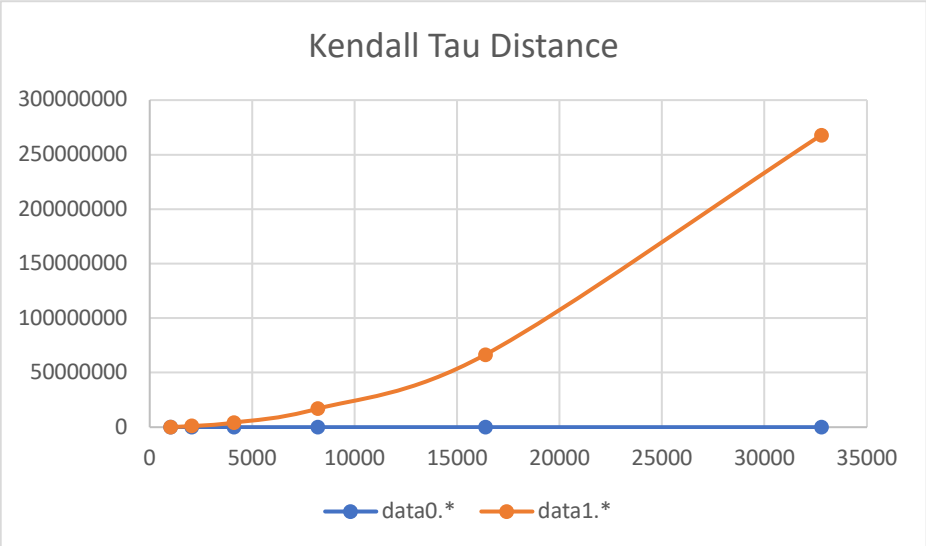
NAME : DIKSHA PRAKASH
NET-ID : dp978

QUESTION 2:

NOTE: In the code "KendallTau.py", the number of inversions is contained in the list "inversionsList".
The table below shows the 'Kendall Tau Distance' obtained for all the datasets. From the table, we can even validate that the code is correct. There exist 0 inversion pairs in data0.* files as they consist of already sorted numbers. Inversion pairs exist only in the data files data1.* as they are unsorted in nature.

DATASET	Number of Inversions	Runtime (ns)
Data0.1024	0	7805347
Data0.2048	0	2724409
Data0.4096	0	5606484
Data0.8192	0	12435388
Data0.16384	0	18656206
Data0.32768	0	47505569
Data1.1024	264541	1167192
Data1.2048	1027236	1741290
Data1.4096	4183804	3411950
Data1.8192	16928767	5025167
Data1.16384	66641183	7206811
Data1.32768	267933908	11956324

The plot for the same can be obtained as under:



Mathematically, to calculate the Kendall Tau distance between two arrays (say, A and C) which essentially act as eigen vectors i.e.

$C[A[i]] = I$ where, $0 \leq I \leq len(a)-1$

And, $D[i] = C[B[i]]$ where, $0 \leq I \leq len(a)-1$

We then calculate the number of inversions in 'D' to get the 'Kendall-Tau' Distance. In my code, I have used 'Merge Sort' to find the number of inversions in 'D'. Essentially, the order of this algorithm is about O(nlgn).

QUESTION 3:

For generating the dataset, I wrote a python code which will open the file "data1.txt" and append the series of 1, 11, 111 and 1111 according to certain set of conditions.

I prefer 'Insertion Sort' as the sorting technique as it works the best and the fastest for the "somewhat" sorted datasets and extremely well for already sorted arrays. The reasoning for this is that when it's inserting elements into the sorted position of the array, it has to barely move any elements. Thus, insertion sort on an already sorted array is linear-time, since it only needs to do one comparison per element.

It is also well known that both 'Insertion Sort' and 'Bubble Sort' have a good performance for sorted arrays. Therefore, for verification purpose, I implemented two sorting algorithms namely, 'Insertion Sort' and 'Bubble Sort'. I compared the runtime for these and 'Bubble Sort' took significantly larger time for execution as compared to 'Insertion Sort'. A tabular result for the same is as under:

SORTING ALGORITHM	Insertion Sort	Bubble Sort
EXECUTION TIME (ns)	4376888.27	5113411903.38

Thus, it can be concluded that Insertion Sort is the best sorting algorithm for "partially" or already sorted data. In the table above, the lower time has been highlighted with 'yellow'.

QUESTION 4:

NOTE: In the code of 'MergeSort.py', the list 'ydata_mergesortTD' represents the number of comparisons performed for top-down approach and in

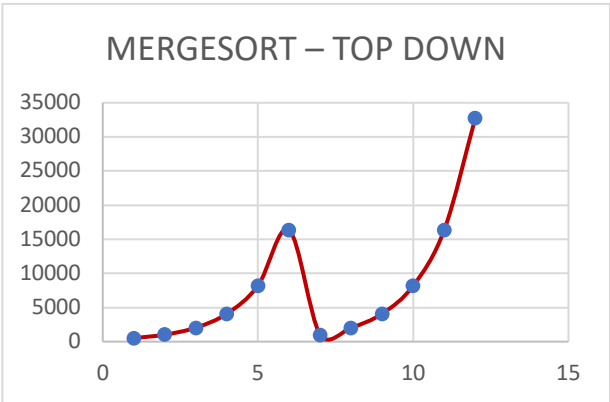
'MergeSort_bkup.py', 'ydata_mergesortBU' represents the number of comparisons performed for bottom-up approach.

