Submitted By:
YASHASWI TAMTA

SORTING TECHNIQUES

A Detailed Comparison Analysis

NOVEMBER 1, 2016

YASHASWI TAMTA
University of Washington, Tacoma

Table of Contents

INTRODUCTION

Since the dawn of computing, the sorting problem has attracted a great deal of research, perhaps due to the complexity of solving it efficiently despite its simple, familiar statement.

Insertion Sort

Overview

The insertion sort algorithm is the sort unknowingly used by most card players when sorting the cards in their hands. When holding a hand of cards, players will often scan their cards from left to right, looking for the first card that is out of place. For example, if the first three cards of a player's hand are 4, 5, 2, he will often be satisfied that the 4 and the 5 are in order relative to each other, but, upon getting to the 2, desires to place it before the 4 and the 5. In that case, the player typically removes the 2 from the list, shifts the 4 and the 5 one spot to the right, and then places the 2 into the first slot on the left. This is insertion sort. Unlike other simple sorts like selection sort and bubble sort which rely primarily on comparing and swapping, the insertion sort achieves a sorted data set by identifying an element that out of order relative to the elements around it, removing it from the list, shifting elements up one place and then placing the removed element in its correct location. Follow the step by step process of sorting the following small list.

Selection Sort

Overview

The Selection Sort is a very basic sort. It works by finding the smallest element in the array and putting it at the beginning of the list and then repeating that process on the unsorted remainder of the data. Rather than making successive swaps with adjacent elements like bubble sort, selection sort makes only one, swapping the smallest number with the number occupying its correct position.

Consider the following unsorted data: 8 9 3 5 6 4 2 1 7 0. On the first iteration of the sort, the minimum data point is found by searching through all the data; in this case, the minimum value is 0. That value is then put into its correct place at the beginning of the list by exchanging the places of the two values. The 0 is swapped into the 8's position and the 8 is placed where the 0 was, without distinguishing whether that is the correct place for it, which it is not.

Now that the first element is sorted, it never has to be considered again. So, although the current state of the data set is 0 9 3 5 6 4 2 1 7 8, the 0 is no longer considered, and the selection sort repeats itself on the remainder of the unsorted data: 9 3 5 6 4 2 1 7 8.

For the running time we have $O(n^2)$ for the calls to minindex and $O(n)$ for swap. Therefore the algorithm has an average case and best case of $O(n^2)$

Bubble Sort

Overview

Because of bubble sort's simplicity, it is one of the oldest sorts known to man. It based on the property of a sorted list that any two adjacent elements are in sorted order. In a typical iteration of bubble sort each adjacent pair of elements is compared, starting with the first two elements, then the second and the third elements, and all the way to the final two elements. Each time two elements are compared, if they are already in sorted order, nothing is done to them and the next pair of elements is compared. In the case where the two elements are not in sorted order, the two elements are swapped, putting them in order.

Consider a set of data: 5 9 2 8 4 6 3. Bubble sort first compares the first two elements, the 5 and the 6. Because they are already in sorted order, nothing happens and the next pair of numbers, the 6 and the 2 are compared. Because they are not in sorted order, they are swapped and the data becomes: 5 2 9 8 4 6 3. To better understand the "bubbling" nature of the sort, watch how the largest number, 9, "bubbles" to the top in the first iteration of the sort.

(5 9) 2 8 4 6 3 --> compare 5 and 9, no swap 5 (9 2) 8 4 6 3 --> compare 9 and 2, swap 5 2 (9 8) 4 6 3 --> compare 9 and 8, swap 5 2 8 (9 4) 6 3 --> compare 9 and 4, swap 5 2 8 4 (9 6) 3 --> compare 9 and 6, swap 5 2 8 4 6 (9 3) --> compare 9 and 3, swap 5 2 8 4 6 3 9 --> first iteration complete

The bubble sort got its name because of the way the biggest elements "bubble" to the top. Notice that in the example above, the largest element, the 9, got swapped all the way into its correct position at the end of the list. As demonstrated, this happens because in each comparison, the larger element is always pushed towards its place at the end of the list.

For the running time we have O(n^2). But when the list is already sorted it comes as best case and it becomes O(n)

Merge Sort

Overview

Merge Sort is frequently classified as a "divide and conquer" sort because unlike many other sorts that sort data sets in a linear manner, Merge Sort breaks the data into small data sets, sorts those small sets, and then merges the resulting sorted lists together. This sort is usually more efficient than linear sorts because of the fact that it breaks the list in half repeatedly, thus allowing it to operate on individual elements in only log(n) operations, rather than the usual n 2 . Given the data (4 3 1 2) to sort, Merge Sort would first divide the data into two smaller arrays (4 3) and (1 2). It would then process the sub list (4 3) in precisely the same manner, by recursively calling itself on each half of the data, namely (4) and (3). When merge sort processes a list with only one element, it deems the list sorted and sends it to the merging process; therefore, the lists (4) and (3) are each in sorted order. Merge sort then merges them into the sorted list (3 4). The same process is repeated with sublist (1 2) --it is broken down and rebuilt in to the list (1 2). Merge Sort now has two sorted lists, (4 3) and (1 2) which it merges by comparing the smallest element in each list and putting the smaller one into its place in the final, sorted data set. Tracing how merge sort sorts and merges the subarrays it creates, makes the recursive nature of the algorithm even more apparent. Notice how each half array gets entirely broken-down before the other half does.

Merge sort has aa running time of O(n log n) as the average and worst case.

Quick Sort

Overview

In some way, the quick sort uses a similar idea to the bubble sort in that it compares items and swaps them if they are out of sequence. However, the idea of the quick sort is to divide the list into smaller lists which can then also be sorted using the quick sort algorithm. This is usually done through recursion. Lists of length 0 are ignored and those of length 1 is sorted.

Quick sort, like Merge Sort, is a divide-and-conquer sorting algorithm. The premise of quicksort is to separate the "big" elements from the "small" elements repeatedly. The first step of the algorithm requires choosing a "pivot" value that will be used to divide big and small numbers. Each implementation of quicksort has its own method of choosing the pivot value--some methods are much better than others. The implementation below simply uses the first element of the list as the pivot value. Once the pivot value has been selected, all values smaller than the pivot are placed toward the beginning of the set and all the ones larger than the pivot are placed to the right. This process essentially sets the pivot value in the correct place each time. Each side of the pivot is then quick sorted.

Ideally, the pivot would be selected such that it was smaller than about half the elements and larger than about half the elements. Consider the extreme case where either the smallest or the largest value is chosen as the pivot: when quicksort is called recursively on the values on either side of it, one set of data will be empty while the other would be almost as large as the original data set. To improve the efficiency of the sort, there are clever ways to choose the pivot value such that it is extremely unlikely to end up with an extreme value. One such method is to randomly select three numbers from the set of data and set the middle one as the pivot. Though the comparisons make the sort slightly slower, a "good" pivot value can drastically improve the efficiency of quicksort.

