Data Structures and Algorithms

ECE 573

Shounak Rangwala

Netid:snr85

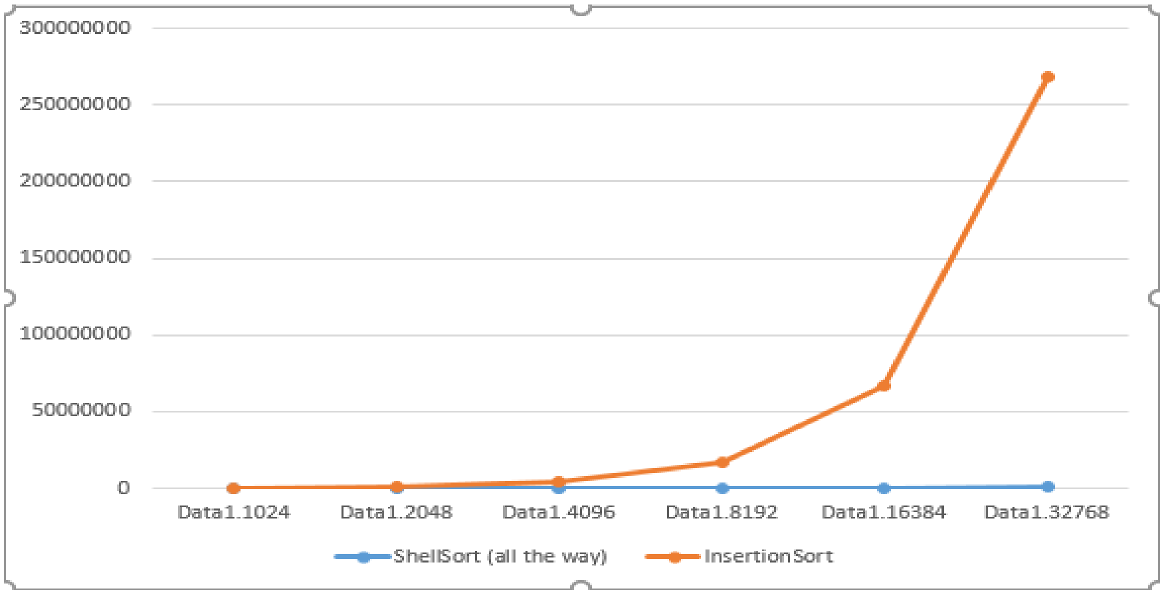
Q1)

I ran the shellsort algorithm on the unsorted data sets data1.1024-data1.32768. I compared the performance between the shellsort and insertion sort as well as shell sort and shellsort that reverts to insertion sort at some cutoff.

The results are in the tables below:

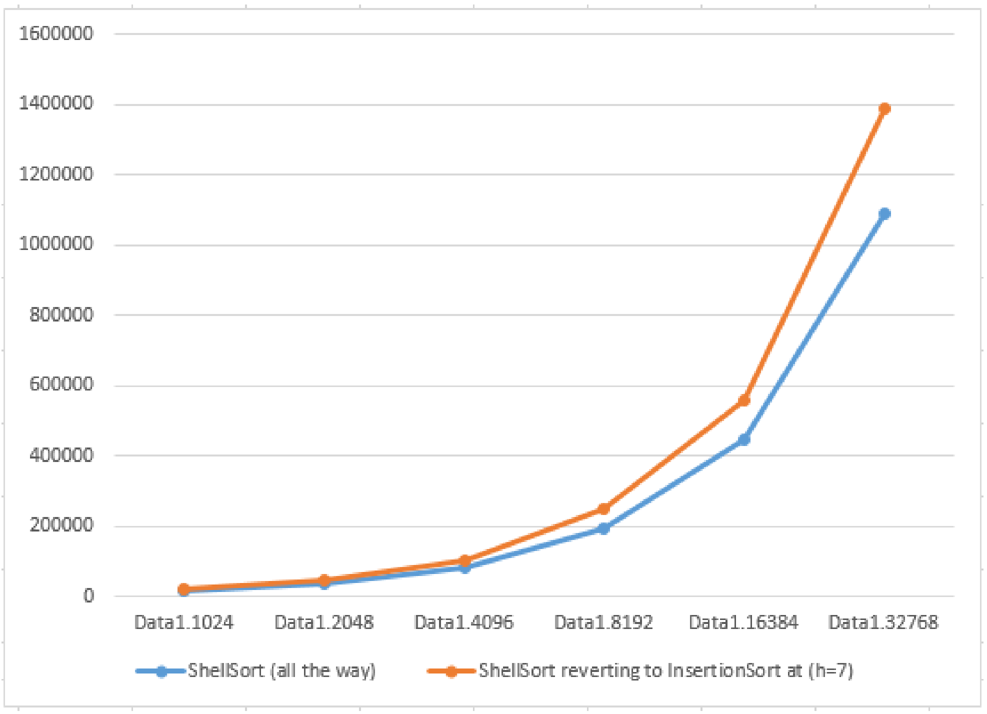
Insertion Sort vs. Shell Sort (number of comparisons)

