**A brief report on the project ‘Simulation of Sorting Techniques’**

Sree Sindhu Sabbineni

**Description:**

The project implements the following sorting techniques using **‘Python’** and performs simulation using the selected datasets. We have also analyzed the performance of the different sorting techniques based on the parameters like runtime, memory usage against the datasize and degree of sortedness. We visualized the results from the graphs drawn using **‘matplotlib’.**

**Implemented Sorting Techniques:**

1.Selection sort

2.Insertion sort

3.Bubble sort

4.Merge sort

5.Quick sort

Here is the brief description of the implemented sorting techniques:

**Selection sort:**

Selection sort algorithm works by finding the minimum element in the array and placing it in the beginning position. This process is repeated on the unsorted part of the array till all the elements are sorted.

**Insertion sort:**

In the insertion sort, the given array is searched sequentially, each element in the array starting from the position two, is placed into the sorted subarray. The lower part of the array is maintained sorted. The element which is to be inserted is placed in the appropriate place in the subarray.

**Bubble Sort:**

Bubble sort algorithm works by traversing through the entire array and swapping adjacent elements into ascending (or descending) order. This process is repeated until all the elements are sorted.

**Merge Sort:**

Merge sort works by dividing an array into two halves and storing the values in two separate lists as ‘left half’ and the 'right half'. All the elements in left-half are sorted by calling the function recursively until there is only one element in the list, the same step is repeated for the right half as well.

In the merge condition, we verify the values of each element in the two lists, when all the elements in the 'right half' are exhausted but 'left half' has still elements, we store the values in 'left half' to a list in the same order because left half is already sorted. Instead, when all the elements in the 'left half' are exhausted but 'right half' has still elements, we store the values in 'right half' to a list in the same order because 'right half' is already sorted. In this way the values are merged to form a sorted array.

**Quick sort:**

Quicksort algorithm works by finding a pivot element in the array (the pivot element can be the value of first index or last index or any value in between), moving all the elements smaller than the pivot to the left side of it, and all the elements greater than the pivot to the right side of it. This yields two subarrays which are: the subarray containing the smaller elements and the subarray containing the greater elements. The process is repeated on each of the subarrays recursively, thereby sorting all the elements.

**Datasets:**

We have taken the below 4 datasets for simulation.

2 data sets (drawn from real time)

2 data sets (generated synthetically)

**Dataset 1:**

It is the data of College Scorecard. It helps students to understand which college good fit for them is. It consists of fields like the institute ID, OPEID (Id given by the postal department), institute name, city, region etc. The data is of the size 444kb.

We have extracted the field of OPEID, which consists of about 7000 values for our study.

The information related to the data can be obtained from:

<https://catalog.data.gov/dataset/college-scorecard>

**Dataset 2:**

It is the data of US State population. The population of all the states has been listed for each year for all genders, male, female etc. The data is of the size 60kb.

We have extracted 50 population values (of 50 states of US) pertaining to the year 2010, and the population of both the sexes for our study.

The information related to the data can be obtained from:

<https://www.census.gov/data/datasets/2017/demo/popest/state-detail.html>

<https://factfinder.census.gov/faces/tableservices/jsf/pages/productview.xhtml?src=bkmk>

**Dataset 3:**

This a synthetic data of 100 decimal values ranging from 0 to 500, these values are analogous to real time values of voltages, physical weights etc.

**Dataset 4:**

This is a Synthetic data of 500 hundred integer numbers ranging from 1 to 10000, these values are analogous to any situation in real time which involves counting like people, things, species etc.

