Simulation of sorting techniques

Submitted by: Bhagyashree Aras

Index

Introduction to sorting techniques sorting algorithm

Insertion sort

Insertion sort is a technique which happens in place. The array is sorted by comparing one item with others sequentially and then arranged in sorted order.The running time for average, best, worst case is mentioned below:

1. Worst Case: O(n^2)
2. Best Case: O(n^2)
3. Average Case: O(n^2)

Selection Sort

Selection sort is a technique which finds a minimum value/ maximum value and swaps it to its correct position. This is also done in place. The algorithm proceeds with auch swaps making the unsorted list to a sorted list from left to right. The running time for average, best, worst case is mentioned below:

1. Worst Case: O(n^2)
2. Best Case: O(n^2)
3. Average Case: O(n^2)

Bubble sort

Bubble sort is a sorting technique which works by repeatedly swapping two adjacent elements if they are unsorted. The running time for average, best, worst case is mentioned below:

1. Worst Case: O(n^2)
2. Best Case: O(n)
3. Average Case: O(n^2)

Merge sort

Merge sort uses the “divide and conquer” strategy. Merge sort breaks the data into small segments, sort it and then merge it back. This sorting technique is preferable because of its efficiency. The data is divided in half and then sorted in log n complexity. Once sorted, it takes n operations to merge it together. Therefore, the running time of merge sort is O(n log n). The running time for average, best, worst case is mentioned below:

1. Worst Case: O(n*log n)
2. Best Case: O(n*log n) or omega(n logn)
3. Average Case: O(n*log n) or theta (n logn)

Quick sort

Quick sort is another divide-and-conquer sorting algorithm. To understand quick sort, let's take an example of students standing in a line in an unordered manner. If we ask them to stand in a sorted way (student with the smallest height at position 1). The student with lowest height will go position 1, same with tallest student. The middle students will swap with each other to get to a correct position. This is how quicksort works. Quicksort starts by choosing a pivot value that is used to divide large and small values. There are multiple methods of choosing a pivot. Once pivot is selected, partitioning procedure for quick sort makes sure that values smaller than pivot are on the left side and values greater than pivot are on the right side. Using recursion, the left and right side will be sorted respectively. The running time for average, best, worst case is mentioned below:

1. Worst Case: O(n^2)
2. Best Case: O(n*log n)
3. Average Case: O(n*log n)

Description of the data sets

This simulation consists of a synthetic dataset and real dataset.

Synthetic Dataset

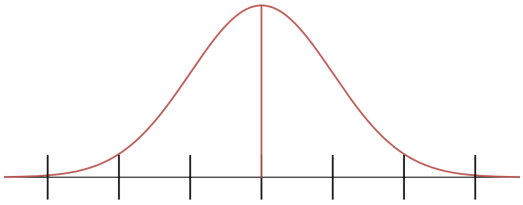
Two Synthetic data sets used are primarily derived from discrete uniform and gaussian distribution.

Uniform Distribution

This distribution is determined by giving equal probability to data points in a given range. For example: tossing a coin is a good example of uniform distribution. The chance of getting a head or tail is the same. Java provides a way to generate samples of uniformly distributed random numbers. For this simulation I am using rand.int( ) with range to generate a sample.

Gaussian Distribution

It is a distribution which peaks in the center and gradually decreases towards the end. When values are clustered around an average, such distribution is plotted using bell-shaped frequency distribution. The curve is plotted using the data points for an item that meets the criteria of Gaussian distribution'. Normal distribution is another name for Gaussian distribution.The area under the curve is used to find probability. The *x* axis shows integer values and the range. The height is marked by how many times a given data point. This point is calculated by a random number generator with a normal distribution.



Graph is created using https://www.desmos.com/calculator

The following graph has a standard deviation of 1 and mean of 50. So, while considering normal distribution, two Factors are taken in account that is:

1. Standard deviation: It determines how much spread out the distribution has.
2. Mean: It is the center of the standard normal distribution.

I am generating dataset 2 with the help of the built java function : nextGaussian( ). This method returns a random number with a mean of 0 and standard deviation 1. For this simulation, I am scaling and shifting the mean of 500 and a standard deviation of 100 to get gaussian distribution. After using this distribution, 70% of values will fall between 400 and 600. 95% of values will fall between 300 and 700.

Real Dataset

For this simulation, two real data sets are extracted. They are as follows:

Weather Outliers

Data source : NOAA

US Consumer Price Index

Data source : U.S. BUREAU OF LABOR STATISTICS

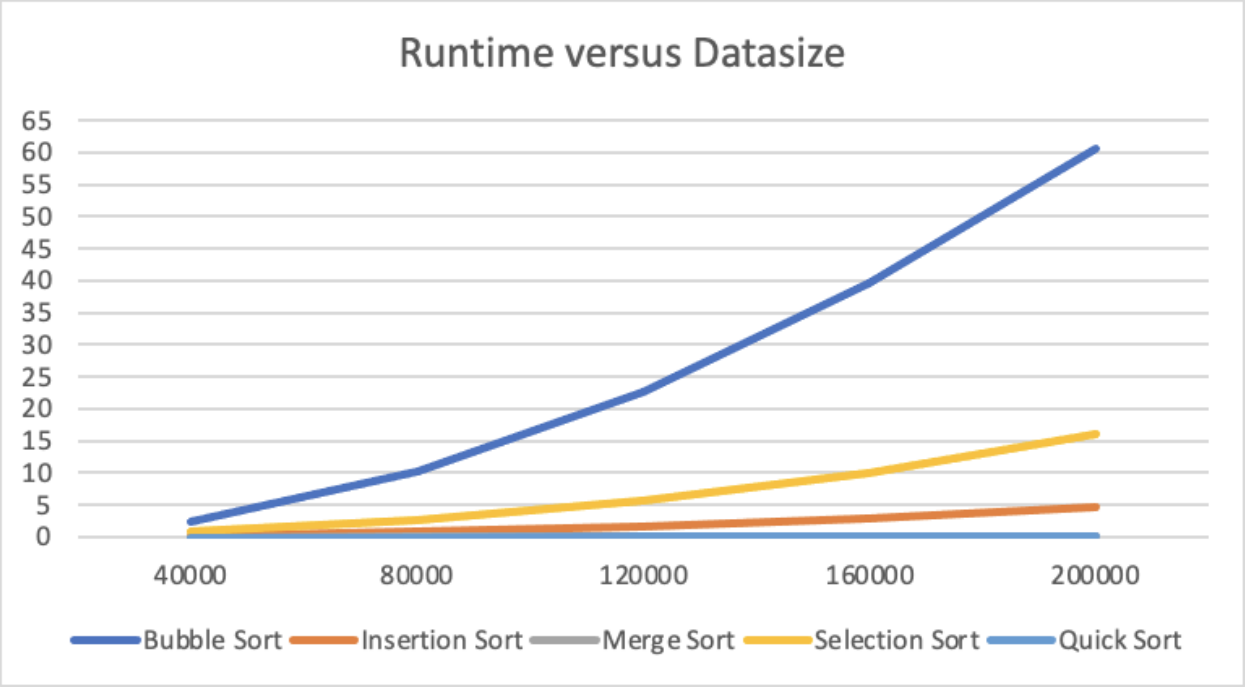
The data contains upto 5 lakh of data. But in this simulation, I am considering upto 2 lakh. This is because of some limitation which is covered in issues faced.

Performance Curves

Synthetic Dataset 1

Graph1: RunTime versus Datasize

Sort	40000	80000	120000	160000	200000
Bubble Sort	2.492	10.161	22.566	39.556	60.667
Insertion Sort	0.183	0.733	1.62	2.926	4.671
Merge Sort	0.006	0.013	0.019	0.027	0.034
Selection Sort	0.787	2.52	5.636	10.067	16.061
Quick Sort	0.004	0.004	0.006	0.008	0.01

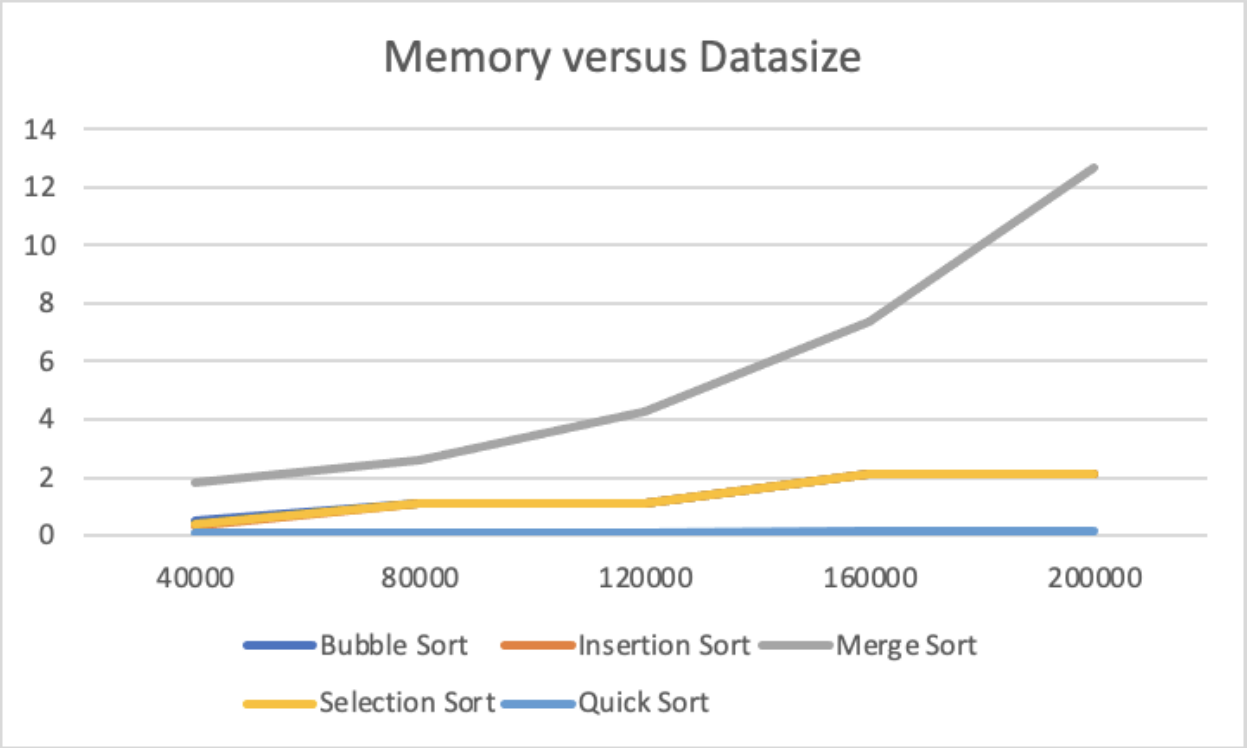


Here the data set had a uniform distribution. The above graph indicates as the data size increases the bubble sort has worse complexity and takes a lot of time to sort. It is followed by a selection sort. Insertion sort is still performing better on large data size. Merge and quicksort are the fastest algorithms although data size is increasing.

Graph2: Memory versus Datasize

Sort	40000	80000	120000	160000	200000

Bubble Sort	**0.491**	**1.109**	**1.096**	**2.102**	**2.137**
Insertion Sort	**0.327**	**1.135**	**1.096**	**2.122**	**2.137**
Merge Sort	**1.803**	**2.613**	**4.259**	**7.391**	**12.675**
Selection Sort	**0.381**	**1.096**	**1.096**	**2.143**	**2.133**
Quick Sort	**0.076**	**0.096**	**0.096**	**0.137**	**0.133**



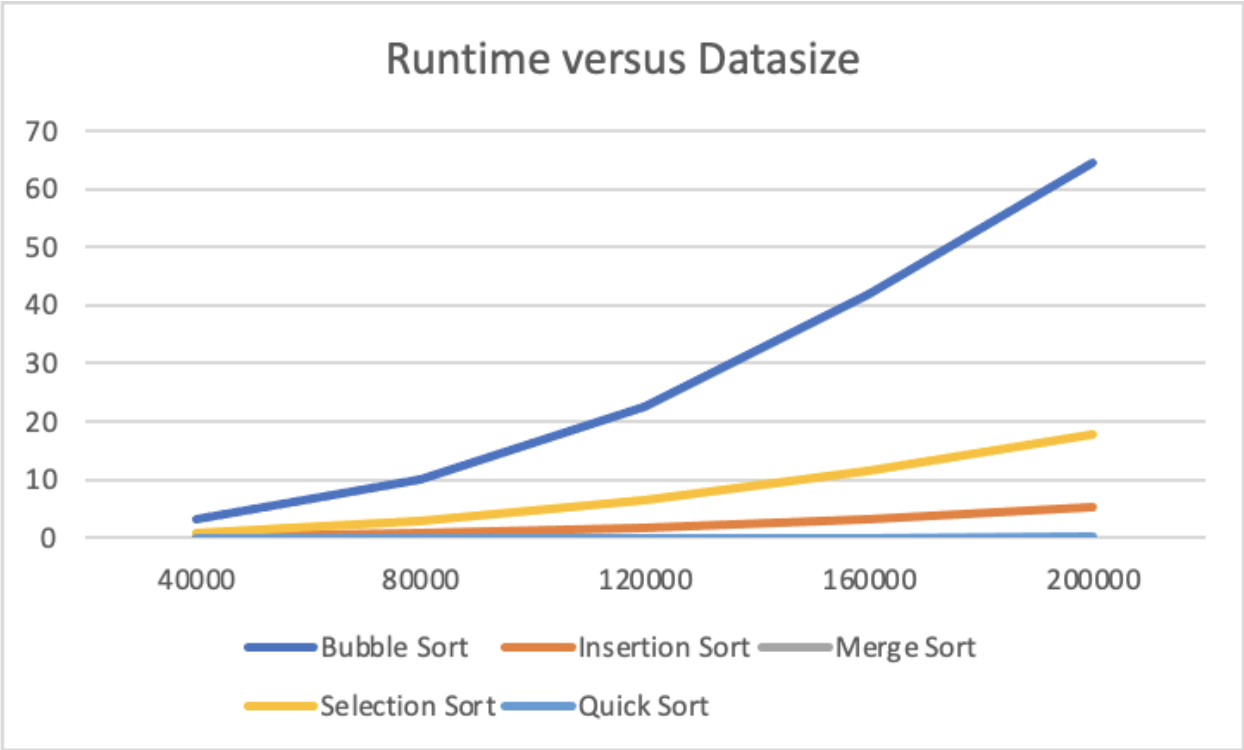
The above graph shows that merge sort uses more space as compared to other sorting algorithms.