Dataset	ShellSort (all the way)	InsertionSort
Data1.1024	15835	265565
Data1.2048	35709	1029284
Data1.4096	82710	4187900
Data1.8192	193716	16936959
Data1.16384	445683	66657567
Data1.32768	1088347	267966676



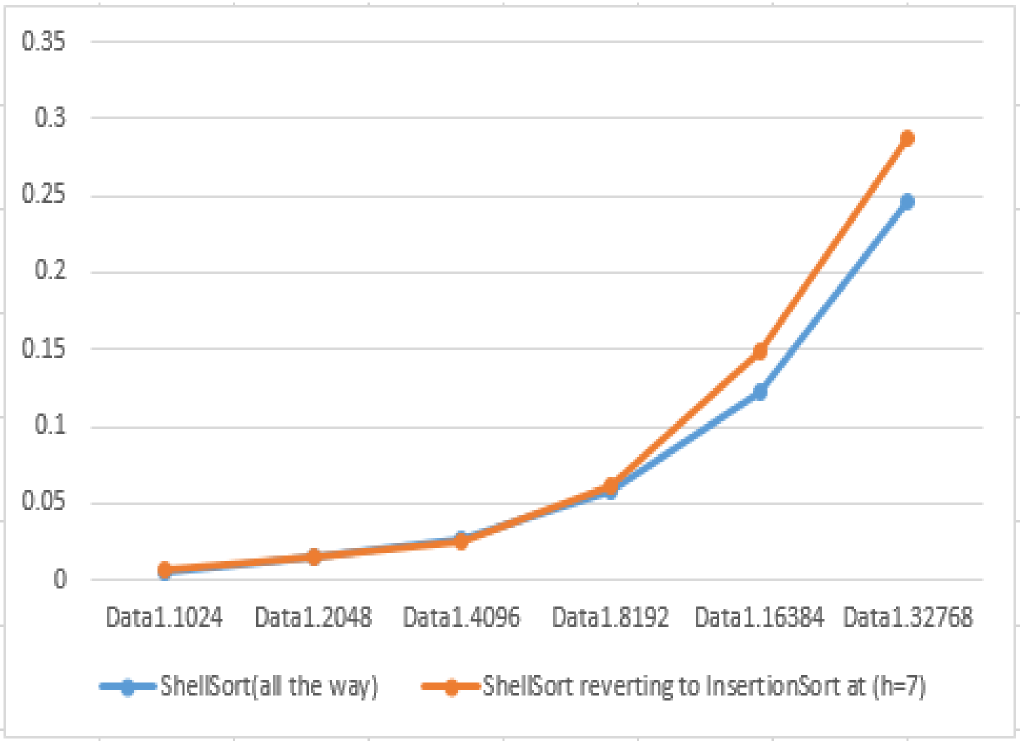
Shellsort vs Shellsort reverting to insertion sort at cutoff = 7 (number of comparisons)

