Project Report

University of Texas Arlington

College of Engineering

Design and Analysis of Algorithm
(DAA 5311-004)
Name: Pruthvik Kakadiya
UTA ID: 1001861545

Instructor – Negin Fraidouni

Aim:

Project #1

Implement and compare the following sorting algorithm:
• Mergesort
• Heapsort
• Quicksort (Regular quick sort* and quick sort using 3 medians)
• Insertion sort
• Selection sort
• Bubble sort

Programming Language:
Python 3,
matplotlib(python library to show graph), Tkinter(python library for GUI)

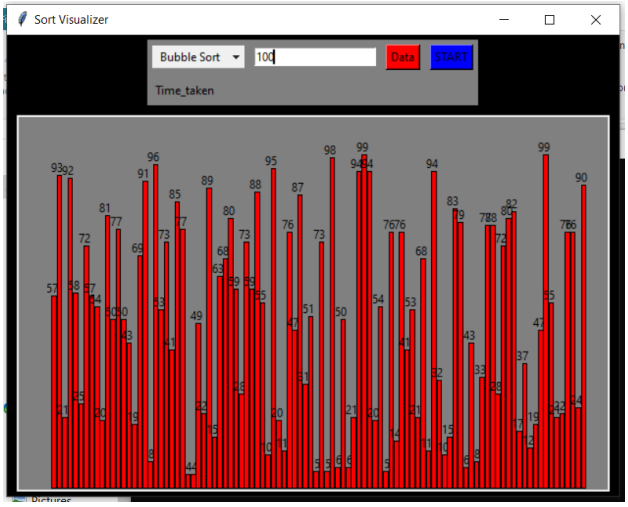
- **Brief:**

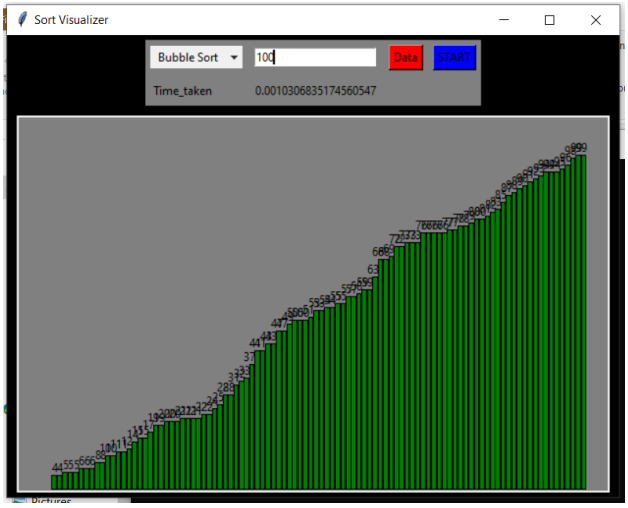
In this project, sorting algorithms(Bubble sort, Selection sort, Insertion sort, Heap sort, Merge sort, Quick sort, Quick sort using three medians) and also method which can be performed to compare run time complexity(in seconds) for 7 different algorithms using array of random integers have been implemented. Also users can choose an option to compare arrays of multiple sizes to compare time complexity of all algorithms in graph formation, and an option to run a selected sorting algorithm for user given data.

```
Choose from following

1: Compare time complexities of all algorithms using user given different Dataset sizes
2: Run sorting algorithm and provide Sorted Dataset using User Given Dataset
3: exit
```

In GUI users can select a sorting algorithm(from a drop down menu) and provide its size(in a text box) to create array of random integers and show it in bar formation of red color(using a button called Data) and click on Start(button) which will show the sorted array in bar formation of green color and run time complexity of that algorithm for given size in seconds.





• Functions:

Project.py :

= To get no of different dataset sizes and create different arrays of random integers:

get_dataset_sizes():	eg: 10, 20, 30 etc
dataset_of_size(n):	eg: n=10 : [0,5,1,…..,10]
datasets(dataset_sizes):	eg: dataset_sizes = [10, 20, 30] { Dataset 10 : [0,5,1,…..,10], Dataset 20 : [0,5,1,…..,20], Dataset 30 : [0,5,1,…..,30] }

=sorting algorithms will run defined sort on array of random integer

• Bubble sort

bubble_sort(x):

This function will run a bubble sort algorithm on an array of random integers(size n) or user given array (if option chosen for user given data) x and will return a sorted array.
(here is an example of user given array sorting)

```
Enter the list of numbers separated by spaces: 10 5 100 1 73 281 15 72 1 2 0 35 1 2 17
Bubble sort    :
[10, 5, 100, 1, 73, 281, 15, 72, 1, 2, 0, 35, 1, 2, 17]
[0, 1, 1, 1, 2, 2, 5, 10, 15, 17, 35, 72, 73, 100, 281]
```

Bubble sort is the simplest sort, the code is just comparing the adjacent elements in the array and swaps it if it is in the wrong position.
And likewise, every iteration will sort the array.
The time complexity of the function is O(n^2)

• Selection sort

selection_sort(x):

This function will run a selection sort algorithm on an array of random integers(size n) or user given array (if option chosen for user given data) x and will return a sorted array.
(here is an example of user given array sorting)

```
Enter the list of numbers separated by spaces: 10 5 100 1 73 281 15 72 1 2 0 35 1 2 17
Selection sort :
[10, 5, 100, 1, 73, 281, 15, 72, 1, 2, 0, 35, 1, 2, 17]
[0, 1, 1, 1, 2, 2, 5, 10, 15, 17, 35, 72, 73, 100, 281]
```