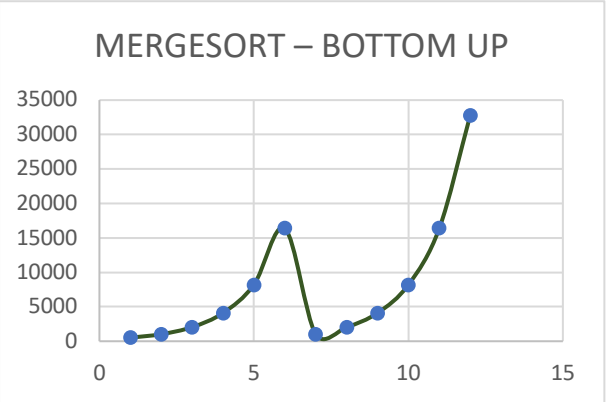
As we know, both the approaches of "Merge Sort" call the same "merge" algorithm wherein the actual comparison takes place. Also, the frequency of calling this method are the same for two versions because the only difference is in the order of the calls. Thus, it is obvious that the number of comparisons is totally same for both the approaches.

The number of comparisons was recorded as under:

DATASET	MERGESORT – TOP DOWN	MERGESORT – BOTTOM UP
Data0.1024	512	512
Data0.2048	1024	1024
Data0.4096	2048	2048
Data0.8192	4096	4096
Data0.16384	8192	8192
Data0.32768	16384	16384
Data1.1024	1022	1022
Data1.2048	2046	2046
Data1.4096	4095	4095
Data1.8192	8191	8191
Data1.16384	16383	16383
Data1.32768	32764	32764

The two can be plotted as under to obtain the same graphs:





The result above can be generalized for the data sets where the size is a power of 2. However, such inferences can't be made on data sets which aren't a power of 2. (I tried to append a few entries to the data sets such that the size is no longer a

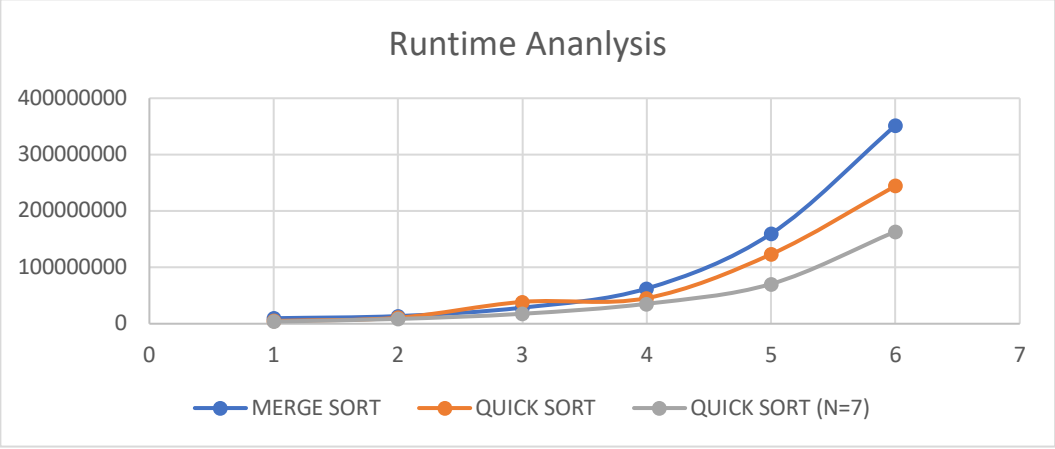
power of 2 and observed that the number of comparisons no longer were the same. I couldn't interpret the reason for it and thus, haven't talked about it much)

QUESTION 5:

NOTE: In the code of 'MergeSort.py', the list 'endTime_MergeSort' represents the time taken by the individual files for execution. Correspondingly, in 'QuickSort.py', the list "ydata_quicksort' represents the runtime.
Following is the tabular representation of the running time Merge Sort vs Quick Sort (without cut-off). Since data0.* is the best case where the data is already shuffled whereas data1.* are the data sets that are shuffled.

DATASET	MERGE SORT	QUICK SORT	QUICK SORT (N=7)
Data0.1024	006273984.91	004620075.23	004895782.47
Data0.2048	010783910.75	009029865.26	008447313.31
Data0.4096	025018930.44	020204782.48	016318011.28
Data0.8192	050668001.17	043055772.78	038665604.59
Data0.16384	117512941.36	096684217.45	078039503.09
Data0.32768	256751775.74	217761039.73	148065495.49
Data1.1024	009800910.95	004918813.71	003925299.64
Data1.2048	013610124.59	010665893.55	008528780.94
Data1.4096	028517007.83	038720130.92	017702722.55
Data1.8192	062466859.82	045467138.29	035420298.58
Data1.16384	159672021.86	123234033.58	070575213.43
Data1.32768	351514816.28	244405984.88	163527822.49

As expected, Quick Sort is faster than Merge Sort in general. When we take cut off = 7, 'Quick Sort' takes less time to finish the sort. The plot of the same is as under:



To experiment for the suitable value of the cut-off, I changed the value of 'N' (in the code) to various integers from 0 to 15 and noted the execution times for all. It was observed that values between 6-10 are good choices.