Synthetic Dataset 2

Graph1: RunTime versus Datasize

Sort	**40000**	**80000**	**120000**	**160000**	**200000**
Bubble Sort	**3.029**	**9.937**	**22.634**	**41.837**	**64.552**
Insertion Sort	**0.196**	**0.748**	**1.722**	**3.042**	**5.112**
Merge Sort	**0.006**	**0.01**	**0.018**	**0.023**	**0.033**

Selection Sort	**0.749**	**2.731**	**6.405**	**11.597**	**17.784**
Quick Sort	**0.004**	**0.008**	**0.017**	**0.03**	**0.044**

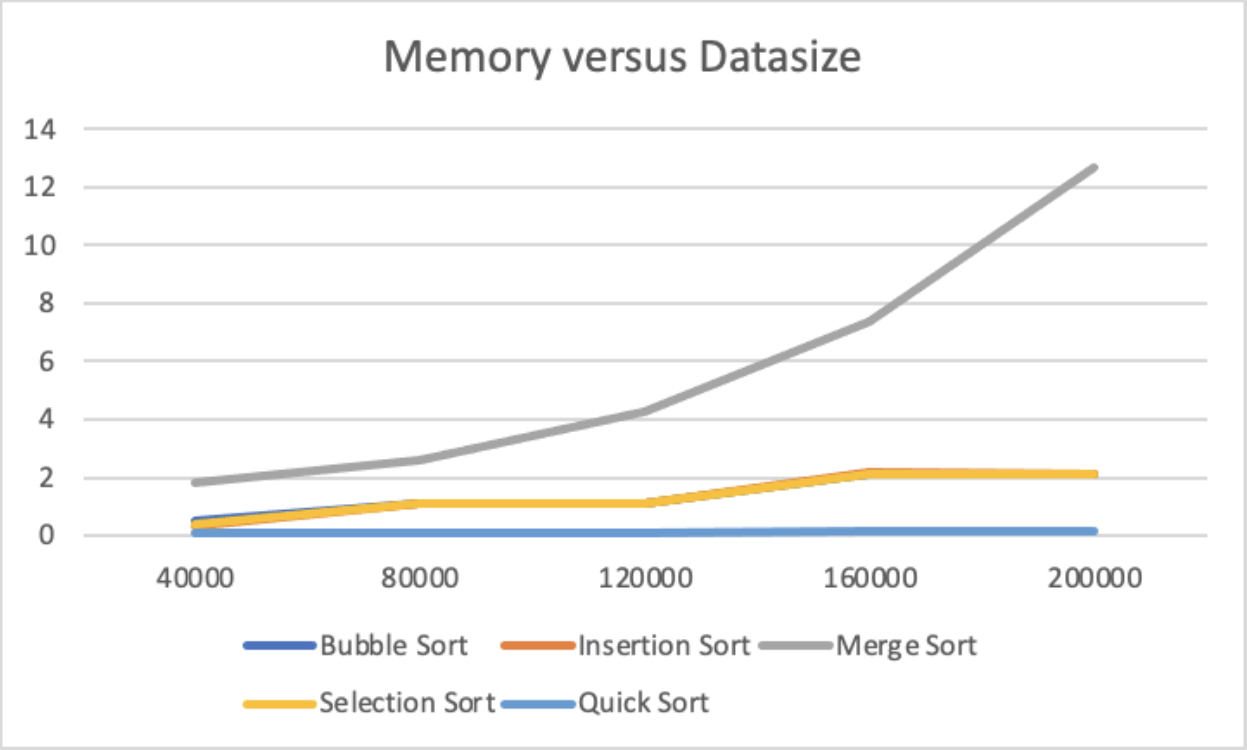


Here the datasets are using normal distribution. The above graph indicates the performance is the same as with synthetic dataset 1. The order of worse performance: Bubble sort > Selection sort > Insertion sort. The merge sort and quicksort are fast for this distribution too.

Graph2: Memory versus Datasize

Sort	40000	80000	120000	160000	200000
Bubble Sort	**0.491**	**1.101**	**1.096**	**2.126**	**2.137**
Insertion Sort	**0.327**	**1.128**	**1.096**	**2.153**	**2.137**
Merge Sort	**1.803**	**2.616**	**4.259**	**7.391**	**12.675**
Selection Sort	**0.381**	**1.096**	**1.096**	**2.146**	**2.133**

Quick Sort	**0.076**	**0.096**	**0.096**	**0.137**	**0.133**



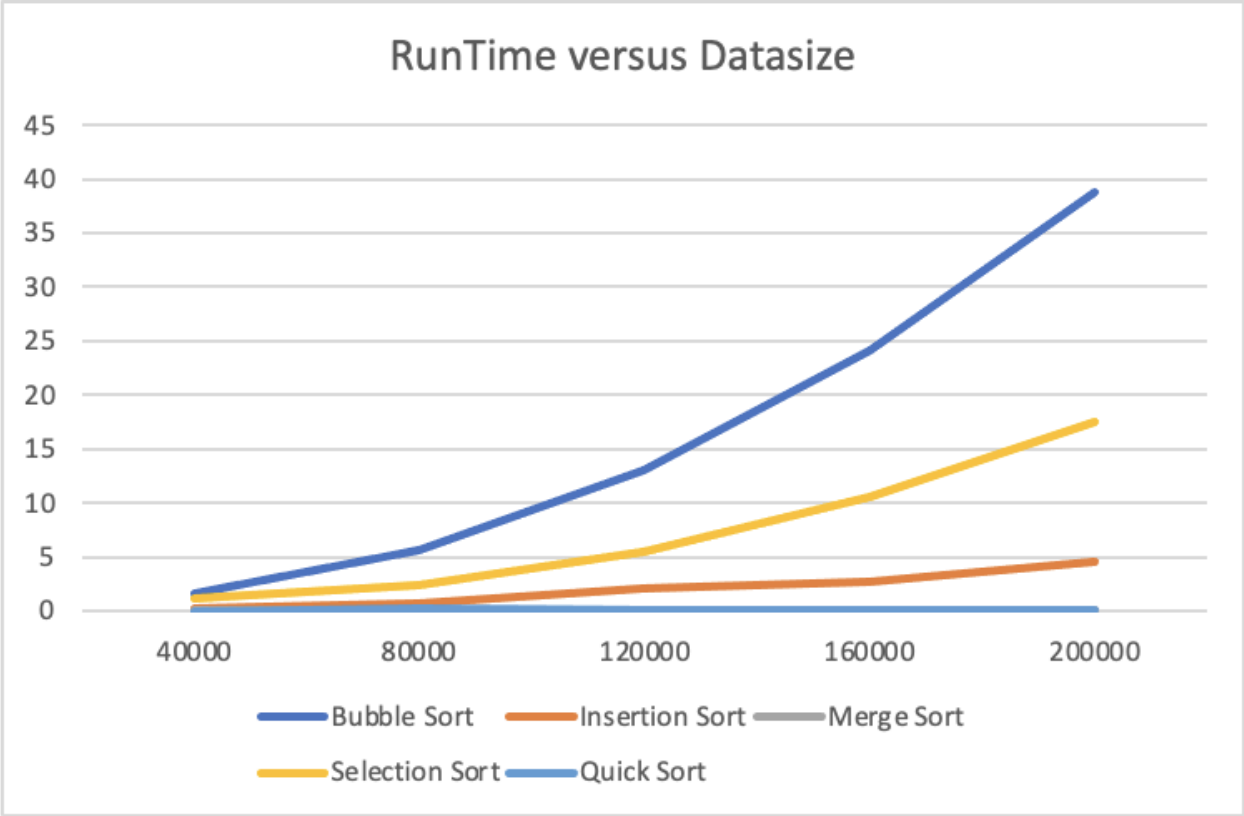
The above graph shows that merge sort uses more space as compared to other sorting algorithms.

Real Dataset 1

Graph1: RunTime versus Datasize

Sort	40000	80000	120000	160000	200000
Bubble Sort	**1.639**	**5.656**	**13.019**	**24.128**	**38.85**
Insertion Sort	**0.32**	**0.639**	**2.099**	**2.762**	**4.547**
Merge Sort	**0.018**	**0.008**	**0.014**	**0.021**	**0.026**

Selection Sort	**1.211**	**2.48**	**5.539**	**10.566**	**17.443**
Quick Sort	**0.007**	**0.23**	**0.049**	**0.087**	**0.139**

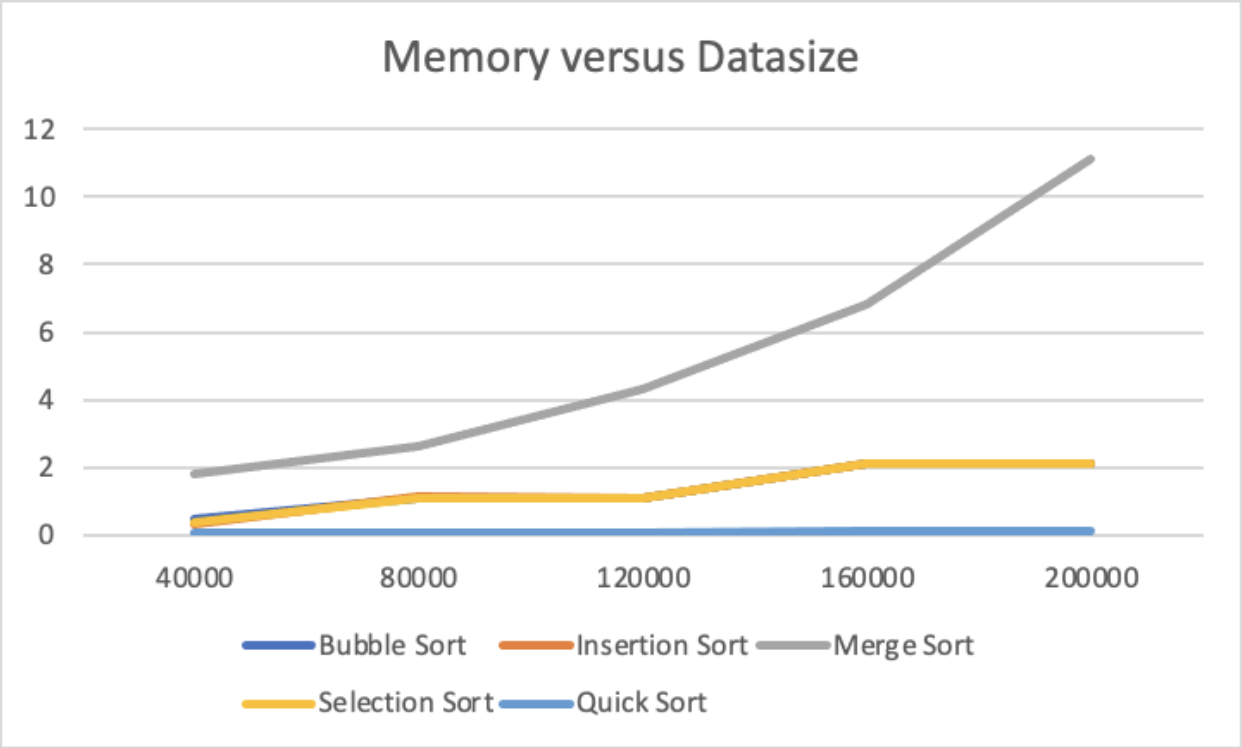


Here the datasets are real thus values can be distorted. The above graph indicates the performance is the same as with synthetic dataset 1. But, it can be seen that run time for bubble , selection, insertion sorts are more than synthetic. This is happening because the degree of sortedness can vary. Still, in general the order of worse performance: Bubble sort > Selection sort > Insertion sort. The merge sort and quicksort are fast for this distribution too.

