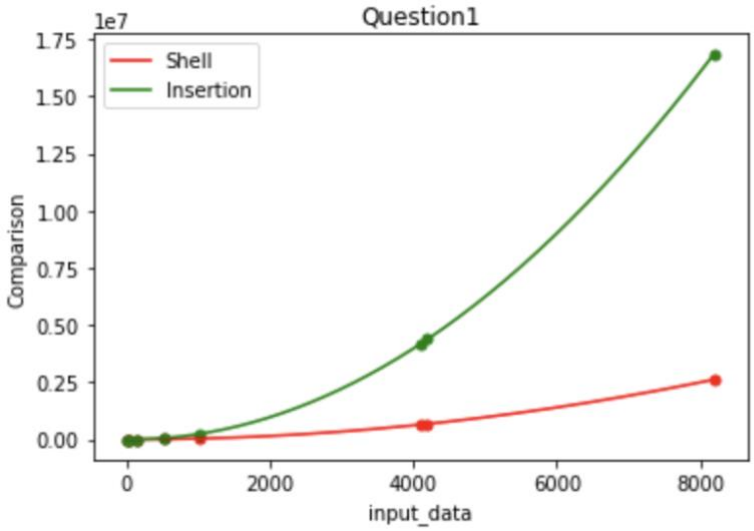
Xin Yang (xy213)
Report of HW 2

Q1:
In this question, I used the datasets in homework 1. As can be seen from the performance diagram, the number of comparisons of shell sort is much smaller than the comparisons made in insertion sort.

In this case, the datasets are much larger than the steps, so the 7, 3, 1 steps can be an effective way to per-sort the original datasets. Since shell sort is based on insertion sort and insertion sort has better performance on sorted data, shell sort with steps 7, 3, 1 can improve the order of the original datasets to avoid the worse cases.

Also there are only three steps so the pre-sort part won't pull the overall performance down.

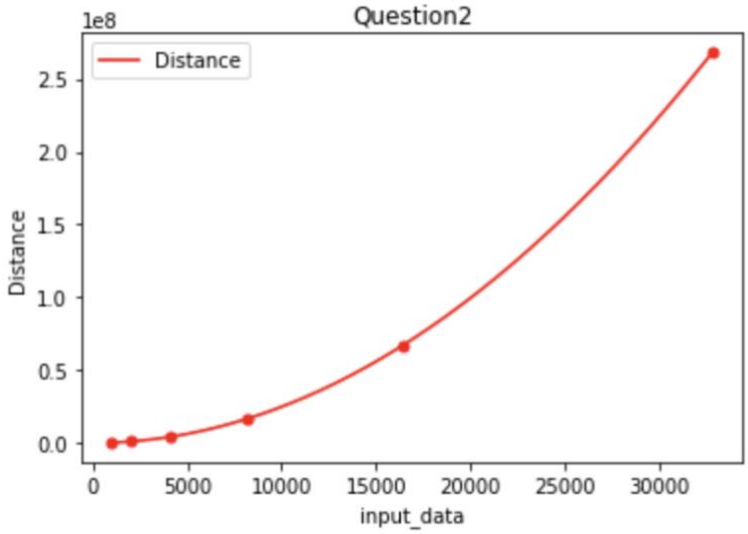


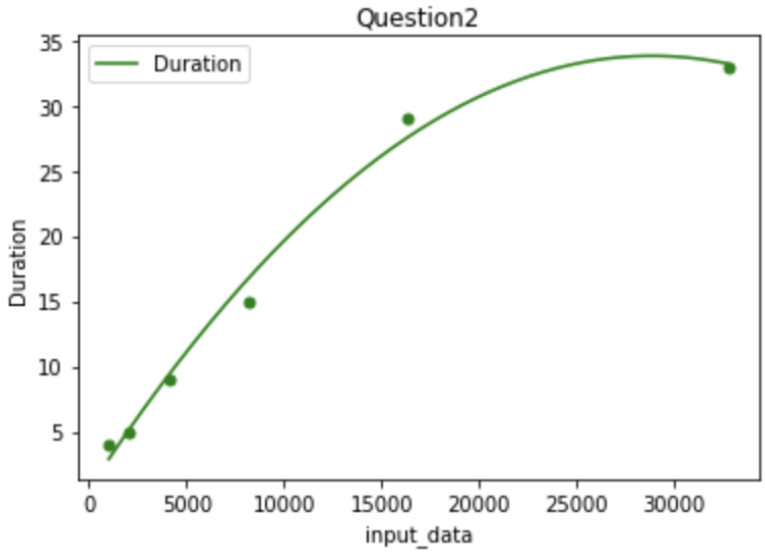
Q2:
To calculate the distance, we can assume the sequence in the first file as the standard reference, and the order in the second file should be rearranged to fit the new standard. The number needs to be exchanged reflects the Kendall Tau distance.

In order to get the Kendall Tau distance with less than quadratic time, I used merge sort as merge sort is a stable sort method, and by analyzing the rule of Kendall Tau distance, we can figure out that when we are doing merge sort, when it comes to the merge part, if the first element of the second array is smaller than the first element of the previous one, then the number of the previous array is the distance for that specific number. By implementing merge sort, the complexity is O(N*logN).

Number	Distance	Duration
1024	264541	4
2048	1027236	5
4096	4183804	9

8192	16928767	15
16384	66641183	29
32768	267933908	33





Q3:

The description of this question is not clear enough as if the data sequence is as described, then it's already sorted. To prevent this situation and make it more meaningful, I suppose the datasets are generated in a random order with fixed repeats as described. The data generator is in the source code. Now I have this dataset, let's also assume that I don't know how many times each number repeats, which means all I know is this dataset contains 4 numbers: 1, 11, 111 and 1111.

Since I already knew there're only four distinct numbers. I can go through the dataset only once and count how many times each number repeats. This is an O(N) operation with an array sized four. Then I'll go through the dataset the second time and change the value of each element according to the counter array we got. E.g. If we count there are 1024 repeats

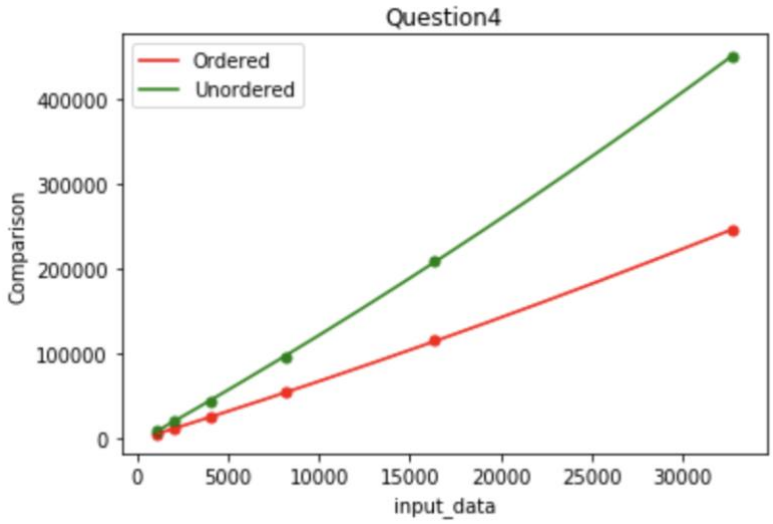
of digit 1, since 1 is the smallest, we set the first 1024 elements of the original dataset to be 1. The rest parts are similar.

In this case we only go through the dataset twice so it's an O(N) algorithm, but this performance is based on the prerequisite that we already know there are only four distinct numbers and what exactly they are.

Q4:
In this question, the comparisons of two implementations of merge sort are exactly the same so I plot only one line for one dataset.