Dataset	ShellSort (all the way)	ShellSort reverting to InsertionSort at (h=7)
Data1.1024	15835	20360
Data1.2048	35709	44032
Data1.4096	82710	102637
Data1.8192	193716	248057
Data1.16384	445683	559660
Data1.32768	1088347	1385872



Shellsort vs Shellsort reverting to insertion sort at cutoff = 7 (time taken in seconds)

Dataset	ShellSort(all the way)	ShellSort reverting to InsertionSort at (h=7)
Data1.1024	0.0049581527	0.0059721469
Data1.2048	0.0151991844	0.0143408775
Data1.4096	0.0267226696	0.0244462490
Data1.8192	0.0577430725	0.0612380504
Data1.16384	0.1219210624	0.1489007472
Data1.32768	0.2465960979	0.2882492542



These findings make perfect sense because, shellsort is basically going through the entire array multiple times and sorting the array in steps to make a more partially sorted array after every h-sort. If the datasets would have been sorted then the shellsort would have taken more comparisons than an insertion sort because insertion sort would just do a N-1 comparisons but shell sort would do N-1+N-3+N-7…. Comparisons. These would still be in the order of N but shell sort works much better than insertion sort on unsorted arrays.

If we would call insertion sort at a cutoff, then the insertion sort would be working on a partially sorted array, this would make it much better than normal insertion and almost as fast as shellsort for smaller N. However, as you can see from the findings, the number of comparisons that would slightly be increased because the insertion sort is being applied to a bigger unsorted array than before and would also take more time.

Q2)

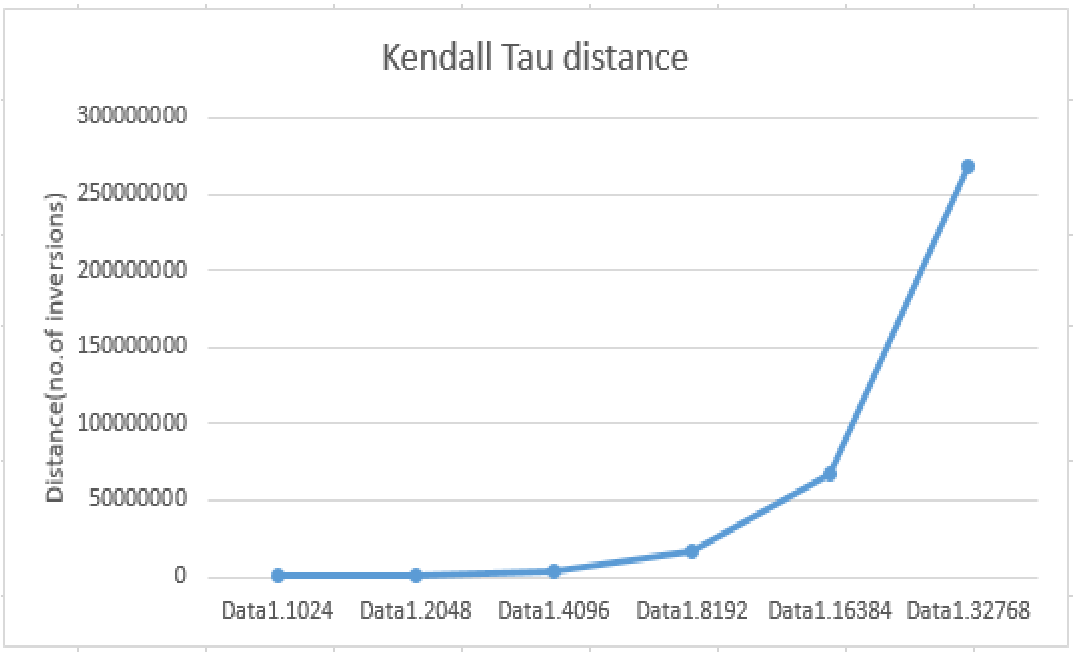
To calculate the Kendall-Tau distance in datasets data1.1024 – data1.32768, I referred to