Quick sort like merge sort has an average case of O(n logn) but in worst case it runs at $O(n^2)$. And the choice of pivot affects the performance of this algorithm. It is strongly advised to not choose pivot as the first element.

Description of Data Sets

Synthetic

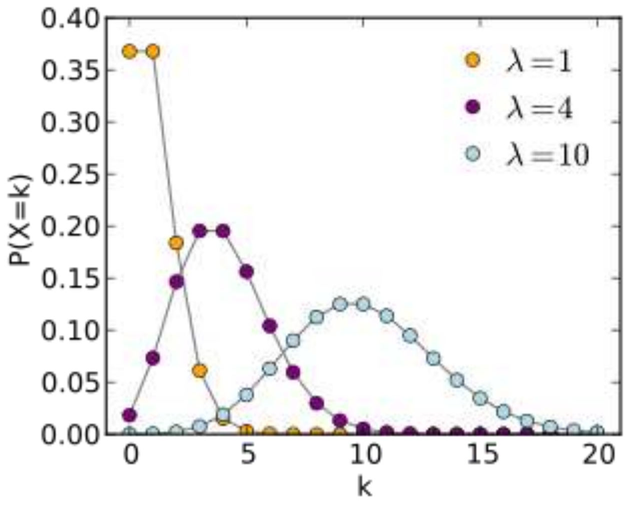
The two Synthetic data sets used are primarily derived from discrete uniform and continuous normal distribution. (Note: a discrete distribution is characterized by a limited number of possible observations as opposed to a continuous distribution which has an infinite number of outcomes.) “A discrete Uniform distribution is a probability distribution whereby a finite number of equally spaced values are equally likely to be observed; every one of n values has equal probability 1/n.” [1] A simple example would be throwing a die.

A continuous Normal Distribution, on the other hand, is a bell-shaped frequency distribution. Here the probabilities found are between two values of the variable. This probability is represented by the area under the curve. The two parameters for the normal distribution are the mean and the standard deviation. The standard deviation gives a measure of the spread of the values around the mean.

Pic taken from:

(http://bestmaths.net/online/index.php/year-levels/year-12/year-12-topic-list/normal-distribution/)

I use a separate function to generate the distribution wherein I pass the id of the desired distribution for example, 1 for Uniform and 2 for Normal. The mean and standard deviation for the normal distribution are calculated randomly therefore, providing a unique distribution in every function call for all the 5 sorting algorithms. I even implemented a Poisson distribution. The Poisson distribution is used to model the number of events occurring within a given time interval where, Lambda is the event rate, also called the rate parameter.



Pic taken from:

(https://en.wikipedia.org/wiki/Poisson_distribution)

Real Data Set

For the real data set I referred the website of UC Irvine Machine learning repository[2]. I took the data set from the above-mentioned website and filtered the data using a self-made python script and chose a specific column with integer values. More is mentioned in the subsection of Real Data Set-1 and Real Data set -2

Synthetic Data Set

For the synthetic data set I had a myriad of options available on the python docs [5]. To help me decide what suits best, I analyzed the output of many distributions available in python before settling for one continuous distribution and one discrete random variable distribution. One other thing I took note of was the degree to which we can control the generation of distribution. Uniform distribution provided the most flexibility and control and therefore was my first choice.

Measure of Sortedness

I adopted a simple measure of sortedness wherein we compare the concurrent positions of an array/list with their sorted version. For example, let assume a list UL for 'Unsorted-List' and its sorted version as SL. Now, compare the UL[i] and SL[i] throughout the length of the lists and maintain a counter whenever we find a match. To get the percentage, divide the total matches found with the length of the list and multiply with 100. This will give us the degree of sortedness for any list.

Although this is not the only measure but, surveying on all the available measures and deciding which should be used, although interesting, is beyond the scope of this assignment.

Performance Analysis

Data set – 1 (synthetic)

This Data set corresponds to Uniform distribution.

Runtime vs Degree of sortedness:

Here I use an input size of 1000. For the degree of sortedness I generated a sorted percentage of x% where 7 instances of x are fixated around 0,25,33,50,75,90 & 100 % whereas once instance is generated randomly and can occur anywhere form 0-50(Note this is achieved by Poisson Distribution). One critical observation I noted was that the degree of sortedness of a uniform distribution was mostly 0% and scarcely in the range of 1-3 but not more than that. This prompted me to use the uniform distribution as a base case to derive the degree of sortedness. To achieve an x% of degree of sortedness I devised a simplistic method where in the array of size n is divide in to two parts x % n and (1-x) % n. The idea is to make sure that the first part is sorted in the whole array and the remaining elements are not in their 'sorted' positions. This will output the degree of sortedness of the concatenated list as x%.

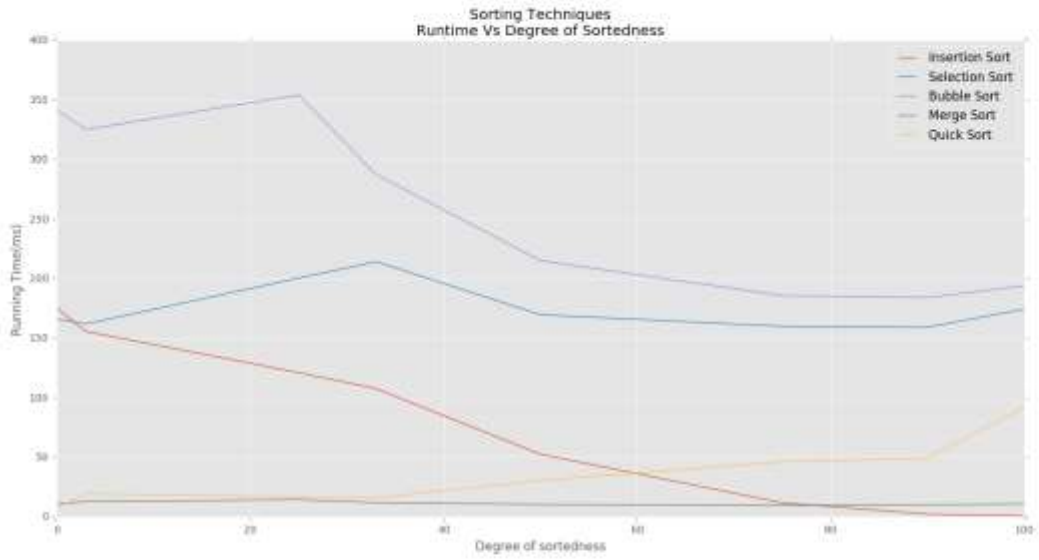
One other advantage of using a uniform distribution was that I could define the intervals to generate the values. i.e. the start and end of the distribution. For a data size of 2000 I can generate a distribution from 0-1000 and sort it and then append it with a raw distribution from 1000-2000. As per my observation on a myriad of runs, a raw uniform distribution mostly results in 0% sortedness with an error of 1-2 %. This way our I can dictate the degree of sortedness of any synthetic array/list and attain a more comprehensive analysis.