**Performance Curves:**

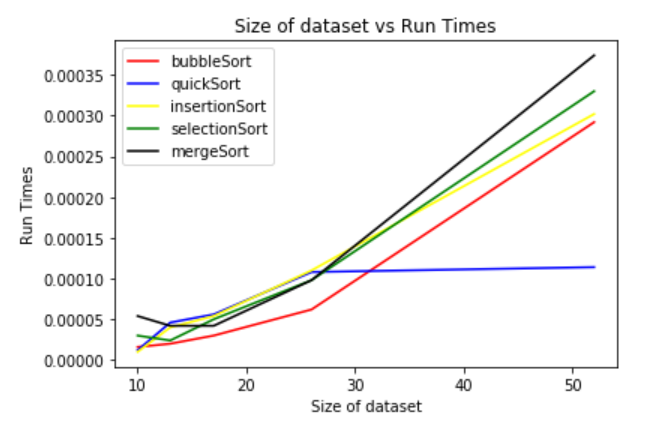
We have evaluated the performance of every sorting technique based on the measures:

1. Datasize vs RunTime
2. Degree of Sortedness vs RunTime
3. Datasize vs MemoryUsage
4. Degree of Sortedness vs MemoryUsage

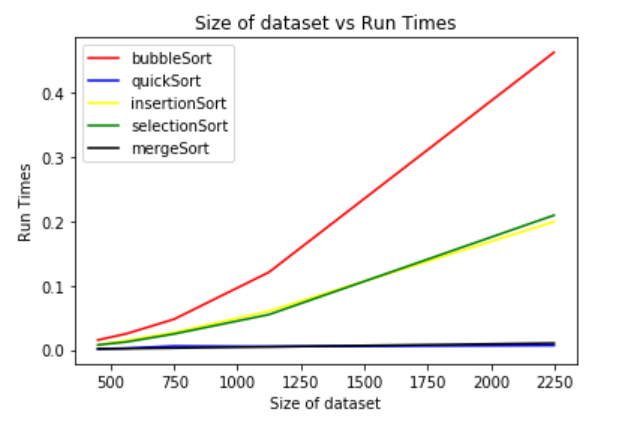
We have plotted the above graphs for each dataset and we made sure that every point on the curve comes from the average of 5 runs. We have also recorded the standard deviation for runtimes of algorithms.

**Datasize vs RunTime:**

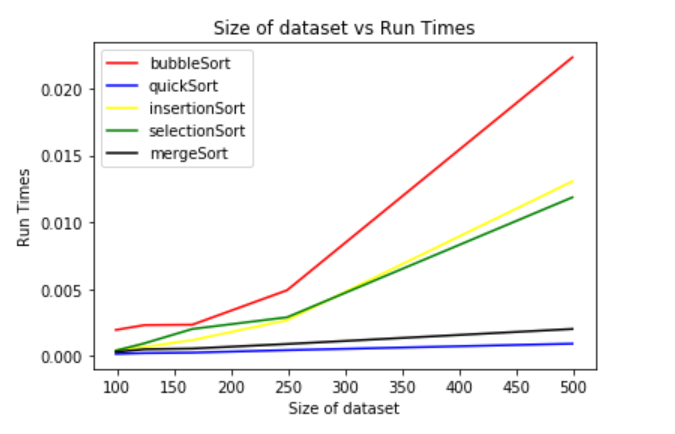
Dataset1:



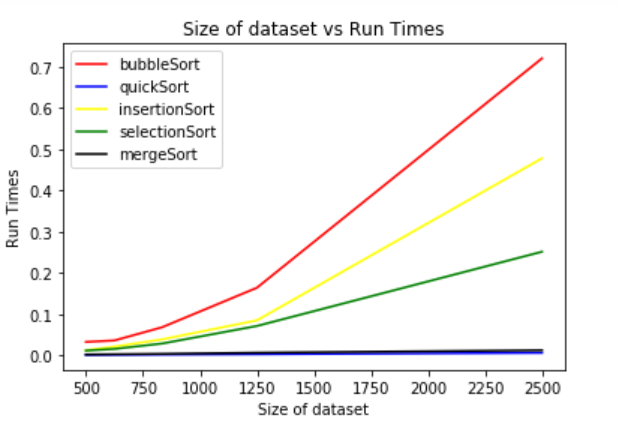
Dataset2:



Dataset3:

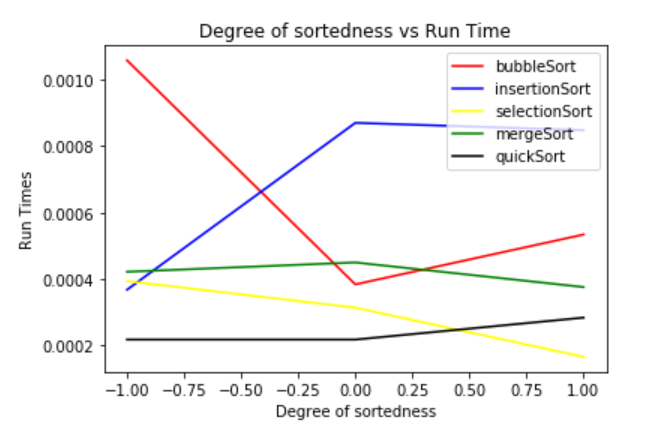


Dataset4:

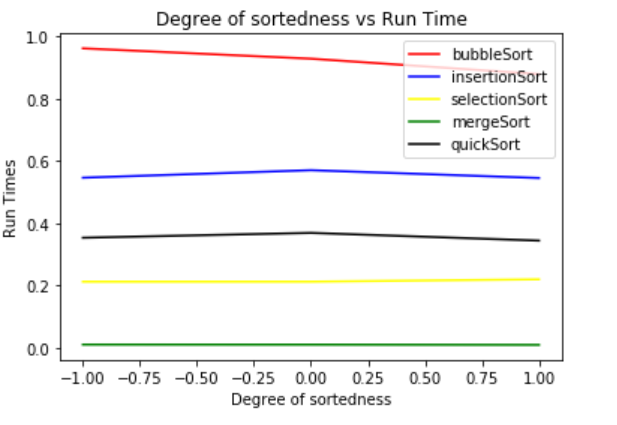


**Degree of Sortedness vs Runtime:**

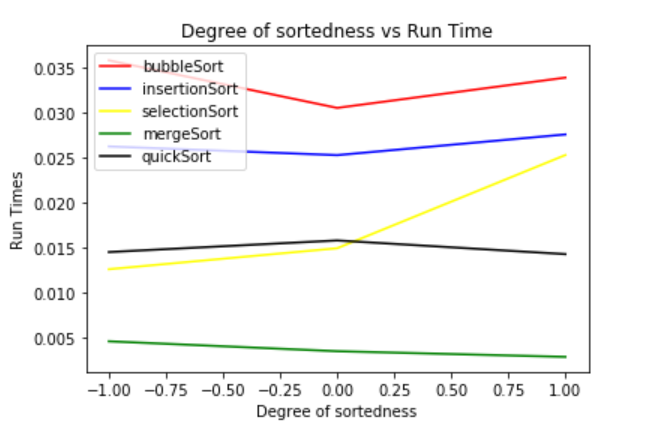
**Dataset1:**



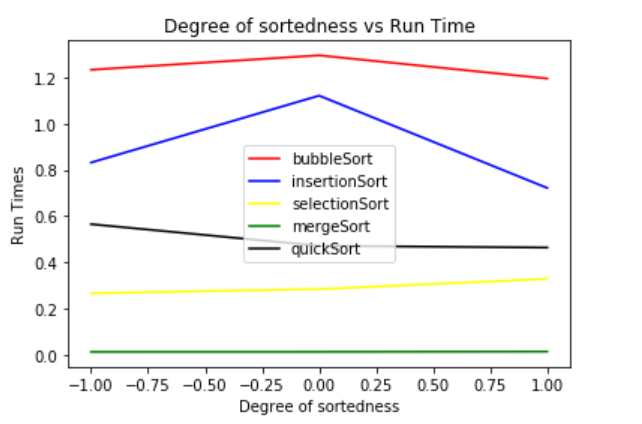
**Dataset2:**

****

Dataset3:

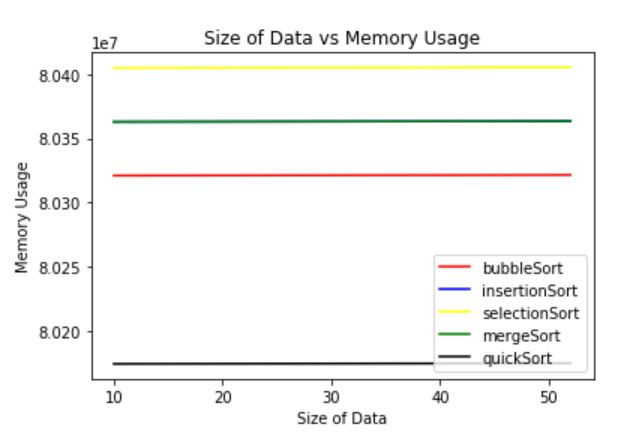


Dataset4:

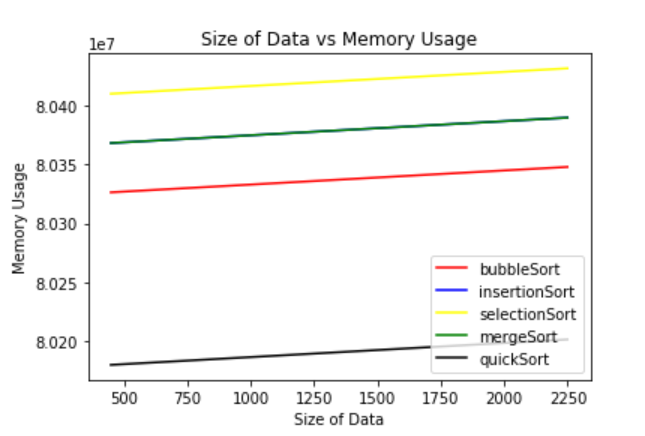


Datasize vs MemoryUsage:

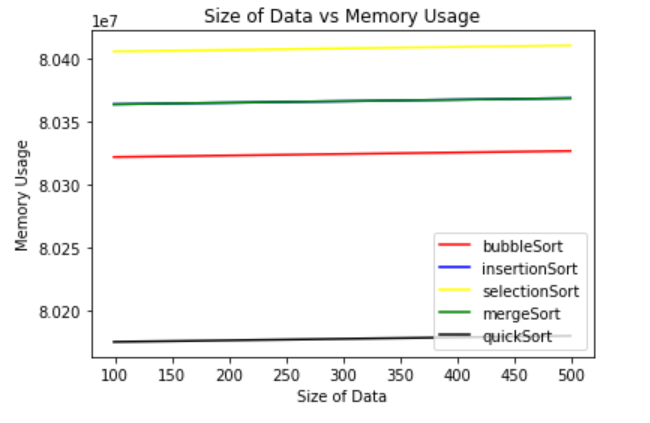
Dataset1:



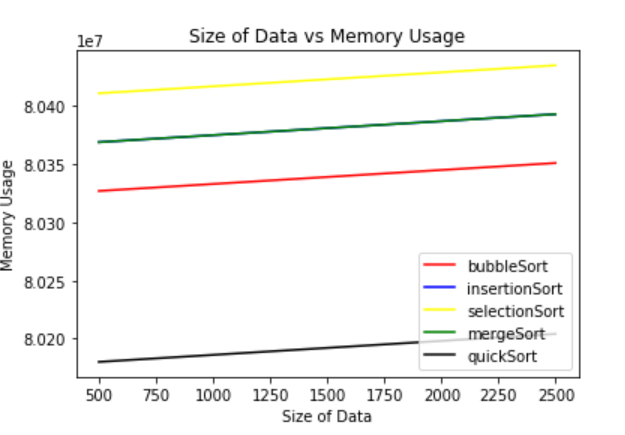
Dataset2:



Dataset3:

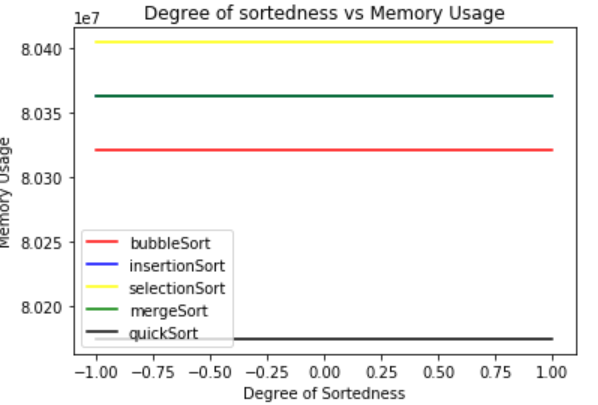


Dataset4:

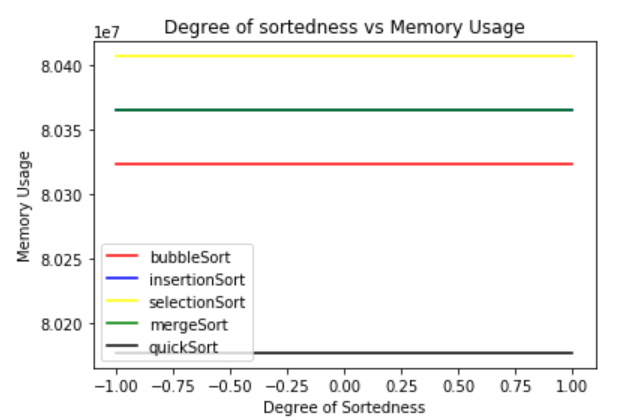


Degree of Sortedness vs Memory Usage:

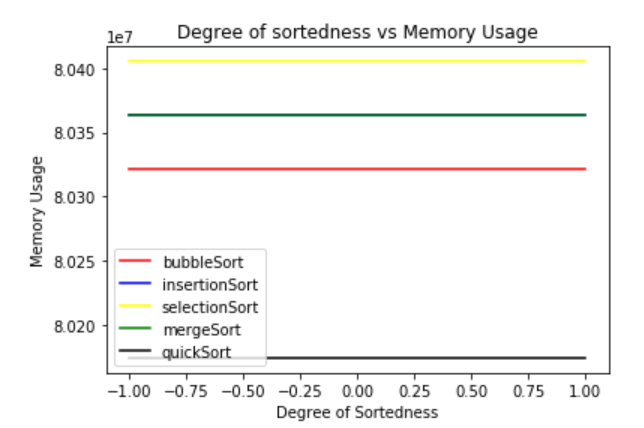
Dataset1:



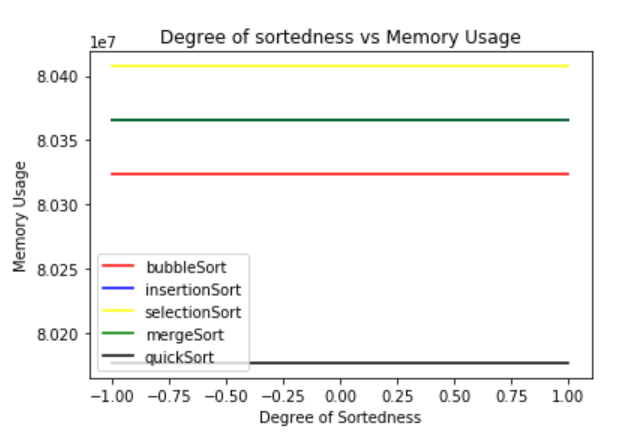
Dataset2:



Dataset3:



Dataset4:



Discussion on Performance Curves:

As we see from the graphs for ‘Datasize vs RunTime’, there is not much difference in the run times for every sorting technique for small size datasets. But, as the size of the data goes on increasing, run times of the sorting techniques like ‘bubble sort’ increases rapidly followed by ‘selection sort’ and ‘insertion sort’. The runtimes of ‘merge sort’ and ‘quick sort’ do not increase rapidly even for large datasizes. Overall, ‘quick sort’ looks like the better sorting technique for these selected datasets.

If we observe the graphs for ‘Degree of Sortedness vs Runtime’, we see some interesting observations. We considered ‘completely inversed list’, ‘randomly unsorted list’ and ‘completely sorted list’ for measuring the degree of sortedness. A few sorting techniques has considerable decrease in run time for already sorted lists and a few have lesser times for inverse lists. Overall, merge sort and quick sort tends to be faster than other three techniques.