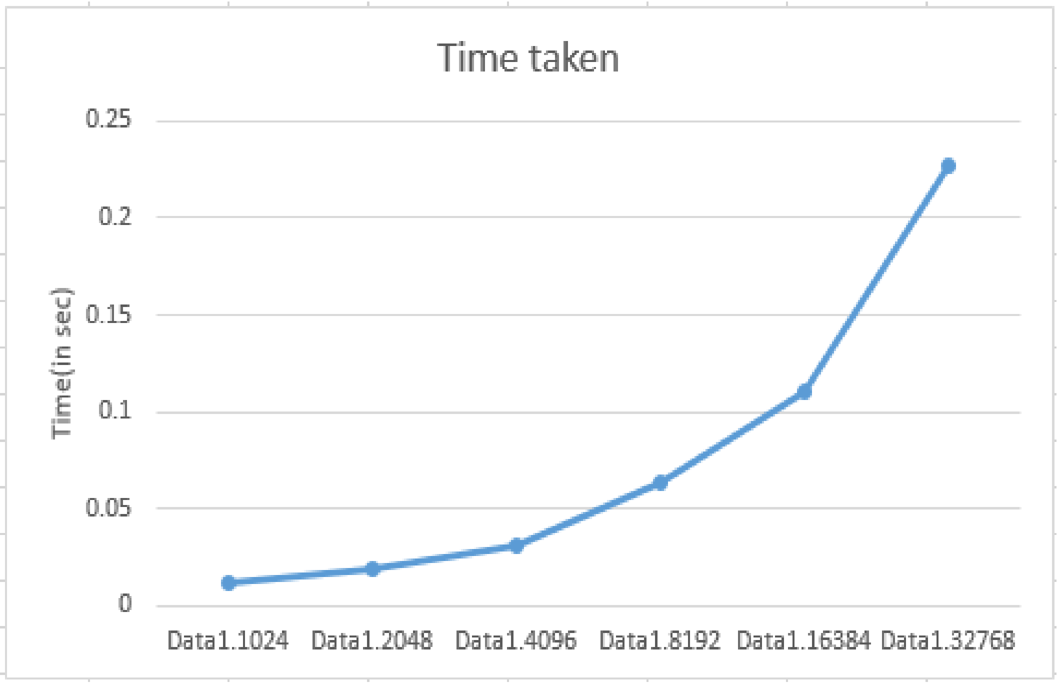
https://en.wikipedia.org/wiki/Kendall_tau_distance and https://stats.stackexchange.com/questions/168602/whats-the-kendall-taus-distance-between-these-2-rankings.

Basically, I assumed the 2 arrays to be compared to be data0.k and data1.k where k is {1024,….,32768 }. Since this distance is basically the inversions in the array which happen during the merge sort algorithm, I counted the number of elements on the left part of the auxiliary array that would be jumped over by an element in the right side of the auxiliary array when it is being merged into the bigger array. This count condition is added only to the case where arr[i]<arr[j] and you copy the jth element into the main array. The total number of these counts will give you the exact number of inversions happening in the mergesort algorithm which coincidentally happens to be the Kendall-Tau distance.

The findings are plotted below and as you can see the complexity for this algorithm is O(NlogN) in worst case which is less than quadratic time.

Dataset	Time taken	Kendall Tau distance
Data1.1024	0.012030839920043945	264541
Data1.2048	0.018375873565673828	1027236
Data1.4096	0.03096294403076172	4183804
Data1.8192	0.06357312202453613	16928767
Data1.16384	0.10990595817565918	66641183
Data1.32768	0.22667217254638672	267933908





Q3)

For mergesort standard version, I used the recursive method used in the slides and also for the bottom-up iterative version of merge sort, I used basic intuition and inspiration from the https://www.geeksforgeeks.org/iterative-merge-sort/ and https://stackoverflow.com/questions/37368921/bottom-up-mergesort-using-python websites.