QUESTION 1:

As we know, for an in-order data (like the files 'data0.* given in the datasets), the shell sort is far less effective than Insertion Sort. Insertion Sort performs well in short sequences and partially sorted sequences but not in the case on unsorted data; as compared to 'Shell Sort'. This is so because 'Shell Sort' is essentially a 'h-stride Insertion Sort'.

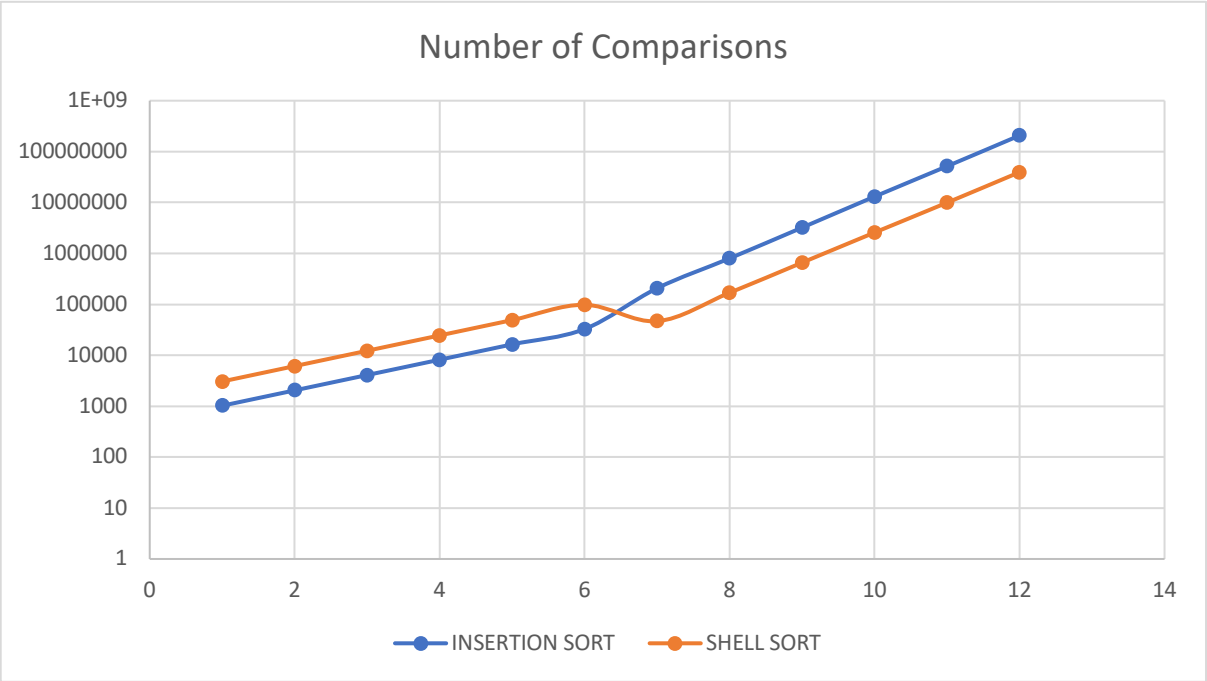
The code generates an output for the number of comparisons for different datasets for both the sorting algorithms. It can be represented tabularly as under:

DATASET	INSERTION SORT	SHELL SORT
Data0.1024	1023	3061
Data0.2048	2047	6133
Data0.4096	4095	12277
Data0.8192	8191	24565
Data0.16384	16383	49141
Data0.32768	32767	98293
Data1.1024	207561	46728
Data1.2048	800630	169042
Data1.4096	3219206	660619
Data1.8192	13082995	2576270
Data1.16384	51577117	9950922
Data1.32768	207305662	39442456

In the table above, the lesser time taken among the two sorting techniques for a given dataset has been highlighted in 'yellow'.

It can be clearly observed that for sorted data i.e. datasets "data0.*", Insertion Sort takes lesser time but for unsorted datasets i.e "data1.*", Shell Sort performs considerably better and efficiently than its counterpart, Insertion Sort.

The plot of the same can be drawn as under which clearly shows that for the sorted datasets i.e. data0.*, the number of comparisons for ‘Insertion Sort’ is less than that for ‘Shell Sort’ and vice versa for unsorted datasets i.e. data1.*.



When we start using an arbitrary data, ‘Shell Sort’ performs more effectively than ‘Insertion Sort’. Shell Sort will first divide the sequence into short subsequences and sort later. Basically, both parts of the Shell Sort are using Insertion Sort, so that’s why it proves to be more effective in this case. Eventually ‘Shell Sort’ will reduce to ‘Insertion Sort’ when the stride length is 1.