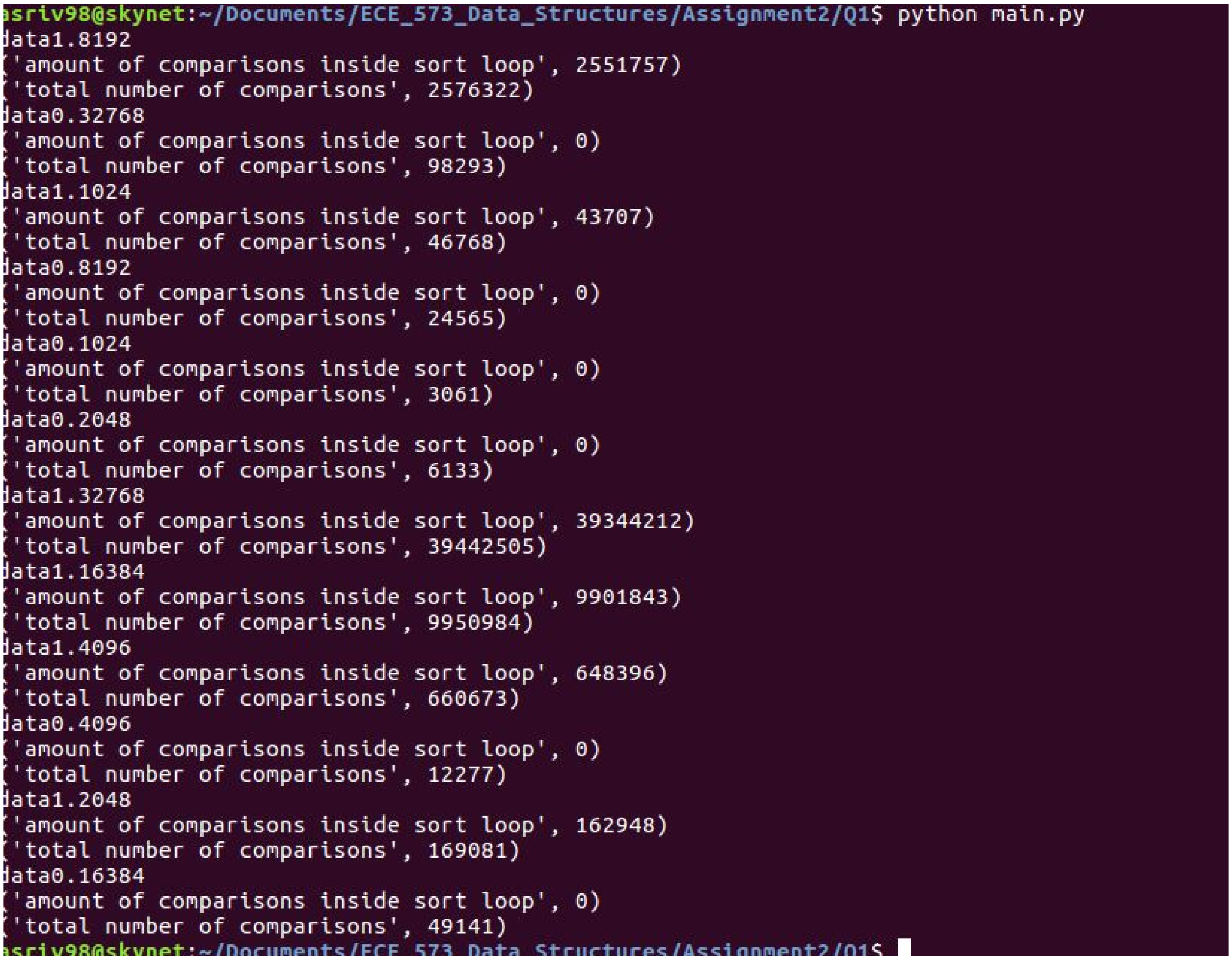
Atul Srivastava
Data Structures and Algorithms Assignment 2
Professor Shantenu Jha
RUID: 170002071

***In all questions with data, I manually ran the program looking for different sizes and configurations and recorded it on an excel sheet to generate all of my graphs.**
***github repo : https://github.com/ASriv98/ECE_573_Data_Structures,**