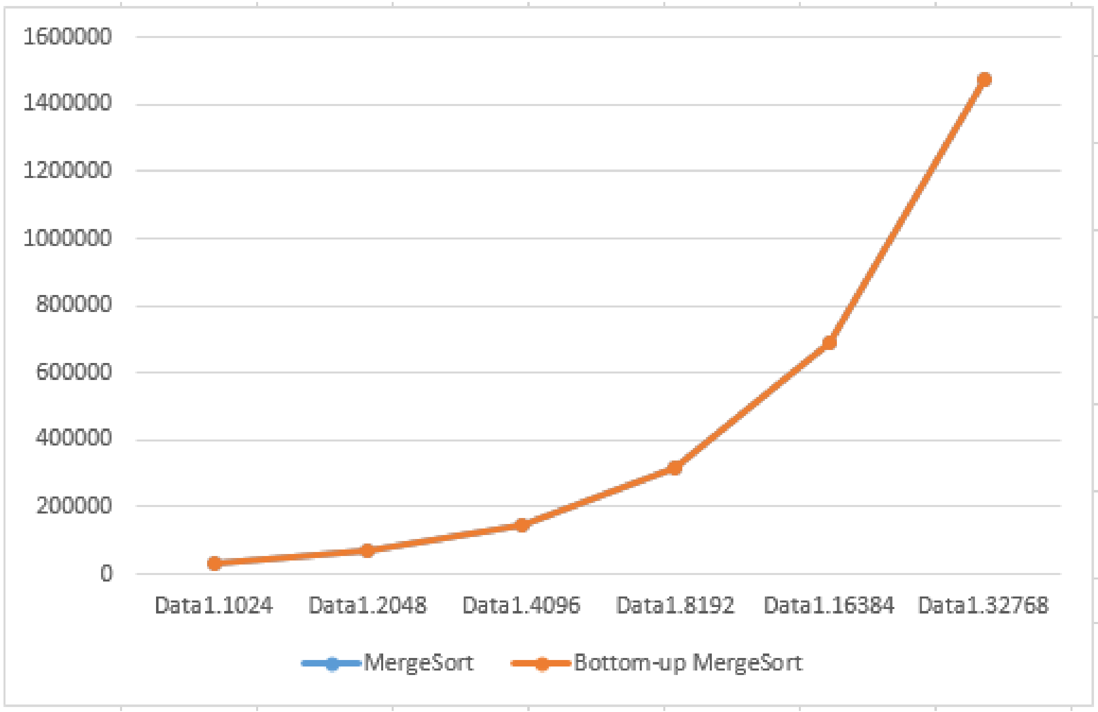
From the results documented below, we can conclude that the number of comparisons being done for both the versions merge sort require the exact same number of operation because the order in which these operations occur are reversed in both these implementations. The datasets are also in powers of 2 which can make merging of 2 equal arrays easier. If there was a different size then, it would take a little longer for the bottom-up version because of the remainder number which would be make the last merge group bigger than the rest.