The selection sort algorithm sorts an array by repeatedly finding the minimum element from the unsorted part of the array and putting it at the end of the sorted part of the array.
The time complexity of the function is O(n^2)

- **Insertion sort**

insertion_sort(x):

This function will run an insertion sort algorithm on an array of random integers(size n) or user given array (if option chosen for user given data) x and will return a sorted array.
(here is an example of user given array sorting)

```
Enter the list of numbers separated by spaces: 10 5 100 1 73 281 15 72 1 2 0 35 1 2 17
Insertion sort :
[10, 5, 100, 1, 73, 281, 15, 72, 1, 2, 0, 35, 1, 2, 17]
[0, 1, 1, 1, 2, 2, 5, 10, 15, 17, 35, 72, 73, 100, 281]
```

Insertion sort is a simple sorting algorithm that builds the final sorted array (or list) one item at a time.
Code will compare from the 1st element of the array with each element of the array and the shortest element will become that element.
Likewise, code does this for other elements and sort the array.
The time complexity of the insertion sort is O(n^2)

- **Heap sort**

heap_sort(x):

This function will run a heap sort algorithm on an array of random integers(size n) or user given array (if option chosen for user given data) x and will return a sorted array.
(here is an example of user given array sorting)

```
Enter the list of numbers separated by spaces: 10 5 100 1 73 281 15 72 1 2 0 35 1 2 17
Heap sort :
[10, 5, 100, 1, 73, 281, 15, 72, 1, 2, 0, 35, 1, 2, 17]
[0, 1, 1, 1, 2, 2, 5, 10, 15, 17, 35, 72, 73, 100, 281]
```

In Heap sort first we make a binary heap tree from the given array and then remove the root of the binary heap tree and place it at the bottom of the array.
The time complexity of the heap sort is O(nlogn)

- **Merge sort**

merge_sort(x):

This function will run a merge sort algorithm on an array of random integers(size n) or user given array (if option chosen for user given data) x and will return a sorted array. (here is an example of user given array sorting)

```
Enter the list of numbers separated by spaces: 10 5 100 1 73 281 15 72 1 2 0 35 1 2 17
merge sort :
[10, 5, 100, 1, 73, 281, 15, 72, 1, 2, 0, 35, 1, 2, 17]
[0, 1, 1, 1, 2, 2, 5, 10, 15, 17, 35, 72, 73, 100, 281]
```

Merge Sort is a divide-and-conquer algorithm. In this algorithm, we take an array as an input and keep on dividing the array into half and separate each element into a single element and then again merging the array while comparing each element and sorting.
The time complexity of merge sort is O(nlogn)

- **Quick sort**

quick_sort(x):

This function will run a quick sort algorithm on an array of random integers(size n) or user given array (if option chosen for user given data) x and will return a sorted array. (here is an example of user given array sorting)

```
Enter the list of numbers separated by spaces: 10 5 100 1 73 281 15 72 1 2 0 35 1 2 17
Quick sort :
[10, 5, 100, 1, 73, 281, 15, 72, 1, 2, 0, 35, 1, 2, 17]
[0, 1, 1, 1, 2, 2, 5, 10, 15, 17, 35, 72, 73, 100, 281]
```

In a quick sort algorithm we choose a pivot (an element) and sort the array or list around that.
The time complexity of quick sort is O(nlogn).

• **Quick sort using 3 medians**

quick_sort_three_medians(x, l, r):

This function will run a Quick sort using 3 medians algorithm on an array of random integers(size n) or user given array (if option chosen for user given data) x and will return a sorted array.

(here is an example of user given array sorting)

```
Enter the list of numbers separated by spaces: 10 5 100 1 73 281 15 72 1 2 0 35 1 2 17
Quick 3 medians:
[10, 5, 100, 1, 73, 281, 15, 72, 1, 2, 0, 35, 1, 2, 17]
[0, 1, 1, 1, 2, 2, 5, 10, 15, 17, 35, 72, 73, 100, 281]
```

code will apply a median of 3 approach.This code will find median from 1st, last and middle element of array and swap their position according to the values and find pivot value, to run sorting algorithms and compute time complexity on those arrays of different sizes these functions will create different arrays of random integers for different dataset sizes and compute time complexity while running sorting algorithm on them.

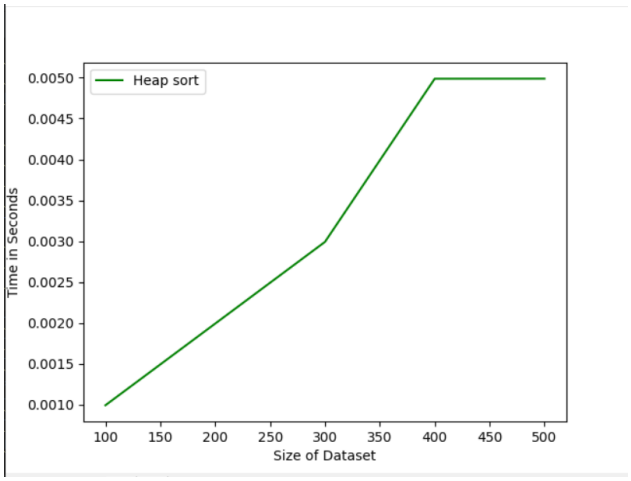
The time complexity of quick sort is O(nlogn).