Question 1

Inspiration for this problem's implemented algorithms came from this blog site:
https://www.geeksforgeeks.org/shellsort/

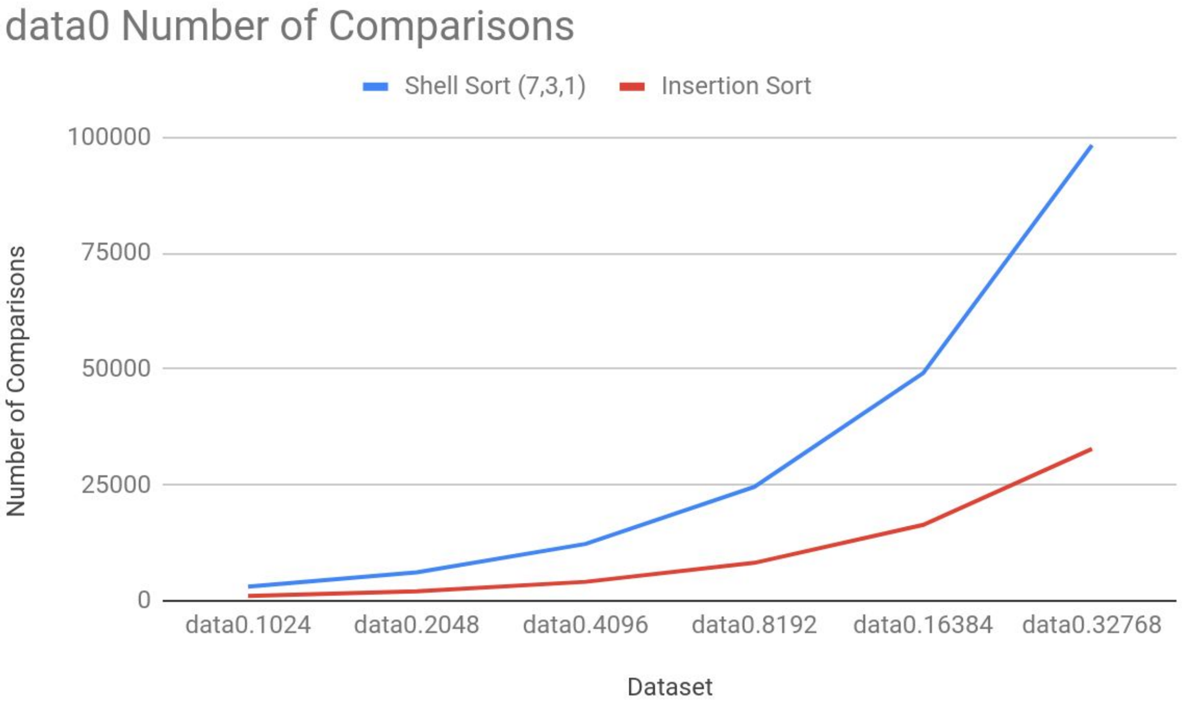


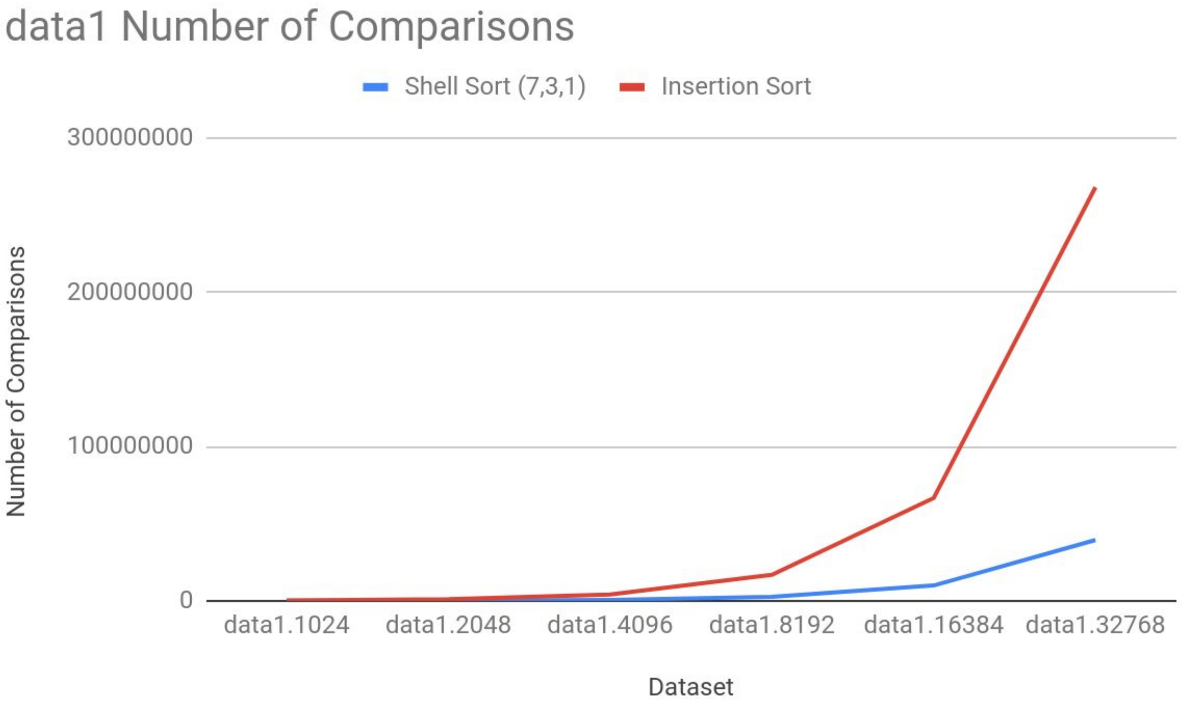
```
asriv98@skynet:~/Documents/ECE_573_Data_Structures/Assignment2/Q1$ python main.py
data1.8192
('amount of comparisons inside sort loop', 2551757)
('total number of comparisons', 2576322)
data0.32768
('amount of comparisons inside sort loop', 0)
('total number of comparisons', 98293)
data1.1024
('amount of comparisons inside sort loop', 43707)
('total number of comparisons', 46768)
data0.8192
('amount of comparisons inside sort loop', 0)
('total number of comparisons', 24565)
data0.1024
('amount of comparisons inside sort loop', 0)
('total number of comparisons', 3061)
data0.2048
('amount of comparisons inside sort loop', 0)
('total number of comparisons', 6133)
data1.32768
('amount of comparisons inside sort loop', 39344212)
('total number of comparisons', 39442505)
data1.16384
('amount of comparisons inside sort loop', 9901843)
('total number of comparisons', 9950984)
data1.4096
('amount of comparisons inside sort loop', 648396)
('total number of comparisons', 660673)
data0.4096
('amount of comparisons inside sort loop', 0)
('total number of comparisons', 12277)
data1.2048
('amount of comparisons inside sort loop', 162948)
('total number of comparisons', 169081)
data0.16384
('amount of comparisons inside sort loop', 0)
('total number of comparisons', 49141)
asriv98@skynet:~/Documents/ECE_573_Data_Structures/Assignment2/Q1$
```

File	Shell Sort (7,3,1)	Insertion Sort
