ECE-573-Homework-2

Name: Zekun Zhang

NetId: zz364

Email: zekunzhang.1996@gmail.com

Note:

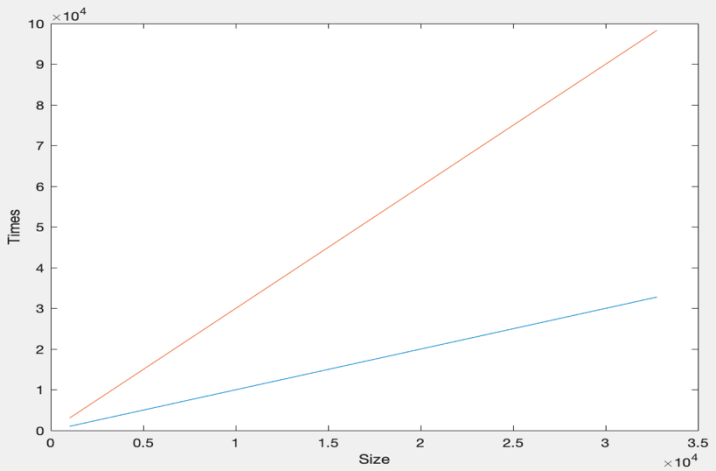
PLEASE first read the 'README' for data file reading part, if you want to test my code with other files.

Q1:

In this case, I use both dataset0 and dataset1 as an input dataset for my program. Therefore, there are two results for this program, and I will analyze them one by one.

In the *Table 1 and Plot* 1, we can see that when we use the dataset0 which is already sorted, the times of comparison is linear with the size of data. In another word, it means in the best case that the data is already sorted, the time complexity is O(N). What's more, since the shell sort is 7-sort, 3-sort and 1-sort combined, the comparison times would be 3 times than the insertion sort. In a summary, when it's in the best case, the insertion sort would be faster than the shell sort.

	Insertion0	Shell0
Size	**Times**	**Times**
1024	1023	3061
2048	2047	6133
4096	4095	12277
8192	8191	24565
16384	16383	49141
32768	32767	98293



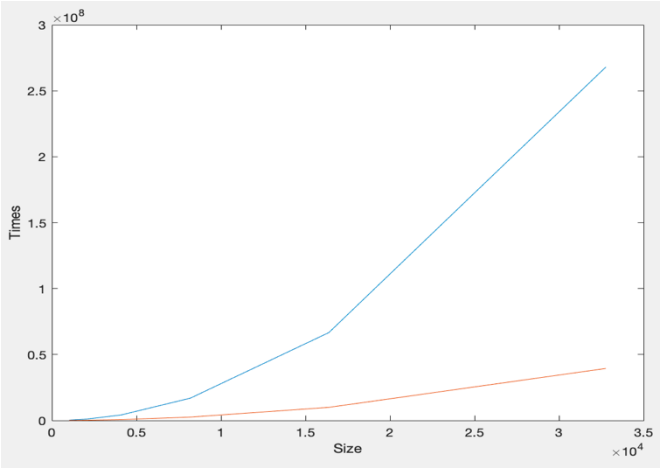
In the *Table 2 and Plot 2*, I use the dataset1 as an input for my program. In this case, the data hasn't been sorted and we can easily find that when the size of data grows larger, more effective the shell sort would be. Moreover, I use the Curve Fitting Tools in the MATLAB to do the power fitting and find that the time complexity of insertion sort in this case is almost O(N^2).

And the time complexity of the shell sort is less than O(N^2) which in the theoretical analysis is O($N \log N$).

So, we can give a conclusion that the shell sort is in some way much faster than the insertion sort, because the shell sort use h-sort to deal with the raw data first, which makes the raw data be pretreated. Then when the 1-sort execute, there would be fewer movements for this sort. This process just like to mill some corns, you should first make them more delicate and finally they become powder-like.

Insertion1 Shell1

Size	Times	Times
1024	265553	46728
2048	1029278	169042
4096	4187890	660619
8192	16936946	2576270
16384	66657561	9950922
32768	267966668	39442456