= to run these sorting algorithms on those arrays of different sizes

- run_heap_sort():

```
Dataset sizes are:  [100, 200, 300, 400, 500]
100 Data_size Sorted by HeapSort   0.0009951591491699219
200 Data_size Sorted by HeapSort   0.001993417739868164
300 Data_size Sorted by HeapSort   0.0029921531677246094
400 Data_size Sorted by HeapSort   0.004987001419067383
500 Data_size Sorted by HeapSort   0.0049877166748046875
Heap sort      :  [0.0009951591491699219, 0.001993417739868164, 0.0029921531677246094, 0.004987001419067383, 0.0049877166748046875]
```

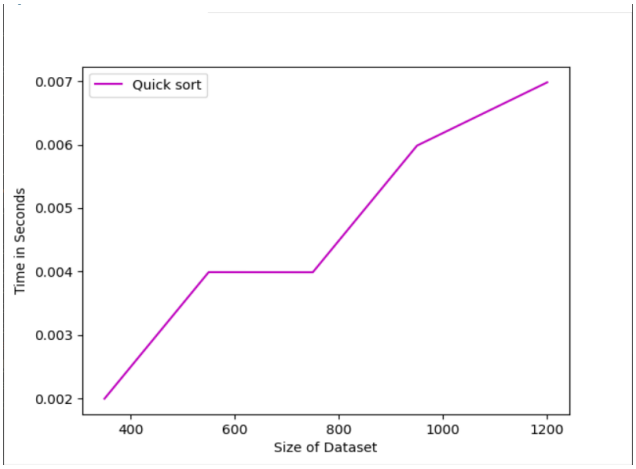
Graphical progression of Heap sort time complexity is shown by providing this data to python matplotlib library



- run_quick_sort():

```
Dataset sizes are:  [350, 550, 750, 950, 1200]
350 Data_size Sorted by QuickSort   0.001995086669921875
550 Data_size Sorted by QuickSort   0.003989696502685547
750 Data_size Sorted by QuickSort   0.0039882659912109375
950 Data_size Sorted by QuickSort   0.005984067916870117
1200 Data_size Sorted by QuickSort   0.006981849670410156
Quick sort     :  [0.001995086669921875, 0.003989696502685547, 0.0039882659912109375, 0.005984067916870117, 0.006981849670410156]
```

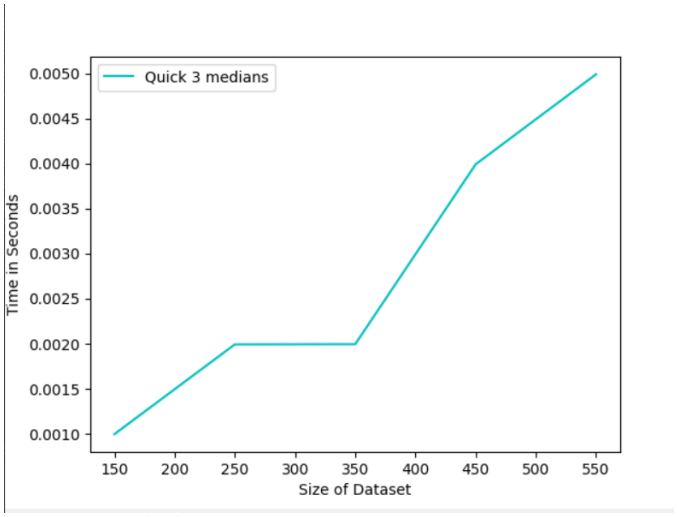
Graphical progression of Quick sort time complexity is shown by providing this data to python matplotlib library



- run_quick_sort_three_medians():

```
Dataset sizes are:  [150, 250, 350, 450, 550]
150 Data_size Sorted by QuickSort with three medians   0.000997304916381836
250 Data_size Sorted by QuickSort with three medians   0.001992940902709961
350 Data_size Sorted by QuickSort with three medians   0.0019953250885009766
450 Data_size Sorted by QuickSort with three medians   0.0039904117584228516
550 Data_size Sorted by QuickSort with three medians   0.0049893856048583984
Quick 3 medians:  [0.000997304916381836, 0.001992940902709961, 0.0019953250885009766, 0.0039904117584228516, 0.0049893856048583984]
```

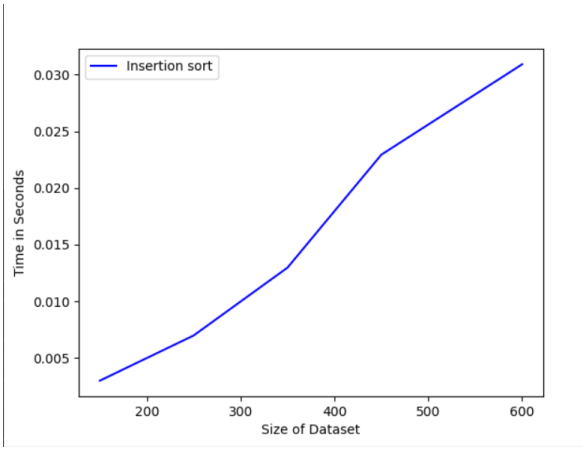
Graphical progression of Quick using 3 medians sort time complexity is shown by providing this data to python matplotlib library