The option of generating degree of sortedness for all values in 0-100% is possible but, I deferred from using it, mainly for reducing the computational overhead.

Also, note that all runs plotted on the graph is the average of 5, that can be verified by referring to the code.

Observation

Graph:



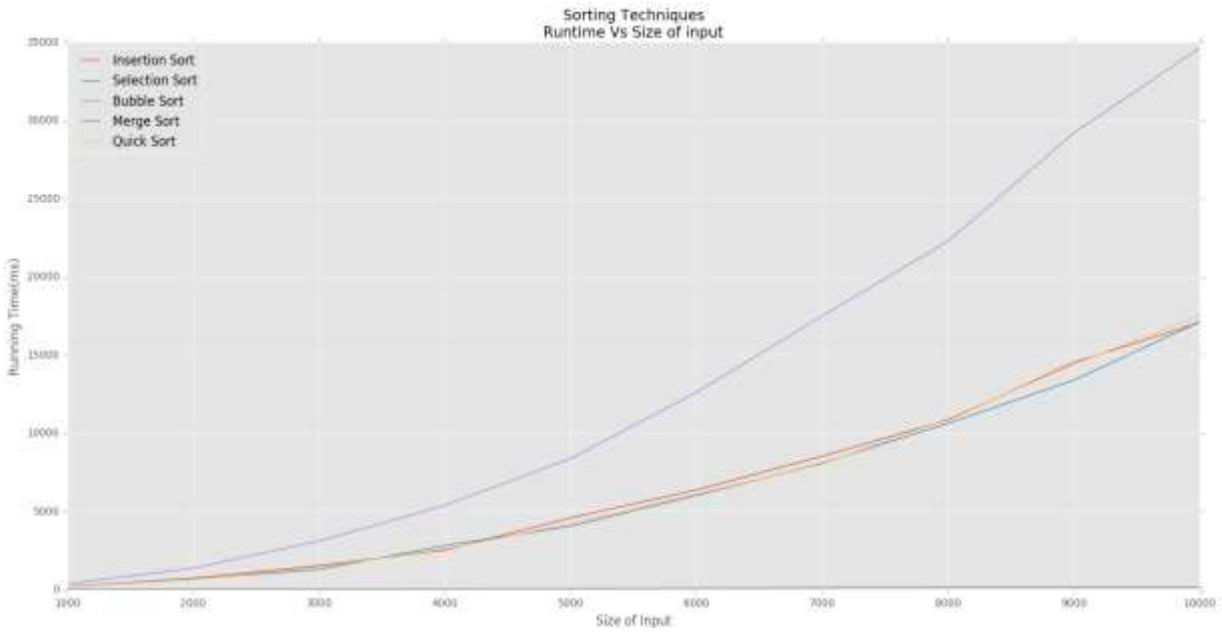
We can observe that the runtime has topped off approximately at 350, taken by bubble sort. We can also observe that as the degree of sortedness increases the runtime of all algorithms except selection and quick sort decreases. And much more drastically for Insertion sort. Bubble sort is observed to have the maximum running time while, merge sort delivers a very good comparative performance.

Runtime vs Size of input:

Here I varied the data set in the range 1000 to 100000 over 10 intervals of 1000 increment each calculated the 'average' running time for every data size.

Observation

Graph:



It's obvious form the graph that as the size of data set increases, the runtime of every sorting algorithms also increases with 'Bubble sort' taking the maximum amount of time. Insertion sort, Selection sort and Quick sort they move in tandem but, even though not directly implacable from the graph, merge sort performs good but it still increases in time. The explanation for its obscurity can be given by the fact that the relative increase in running time of merge sort is very less when compared to that of other sorting techniques.

(Every sorting technique has been validated with real human verification and guaranteed to output a sorted array.)

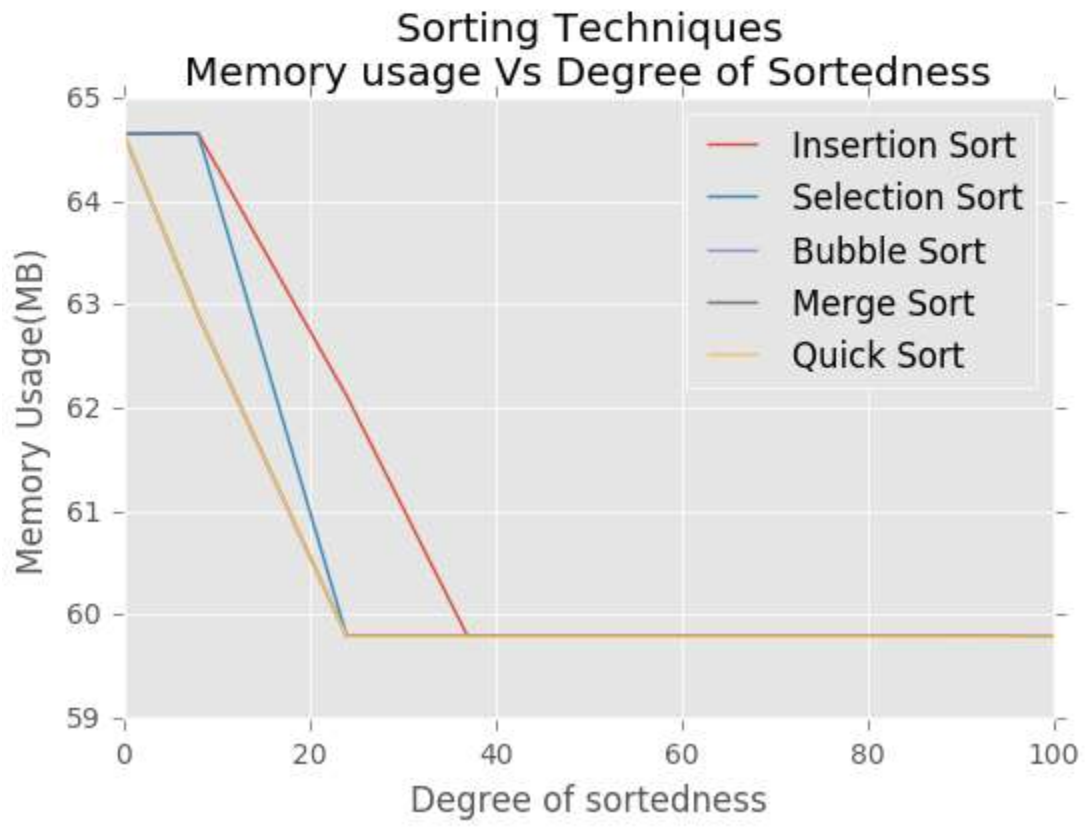
Memory usage vs Degree of sortedness:

The memory usage vs degree of sort is calculated by the using the memory_profiler package in python. We use the memory_usage function on each sorting technique. There were many complications involved in calculating memory, specifically in python. But somehow, it was completed and executed using Jupyter.