data0.1024	3061	1023
data0.2048	6133	2047
data0.4096	12277	4095
data0.8192	24565	8191
data0.16384	49141	16383
data0.32768	98293	32767
data1.1024	46768	265564
data1.2048	169081	1029283
data1.4096	660673	4187899
data1.8192	2576322	16936958
data1.16384	9950984	66657566
data1.32768	39442505	267966675

This question asked us to use the datasets and implement shell sort on them. When you are doing shellsort with a gap size of 1, it is exactly the same as insertion sort, so in order to compare shellsorts number of comparisons with insertion sort's number of comparisons, I simply ran the shellsort algorithm with a gap size of 1.

As you can see from the data, when the lists are already sorted (the data0 sets), insertion sort takes significantly less comparisons. This makes sense; insertion sort executes (n-1) comparisons whereas shellsort has to compute (n-7) + (n-3) + (n-1) comparisons. This applies to the best case for insertion sort and selection sort, which yields 3n -11 times for shell sort in the best case. Shell sort sees enhanced performance when using a dataset that is unsorted, as it shows a significant improvement over insertion sort. This occurs because when doing shell sort, we achieve a semi ordering by essentially doing insertion sort at gaps of size 7 and 3, before reverting back to original insertion sort. Essentially, shell sort is dividing the array into smaller semi sorted sequences so that the execution of insertion sort takes significantly less comparisons, because on large datasets, insertion sort can have the tendency to make many comparisons.