- run_insertion_sort():

```
Dataset sizes are:  [150, 250, 350, 450, 600]
150 Data_size Sorted by InsertionSort   0.0029897689819335938
250 Data_size Sorted by InsertionSort   0.00698089599609375
350 Data_size Sorted by InsertionSort   0.012969255447387695
450 Data_size Sorted by InsertionSort   0.022938013076782227
600 Data_size Sorted by InsertionSort   0.030913829803466797
Insertion sort :  [0.0029897689819335938, 0.00698089599609375, 0.012969255447387695, 0.022938013076782227, 0.030913829803466797]
```

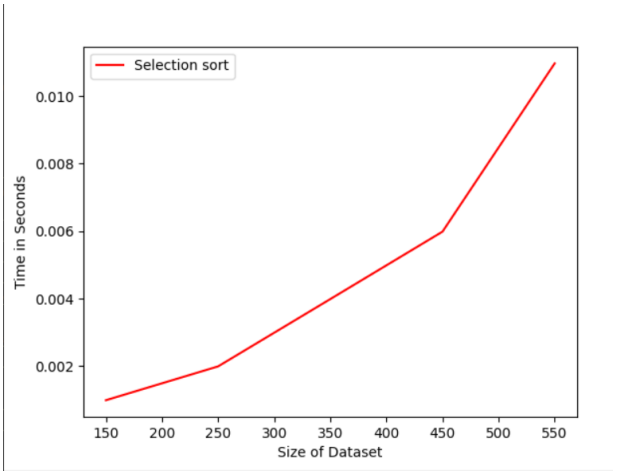
Graphical progression of Insertion sort time complexity is shown by providing this data to python matplotlib library



- run_selection_sort():

```
Dataset sizes are:  [150, 250, 350, 450, 550]
150 Data_size Sorted by SelectionSort   0.0009932518005371094
250 Data_size Sorted by SelectionSort   0.001995086669921875
350 Data_size Sorted by SelectionSort   0.003988504409790039
450 Data_size Sorted by SelectionSort   0.005985498428344726
550 Data_size Sorted by SelectionSort   0.010968923568725586
Selection sort :  [0.0009932518005371094, 0.001995086669921875, 0.003988504409790039, 0.005985498428344726, 0.010968923568725586]
```

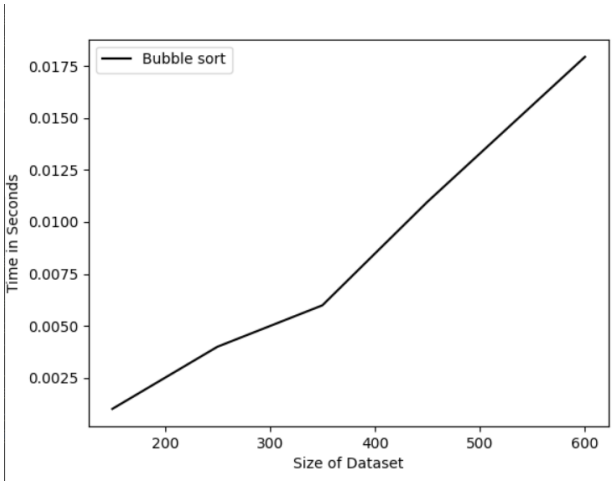
Graphical progression of Selection sort time complexity is shown by providing this data to python matplotlib library



- run_bubble_sort():

```
Dataset sizes are:  [150, 250, 350, 450, 600]
150 Data_size Sorted by BubbleSort   0.0009958744049072266
250 Data_size Sorted by BubbleSort   0.003987550735473633
350 Data_size Sorted by BubbleSort   0.0059850215911865234
450 Data_size Sorted by BubbleSort   0.010970592498779297
600 Data_size Sorted by BubbleSort   0.017952919006347656
Bubble sort    :  [0.0009958744049072266, 0.003987550735473633, 0.0059850215911865234, 0.010970592498779297, 0.017952919006347656]
```

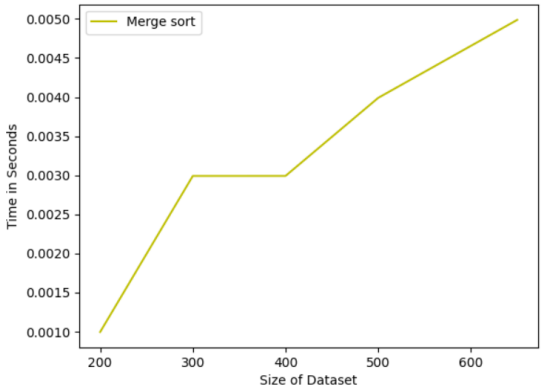
Graphical progression of Bubble sort time complexity is shown by providing this data to python matplotlib library



- run_merge_sort():