Methods like psutil were also tried but they resulted in incomprehensible values and readings. In the list returned by the memory_usage function we select only the relevant *rss* value which is the 'Resident Set Size'.

Observation:

Graph:



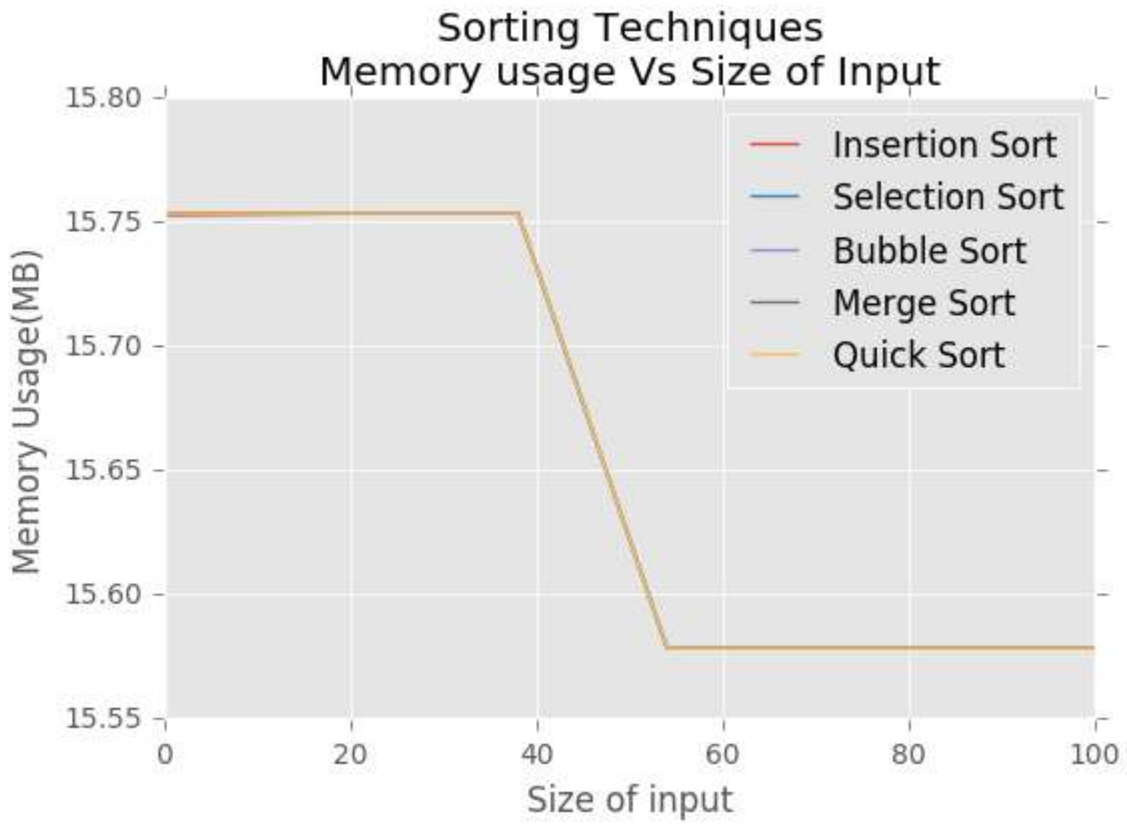
We can observe that the memory usage is very relative to the size of variable stored and processed, we can clearly see the differences implicated by the algorithms in term of using the memory and generating or creating variables. Python interpreter, as we all know, stores the variables dynamically and therefore, causes a major splurge in the usage.

Memory sage vs Size of input:

Due to performance issues of my machine and the unavailability of another machine at that time, I was forced to run the memory analysis on a date size till max = 100.

Observation:

Graph:



Data set – 2 (synthetic)

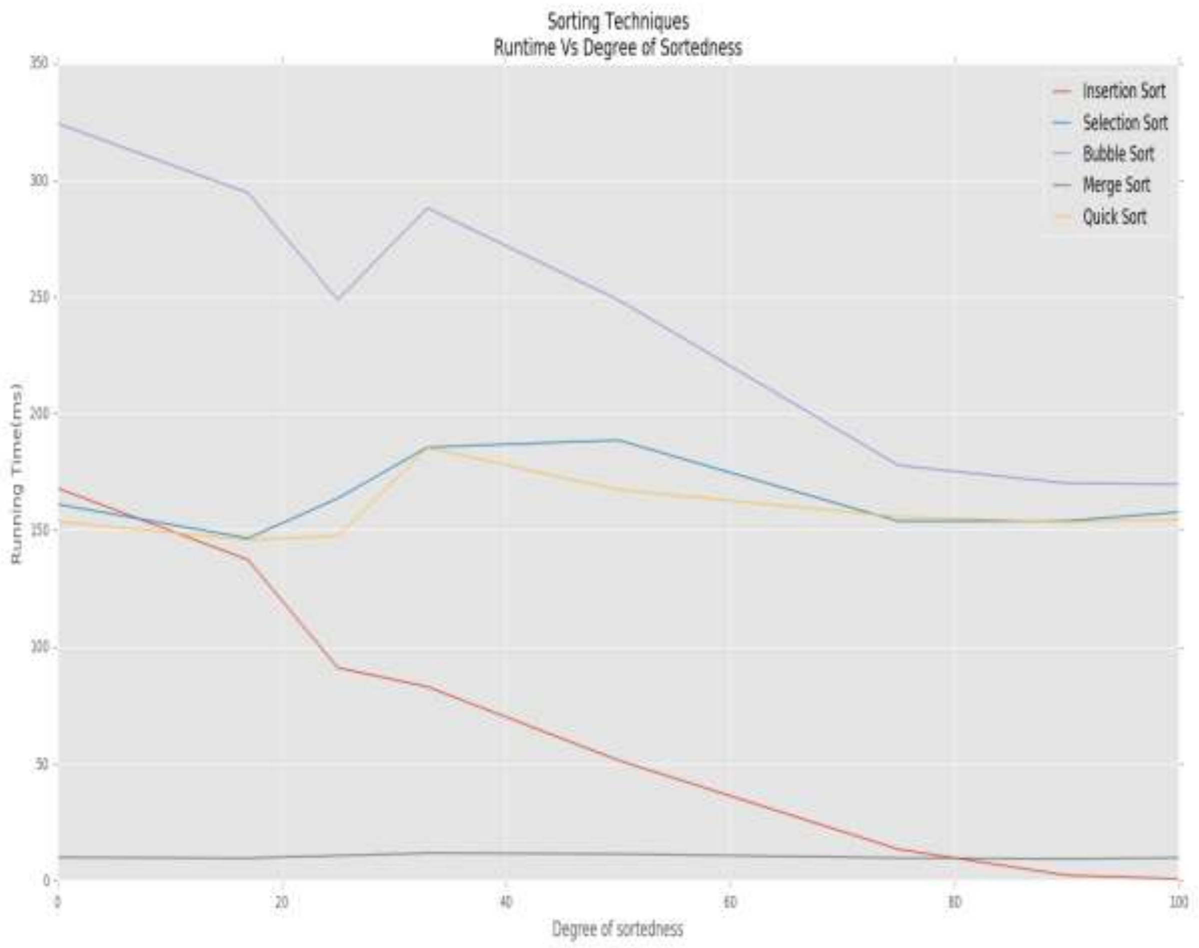
This Dataset corresponds to a normal distribution.

