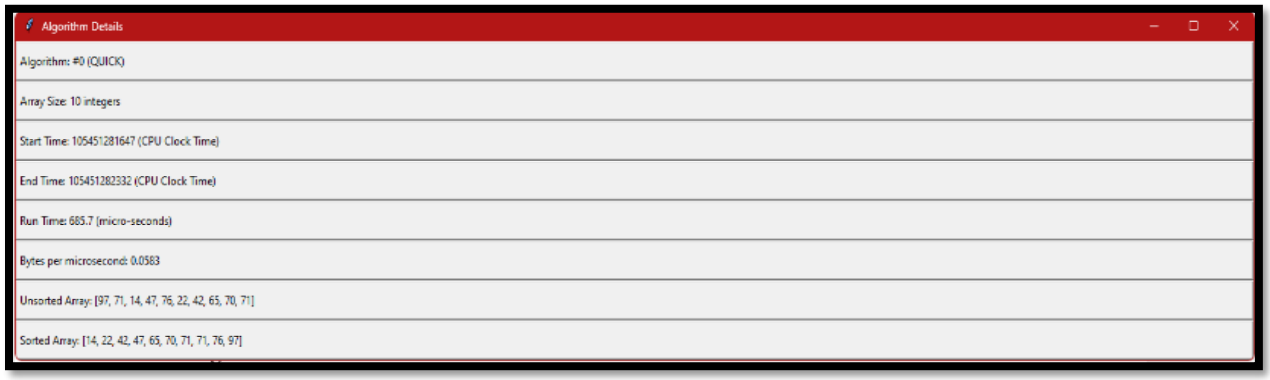
As for improving their running time, it depends on the specific algorithm and the data being sorted. For example, if the data is already partially sorted, insertion sort may be faster than quick sort. Additionally, some sorting algorithms, such as quick sort, can be optimized by choosing a better pivot element, or by using parallelization techniques.

In terms of which algorithm is better, it really depends on the specific use case and data being sorted. For small input sizes, any of the algorithms could be used since their running times are relatively similar. However, for larger input sizes, quick sort (regular or median) or heap sort may be preferable due to their faster running times. But again, this could vary depending on the specific data being sorted and other considerations, such as memory usage.

Appendix

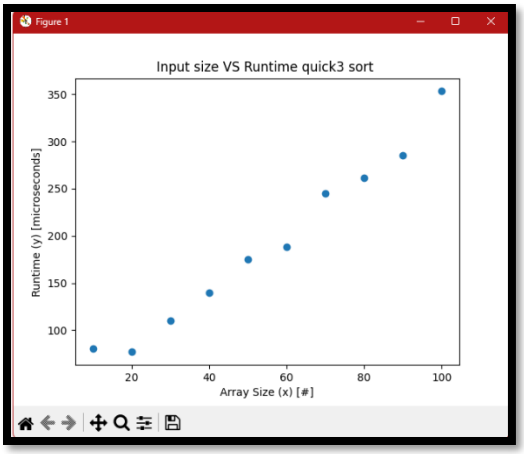


(Figure 1) Sorting Algorithms Window



(Figure 2) Algorithm Details Window

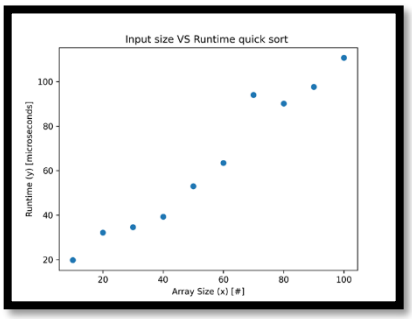
(Figure 3) Comparison Window



(Figure 4) Graph Window Graph Window

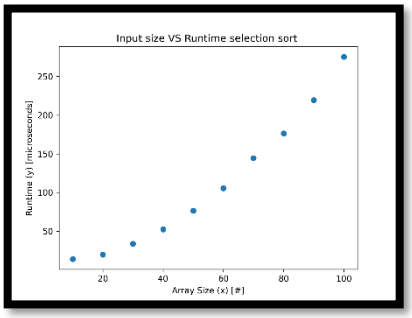
```
Unsorted Array: [34, 82, 44, 45, 5, 47, 70, 93, 93, 75]
SELECTION SORT
ARRAY SIZE: 10
START TIME: 187574755564.5
END TIME: 187574755781.8
RUN TIME: 217.29998779296875 MICROSECONDS
BYTES PER MICROSECOND: 0.18407732281195413
UNSORTED: [34, 82, 44, 45, 5, 47, 70, 93, 93, 75]
SORTED: [5, 34, 44, 45, 47, 70, 75, 82, 93, 93]
```

(Figure 5)