Graph2: Memory versus Datasize

Sort	40000	80000	120000	160000	200000
Bubble Sort	**0.493**	**1.11**	**1.097**	**2.108**	**2.138**
Insertion Sort	**0.328**	**1.141**	**1.097**	**2.115**	**2.138**

Merge Sort	**1.806**	**2.612**	**4.305**	**6.828**	**11.094**
Selection Sort	**0.381**	**1.096**	**1.097**	**2.143**	**2.134**
Quick Sort	**0.076**	**0.096**	**0.097**	**0.137**	**0.134**



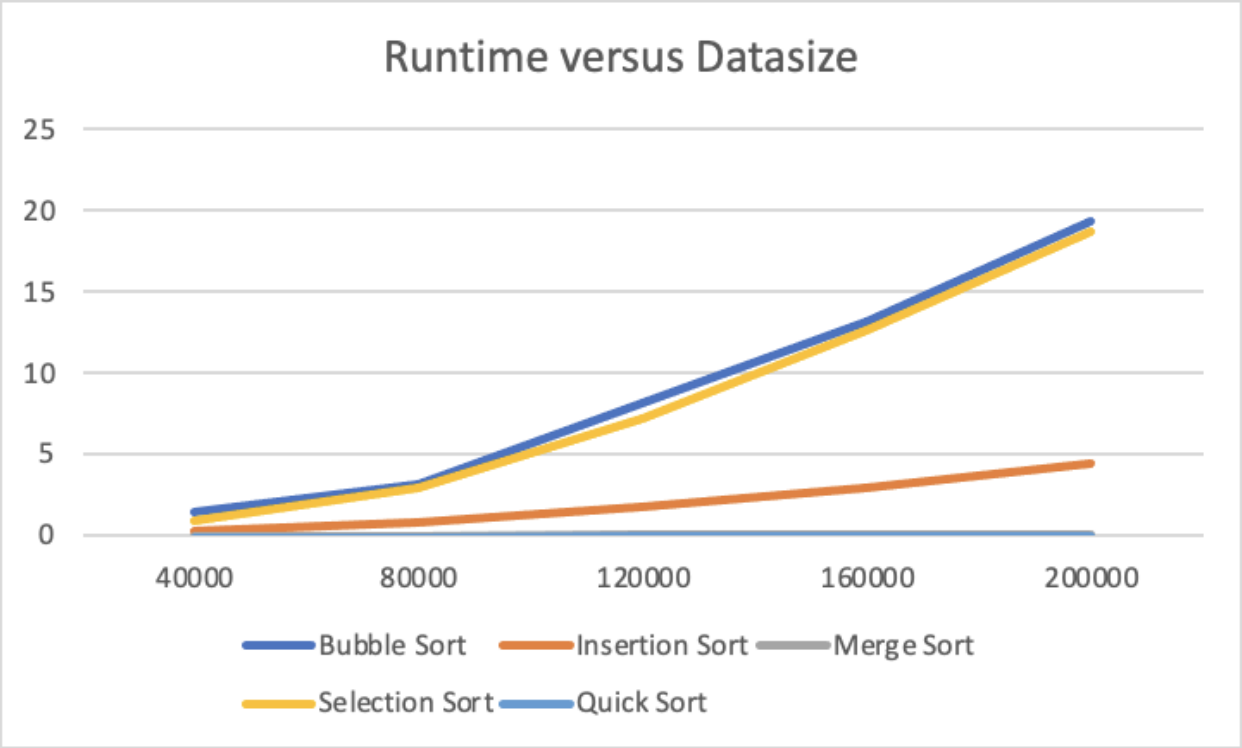
The above graph shows that merge sort is having worse space complexity as compared to other sorting algorithms. Bubble sort is followed by selection sort. But if compared to bubble sort, it's better.

Real Dataset 2

Graph1: RunTime versus Datasize

Sort	40000	80000	120000	160000	200000
Bubble Sort	**1.475**	**3.165**	**8.106**	**13.19**	**19.277**
Insertion Sort	**0.251**	**0.765**	**1.744**	**2.961**	**4.385**

Merge Sort	**0.004**	**0.007**	**0.013**	**0.016**	**0.019**
Selection Sort	**0.877**	**2.938**	**7.152**	**12.572**	**18.683**
Quick Sort	**0.003**	**0.003**	**0.005**	**0.008**	**0.01**

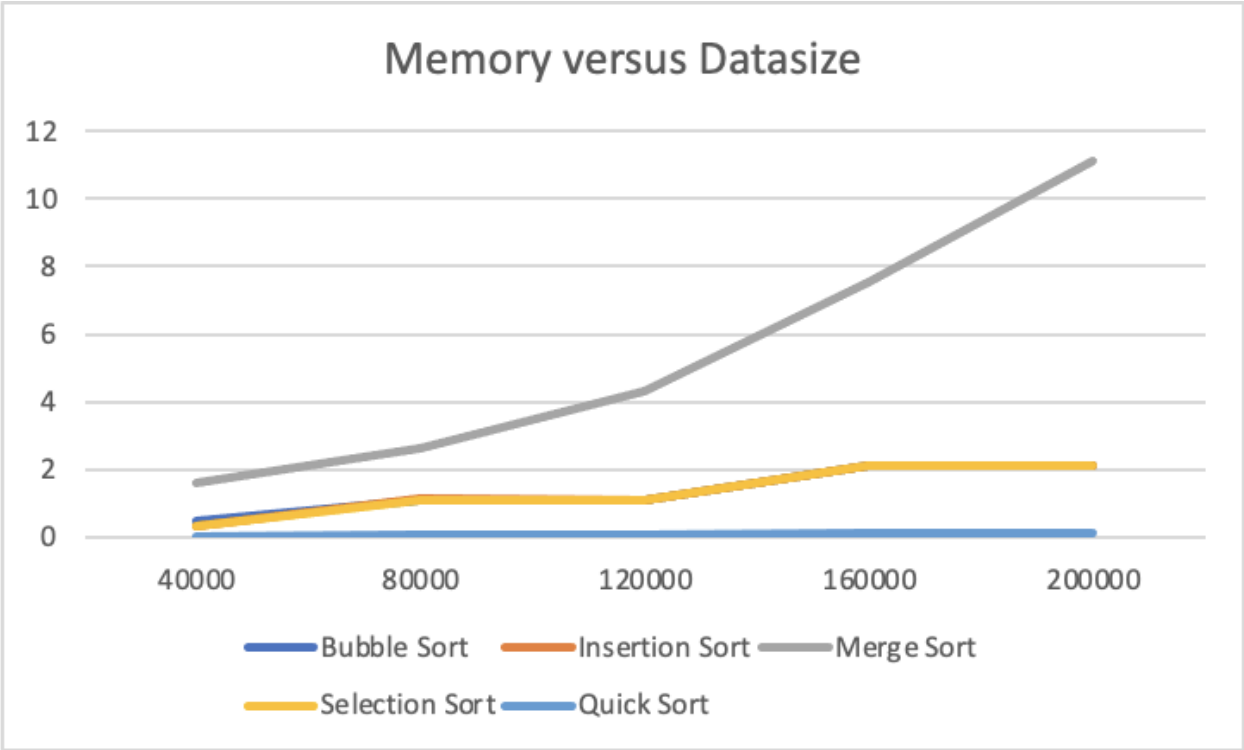


The above graph shows that bubble sort and selection sort, inplace comparison sorting algorithms, are highly inefficient for large datasets. Insertion sort is not highly efficient but not worse too. Merge and quicksort have a tandem performance.

Graph2: Memory versus Datasize

Sort	40000	80000	120000	160000	200000
Bubble Sort	**0.492**	**1.106**	**1.097**	**2.121**	**2.13**
Insertion Sort	**0.327**	**1.15**	**1.097**	**2.143**	**2.13**
Merge Sort	**1.622**	**2.612**	**4.305**	**8.516**	**11.094**

Selection Sort	**0.361**	**1.096**	**1.097**	**2.139**	**2.134**
Quick Sort	**0.056**	**0.096**	**0.097**	**0.136**	**0.134**



The above graph shows that merge sort is having worse space complexity followed by other sorts except quick sort.

Synthetic Dataset 1

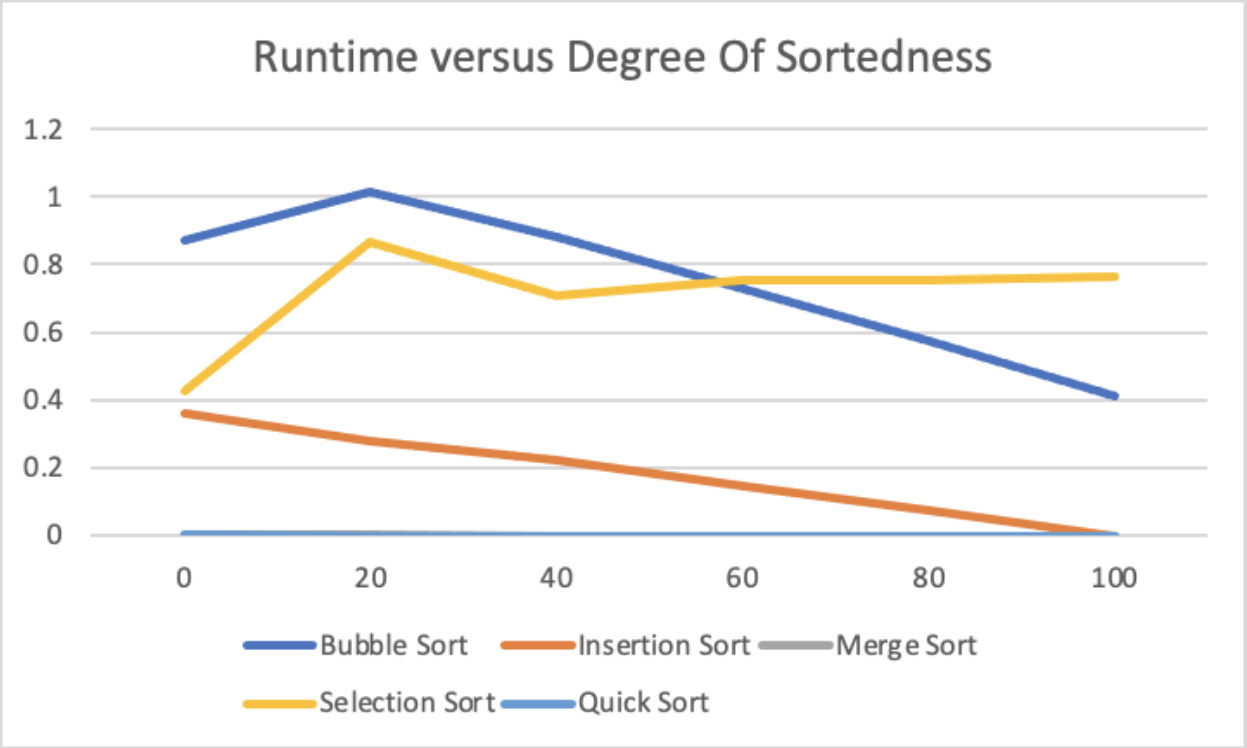
In this simulation, I am comparing a dataset of 40000 and 200000. The simulation for other data sizes can be done in a similar way.

For 40000 dataset:

Graph1: Runtime versus Degree of Sortedness

Sort	0	20	40	60	80	100
Bubble Sort	**0.873**	**1.015**	**0.883**	**0.729**	**0.577**	**0.413**
Insertion Sort	**0.362**	**0.277**	**0.224**	**0.147**	**0.073**	**0**

Merge Sort	**0.003**	**0.003**	**0.002**	**0.002**	**0.002**	**0.002**
Selection Sort	**0.425**	**0.865**	**0.709**	**0.754**	**0.753**	**0.766**
Quick Sort	**0.002**	**0.001**	**0.001**	**0.001**	**0.001**	**0.001**



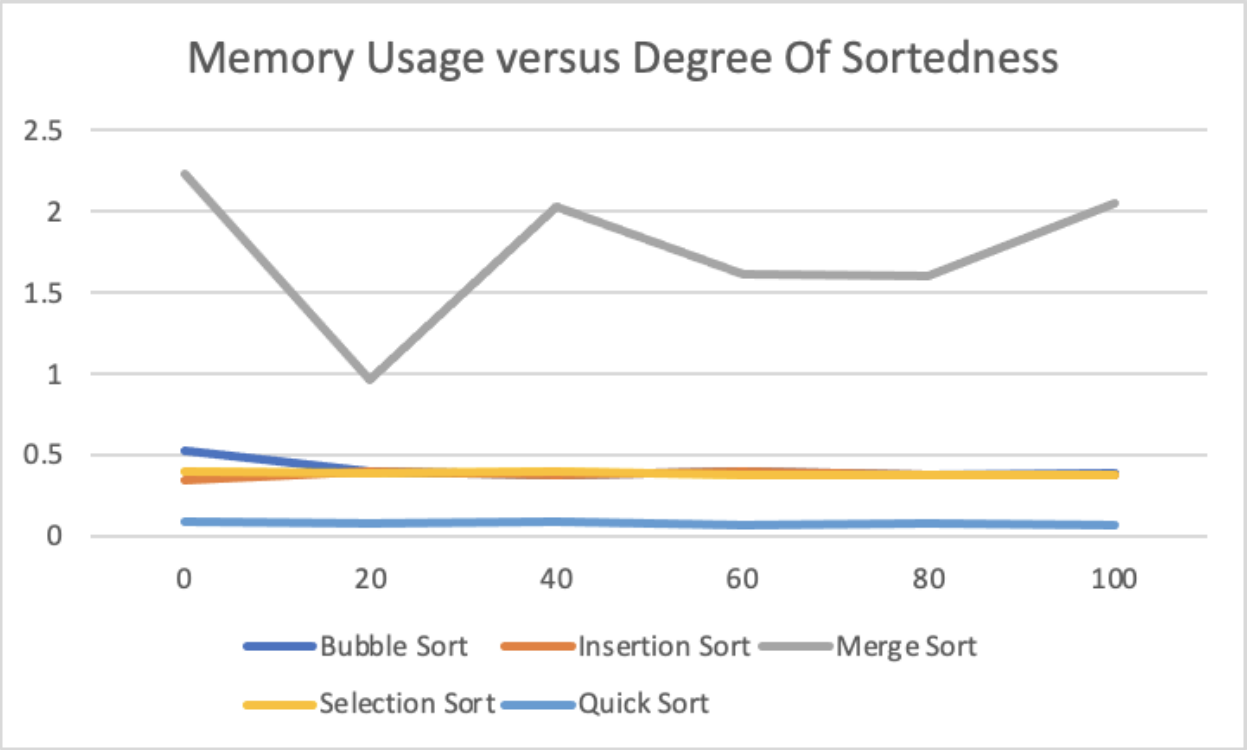
Through the graph I observed following points:

1. *With 0 degree of sortedness, bubble sort is most inefficient followed by selection. But with an increase in degree of sortedness bubble sort is going down while selection sort is performing the same(being stable).*
2. *Merge and quick sort is efficient and stable for any degree of sortedness.*
3. *Insertion sort line is decreasing with increase in degree of sortedness.*

Graph2: Memory versus Degree of Sortedness

Sort	0	20	40	60	80	100
Bubble Sort	**0.529**	**0.397**	**0.384**	**0.397**	**0.378**	**0.388**

Insertion Sort	**0.349**	**0.397**	**0.384**	**0.404**	**0.378**	**0.383**
Merge Sort	**2.227**	**0.959**	**2.032**	**1.614**	**1.606**	**2.049**
Selection Sort	**0.397**	**0.392**	**0.397**	**0.378**	**0.383**	**0.379**
Quick Sort	**0.092**	**0.079**	**0.092**	**0.07**	**0.083**	**0.074**



Through the graph I observed following points:

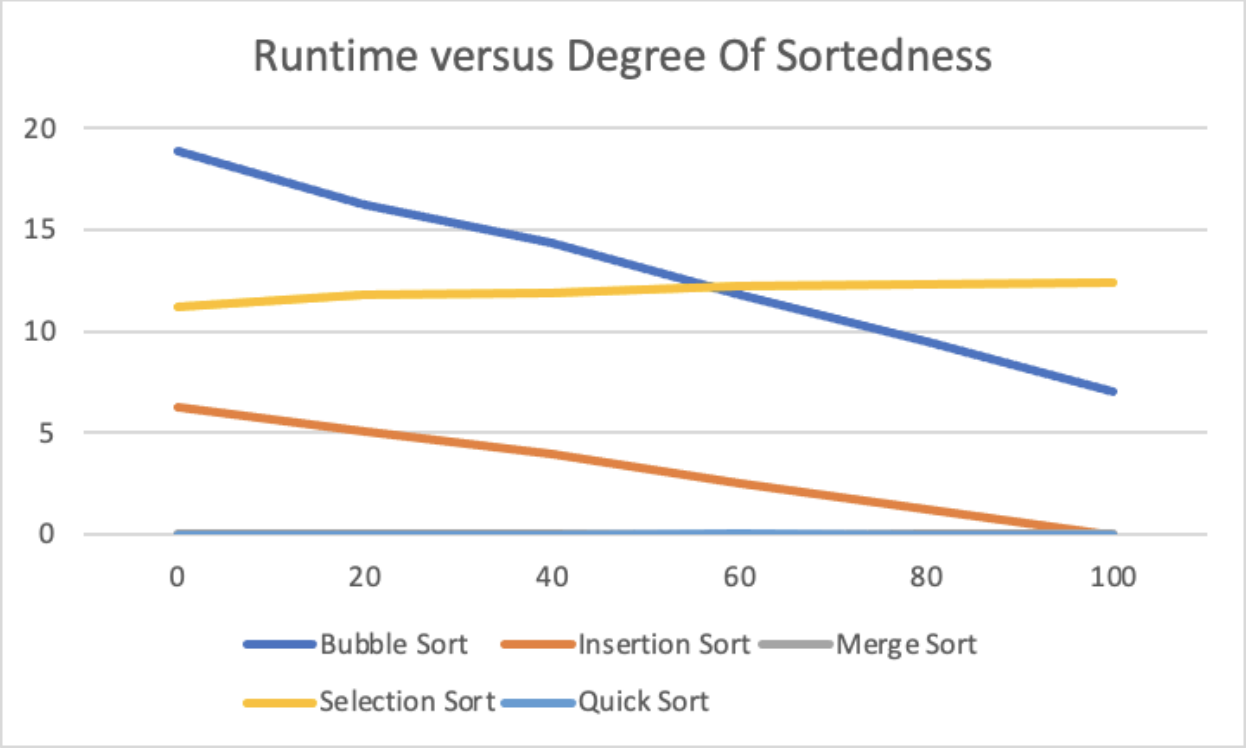
1. *Merge sort requires an auxiliary array due to which memory utilisation for merge sort is much higher than other.*
2. *Insertion, selection, bubble sorts are performing on the same level. Quick sort is efficient.*

For 160000 dataset:

Graph1: Runtime versus Degree of Sortedness

Sort	**0**	**20**	**40**	**60**	**80**	**100**
Bubble Sort	**18.836**	**16.249**	**14.352**	**11.824**	**9.518**	**7.045**

Insertion Sort	**6.258**	**5.038**	**3.931**	**2.55**	**1.273**	**0**
Merge Sort	**0.011**	**0.011**	**0.01**	**0.009**	**0.011**	**0.011**
Selection Sort	**11.215**	**11.806**	**11.911**	**12.218**	**12.285**	**12.381**
Quick Sort	**0.006**	**0.007**	**0.007**	**0.009**	**0.007**	**0.006**



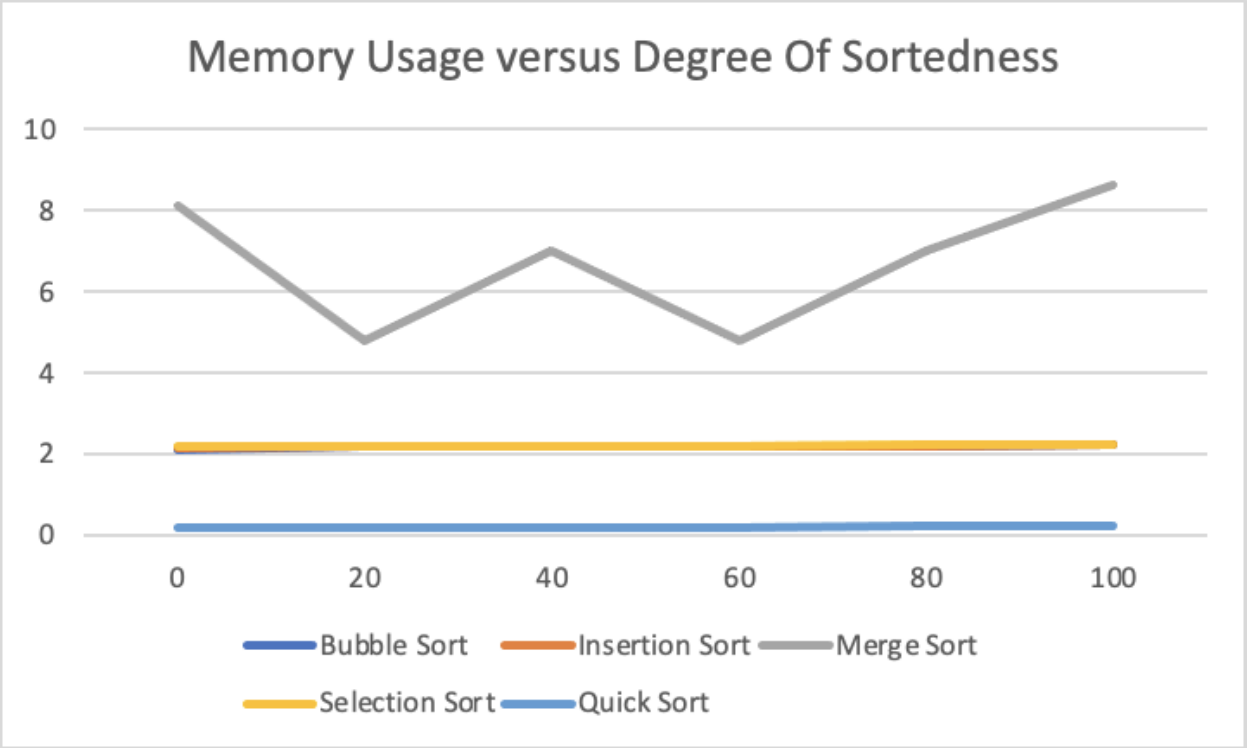
Through the graph I observed following points:

1. *Merge sort and quick sort are efficient*
2. *With a large data set, bubble sort is inefficient followed by selection sort. With an increase in degree of sortedness bubble sort is going down while selection sort is performing the same.*
3. *With increase in sortedness insertion sort is decreased till 0*

Graph2: Memory versus Degree of Sortedness

Sort	0	20	40	60	80	100
Bubble Sort	**2.111**	**2.216**	**2.217**	**2.217**	**2.217**	**2.218**
Insertion Sort	**2.166**	**2.216**	**2.217**	**2.217**	**2.217**	**2.218**

Merge Sort	**8.094**	**4.789**	**7.019**	**4.79**	**7.019**	**8.619**
Selection Sort	**2.216**	**2.217**	**2.217**	**2.217**	**2.218**	**2.218**
Quick Sort	**0.216**	**0.217**	**0.217**	**0.217**	**0.218**	**0.218**



Through the graph I observed following points:

1. *Merge sort requires more storage than other sorting algorithms.*
2. *Bubble, insertion and selection sorts are performing on the same level. Quick sorting takes the least memory.*

Note : All the other data sets are analysed in a similar way and expected to result in the same graphs.

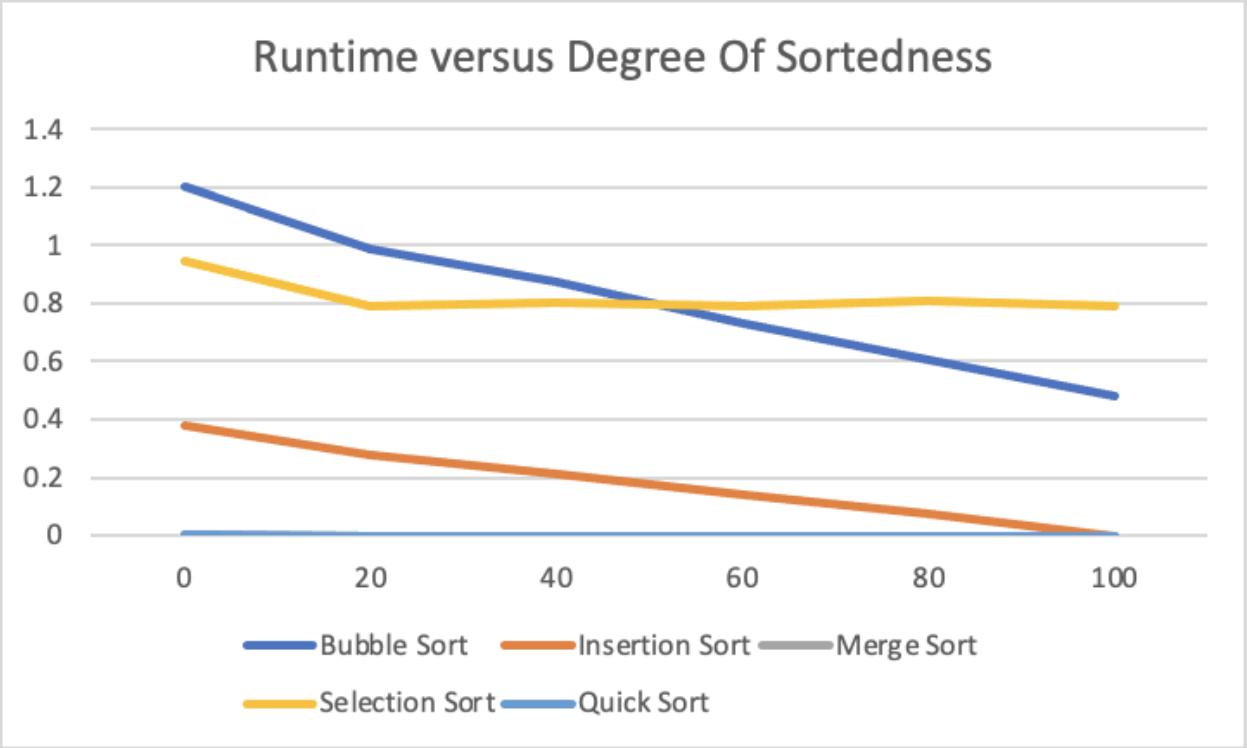
Synthetic Dataset 2

For 40000 dataset:

Graph1: RunTime versus Degree of Sortedness

Sort	**0**	**20**	**40**	**60**	**80**	**100**

Bubble Sort	**1.202**	**0.986**	**0.873**	**0.729**	**0.605**	**0.482**
Insertion Sort	**0.379**	**0.276**	**0.215**	**0.143**	**0.073**	**0**
Merge Sort	**0.003**	**0.002**	**0.002**	**0.002**	**0.002**	**0.002**
Selection Sort	**0.948**	**0.792**	**0.805**	**0.791**	**0.807**	**0.792**
Quick Sort	**0.004**	**0.002**	**0.002**	**0.002**	**0.002**	**0.002**