Runtime vs Degree of sortedness:

The size of data set used is 1000. The Degree of sortedness was generated in the same way as it seems efficient enough. We use a normal distribution here for testing purposes. Although in my opinion the value of the data set does not impact the performance unless there is a huge gap in the size of each element. Going down to the assembly level the time taken for comparing two 4-digit numbers shouldn't be different form comparing two 1-digit numbers. But on the other hand, given many recursive runs, comparing two 1-digit numbers will be faster than comparing two 34 digit numbers. But strictly speaking, we rarely find integers of that size in real-life scenarios unless we look at the data in NASA or aerospace related projects, they seem to produce huge integers even after the decimal point.

Observation

<u>Graph:</u>



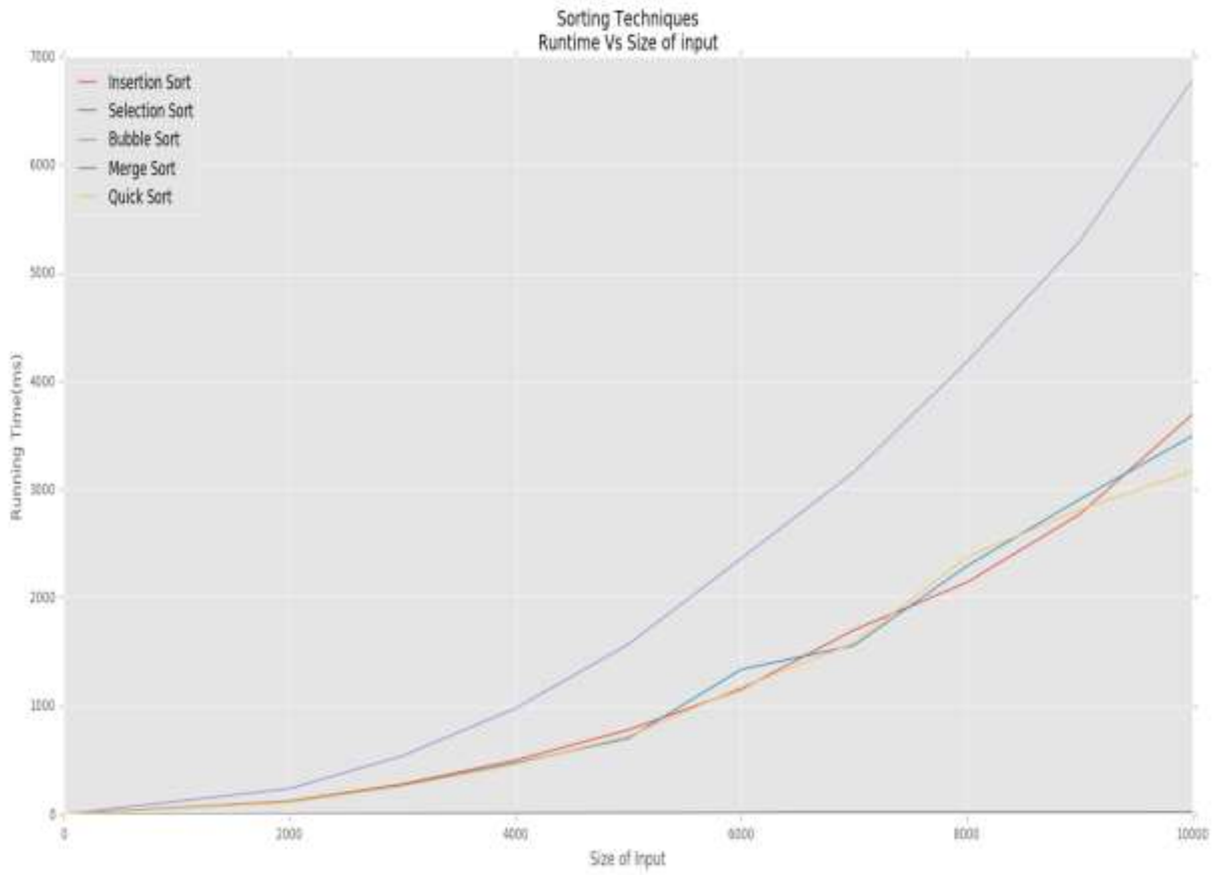
The graph appears different when compared to the one plotted for synthetic data set-1. Although the underlining facts remain intact. All algorithm except Selection sort and Quick sort have a declining trajectory, Insertion sort decline drastically to almost the fastest when the list is 100% sorted, Merge sort maintains a good performance not much affected by the degree of sortedness(relatively). Quick sort seems to be taking much more tie than in the previous case. That explains a very interesting observation on the quick sort algorithm; It's performance largely depends on the pivot position. It is proven that quick sort performs better when the pivot position is randomized at every partition. Even when we randomize it, there is a level of uncertainty attached to it. Sometimes even the randomized pivot cannot lead to an optimal pivot. And hence the fluctuating performance.

Runtime vs Size of input:

To maintain a fair comparison between datasets we use the same increments in data size as before. Starting from 1000 incremented by 1000 and ending at 10000.

Observation

Graph:



As mentioned before, the size of an individual element does not majorly affect the running time of the algorithm due to negligible differences when we compare large integers. We can observe here that, Bubble sort is taking the longest time followed by Insertion sort, Quick sort, Selection sort in tandem. Merge sort takes the least amount of time.