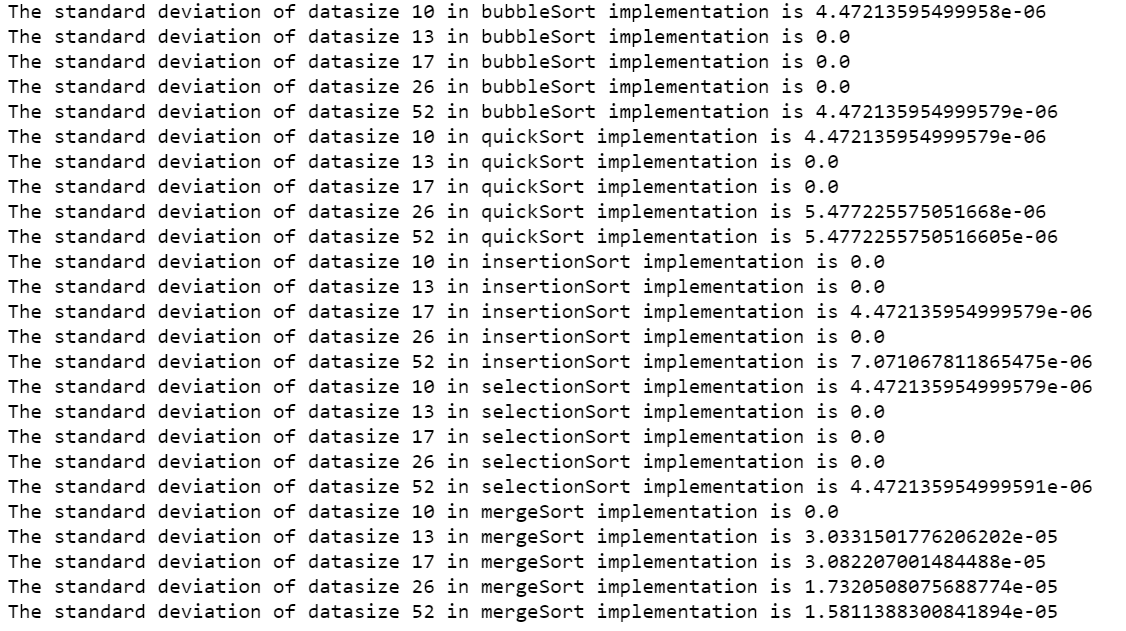
To measure the memory usage, we have loaded **‘memory\_profiler’** and used **%memit** magic function. **%memit -r 5** gives the average memory usage for 5 simulations. We have also considered the input datasize to calculate the memory usage using **‘sys.getsizeof(object)’.** We combined these two values to get the overall memory usage of the sorting techniques. We observed that the standard deviation is very less. The memory usage gradually increases as the size of data increases for every sorting technique as displayed in the graphs for Datasets 2 and 4. However, for datasets 1 and 2, we hardly see any difference because they are small datasets. Although there will be an increase of memory usage in bytes, we cannot notice it in graphs.

As seen in the graphs for ‘Degree of sortedness vs Memory Usage’, the degree of sortedness doesn’t have affect on the memory usage of the sorting techniques. It might be because of the fact that all the techniques are inplace algorithms and it will have the same memory irrespective of the arrangement of numbers in the list.