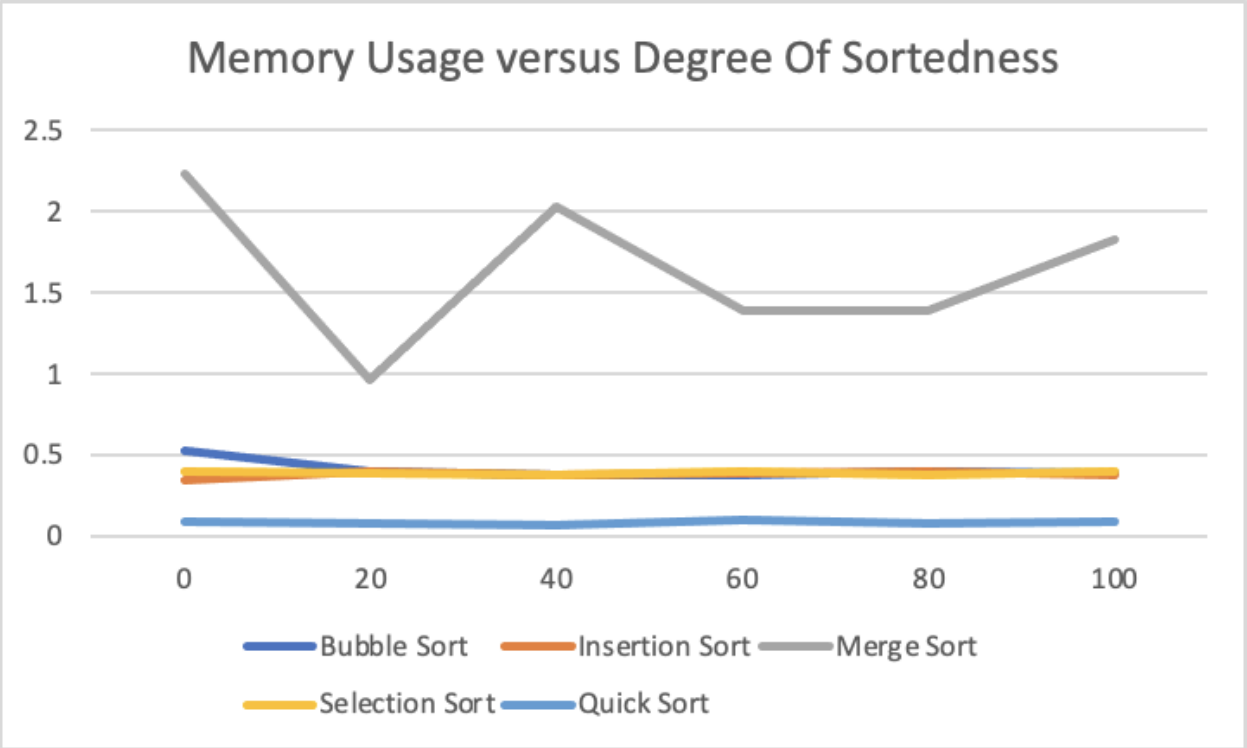


Through the graph I observed following points:

1. *Merge sort and quick sort are efficient*
2. *With a large data set, bubble sort is inefficient followed by selection sort. With an increase in degree of sortedness bubble sort is going down while selection sort is performing the same.*
3. *With increase in sortedness insertion sort is decreased till 0*

Graph2: Memory versus Degree of Sortedness

Sort	0	20	40	60	80	100
Bubble Sort	0.529	0.397	0.384	0.377	0.403	0.386
Insertion Sort	0.349	0.397	0.384	0.385	0.403	0.378
Merge Sort	2.227	0.959	2.029	1.389	1.388	1.822
Selection Sort	0.397	0.392	0.377	0.403	0.378	0.399
Quick Sort	0.092	0.079	0.072	0.097	0.081	0.094



Through the graph I observed following points:

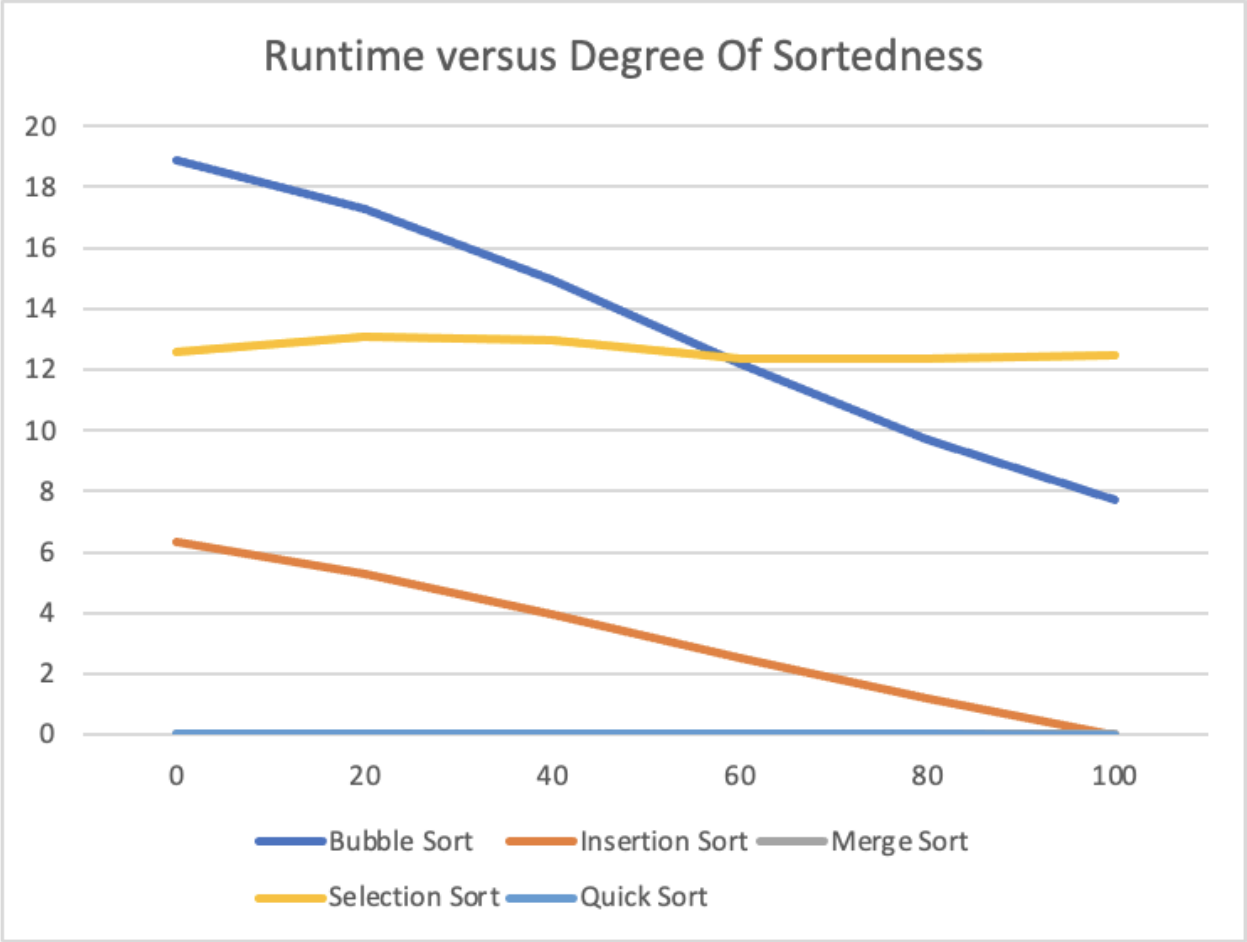
1. Merge sort is highly inefficient in terms of space.
2. *Insertion, selection, bubble sorts are in tandem. Quick sort is efficient.*

For 160000 dataset:

Graph1: RunTime versus Degree of Sortedness

Sort	0	20	40	60	80	100
Bubble Sort	18.891	17.3	14.959	12.183	9.714	7.735

Insertion Sort	**6.323**	**5.264**	**3.943**	**2.519**	**1.209**	**0**
Merge Sort	**0.013**	**0.011**	**0.01**	**0.011**	**0.013**	**0.014**
Selection Sort	**12.568**	**13.063**	**12.984**	**12.382**	**12.383**	**12.465**
Quick Sort	**0.028**	**0.028**	**0.029**	**0.03**	**0.03**	**0.026**

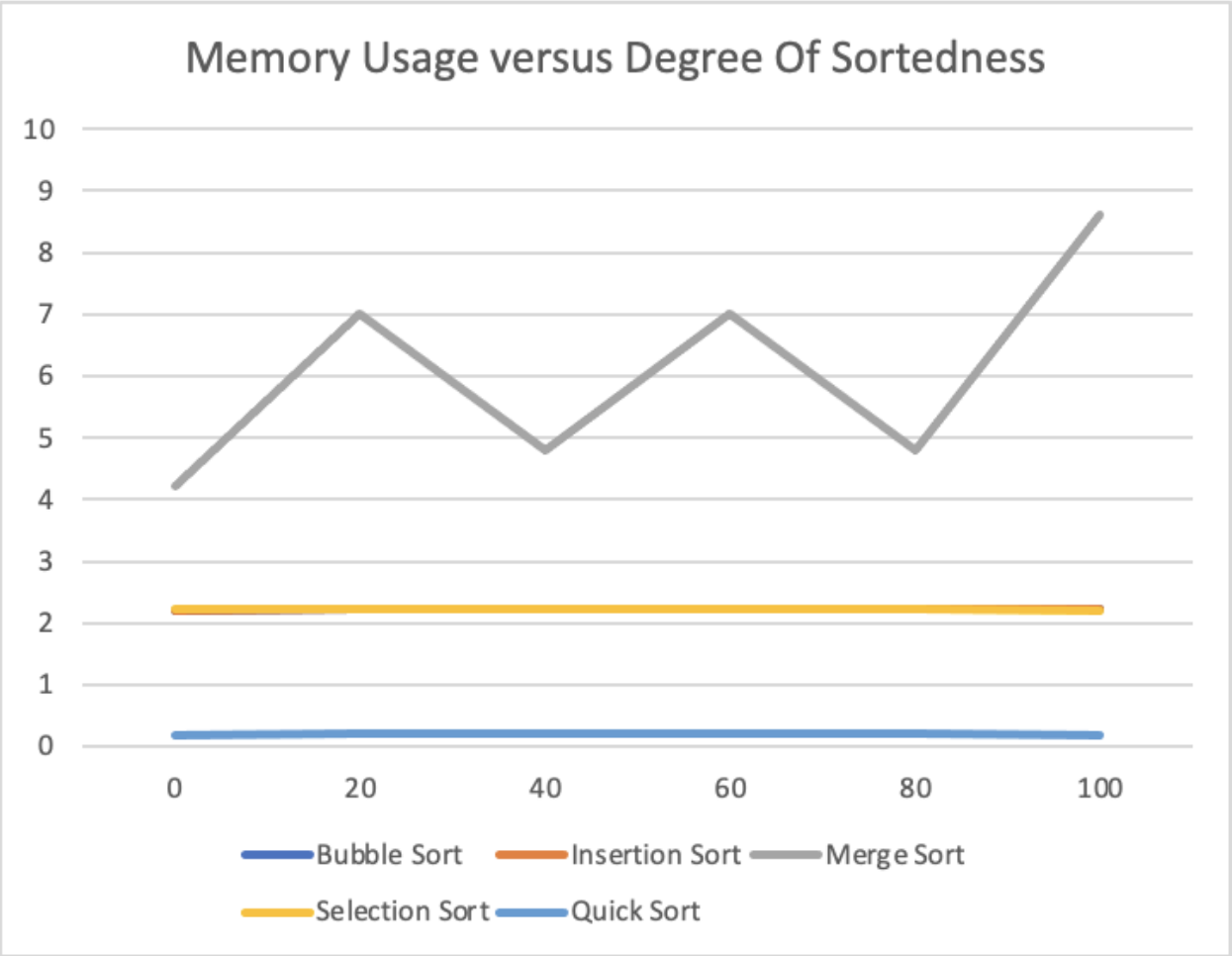


The observation for this graph is the same as the 40000 synthetic dataset2(Runtime versus Data size graph).

Graph2: Memory versus Degree of Sortedness

Sort	**0**	**20**	**40**	**60**	**80**	**100**
Bubble Sort	2.207	2.217	2.218	2.218	2.218	2.218
Insertion Sort	2.207	2.217	2.218	2.218	2.24	2.218

Merge Sort	4.227	7.019	4.789	7.019	4.79	8.619
Selection Sort	2.239	2.24	2.218	2.218	2.218	2.217
Quick Sort	0.217	0.218	0.218	0.218	0.218	0.217



The observation for this graph is the same as the 40000 synthetic dataset2(Memory versus Data size graph).