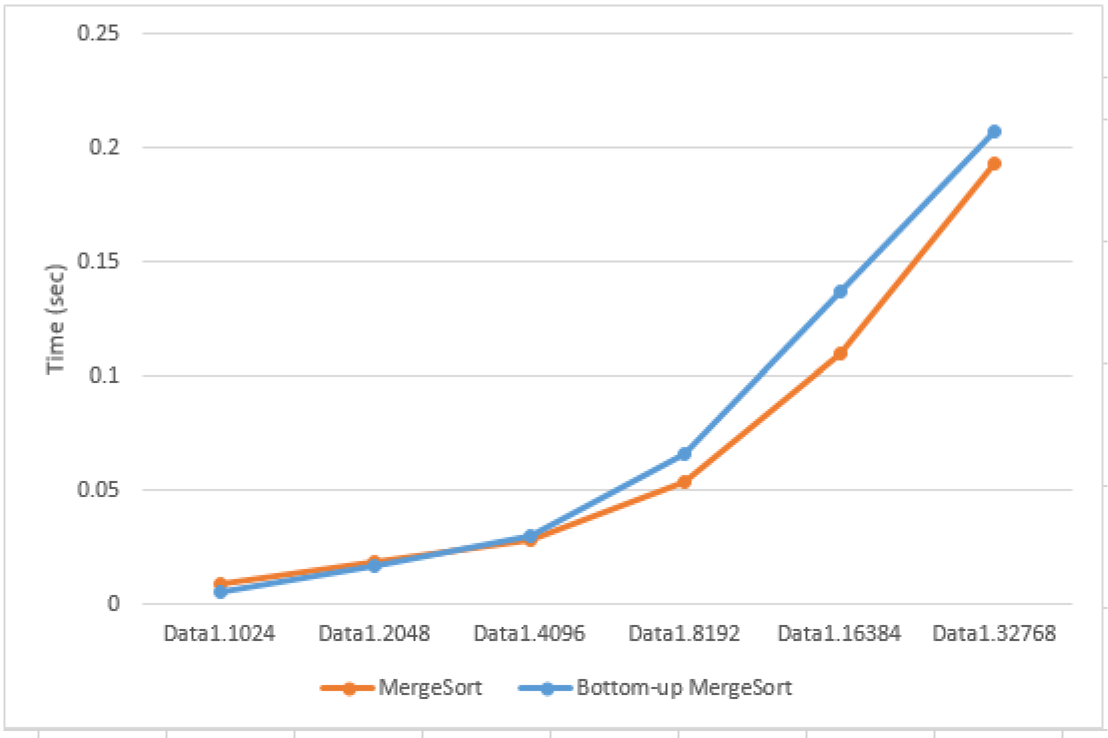
Number of operations

Dataset	MergeSort	Bottom-up MergeSort
Data1.1024	30720	30720
Data1.2048	67584	67584
Data1.4096	147456	147456
Data1.8192	319488	319488
Data1.16384	688128	688128
Data1.32768	1474560	1474560

Time taken (in sec)

Dataset	MergeSort	Bottom-up MergeSort
Data1.1024	0.008961200714111328	0.005517244338989258
Data1.2048	0.01836228370666504	0.016993045806884766
Data1.4096	0.02835822105407715	0.029475927352905273
Data1.8192	0.05371499061584473	0.0662539005279541
Data1.16384	0.1094362735748291	0.13715410232543945
Data1.32768	0.1928119659423828	0.20685696601867676





Q4)

Since you are making a sorted array with a large number of similar entries we can try to capitalize on the fact that there are very few different entries and come up with an algorithm such as the insertion sort. We need a stable algorithm in this case because while traversing through the entire array we need to make sure that the comparisons are being stopped at the first equal key and not getting carried over till the very end. Merge sort is also a viable option because it is stable but however, since the array is sorted, we can assume that Merge sort would take NLogN time as compared to insertion sort.

If this same array was shuffled, then merge sort would be better than insertion sort because the worst case complexity is NlogN as compared to insertions N^2.

Also we can try to incorporate the duplicate keys principle for quick sort in this case and try to make the sorting in place and in NlogN time. This should be run on the unsorted version of this same array. The Djikstras 3 – way partition would take comparably lesser time because a block of the same key values would be used as the partition in each iteration. The 1 consequence of this sorting is that it may not always be stable and to make it stable we much ensure that the comparison of similar items stops at the first instance and doesn't carry on. This can reduce the computation to linear or linearithmic time.

Q5)

The results for the comparison of the median-of-3 quicksort and quicksort along with cutoff to insertion sort can be seen in the tables and the plots below. I have also compared the mergesort performance to this variation of quicksort.