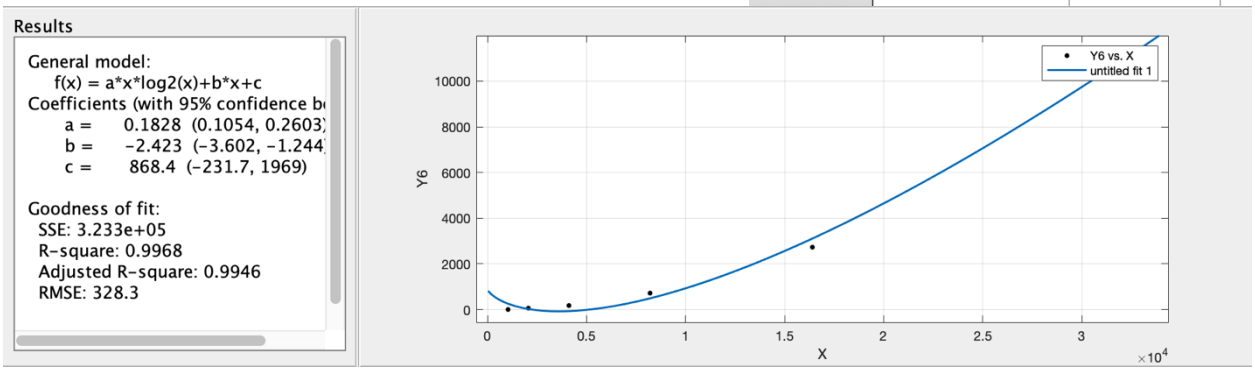


Q2:

I've found and borrowed some code from the websites I listed in my code, which helps me understand the deep meaning of Kendall Tau distance, and I find that the Kendall Tau distance problem is actually a sort problem. Since sort can somehow make up the inversion which in the Kendall Tau distance computation, the distance can be considered as an adaptive sort-comparison count. So, we can find that

Kendall_Tau_dist Time

Size	Times	Time(ms
1024	264340	11.988
2048	1053573	46.894
4096	4205430	178.991
8192	16801639	728.937
16384	67161944	2739.14



in the table on the right, the times which represents the adaptive sort-comparison count. And from the execution time in the table, I use Curve Fitting Tools in MATLAB to do the curve-fitting work (the figure below), find that it fits the O($N \log N$) really well. Therefore, we can make a summary that using the merge sort algorithm is quite an effective way to compute the Kendall Tau distance and the time complexity is near O($N \log N$).

Q3:

According to this problem, the difference between the normal sorting problem is that the dataset contains too much same elements, and moreover, the same elements are always stored together. So, in this case, I would like to consider the unit for elements is 1024, which means 1024 elements would team up as a new 'big' element. Then we can choose Bubble Sort or Insertion Sort and use them on the 'big' element unit, then, it's quite easy to finish this problem. Since some sorting method like Quick Sort and Merge Sort would contains recursive operations, it's hard to deal with both judgement of 'small' elements and sorting of 'big' elements.

Q4:

I used the dataset0 and dataset1 as inputs to test both method--*Merge Sort and Bottom-up Merge Sort*. The two tables below are the results with separately using dataset0 and dataset1 as input. From the results, we can find that when the dataset is fixed, no matter which merge method is used, the times of comparison are always the same. So, we can make a summary that the merge methods work with same times of comparison. What's more, from the tables, we can also find that the comparison times are different from different datasets when the size of dataset is fixed.

	Merge0	BUMer
Size	**Times**	**Times**
1024	5120	5120
2048	11264	11264
4096	24576	24576
8192	53248	53248
16384	114688	114688
32768	245760	245760

	Merge1	BUMer
Size	**Times**	**Times**
1024	8954	8954
2048	19934	19934
4096	43944	43944
8192	96074	96074
16384	208695	208695
32768	450132	450132

Q5:

In this case, I first used the dataset0 and dataset1 as inputs to test the original Quick Sort, which without any performance improvements. However, when I test the execution time of

Quick Sort and Merge Sort, I find that there's a really big difference between these two sorting-method. So, I checked the code again and found that I used an 'assert' in the Merge Sort algorithm and it really takes a lot of time to check the data is sorted or not, since the Merge Sort is recursively to sort every single data. Finally, I fixed this and give out the following two tables. According to these two tables, we can find that Quick Sort in some way is really faster than the Merge Sort, however, in the meantime, Merge Sort is much more stable than the Quick Sort. I think there's really a tradeoff between speed and stabilization.

Moreover, we can find that when the size of data grows up, the cut-off insertion sort would reduce the execution time and improve the algorithm. Since the Insertion Sort is an O(N^2) algorithm and Quick Sort is an O($N \log N$) algorithm, it would be easy to understand why in the small dataset, Quick Sort with cut-off would cost more time. In the meantime, when the data size get larger, the insertion-cut-off would reduce the recursion times the Quick Sort would take and in this way, it would reduce the execution time.

	Merge	Quick	Cutoff
Size	**Times**	**Times**	**Times**
1024	0.165	0.068	0.116
2048	0.355	0.136	0.139
4096	0.755	0.289	0.255
8192	1.546	0.61	0.547
16384	2.553	1.364	1.269
32768	4.679	3.424	2.719

	Merge	Quick	Cutoff
Size	**Times**	**Times**	**Times**
1024	0.215	0.155	0.138
2048	0.501	0.364	0.291
4096	0.979	0.708	0.623
8192	2.173	1.627	1.342
16384	4.269	3.601	2.939
32768	9.191	7.144	6.178

From the two figures, we can easily find that these three curves' growth rates are much near the time complexity O($N \log N$), which we've already known theoretically. Moreover, it's the relationship of execution time between these three sort-method.