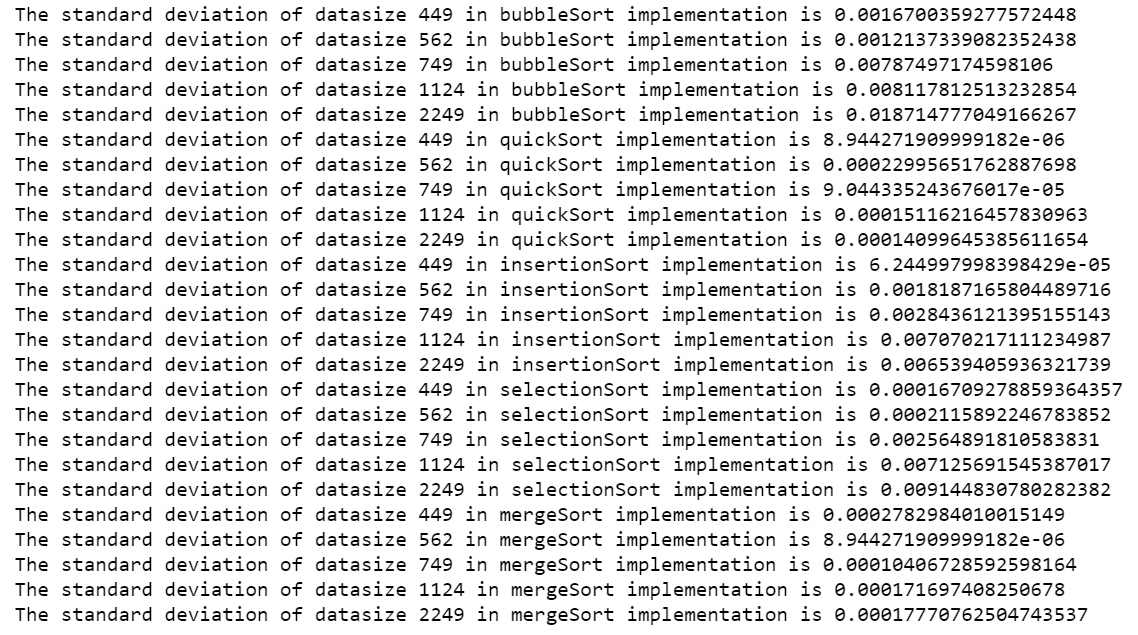
Recorded Standard Deviation:

We ran each function for 5 times and took the average to plot a point on the curves. We have also recorded the standard deviation of runtimes for 4 datasets.

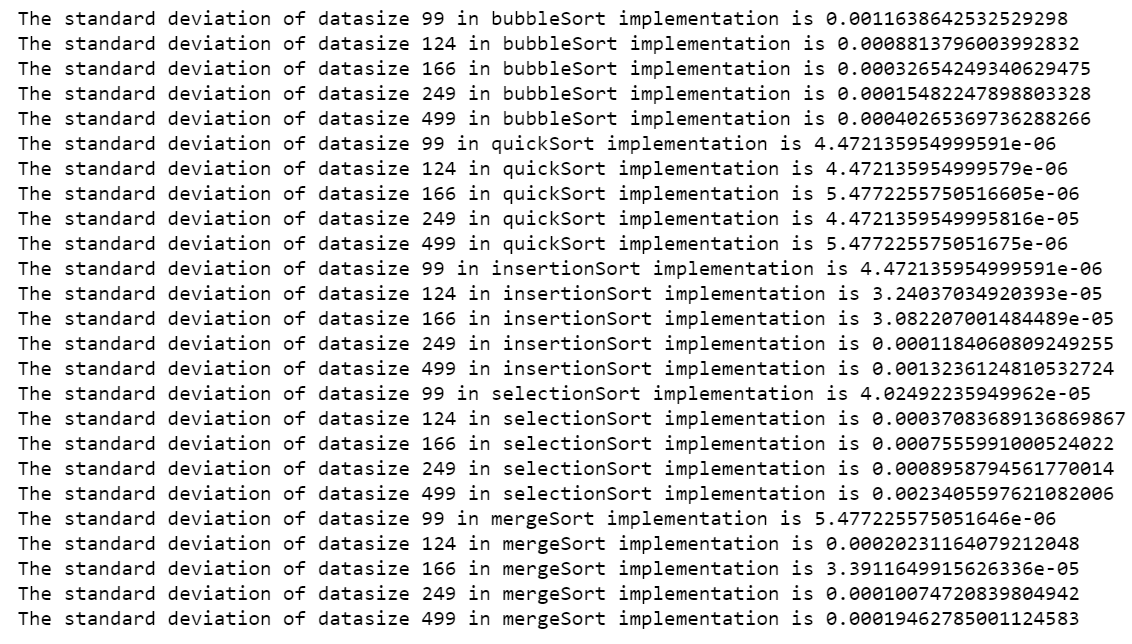
Standard Deviation for Dataset1:



Standard Deviation for Dataset2:



Standard Deviation for Dataset3:



Standard Deviation for Dataset4:

