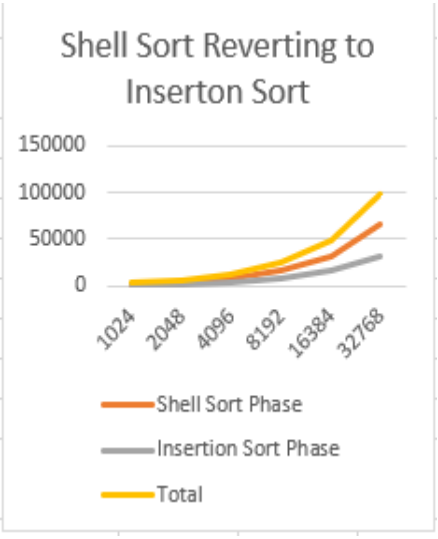
DATA STRUCTURES AND ALGORITHMS

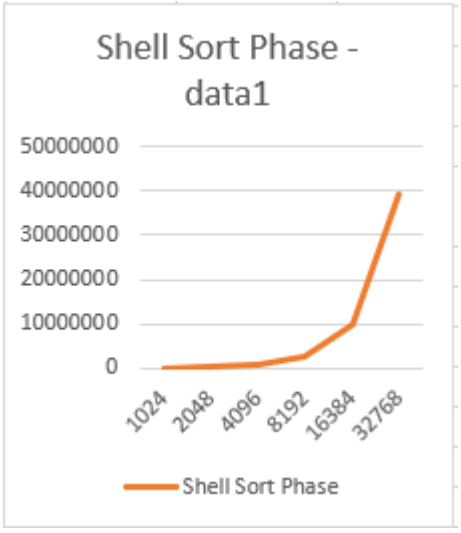
HOMEWORK - 2

1. The tabular and graphical results for the shell sort and insertion sort phase for data0 and data1 is shown below:

Input Size	Shell Sort Phase	Insertion Sort Phase	Total
1024	2038	1023	3061
2048	4086	2047	6133
4096	8182	4095	12277
8192	16374	8191	24565
16384	32758	16383	49141
32768	65526	32767	98293

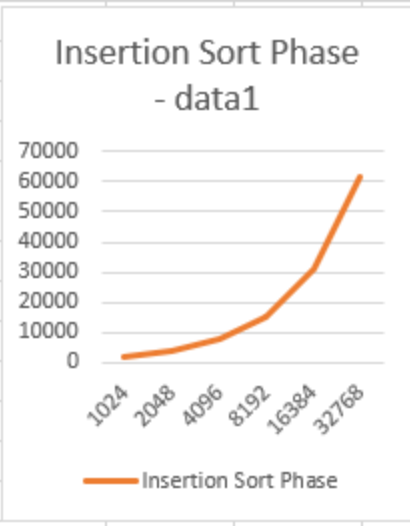
Input Size	Shell Sort Phase	Insertion Sort Phase	Total
1024	44807	1961	46768
2048	165208	3873	169081
4096	652786	7887	660673
8192	2560849	15473	2576322
16384	9920179	30805	9950984
32768	39380997	61508	39442505

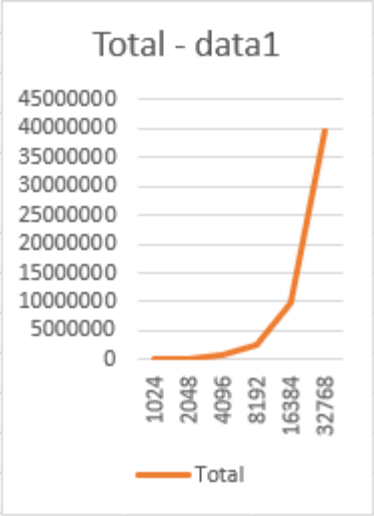




In the case of data0, the data is pre-sorted, so the shell sort reverts to the best case of the algorithm. As a result, the number of comparisons is lesser compared to that of data1.

However, for data1, since the elements are not in order, the algorithm takes longer to sort the array. This is observed in both the shell sort and insertion sort phases in the tables.





The shell sort algorithm is more effective than the insertion sort algorithm because the shell sort algorithm sorts the array using gaps, called h-sorting. Since we first 7-sort it and then 3-sort the array, when the gap reduces to 1, the list is already sorted, which reduces the number of comparisons, when compared to insertion sort.

The results for the relative time taken for when shell sort reverts to insertion sort is shown for data0 and data1 in the tables below:

For data0:

Input Size	Shell Sort Reverting	Shell Sort all the way
1024	0.00299883	0.006015539
2048	0.00698996	0.010987759
4096	0.01099324	0.024983406
8192	0.02798271	0.055967093
16384	0.07795691	0.178898811
32768	0.07795644	0.34780097

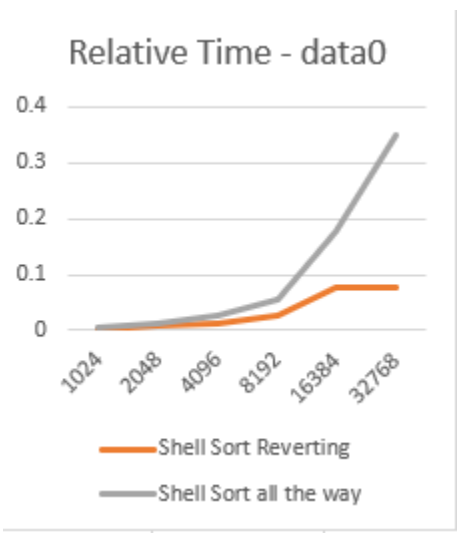
For data1:

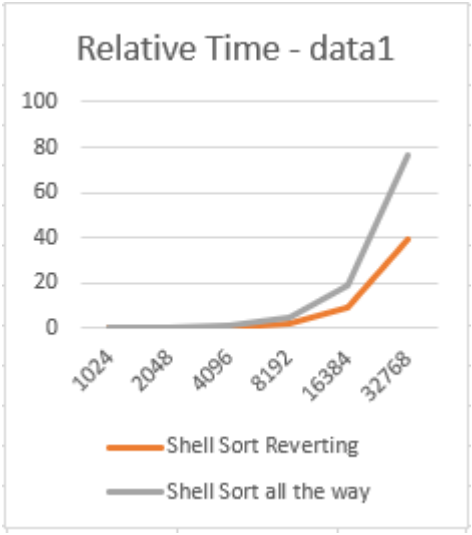
Input Size	Shell Sort Reverting	Shell Sort all the way
1024	0.035979986	0.07695627
2048	0.143918753	0.32081604
4096	0.567677259	1.17032957
8192	2.37963891	4.55098391
16384	9.43949151	18.7642505
32768	39.19515657	76.502331

For the case of data0, since the data is already sorted, the sorted list takes time in the order ranging from 3 milliseconds to 347 milliseconds. The increase in the time can be attributed to the increase in the data size. Further, as the data size increases, the time taken to compute the sorted array also increases, as seen in the graph below.

For data1, the data is out of order, so a considerable amount of time is taken when compared to data0; ranging from 4 milliseconds to around 75 seconds. Once again, the time taken increases with increase in the data size.

The corresponding graphs are shown below:





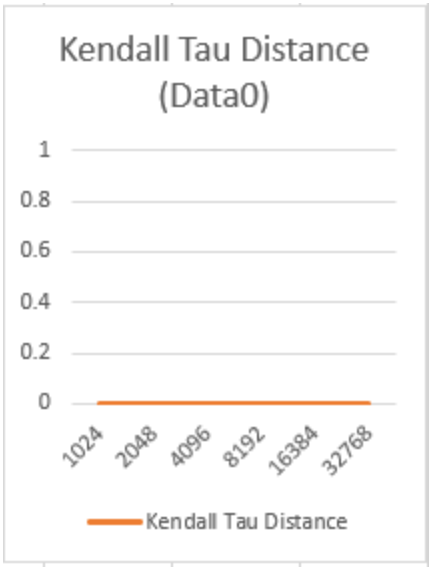
2. In order to compute the Kendall Tau distance with an algorithm in less than quadratic time (O(N^2)), I used the merge sort algorithm, which has a best and worst case of O(NlogN). The results are shown below:

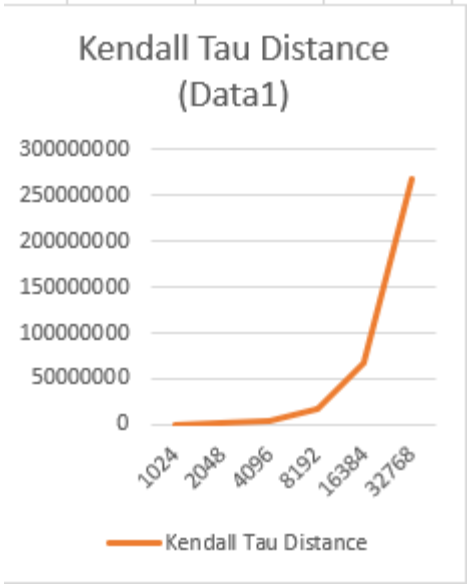
For data0:

N	Kendall Tau Distance
1024	0
2048	0
4096	0
8192	0
16384	0
32768	0

For data1:

N	Kendall Tau Distance
1024	264541
2048	1027236
4096	4183804
8192	16928767
16384	66641183
32768	267933908





The values for the Kendall Tau distance for any data size in the case of data0 are all 0 since the array is already sorted; and hence, there are no swaps to be made. For data1, the number of swaps steadily increase with increase in the data size.

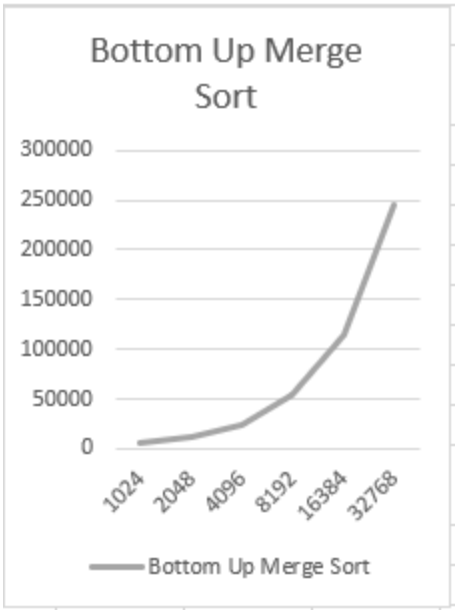
3. For data0, the number of comparisons and physical time for recursive merge sort and the bottom up merge sort algorithms are shown below:

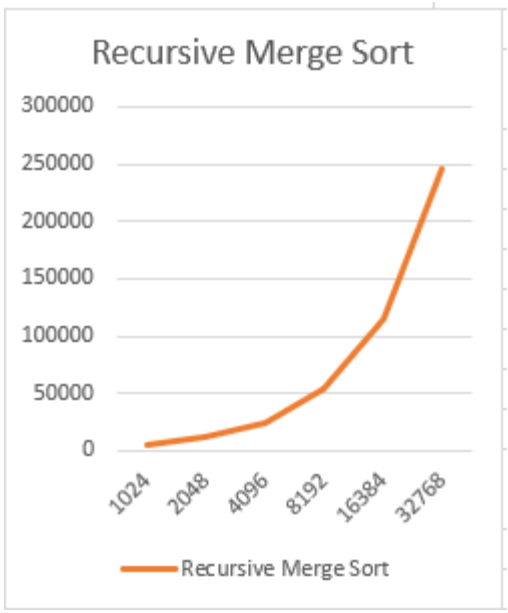
For data0:

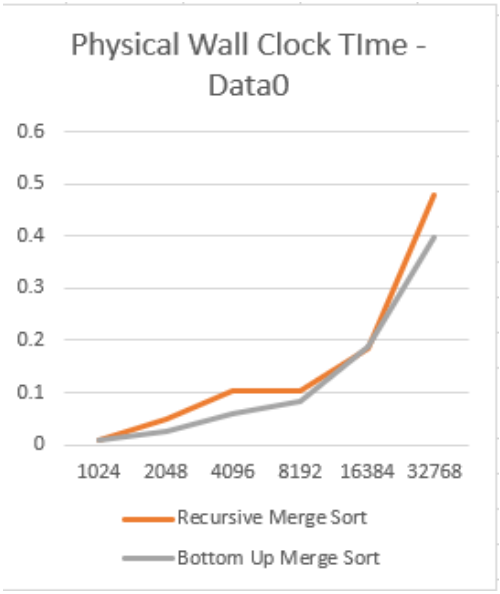
Input Size (N)	Recursive Merge Sort	Bottom Up Merge Sort
1024	5120	5120
2048	11264	11264
4096	24576	24576
8192	53248	53248
16384	114688	114688
32768	245760	245760

For data1:

Input Size (N)	Recursive Merge Sort	Bottom Up Merge Sort
1024	0.010013342	0.007995605
2048	0.049960136	0.023980379
4096	0.102940798	0.058980942
8192	0.102941036	0.081973314
16384	0.183894634	0.187894106
32768	0.479697943	0.398770332







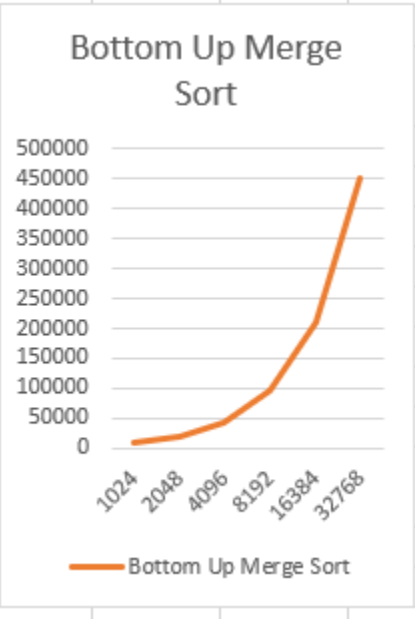
The observations noticed from the graph and tables are:

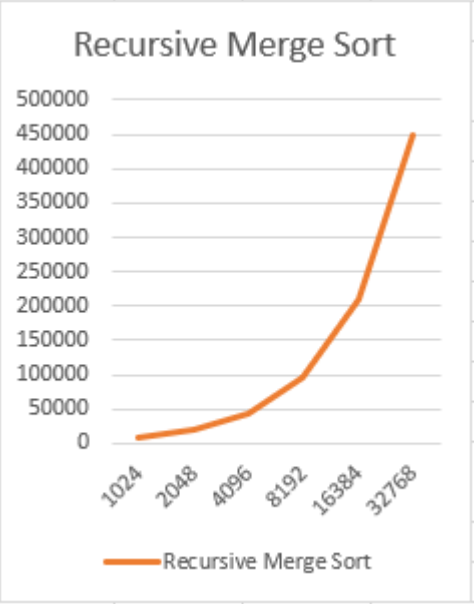
- The number of comparisons for both data0 for the various data sizes remains the same.
- The physical time for bottom up merge sort is smaller compared to recursive merge sort.

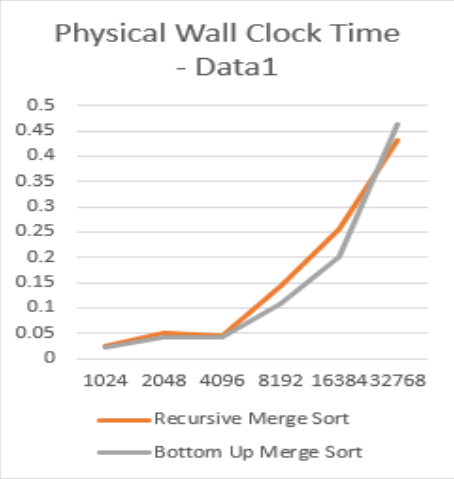
For data1, the number of comparisons and physical time for recursive merge sort and the bottom up merge sort algorithms are shown below:

Input Size (N)	Recursive Merge Sort	Bottom Up Merge Sort
1024	8954	8954
2048	19934	19934
4096	43944	43944
8192	96074	96074
16384	208695	208695
32768	450132	450132

Input Size (N)	Recursive Merge Sort	Bottom Up Merge Sort
1024	0.024986029	0.020987988
2048	0.04997015	0.042972326
4096	0.044974089	0.041975975
8192	0.141919374	0.107935667
16384	0.254872322	0.201884031
32768	0.431738377	0.462727785







OBSERVATIONS

- In the case of data1, like data0, the number of comparisons remain the same for the various data sizes and again, the physical time taken for computation in bottom up merge sort is smaller compared to recursive merge sort.
- Both the number of comparisons and physical time for data1 is larger than data0, since more comparisons need to be made.

4. Since the data given to us consists of repeats of 1024 ‘1’s’, 2048 ‘11’s’, 4096 ‘111’s’ and 1024 ‘1111’s’, the list is already in a sorted order. So we need to use an algorithm that can pass through the entire list in O(N) time. The efficient algorithm which I chose was the insertion sort algorithm, and the number of comparisons was 8192, which is the length of the list.

5. The comparison of quick sort with the merge sort implementation in Q3 is shown in the table below, for data0 and data1 respectively:

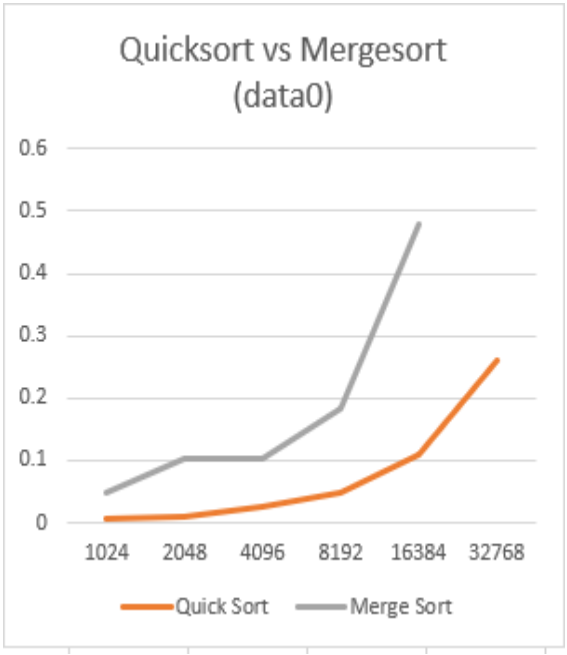
For data0:

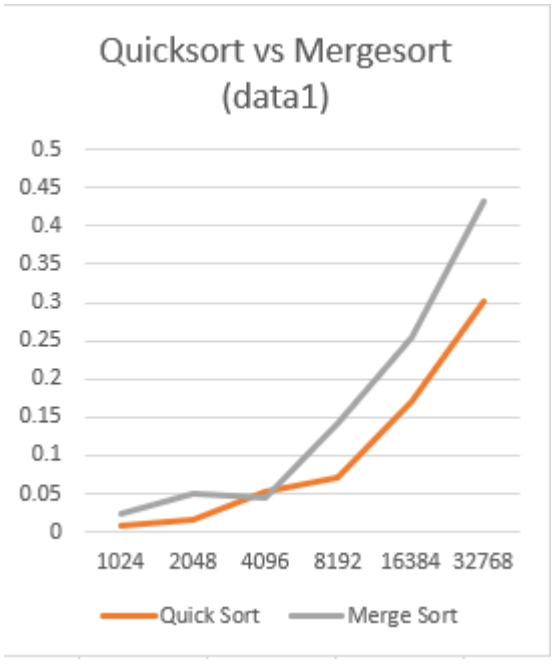
Input Size (N)	Quick Sort	Merge Sort
1024	0.007009	0.01001334
2048	0.00999	0.04996014
4096	0.026964	0.1029408
8192	0.048952	0.10294104
16384	0.10896	0.18389463
32768	0.25985	0.47969794

For data1:

Input Size (N)	Quick Sort	Merge Sort
1024	0.007994	0.02498603
2048	0.01497	0.04997015
4096	0.053477	0.04497409
8192	0.06996	0.14191937
16384	0.170902	0.25487232
32768	0.300375	0.43173838

The graphs for the above are shown below:





As seen from both the tables and the graphs, for both data0 and data1, quicksort outperforms merge sort; it takes a lesser time to compute the sorted array compared to merge sort.

The results obtained when using a cutoff at N = 7 are shown below:

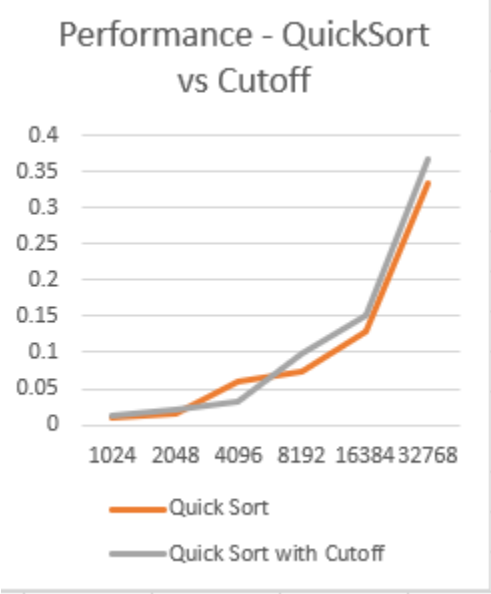
For data0:

Input Size (N)	Quick Sort	Quick Sort with Cutoff
1024	0.00999	0.01199126
2048	0.015988	0.02198839
4096	0.060496	0.03196168
8192	0.072959	0.09794426
16384	0.128073	0.15191197
32768	0.334881	0.36815882

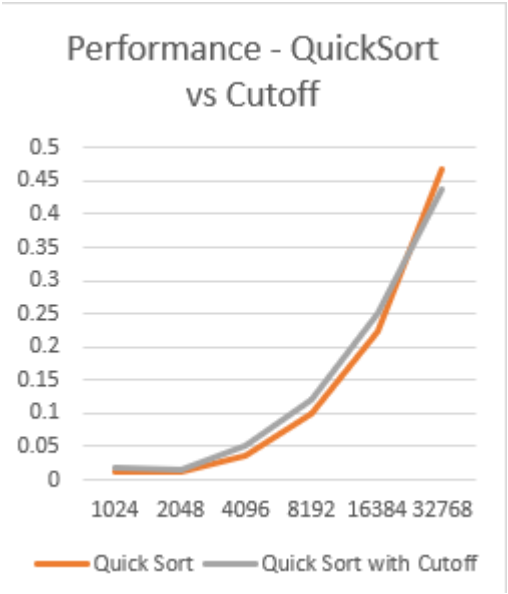
For data1:

Input Size (N)	Quick Sort	Quick Sort with Cutoff
1024	0.011992	0.01898932
2048	0.012953	0.01599026
4096	0.036977	0.05296612
8192	0.100944	0.12196517
16384	0.221923	0.24985671
32768	0.46843	0.43674922

For data0:



For data1:



Adding a cutoff at N = 7, the time taken decreases by a small amount compared the previous algorithms.

Varying the cutoff from 10 to 10000 in steps of 100, I obtained the results shown in the next page. The performance of the algorithm increases with increase in the data size, so a value for the cutoff to insertion when the performance inverts could not be determined.

For data0:

Input Size (N)	Time (s)
1024	0.011992
2048	0.01599
4096	0.043974
8192	0.078956
16384	0.33181
32768	0.470253

For data1:

Input Size (N)	Time (s)
1024	0.014991
2048	0.017007
4096	0.047983
8192	0.122929
16384	0.184894
32768	0.414762