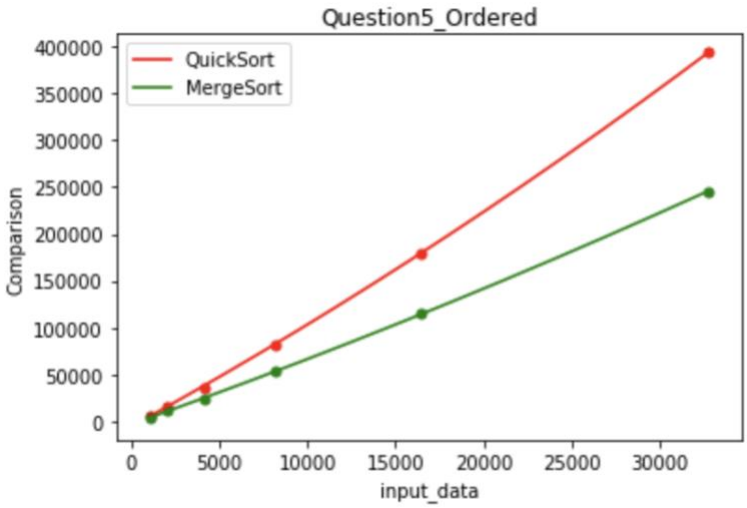
Since the datasets are all even numbers, the comparisons of the two ways are the same because the only difference is the order of comparisons, the recursive one will merge the previous part first then the back part, but the bottom-up one will merge the whole part and enlarge the merge range. But if the datasets are odd numbers, the result will be different as the recursive one will divide the whole part into two parts with different length, and this can cause the difference between the bottom-up method.

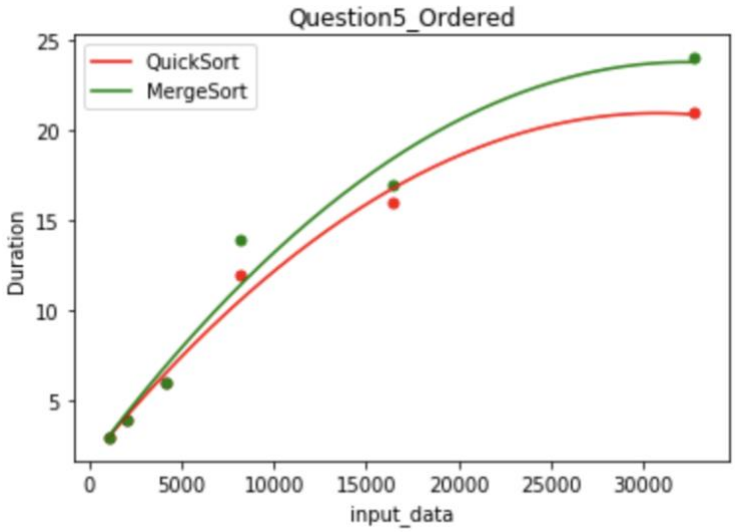
The difference between sorted and unsorted datasets are caused by the convenience of merge part. If it's already sorted, after comparing the previous(smaller) part, we can directly put the after part into the rest positions without doing other comparisons.