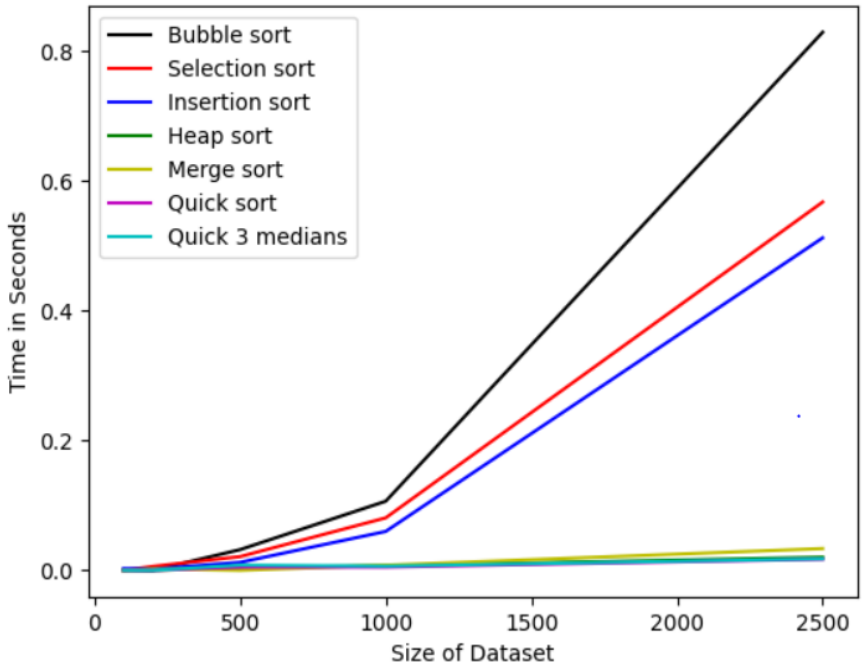
```
200 Data_size Sorted by MergeSort   0.0009963512420654297
300 Data_size Sorted by MergeSort   0.0029916763305664062
400 Data_size Sorted by MergeSort   0.002991914749145508
500 Data_size Sorted by MergeSort   0.0039899349212646484
650 Data_size Sorted by MergeSort   0.004987001419067383
Merge sort     :  [0.0009963512420654297, 0.0029916763305664062, 0.002991914749145508, 0.0039899349212646484, 0.004987001419067383]
```

Graphical progression of Merge sort time complexity is shown by providing this data to python matplotlib library

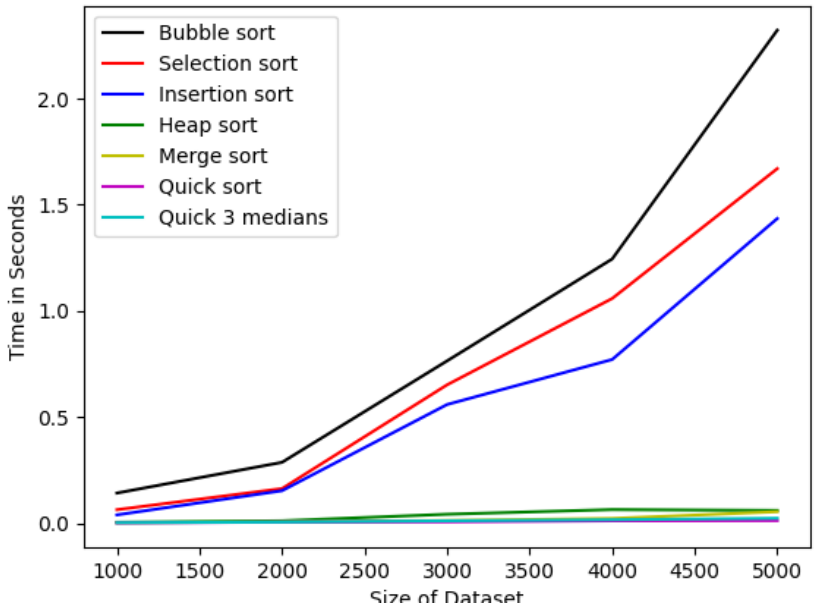


Performance comparisons:

By running all the algorithms at a same time on arrays having random integers of sizes 500, 1000, 1500, 2000, 2500 and providing all the time complexities to python library matplotlib we get comparison between all algorithms showing following results.