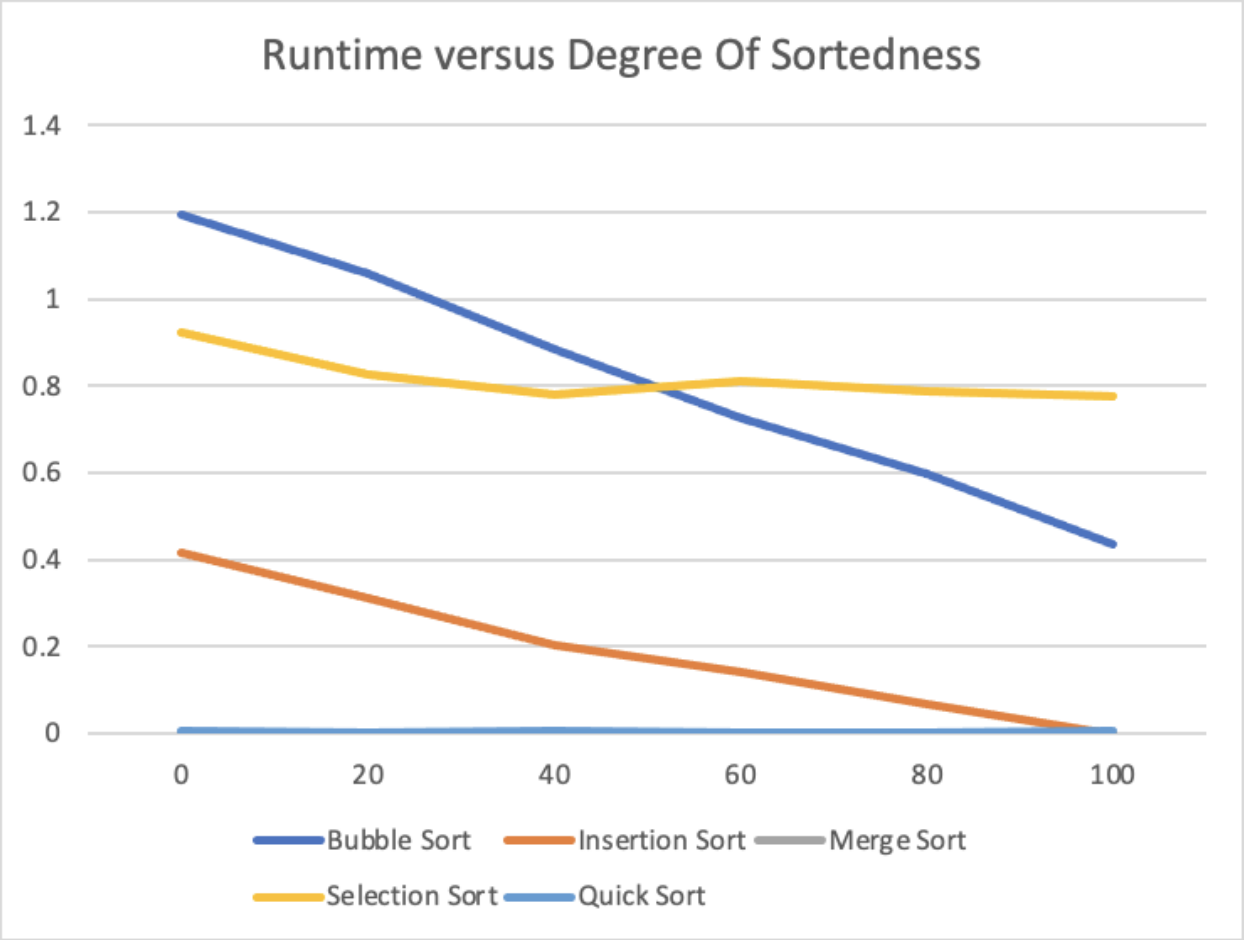
Real Dataset 1

For 40000 dataset:

Graph1: RunTime versus Degree of Sortedness

Sort	0	20	40	60	80	100

Bubble Sort	**1.194**	**1.058**	**0.885**	**0.724**	**0.597**	**0.436**
Insertion Sort	**0.418**	**0.312**	**0.203**	**0.141**	**0.069**	**0**
Merge Sort	**0.004**	**0.002**	**0.002**	**0.002**	**0.002**	**0.002**
Selection Sort	**0.923**	**0.825**	**0.781**	**0.812**	**0.788**	**0.775**
Quick Sort	**0.007**	**0.005**	**0.007**	**0.005**	**0.005**	**0.006**

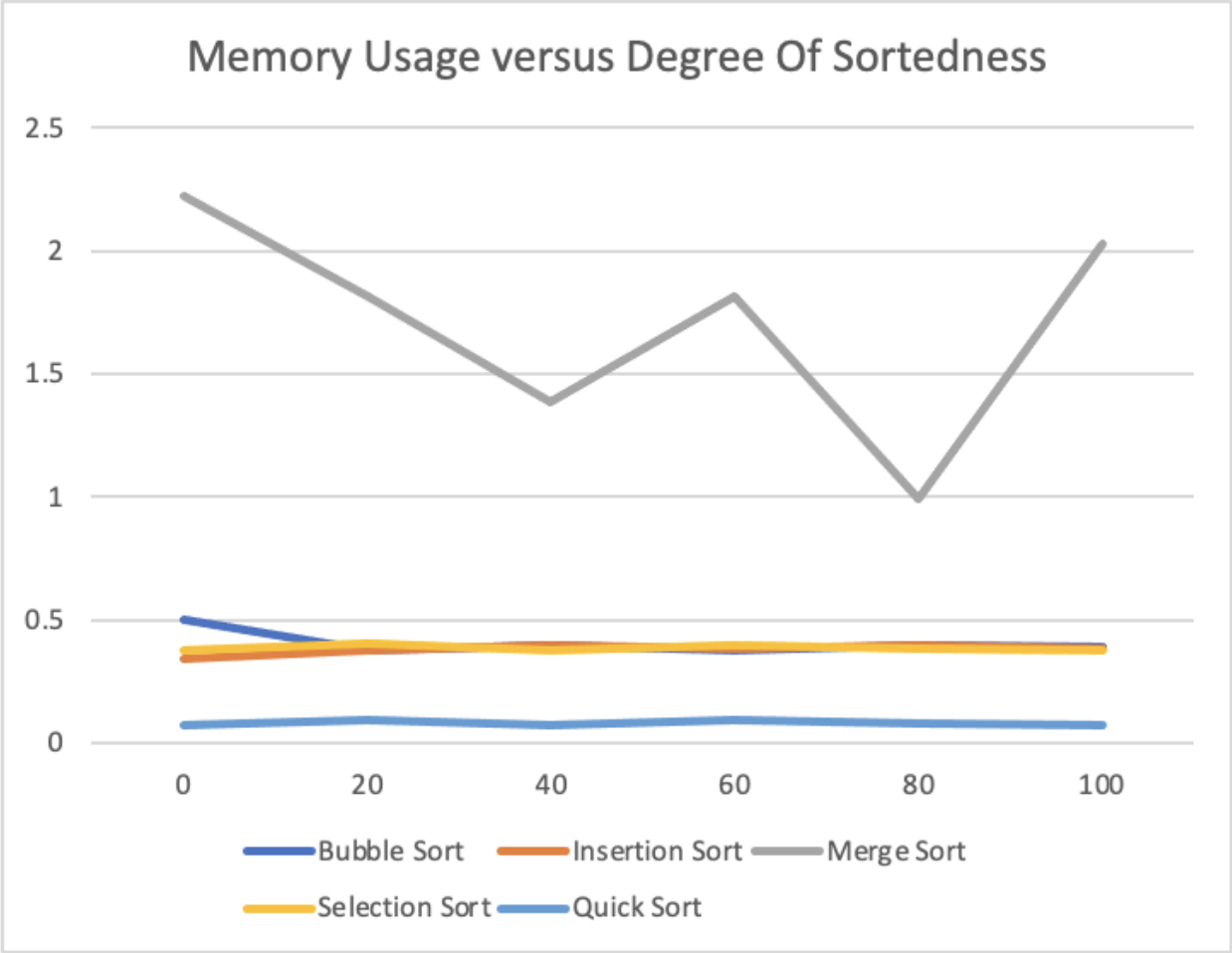


Through the graph I observed following points:

1. *Merge sort and quick sort are efficient*
2. *With increase in sortedness insertion sort is decreased till 0*
3. *With a large data set, bubble sort is inefficient followed by selection sort. But selection sort is giving stable performance irrespective of degree of sortedness.*

Graph2: Memory versus Degree of Sortedness

Sort	0	20	40	60	80	100
Bubble Sort	0.504	0.377	0.4	0.378	0.4	0.389
Insertion Sort	0.34	0.377	0.4	0.384	0.4	0.384
Merge Sort	2.223	1.817	1.387	1.817	0.991	2.029
Selection Sort	0.377	0.406	0.378	0.4	0.384	0.377
Quick Sort	0.072	0.095	0.07	0.095	0.079	0.072



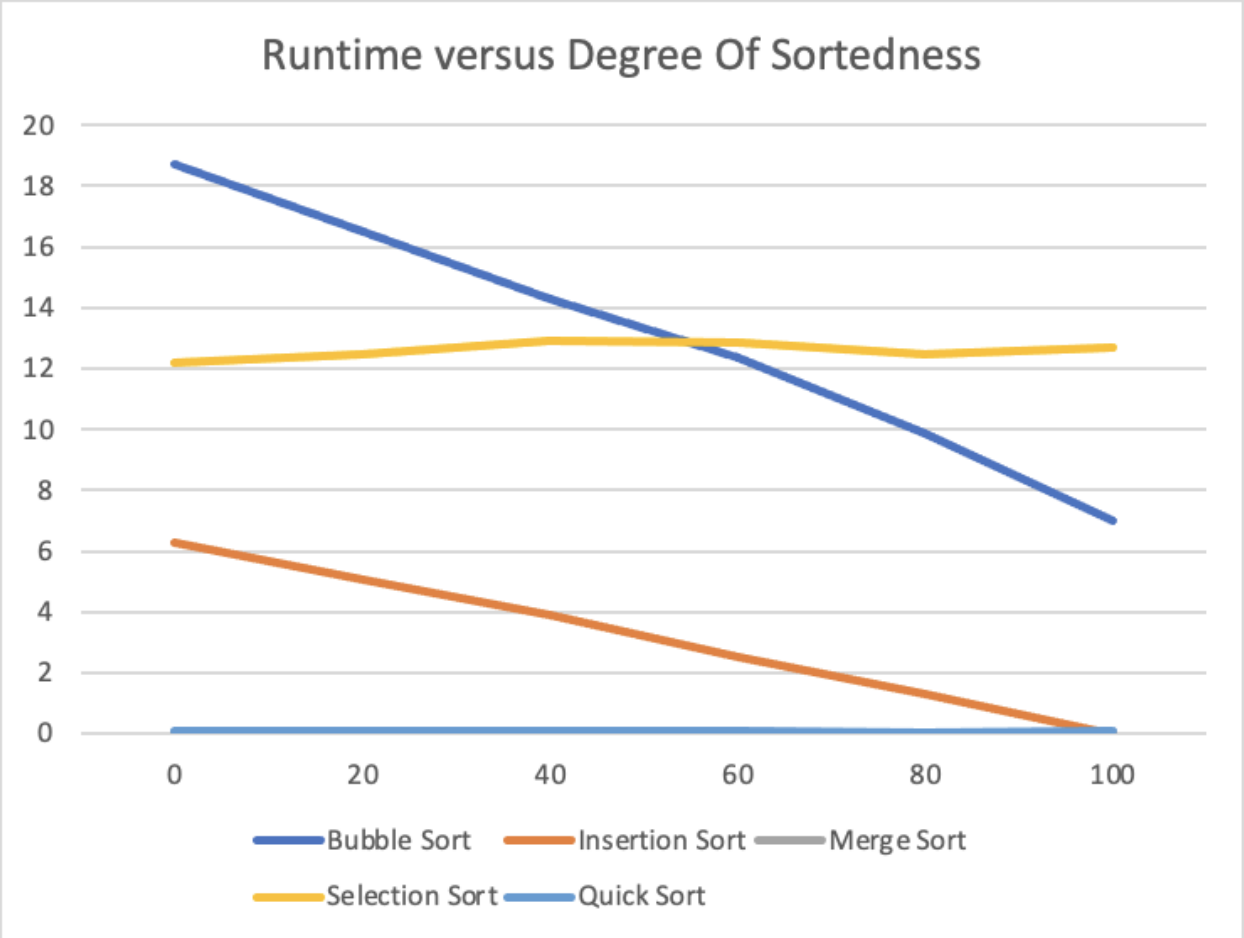
Through the graph I observed following points:

1. *Merge sort is highly inefficient in terms of space.*
2. *Insertion, selection, bubble sorts are in tandem. Quick sort is efficient.*

For 160000 dataset:

Graph1: RunTime versus Degree of Sortedness

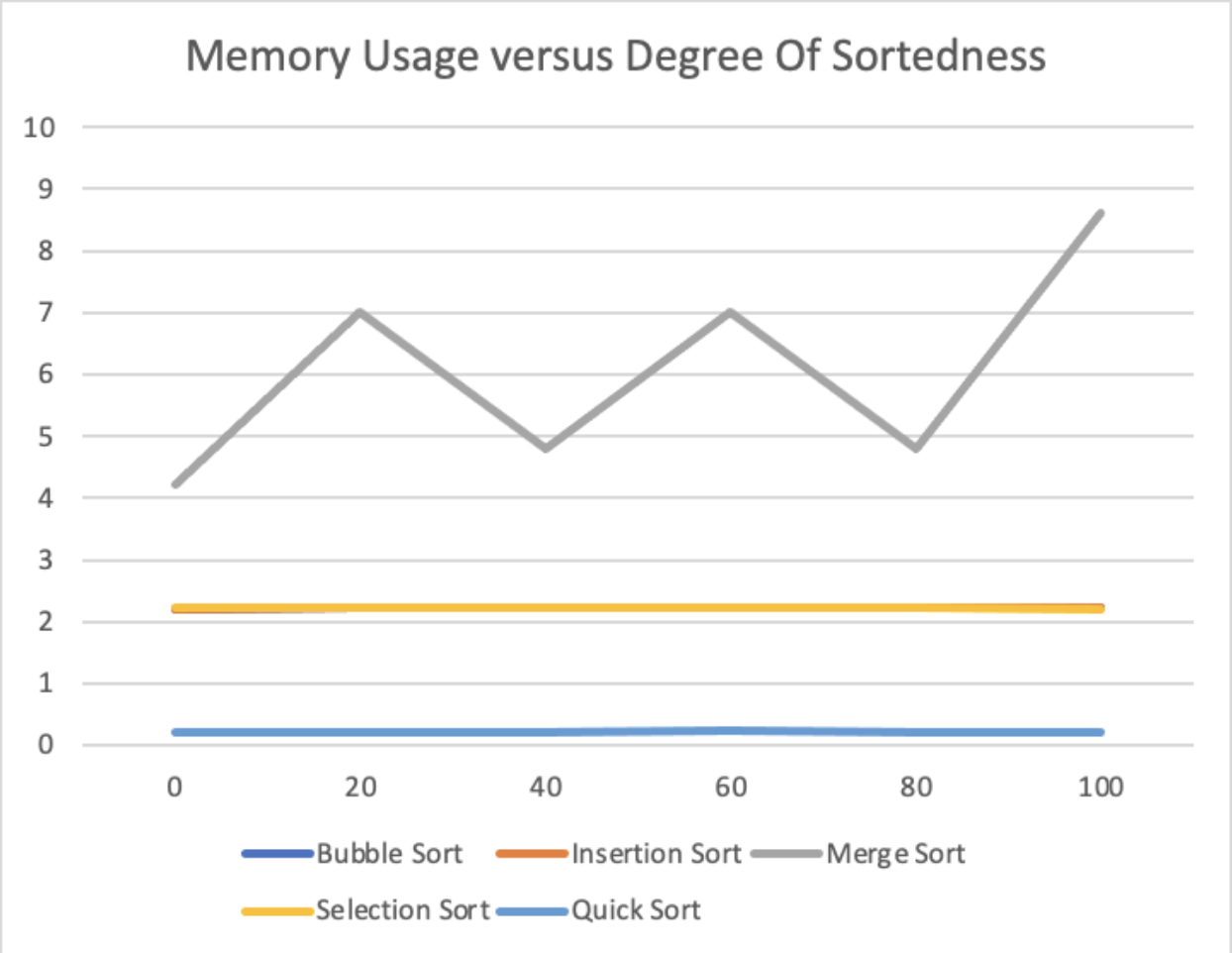
Sort	0	20	40	60	80	100
Bubble Sort	18.705	16.49	14.278	12.374	9.896	7.021
Insertion Sort	6.257	5.082	3.89	2.544	1.286	0
Merge Sort	0.011	0.013	0.01	0.01	0.011	0.012
Selection Sort	12.212	12.458	12.911	12.876	12.472	12.714
Quick Sort	0.089	0.092	0.089	0.09	0.087	0.088



1. *The observation for this graph is the same as the 40000 real dataset1(Runtime versus Data size graph).*

Graph2: Memory versus Degree of Sortedness

Sort	0	20	40	60	80	100
Bubble Sort	2.206	2.217	2.218	2.218	2.218	2.218
Insertion Sort	2.206	2.217	2.218	2.218	2.232	2.218
Merge Sort	4.227	7.019	4.789	7.019	4.79	8.619
Selection Sort	2.231	2.232	2.218	2.218	2.218	2.217
Quick Sort	0.217	0.218	0.218	0.232	0.218	0.217



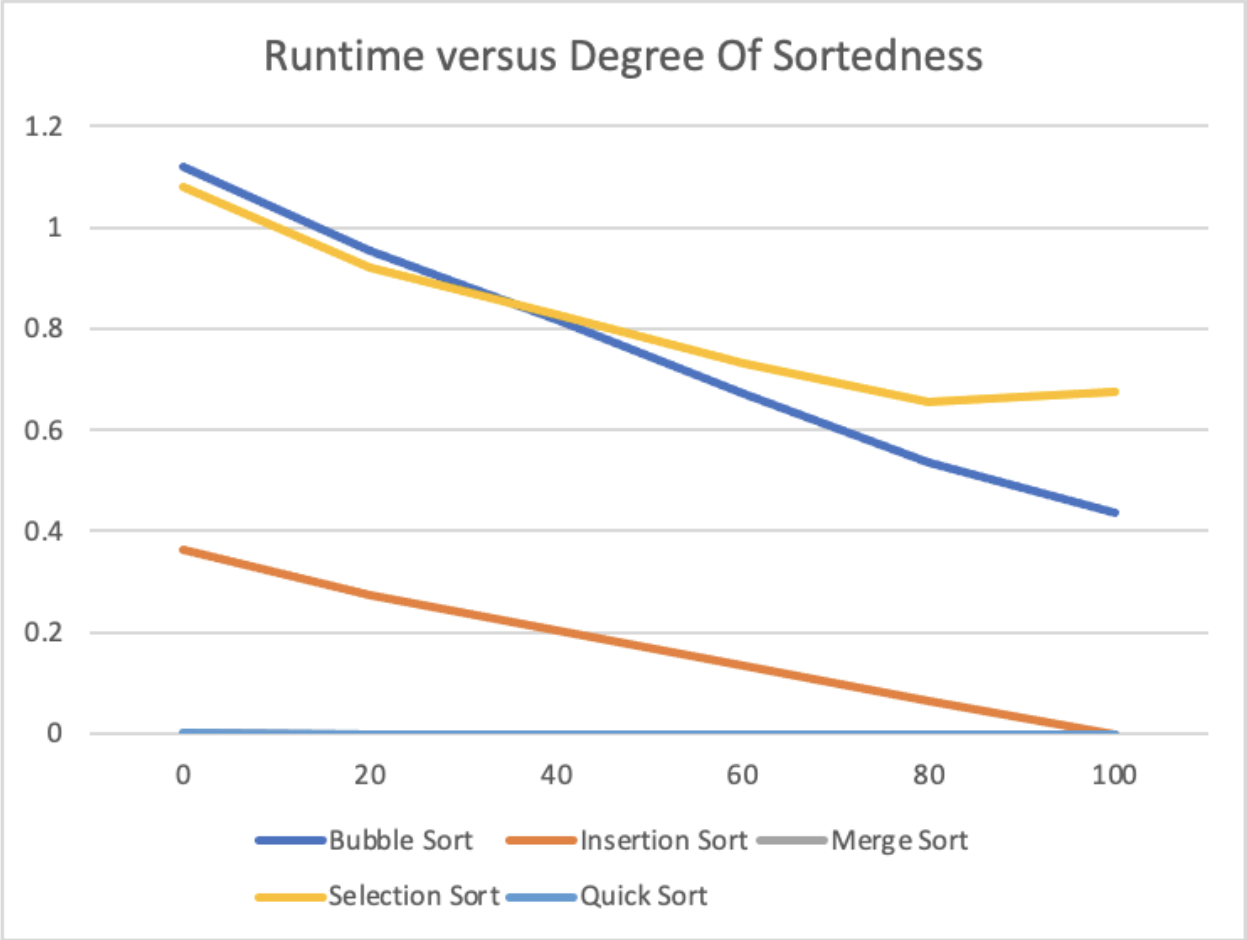
1. *The observation for this graph is the same as the 40000 real dataset(Memory versus Data size graph).*

Real Dataset 2

For 40000 dataset:

Graph1: RunTime versus Degree of Sortedness

Sort	0	20	40	60	80	100
Bubble Sort	1.118	0.953	0.816	0.673	0.535	0.438
Insertion Sort	0.362	0.274	0.204	0.136	0.066	0
Merge Sort	0.003	0.002	0.002	0.002	0.002	0.002
Selection Sort	1.081	0.92	0.827	0.732	0.656	0.675
Quick Sort	0.002	0.001	0.001	0.001	0.001	0.001



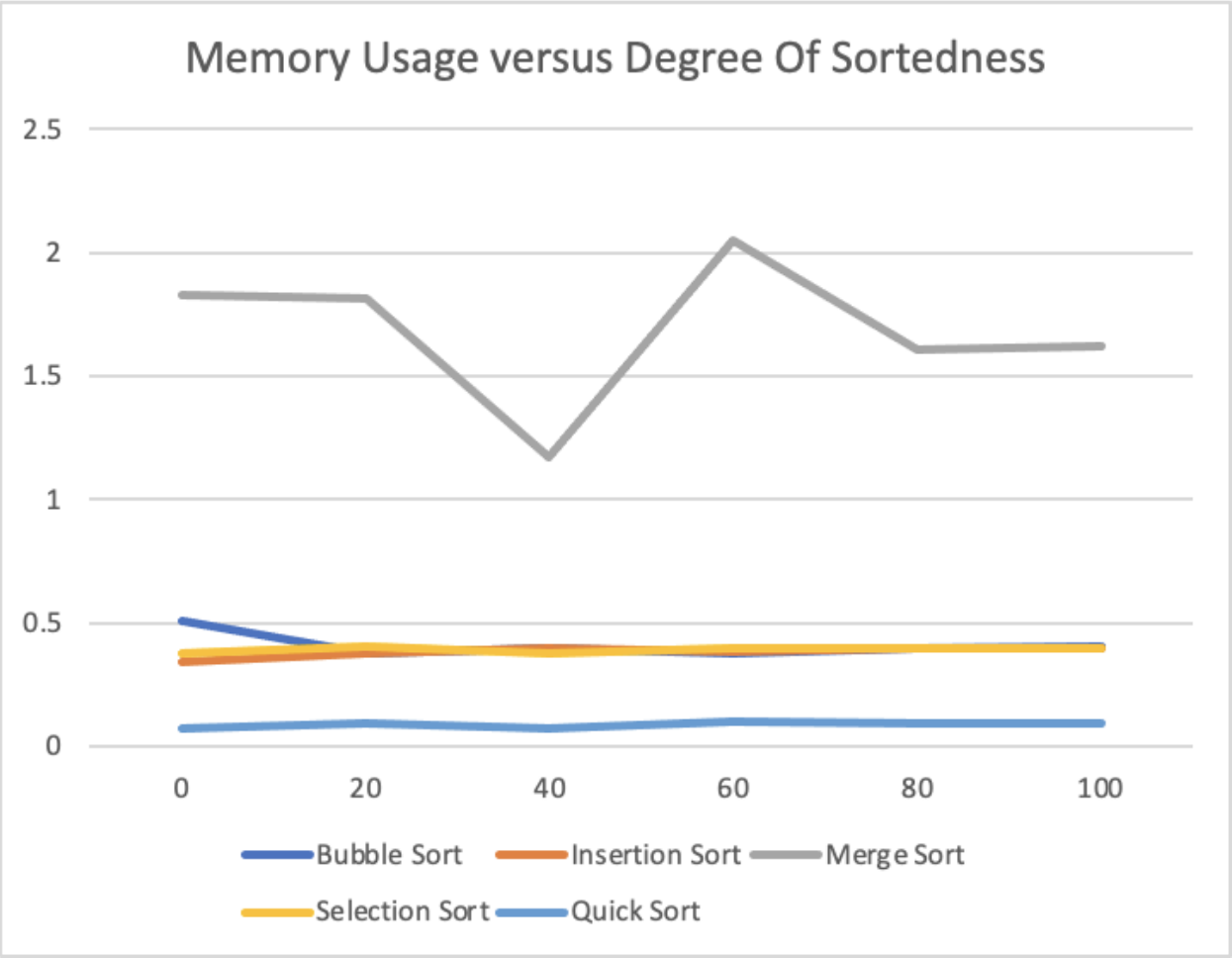
Through the graph I observed following points:

1. *Merge sort and quick sort are efficient*
2. *With a large data set, bubble sort is inefficient followed by selection sort. With an increase in degree of sortedness bubble sort is going down while selection sort is performing the same.*
3. *With increase in sortedness insertion sort is decreased till 0*

Graph2: Memory versus Degree of Sortedness

Sort	0	20	40	60	80	100
Bubble Sort	0.505	0.378	0.4	0.38	0.398	0.406
Insertion Sort	0.34	0.378	0.4	0.385	0.398	0.399

Merge Sort	**1.83**	**1.817**	**1.173**	**2.051**	**1.605**	**1.623**
Selection Sort	**0.378**	**0.406**	**0.38**	**0.398**	**0.399**	**0.401**
Quick Sort	**0.072**	**0.095**	**0.075**	**0.099**	**0.094**	**0.096**