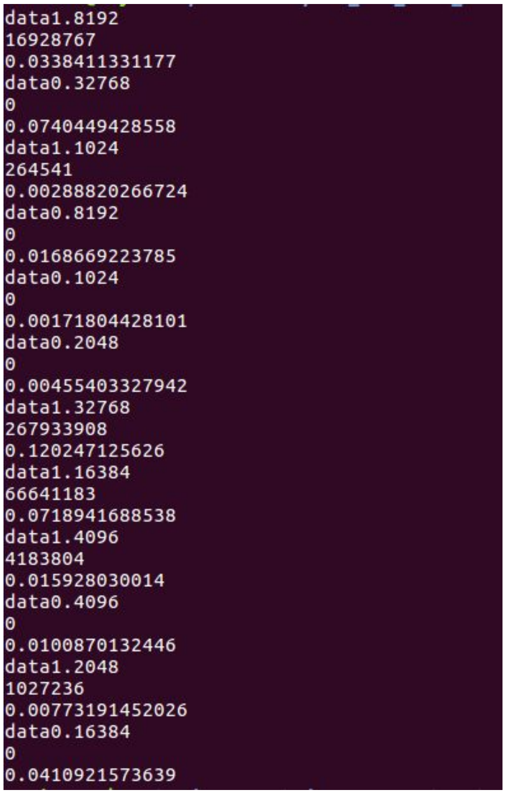
The graphs show the trend between shellsort and insertion sort with data0 (sorted sets) and data1(unsorted sets)

Question 2

https://www.jstor.org/stable/2282833?seq=1#metadata_info_tab_contents

This paper details efficient Kendall tau algorithm that uses n*log(n) opposed to the typical n^2 algorithm. At first thought of this problem, I essentially wanted to develop an algorithm that would have an n^2 running time, which would simply compare all of the possible pairs (essentially a bubble sorting type algorithm).

Algorithm inspiration from this blog:
https://medium.com/@ssbothwell/counting-inversions-with-merge-sort-4d9910dc95f0

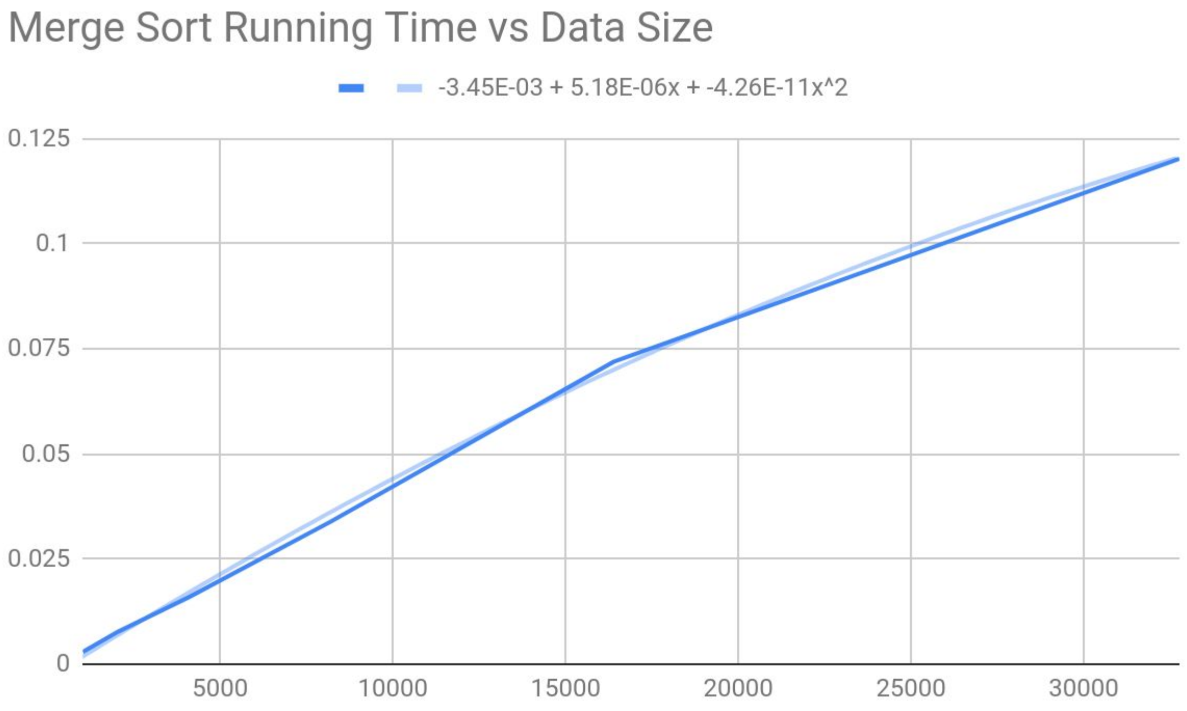


```
data1.8192
16928767
0.0338411331177
data0.32768
0
0.0740449428558
data1.1024
264541
0.00288820266724
data0.8192
0
0.0168669223785
data0.1024
0
0.00171804428101
data0.2048
0
0.00455403327942
data1.32768
267933908
0.120247125626
data1.16384
66641183
0.0718941688538
data1.4096
4183804
0.015928030014
data0.4096
0
0.0100870132446
data1.2048
1027236
0.00773191452026
data0.16384
0
0.0410921573639
```