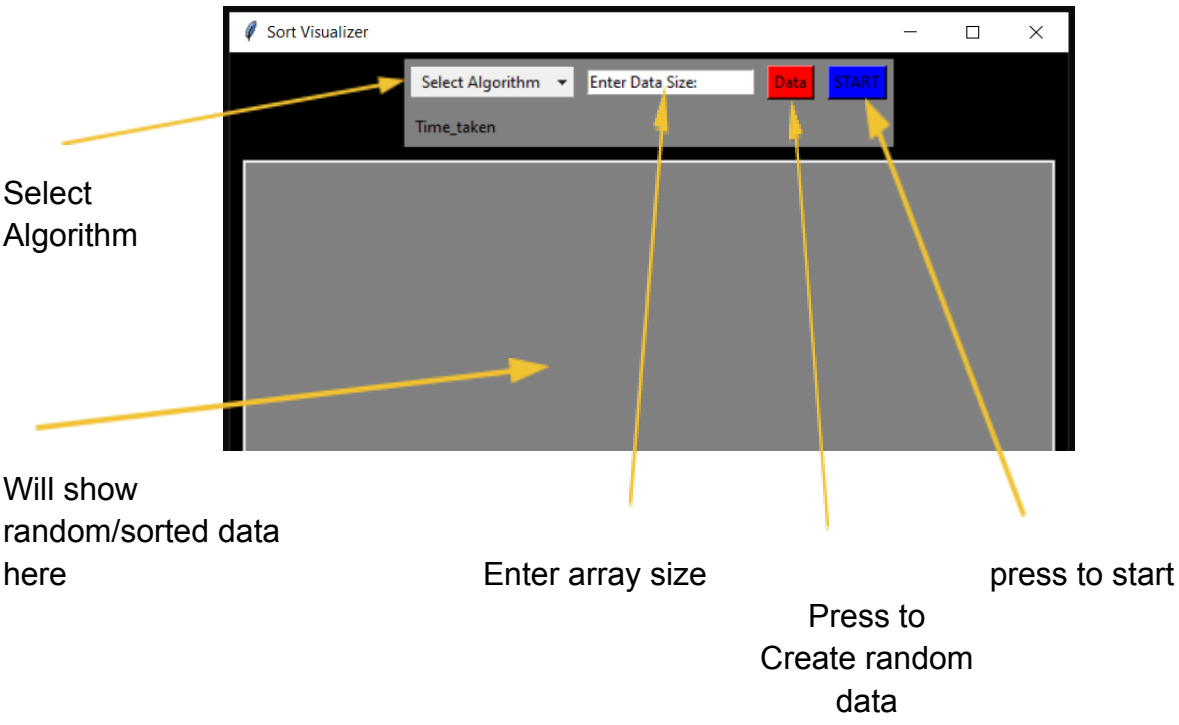


Running those algorithms on even bigger size of arrays shows following results



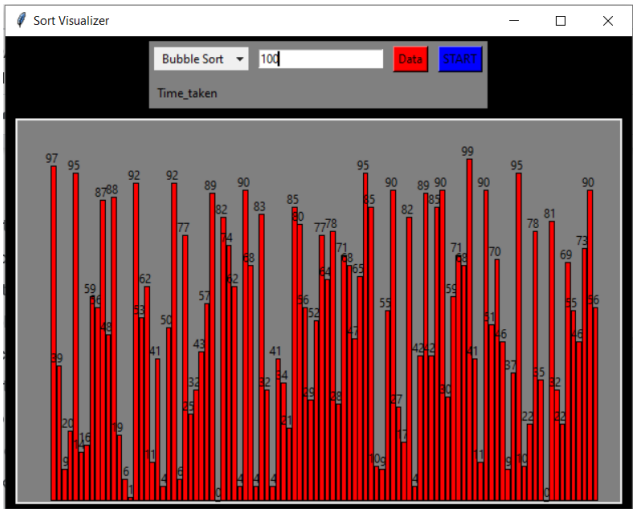
- **GUI Part**

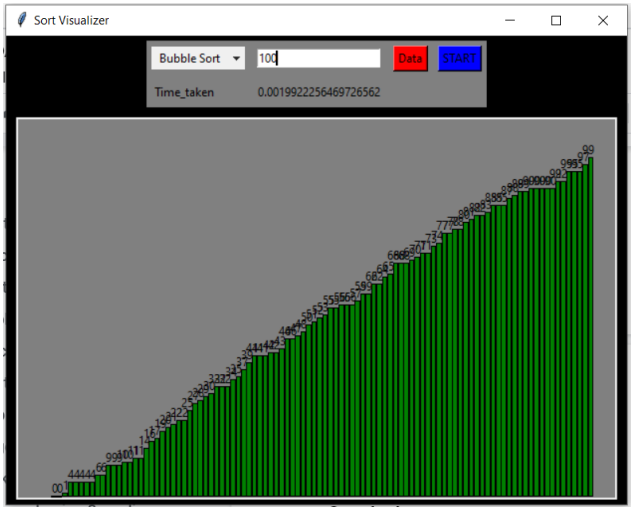
In gui implementation there is a drop down menu to select from sorting algorithms to perform and provide run time complexity for user provided dataset size.



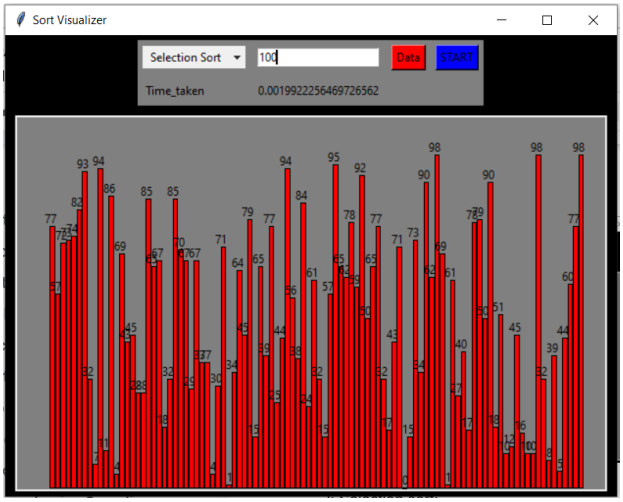
- Select sorting algorithm from drop down menu to perform sorting on array of random integers and show time complexity.
- Enter dataset size
- Press Data(button) to create random dataset
- Press Start(button) to start sorting of the array of given size
- Canvas will show random array / sorted array(In bar formation)
- Label Time_taken will show run time complexity in seconds.