Data set – 3 (real)

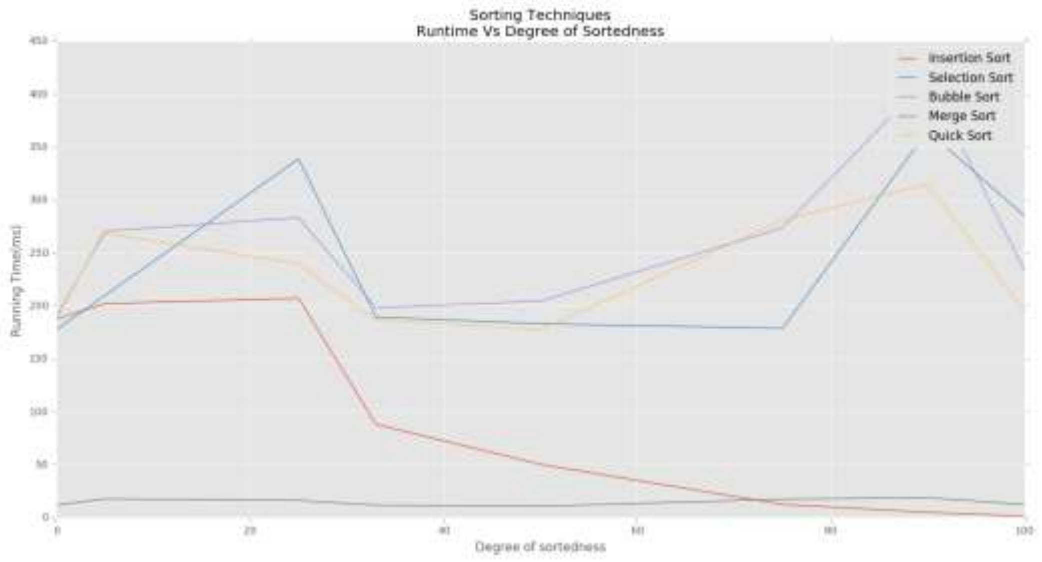
For a real data set I looked up the UCI website of Machine learning repository and took Iris Data set[4]. I imported the data set in to a local script and filtered a specific column thus resulting in an array like structure. In the main python script, I converted the single column file into a list which can be worked processed.

Runtime vs Degree of sortedness:

The run time of a real data set can be computed in a very similar way as before; we plug in the data we extracted by the method mentioned above. Plugging in the data is relatively easy albeit we need to take care of the type conversions.

Observation

<u>Graph:</u>



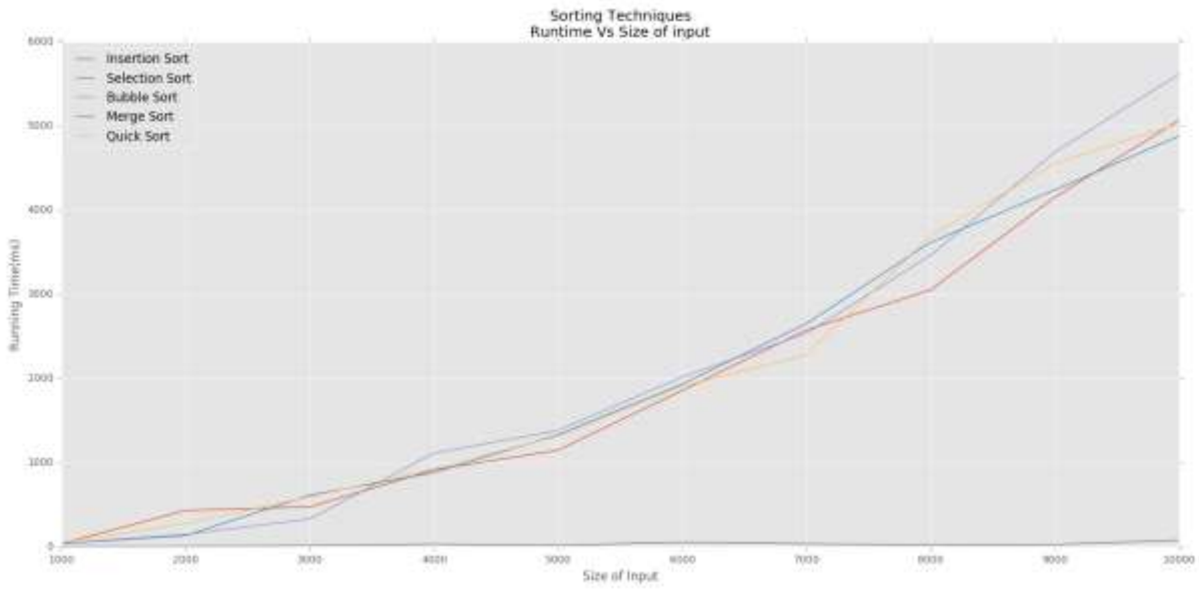
We can observe some unusual spikes in the data but the underlying facts remain consistent

Runtime vs Size of input:

The runtime of even the most efficient algorithm will get affected by the input size. And so is witnessed here.

Observation

<u>Graph:</u>



Particularly for the real data set we see the sorting algorithms move in tandem and take similar amount of time.

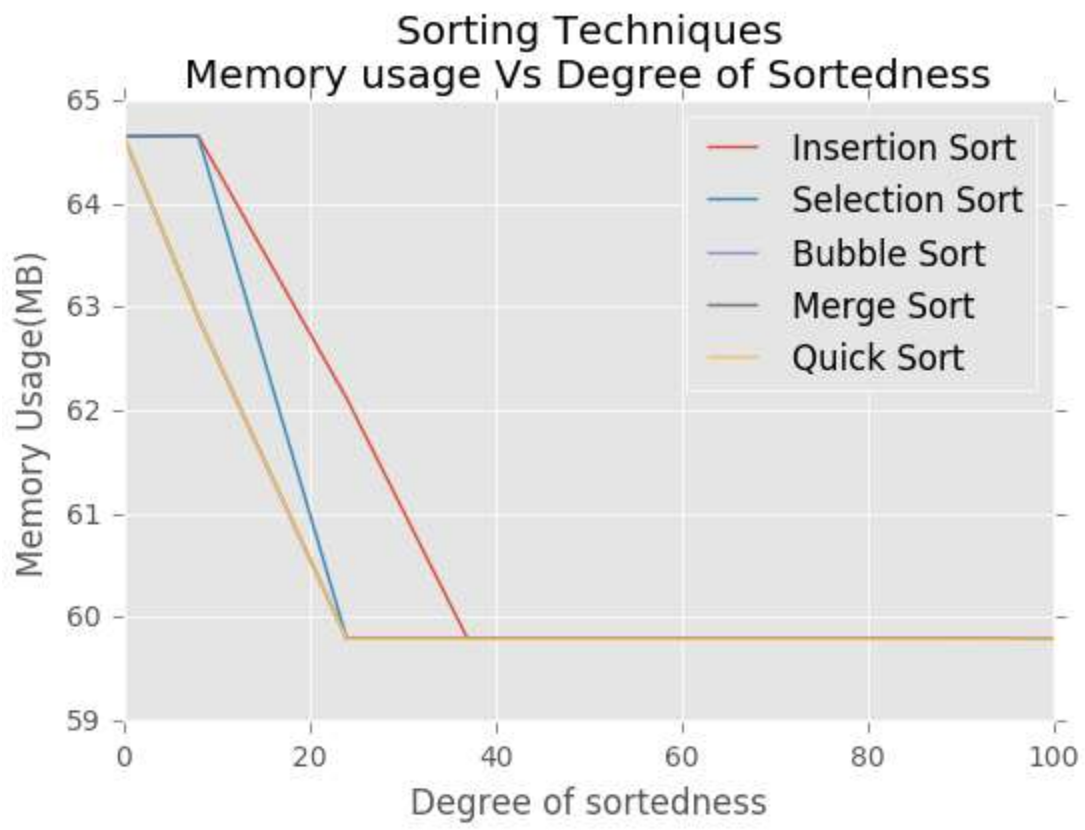
Memory usage vs Degree of sortedness:

The memory usage vs degree of sort is calculated by the using the memory_profiler package in python. We use the memory_usage function on each sorting technique. There were many complications involved in calculating memory, specifically in python. But somehow, it was completed and executed using Jupyter.

Methods like psutil were also tried but they resulted in incomprehensible values and readings. In the list returned by the memory_usage function we select only the relevant *rss* value which is the 'Resident Set Size'.

Observation

<u>Graph:</u>



Memory Usage vs Size of input:

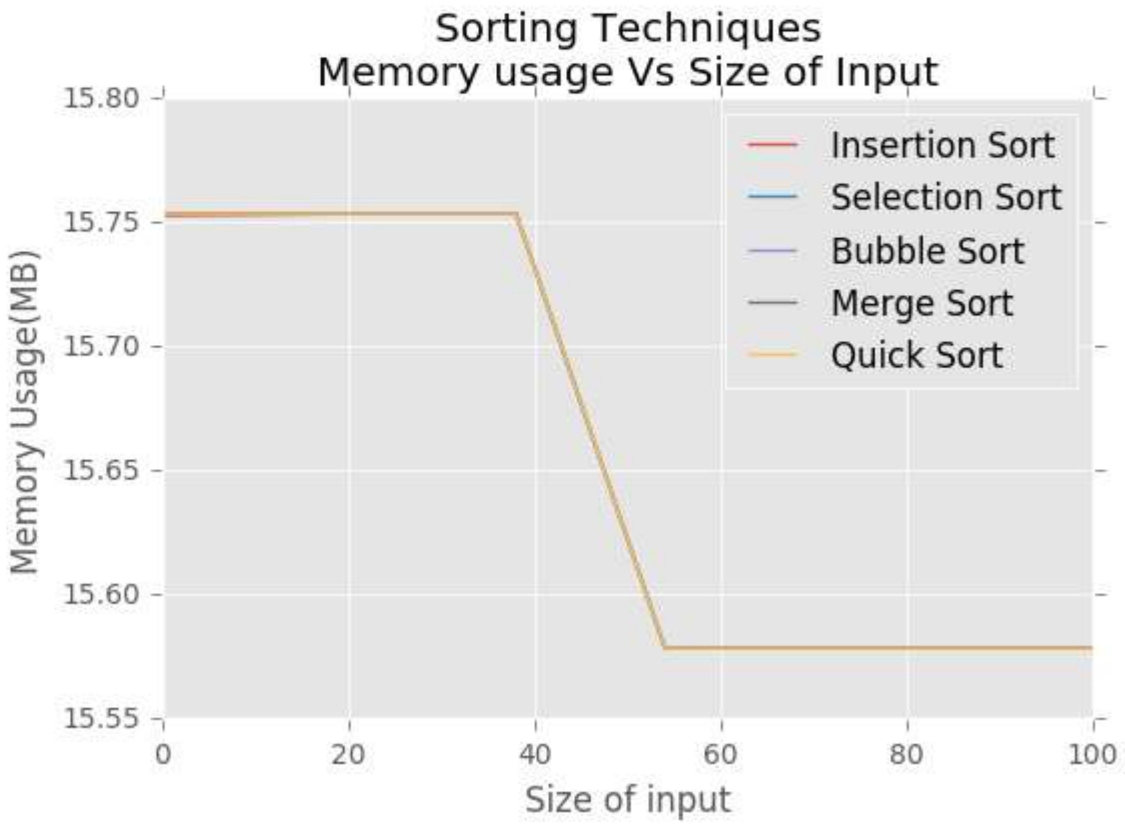
The memory usage of an algorithm is vaguely related to the structure if the code and also the degree to which the variables use the memory. Python, uses dynamic allocation and therefore is directly dependent on the variables assignment. Reframing the code and then calculating the memory usage was planned but, then I felt it's better to optimize the code and then finally compare the memory usage.

More discussed in Issues and Hurdles

Observation

Here we have the graph (on the page below) for the mapping of memory and size of input. Due to unexpected system complications, the test are run only on 100 input data size.

<u>Graph:</u>



When I considered the python, methods used for memory profiling, I found that most of the methods mentioned in the documentation were using psutil functions, which track memory using the pid or the process id whenever invoked.

More discussion in Issues & Hurdles

Data set – 4 (real)

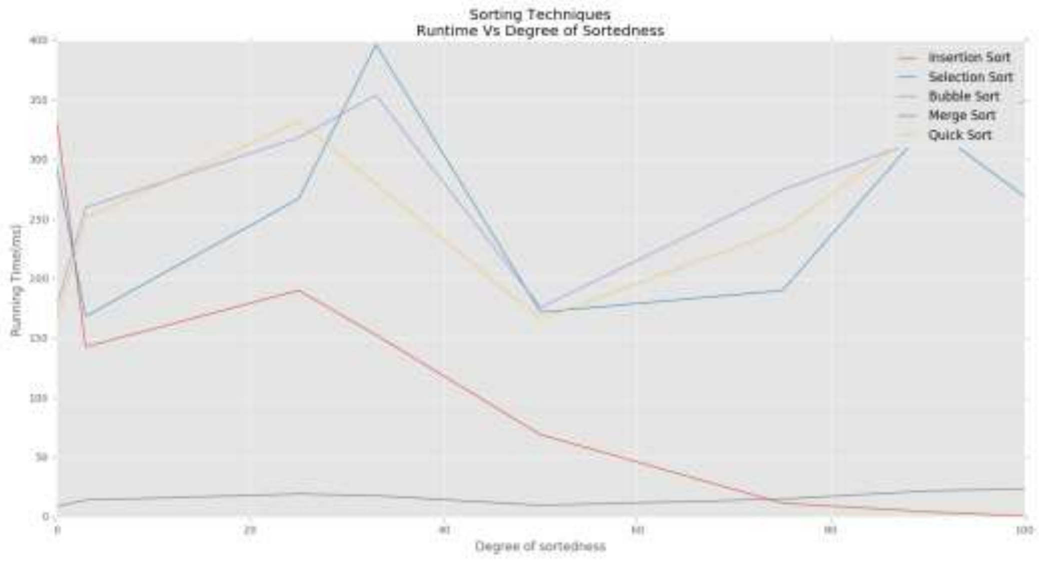
For a real data set I looked up the UCI website of Machine learning repository and took Bands Data set[3]. I imported the data set in to a local script and filtered a specific column thus resulting in an array like structure. In the main python script, I converted the single column file into a list which can be worked processed.