For this problem, I ignored the data0 dataset because for all of them, there are no inversions and hence the Kendall Tau distance is 0.

File	Running Time (second)	Number of Comparisons
data1.1024	0.002888202667	264541

data1.2048	0.00773191452	1027236
data1.4096	0.01592803001	4183804
data1.8192	0.03384113312	16928767
data1.16384	0.07189416885	66641183
data1.32768	0.1202471256	267933908



In order to efficiently calculate to the Kendall Tau distance in n*logn time, we use a merge sort algorithm. In particular, we split our input array into left and right halves, and recursively keep splitting until we have only single elements remaining. Then, we append appropriate values to a sorted_list which is going to be returned (O(n) time merging functionality). We know that in Kendall Tau distance, an inversion is a pair of numbers in which a higher number comes before a lower number. When doing merge sort, we kind of accomplish this. In particular, when we are merging halves of lists, the elements from the left side that are smaller than elements in the right side are appended to the sorted list before. Then, all the remaining numbers on the right side are inversion pairs with the numbers in the left side that were not added to the sorted list.

Q3

I think a sorting algorithm that will sort this most effectively is merge sort, because the merge method will be skipped all the time because the data we have is already in order. This will lead to a linear runtime of O(n).

If the data is ordered as is given in the problem, bubble sort and insertion sort will handle sorting well because it will make n comparisons. However, if the dataset is shuffled, mergesort will perform the best because of even partitioning of the dataset.

https://www.sanfoundry.com/python-program-implement-bucket-sort

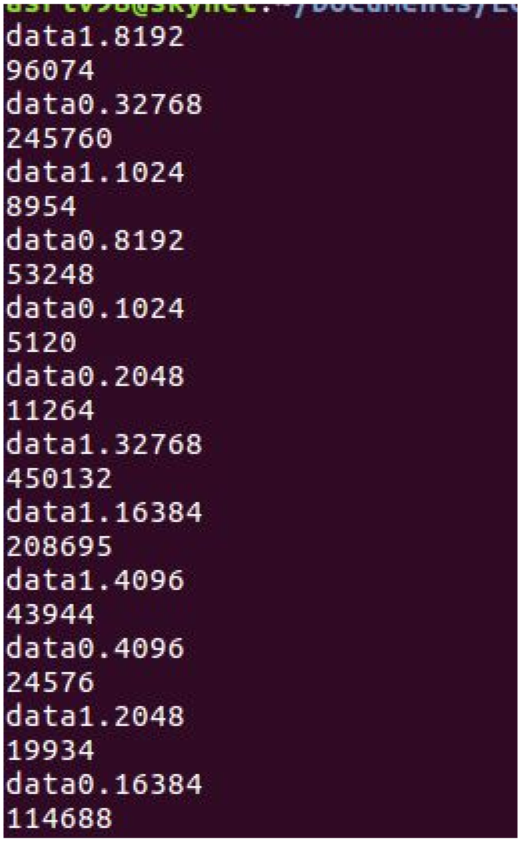
I also researched some other types of sorting algorithms that don't necessarily use comparisons first to sort. In particular, I tested an implementation of bucket sort, which also works pretty fast and has a linear running time. The way bucket sort works, it sorts the data into "buckets" which are data that are in the same range of values, and then it performs insertion sort on this data. For this problem in particular, we can split up into buckets easily and because the data is already sorted, insertion sort works very well and only has to do single comparisons, as mentioned before. The downfall of bucketsort is that it requires additional space, and in particular, the implementation I used seems to have some extra overhead which seems to be causing a timing slowdown.