Dataset	Quicksort	Mergesort
Data1.1024	0.0038390159606933594	0.008961200714111328
Data1.2048	0.011981964111328125	0.01836228370666504
Data1.4096	0.02134227752685547	0.02835822105407715
Data1.8192	0.0408329963684082	0.05371499061584473
Data1.16384	0.0835731029510498	0.1094362735748291
Data1.32768	0.17377090454101562	0.1928119659423828

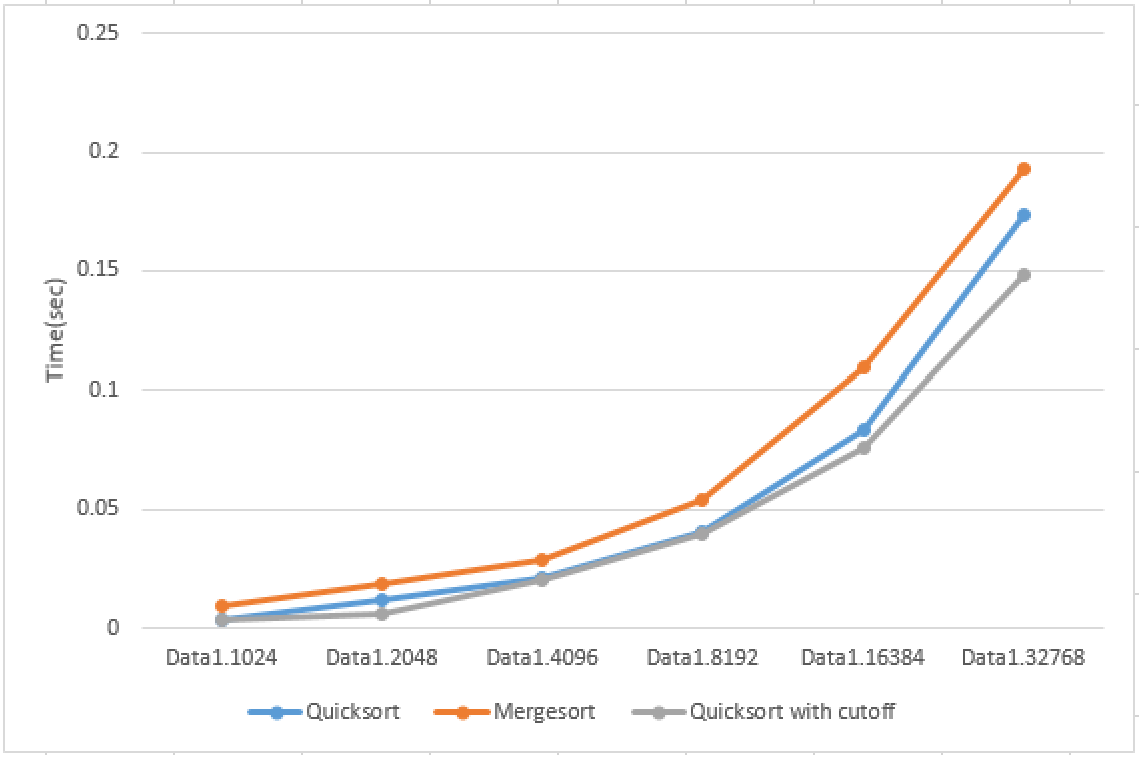
Dataset	Quicksort	Quicksort with cutoff
Data1.1024	0.0035550594329833984	0.003348112106323242
Data1.2048	0.011981964111328125	0.00630497932434082
Data1.4096	0.02134227752685547	0.02061009407043457
Data1.8192	0.0408329963684082	0.03979802131652832
Data1.16384	0.0835731029510498	0.07587289810180664
Data1.32768	0.17377090454101562	0.14847993850708008