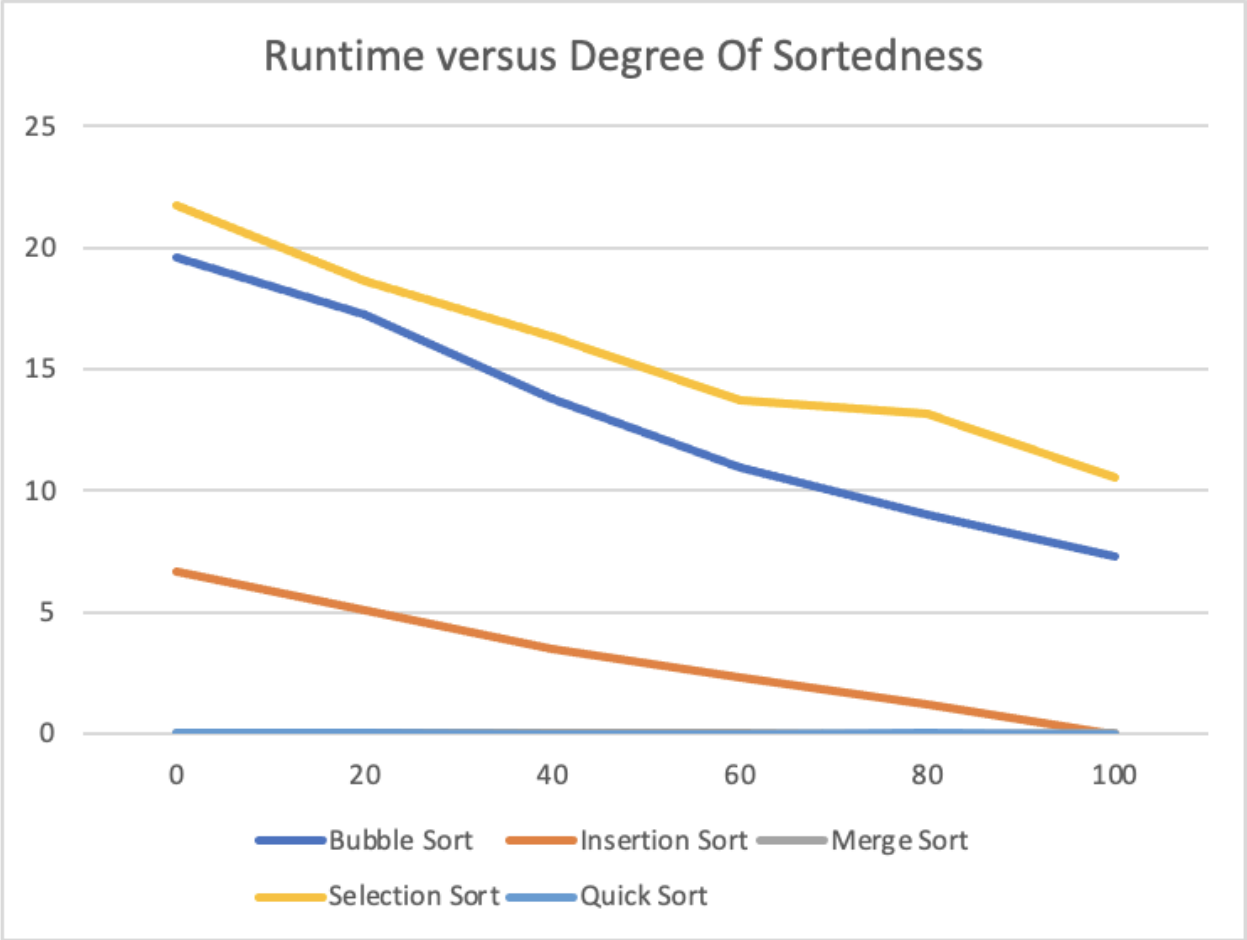
Through the graph I observed following points:

1. *Merge sort is highly inefficient in terms of space.*
2. *Insertion, selection, bubble sorts are in tandem. Quick sort is efficient.*

For 160000 dataset:

Graph1: RunTime versus Degree of Sortedness

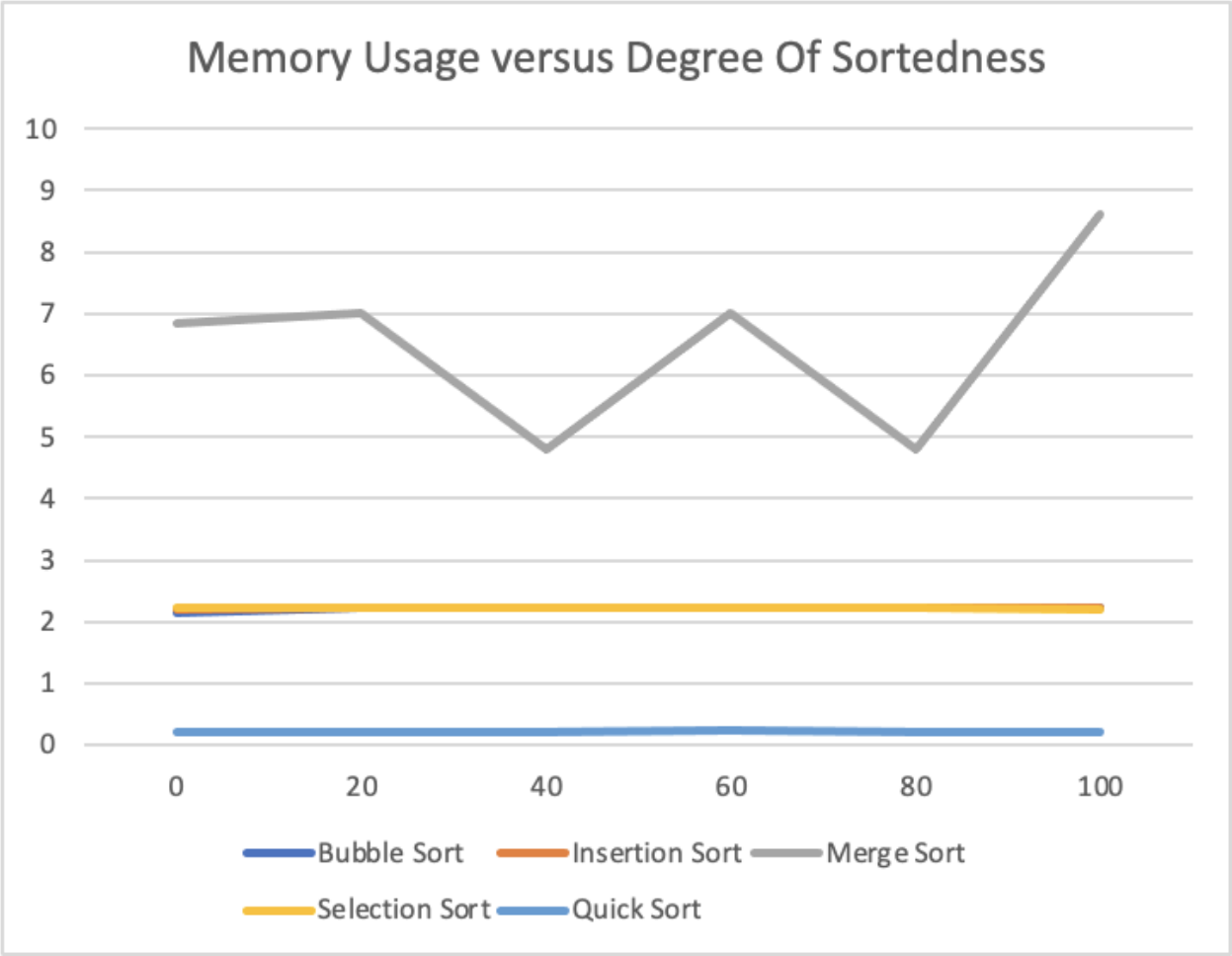
Sort	0	20	40	60	80	100
Bubble Sort	19.612	17.227	13.799	10.932	9.022	7.277
Insertion Sort	6.675	5.113	3.475	2.35	1.189	0
Merge Sort	0.017	0.012	0.01	0.01	0.009	0.011
Selection Sort	21.721	18.621	16.367	13.751	13.132	10.523
Quick Sort	0.008	0.008	0.007	0.007	0.008	0.007



1. *The observation* for this graph is the same as the 40000 real dataset2(Runtime versus Data size graph).*

Graph2: Memory versus Degree of Sortedness

Sort	0	20	40	60	80	100
Bubble Sort	2.139	2.216	2.218	2.218	2.218	2.218
Insertion Sort	2.187	2.216	2.218	2.218	2.232	2.218
Merge Sort	6.84	7.019	4.79	7.019	4.79	8.619
Selection Sort	2.23	2.232	2.218	2.218	2.218	2.217
Quick Sort	0.216	0.218	0.218	0.232	0.218	0.217



1. *The observation for this graph is the same as the 40000 real dataset2(Memory versus Data size graph).*

Analysis of curves

Irrespective of which dataset is being simulated the following observations are drawn from experiment. When started the simulation with large data sets from range 40000 to 200000 and the observations are following:

Runtime versus Datasize:

1. Quicksort and Mergesort are the fastest sorting algorithms for large datasets.

2. Bubble sort and Selection sort are the slowest algorithm for large datasets
3. Insertion sort is seen to be stable irrespective of data size being used but comes under the category of slow algorithm.

Note: For a large dataset, bubble sort is not a practical option to go for. If compared between bubble and selection sort, selection sort is better although both have O(n^2) but selection sort performs less number of swaps than bubble sort. Between merge and quick, mergesort is more efficient. For a small dataset quick sort's performance is better than merge sort.

For reference, in any table of runtime versus datasize, it is visible that for data size 40000 quicksort is doing better but in data size of 200000, merge sort is efficient.

Memory versus Datasize:

1. Mergesort takes the maximum space and this shows space complexity is in O(n).
2. Bubble sort and Selection sort and insertion sort are performing in a similar range and space complexity is in O(1)
3. Quick sort is using less memory if compared to others and space complexity is seen as an O(log n).

Runtime versus Degree of sortedness:

1. Irrespective of what degree of sortedness is bubble sort and selection sort are the slowest algorithm for large datasets
2. Quick and merge sort is efficient.
3. Insertion sort is decreasing with an increase of degree of sortedness being used but comes under the category of slow algorithm.

Memory versus Degree of sortedness:

1. For a large data size of 160000, the performance of merge sort is worse. I observed fluctuations in merge sort simulations. Those oscillations do not change the fact that merge sort is still worse as compared to others.

After considering all the parameters, I conclude the following:

1. I would use merge sort if I need to consider sorting of large datasets. And quick sort if I need to consider a small dataset.
2. If I need to make a tradeoff between memory and run time, I will choose quick sort for all data sizes as space complexity is O(log n) and time complexity is O(n*log n)
3. I differentiate the following sorting algorithm in three categories:
    a. Slow Algorithm: Bubble Sort > Selection Sort
    b. Stable/Slow Algorithm: Insertion Sort
    c. Fast Algorithm:
        i. Mergesort > Quicksort (Large dataset)
        ii. Quicksort > Mergesort (Small dataset)
4. The degree of sortedness may vary but the behaviour of the algorithm will always remain the same. Thus the graphs for any kind of data set will result in a similar graph. This shows how each sorting algorithm would perform if degree of sortedness is specified.

Conclusion on each Sorting Algorithm:

1. **Bubble Sort:**
    - Bubble sort is better than merge sort if we are considering a small data set. With increase in data size, runtime of bubble sort is increasing and that is the reason. It is not practical to use for large data sets.
    - Insertion sort is faster than bubble sort as it considers what happens in each pass.
    - Selection sort is better than bubble sort due to less swapping required.

- Quick sort is highly efficient than bubble sort when data size is greater than 1000.
- Merge sort is better than bubble sort for large data sets. For small data set performance of both are same.

2. **Insertion Sort:**
    - Insertion sort can be used for small datasets. In this simulation, I observed that with a large dataset, It was not efficient but it was stable.
    - Insertion sort is efficient when data set is sorted or close to sorted(refer runtime versus degree of sortedness)
3. **Merge Sort:**
    - Merge sort is better than all other sorting techniques when considering large datasets.
    - Quicksort will outperform merge sort when the data set is small.
4. **Selection Sort:**
    - Insertion sort is faster than selection sort.
    - Selection sort is better than bubble sort
5. **Quick Sort:**
    - Quick sort performs efficiently when data size is greater than 1000. For a very small data set, quick sort is not preferred as it will have overhead.
    - In this simulation quick sort performed efficiently.

Measure of sortedness

Issues

Issue 1: I started the project with the thought that it is easy and can be completed very quickly. I thought I would have enough time to make simulations in java swing. But due to time constraints,

Resolutions: I decided to go with MS EXCEL to get the graphs.

Issue2: While doing simulation quick sort was unstable after 70000 data size. The reason I encountered more recursive calls was generating stack overflow. On analysis, I realised that the algorithm is dependent on choosing a correct pivot.

Resolution: There are multiple methods of choosing pivots but to optimize my code I decided to go for random pivoting.

Issue3: I had data of more than 5 lakhs. While running the whole program, it was taking a lot of time due to which

Resolution: I decided to do the simulation of data between 40000 and 200000.

Issue4: How to define degree of sortedness?

Resolution: Explained below.

Definition for Sortedness

When I started working on a measure of sortedness. I was not sure how I could define it. I came up with multiple definitions but was not satisfied so I decided to model the data to generate data with a specified degree of sortedness. It took a lot of time but the effort was worth it.

Define: In easy words, 0% sortedness implies data is reversely sorted(maximum number of inversion count) and 100% sortedness means the data is completely sorted(ascending order).

Let's take an example to understand how I am modeling the data to a specified degree of sortedness.

Suppose there is an array: 5,4,3,2,1. This array is 0% sorted. To sort this array, let's check how many inversion counts will be needed. The given array will have 10 inversion counts. Inversion count can be calculated by using formula n(n-1)/2 where n is the size of the array.Then, the number of inversion counts is

calculated linearly between 0 to n(n-1)/2. After knowing inversion count, the algorithm will prepare the data with a given degree of sortedness.

Limitations

1. *In real dataset 2, I observed an anomaly that is selection sort is behaving worse than bubble sort. This can be ignored both share same run time complexity
2. The graph lines of quick and merge are not clearly visible as they are very close to the x axis. (refer values in table)

Note*: I am considering the following:*

1. *On the y-axis (a) run time (b) memory usage against the following parameters*
2. *On the x-axis (a) data size, (b) degree of data sortedness (please research a measure of sortedness)*