When I ran the different sorts on the data set, merge sort gave me the best running time on average. However, I think this is simply because of the implementation and in this particular data set, many sorts will have a linear running time because it is already all sorted.

Q4

Although the top down merge sort was basically done in question 2, the bottom up (iterative) algorithm was a new approach and I learned about it and used the algorithm from this website: https://www.geeksforgeeks.org/iterative-merge-sort/

https://stackoverflow.com/questions/42608630/how-to-add-a-comparison-counter-for-merge-sort-in-python



```
data1.8192
96074
data0.32768
245760
data1.1024
8954
data0.8192
53248
data0.1024
5120
data0.2048
11264
data1.32768
450132
data1.16384
208695
data1.4096
43944
data0.4096
24576
data1.2048
19934
data0.16384
114688
```

File	Top Down (Recursive)	Bottom up (Iterative)
data0.1024	5120	5120
data0.2048	11264	11264
data0.4096	24576	24576
data0.8192	53248	53248
data0.16384	114688	114688
data0.32768	245760	245760
data1.1024	8954	8954
data1.2048	19934	19934
data1.4096	43944	43944
data1.8192	96074	96074
data1.16384	208695	208695
data1.32768	450132	450132

Because both the top down(recursive) and bottom up (iterative) merge sort share the same way of merging the broken up arrays. The same types of comparisons happen each time when dealing with this. Also, the amount of times the two merges happen is also the same and because of this, we have the same amount of comparisons.

We have to take into account that the datasets are also a power of 2. If the datasets are not a power of 2, we will have different comparison times.

Q5

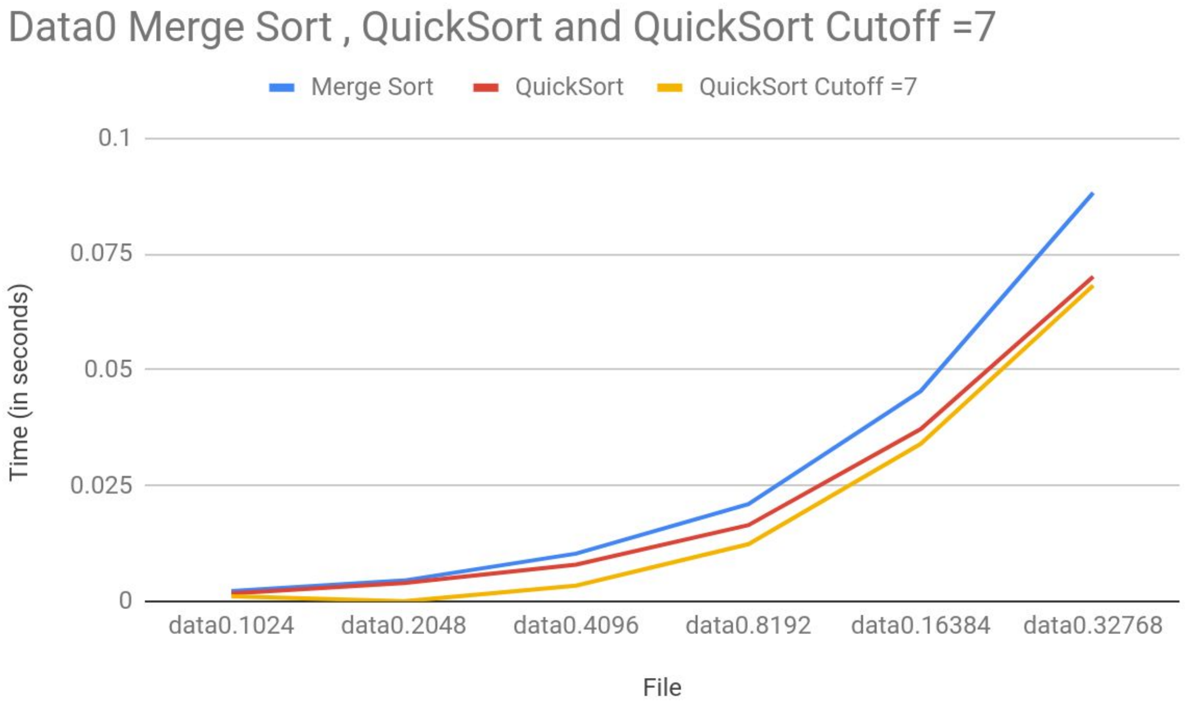
First, I added timing onto Q4, bottom up, naive merge sort algorithm to get the timing data for that.