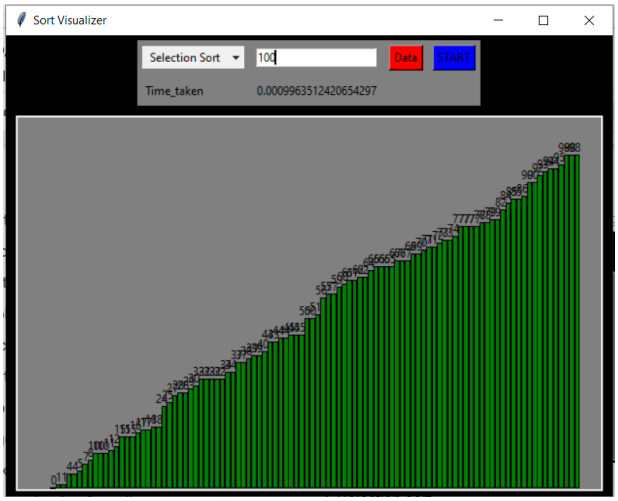
1: Bubble sort:



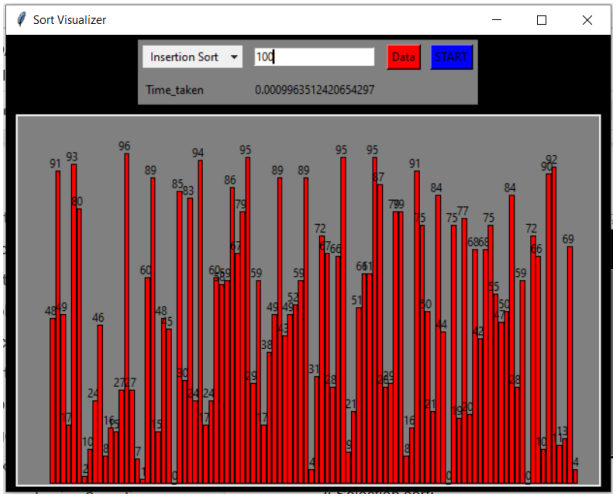


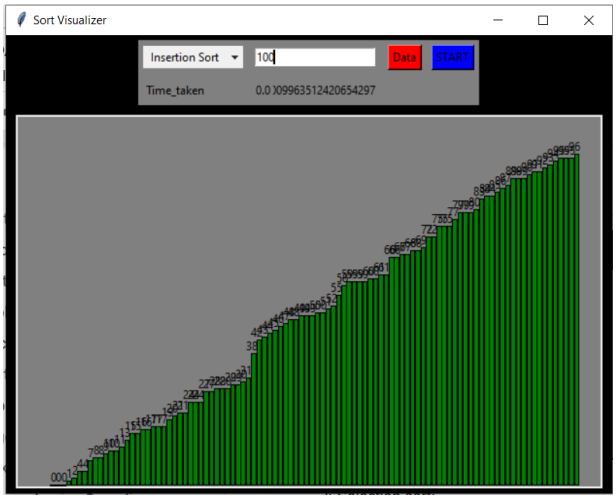
2: Selection sort:



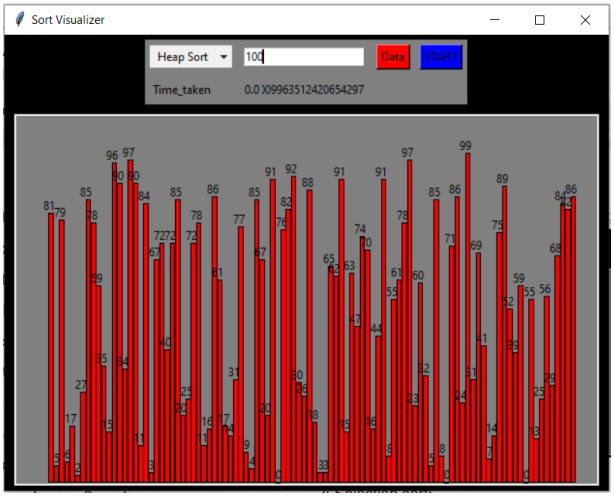


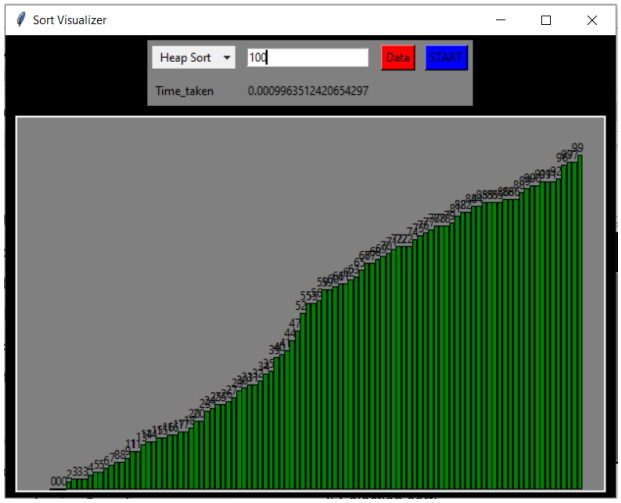
3: Insertion sort:



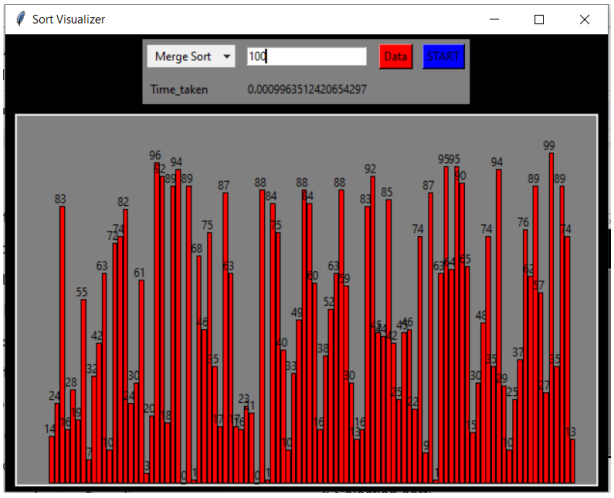


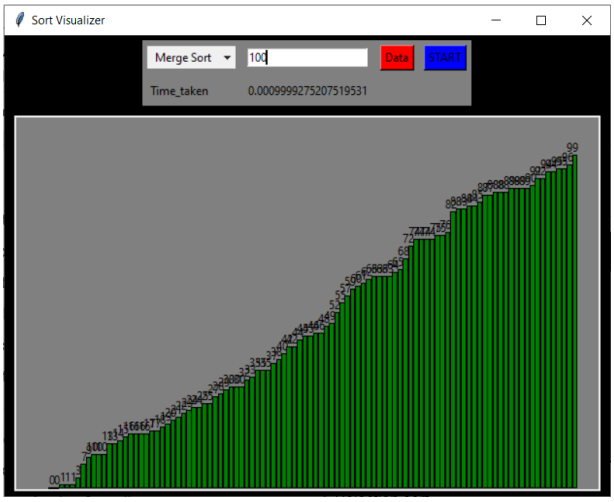
4: Heap sort:



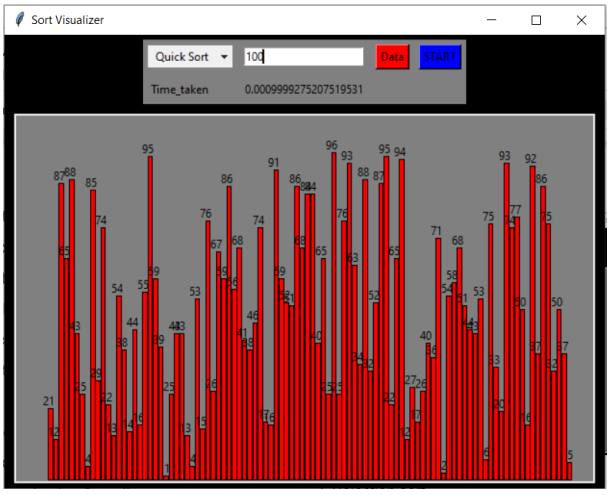


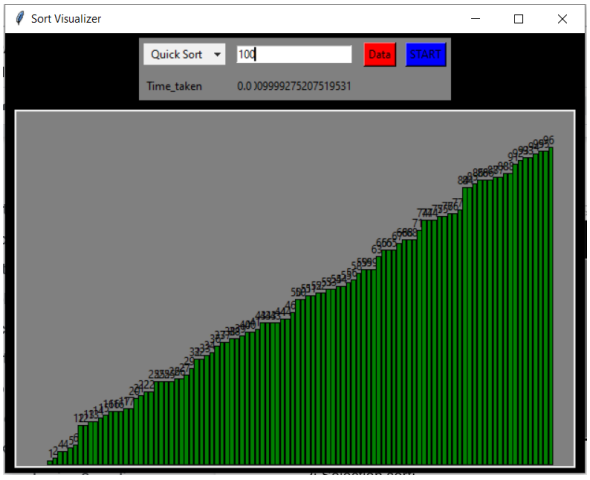
5: Merge sort:



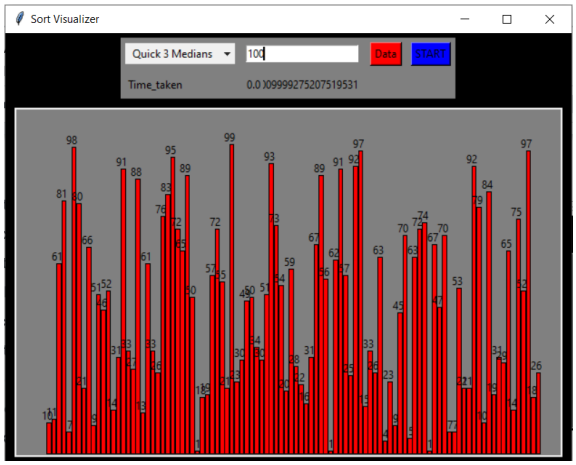


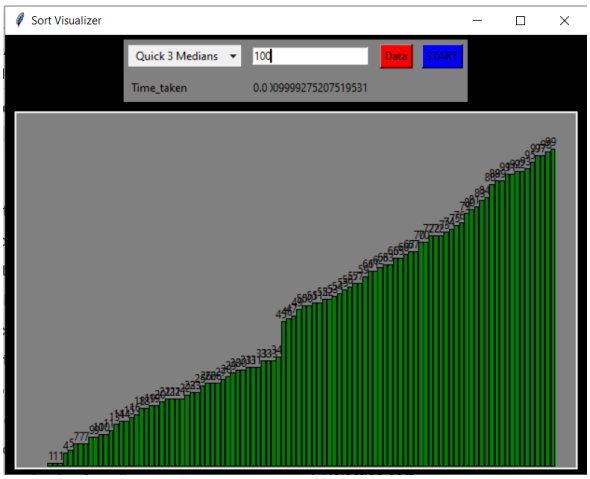
6: Quick sort:





7: Quick sort 3 medians:





Conclusion

According to output the conclusion for a smaller number of values in array every sorting elements takes almost same time but for larger number of values in array there is a significance difference in time taking by algorithms so for larger number of values in array heap sort, merge sort and quick sort (both kind) are preferable rather than bubble sort, selection sort and insertion sort.