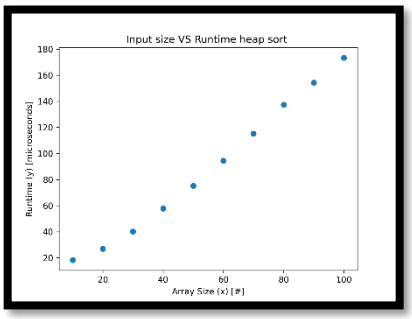
X values: [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
Y values: [19.8, 32.2, 34.6, 39.3, 53.0, 63.5, 94.0, 90.2, 97.6, 110.7]

Quick sort



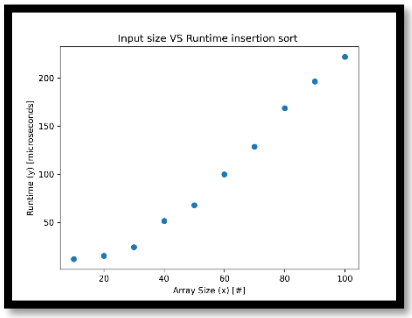
X values: [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
Y values: [13.9, 19.7, 33.8, 52.3, 76.5, 105.5, 144.4, 176.3, 219.5, 275.5]

Selection sort



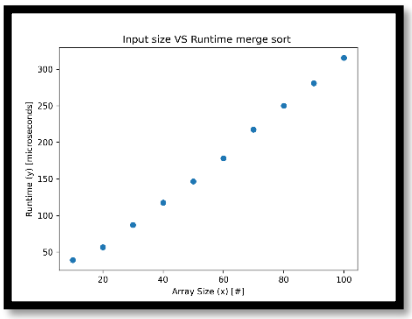
X values: [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
Y values: [18.3, 27.0, 40.2, 57.9, 75.2, 94.5, 115.2, 137.4, 154.3, 173.3]

Heap sort



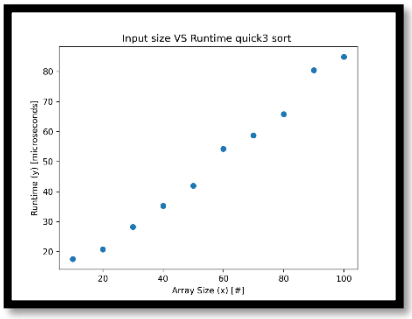
X values: [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
Y values: [11.7, 15.1, 24.1, 51.3, 67.6, 99.8, 128.6, 168.7, 196.4, 222.0]

Insertion sort



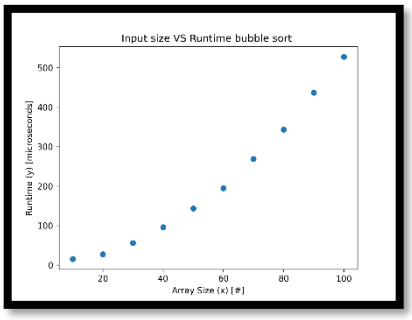
X values: [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
Y values: [38.9, 56.6, 87.1, 117.4, 146.5, 178.2, 217.2, 250.1, 280.8, 315.5]

Merge Sort



X values: [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
Y values: [17.5, 20.7, 28.2, 35.2, 41.9, 54.2, 58.7, 65.8, 80.4, 84.9]

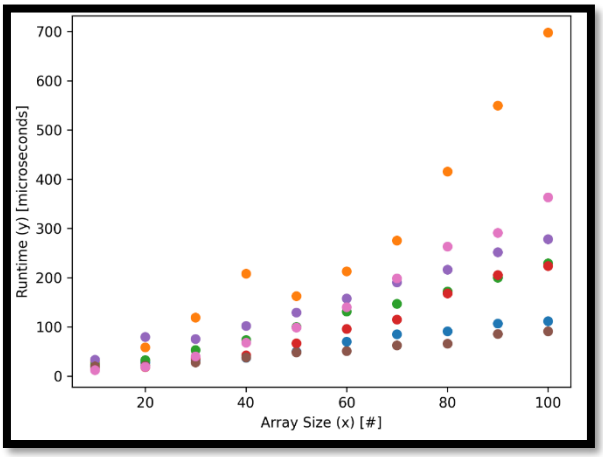
Quick Median Sort



X values: [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
Y values: [15.0, 27.5, 55.8, 96.1, 143.1, 195.0, 269.1, 343.4, 437.0, 528.0]

Bubble Sort

(Figure 6)



Legend
Blue: quick sort
Orange: selection sort
Green: Heap sort
Red: Insertion sort
Purple: Merge sort
Brown: Quick median sort
Pink: Bubble sort

(Figure 7) – A graph that plots the input size vs run time of all seven sorting algorithms.