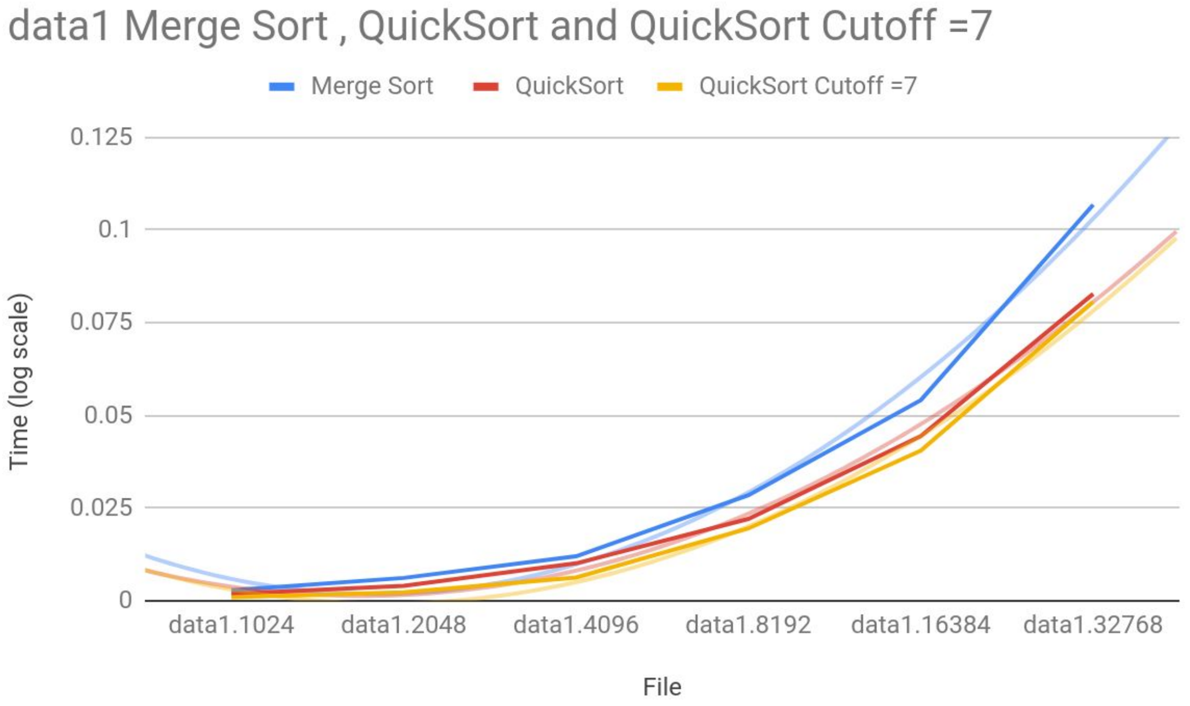
https://ownagezone.wordpress.com/2013/03/04/quick-sort-median-of-three-python-implementation/
The quicksort median of three algorithms was researched from this blog site.