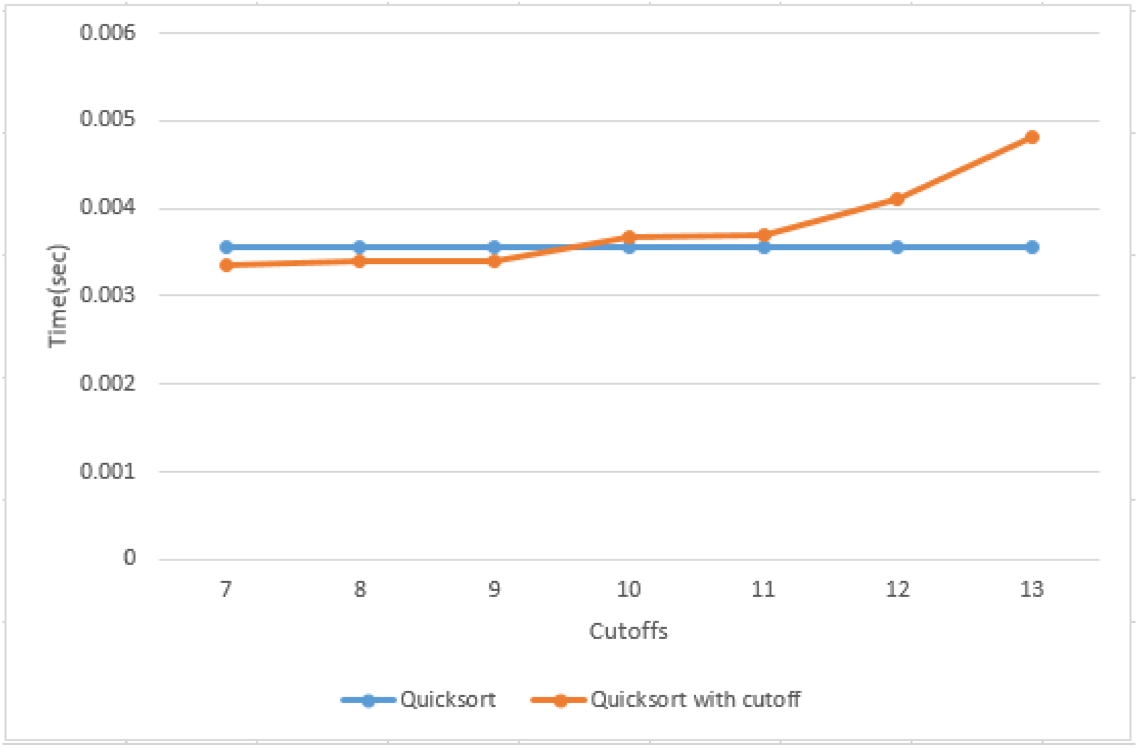
Dataset used (data1.1024)

Cutoff	Quicksort	Quicksort with cutoff
7	0.0035550594329833984	0.003348112106323242
8	0.0035550594329833984	0.003402116928100586
9	0.0035550594329833984	0.003403902053833008
10	0.0035550594329833984	0.003681182861328125
11	0.0035550594329833984	0.003693103790283203
12	0.0035550594329833984	0.004099845886230469
13	0.0035550594329833984	0.004824161529541016

I think that the performance of mergesort is slightly poorer than Quicksort because of the copying and pasting operations being done to the auxiliary array. As expected, quicksort with a cutoff ensures that after a certain level, insertion sort gets a partially sorted array that would be easier to sort and involve less comparisons because of its stable nature. This also reduces the overhead of having too many tiny subarrays that would be added to the stack and cause stress no the cache memory of the computer.

As for the optimal cutoff value, I tried different values in the vicinity of 7 and observed that with each cutoff increase there was an increase in the time taken for quicksort. The flip occurs between the values of 9 and 10 where the time taken for normal quicksort is better than the time taken with the cutoff value. This is when the insertion sort comparisons overshadow the quicksort subarray overhead.





Q6) By tracing the algorithms in the dataset given:

2nd column = Bottom-up mergesort

3rd column = Standard quicksort

4th column = Quicksort 3 way

5th column = Mergesort

6th column = insertion sort

7th column = heapsort

8th column = selection sort

9th column = Knuth shuffle