Runtime vs Degree of sortedness:

The size of data set used is 1000. The Degree of sortedness was generated in the same way as it seems efficient enough. We use a normal distribution here for testing purposes. Although in my opinion the value of the data set does not impact the performance unless there is a huge gap in the size of each element. Going down to the assembly level the time taken for comparing two 4-digit numbers shouldn't be different form comparing two 1-digit numbers. But on the other hand, given many recursive runs, comparing two 1-digit numbers will be faster than comparing two 34 digit numbers. But strictly speaking, we rarely find integers of that size in real-life scenarios unless we look at the data in NASA or aerospace related projects, they seem to produce huge integers even after the decimal point.

Observation

Graph:



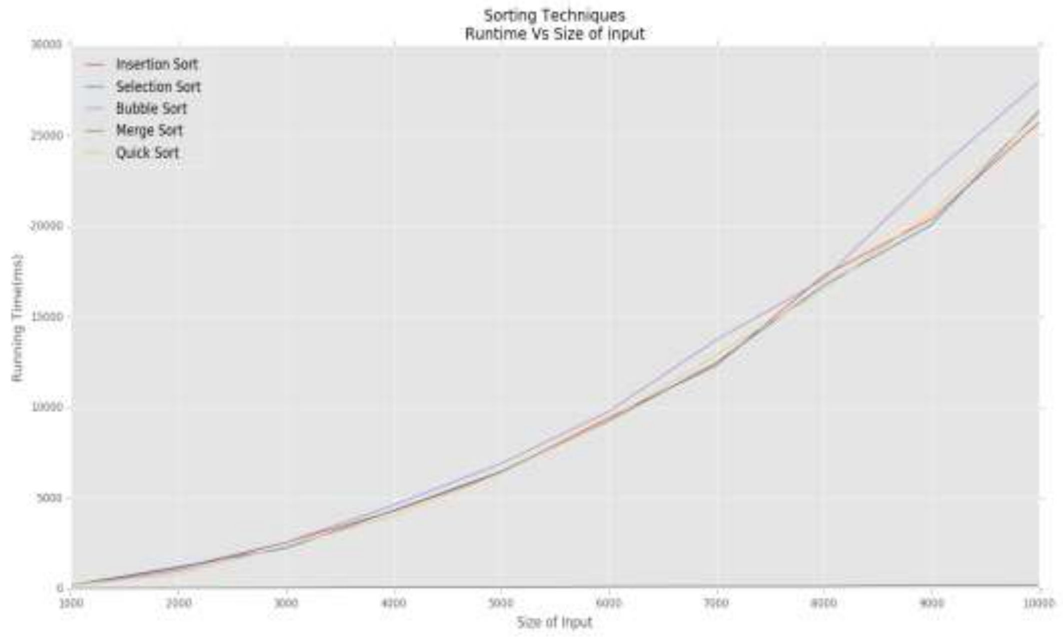
The graph obtained here might look weird but, on analysis I found that the run time has been affected by the busy processors of my system. I was running multiple processes and was using my core to the fullest. That's the reason for spiking in the graph. The underlying facts remain the same. Inerstion sort decreases drastically.

Runtime vs Size of input:

For the size of input, I proceeded the same way as in previous methods to maintain the congruity of the performance analysis.

Observation

Graph:



Issues and Hurdles:

From the very beginning I decided to use python as the coding base for this assignment. Thinking that the performance analysis, especially the graph will be relatively easier. Thanks to this assignment I have learnt my lesson. My first approach was to create and executable and generate all the graphs with just one click but, keeping the relative time constraint I decided to right the raw working code and then work on achieving fine modularity. Threading was my first option to compute all the tasks in parallel I tried implementing the sorting techniques into a multithreaded program, my first time in python but, I couldn't execute it properly. Later, I had to drop the threading functionality because of the time it was taking to rectify it.

Getting real data sets was hard at first then I figured out a way to filter out the required column from a multivariate Data set. So, that problem was solved. Although, my processing was to first generate analysis for synthetic data and then plug in the real data. Everything in concern with the running time analysis on synthetic was done and took almost a day.

The major hurdle came on the last day when I was left with the task of memory. Memory profiling in Python is the worst, I know that even other people using python faced the same problems and therefore have been active on the discussion forum for the very same topic. After hours and hours of trial and error methods and comparison with my perceptive knowledge of memory consumption I finally found a way to make it work.

The problem was that my IDE was not able to execute the memory profiler code without errors, I tried everything I possibly could but it wouldn't budge. My last resort was to run it on my friend's computer and it didn't work on his as well. Then Jupyter came to my rescue and I could run it in Jupyter. Although Jupyter is not very responsive to the python code, that's the only thing working for me.

Reframing the code and then calculating the memory usage was planned but, then I felt it's better to optimize the code and then finally compare the memory usage.

But I'm confident that before the next weekend I can optimize my code and solve all the errors.

Conclusion

From the above analysis, we find that merge sort performs well in almost all situations except in optimizing memory consumption. Quick sort comes next with average 2nd best performance but it should not be used when we already have a sorted array. For a sorted array, we should use Insertion sort. Similarly, our data suggests that Selection sort is an average performer. But, since we hardly have a sorted array to plug into a sorting algorithm we should prefer to use quick sort over merge sort as merge sort used much more memory when treating large datasets.

References

[1] http://slideplayer.com/slide/1424452/

[2] https://archive.ics.uci.edu/ml/datasets.html

[3] https://archive.ics.uci.edu/ml/datasets/Cylinder+Bands

[4] https://archive.ics.uci.edu/ml/datasets/Iris

[5] https://docs.scipy.org/doc/numpy/reference/routines.random.html