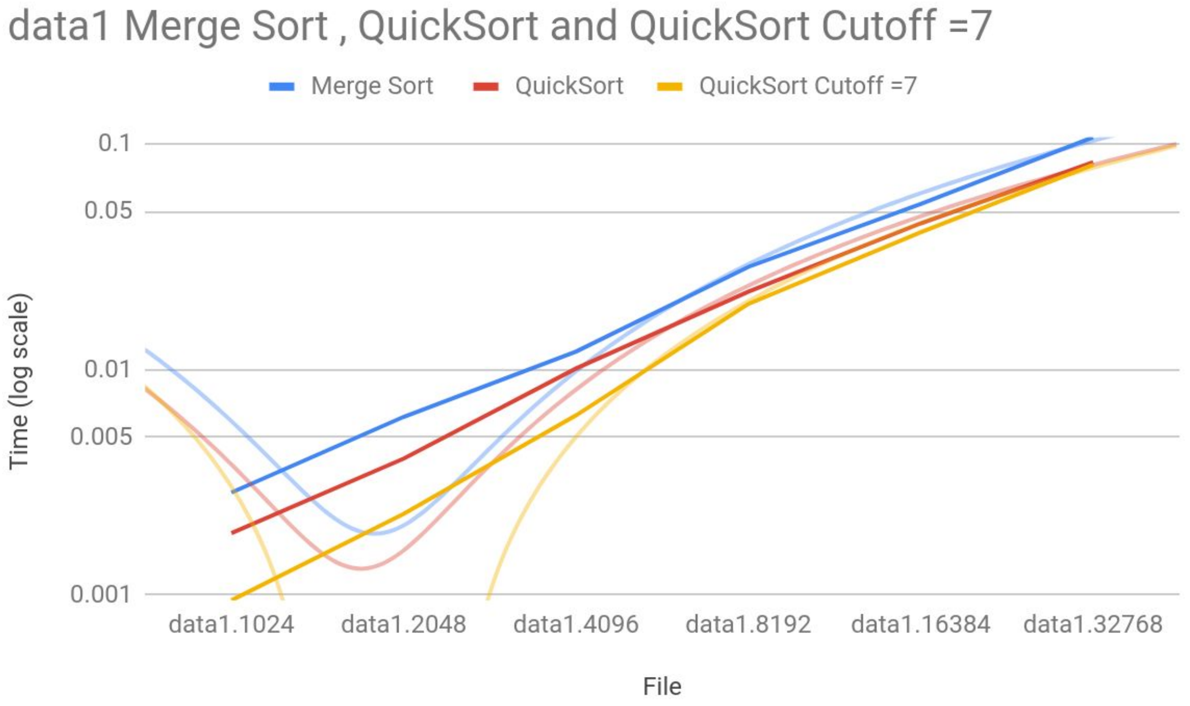
File	Merge Sort	QuickSort	QuickSort Cutoff =7
data0.1024	0.002243041992	0.001715898514	0.00111939432
data0.2048	0.0044901371	0.003942966461	1.08E-05
data0.4096	0.01031088829	0.007947921753	0.003394956589
data0.8192	0.02100396156	0.01647305489	0.01235401154
data0.16384	0.04543113709	0.03721785545	0.03401910782
data0.32768	0.08821487427	0.07011198998	0.06818091393
data1.1024	0.002847909927	0.001883029938	0.000948384332
data1.2048	0.006166934967	0.004026889801	0.002294023244
data1.4096	0.01200604439	0.01010012627	0.006262893677
data1.8192	0.02847719193	0.02209997177	0.01951300621
data1.16384	0.05412602425	0.04440379143	0.04051410675

data1.32768	0.1068291664	0.08276891708	0.08060609818

The data shows that in general, quick sort is slightly faster than merge sort. In addition, quick sort with a cutoff = 7 is even faster than running quicksort without any cutoff.







Since we know that data0 is all of the lists in sorted order, and data1 is shuffled, and we already have prior knowledge of how the algorithms run, we can get a sense of what we expect to see on the graphs. Data0 is the best case for both merge sort and quick sort (merge sort is always n log n average case), and so we expect to see an nlogn relation. When we plot the data1 on a log scale, we can see that it isn't complete a quadratic relation, but quicksort doesn't fare as well with an unsorted list. This is because insertion sort works much better when the list is sorted compared to when it is unsorted.

As per the last part of the question asks, I experimented with a few different cutoff values to see if too small or too large of a cutoff value is actually worse than just using quick sort naively.

In the range of 6-10, cutoff values tend to speed up quicksort, but beyond 10, I was getting results that were slowing down the sorting. So, based on my testing, at values of above 10, the use of a cutoff reverted the way quicksort worked in terms of its efficiency.