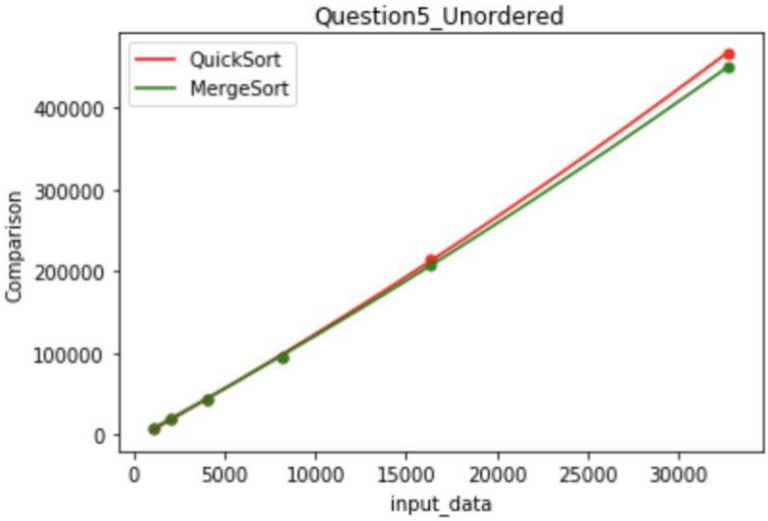


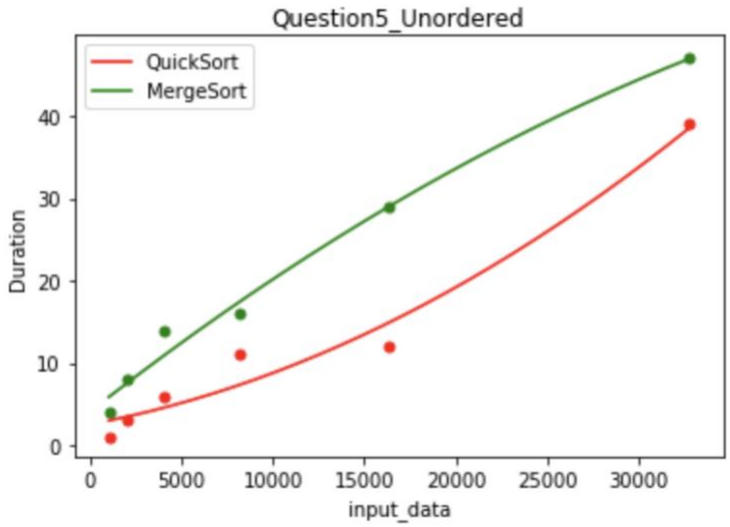
Q5:
The four diagrams below show the comparisons between quicksort with median-of-three and merge sort with recursion in question 4. Datasets are classified as sorted and unsorted. The comparisons of merge sort are fewer than quick sort in Sorted conditions as merge sort can save lots of comparisons with sorted sequence as mentioned in question 4. But when it comes to unsorted arrays, the comparisons of two sorts are very close as they are all algorithms based on comparisons and this is the average case situation for both, since they are all O(N*logN) algorithms, getting this result is as expected.

Opposite to the comparisons, the running time of quick sort is faster than merge sort because merge sort wastes lots of time on determining the corner situations and copying data between auxiliary arrays. Meanwhile the median-of-three method avoids the worst case and make quick sort more efficient on a random sequence.



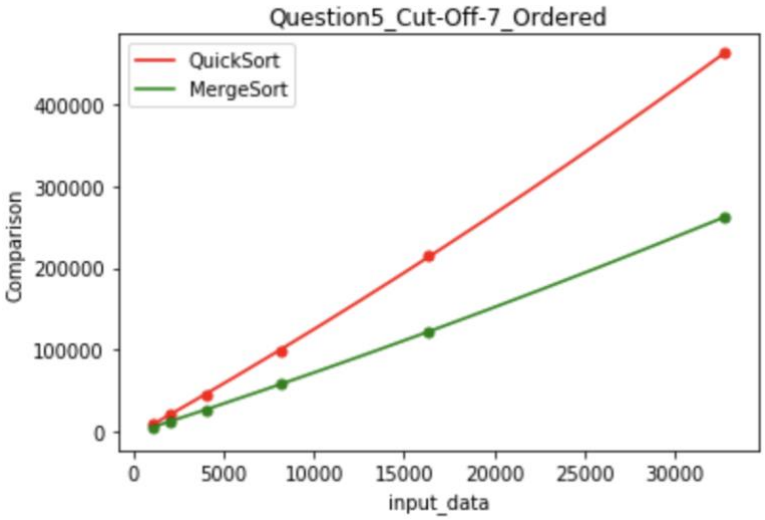


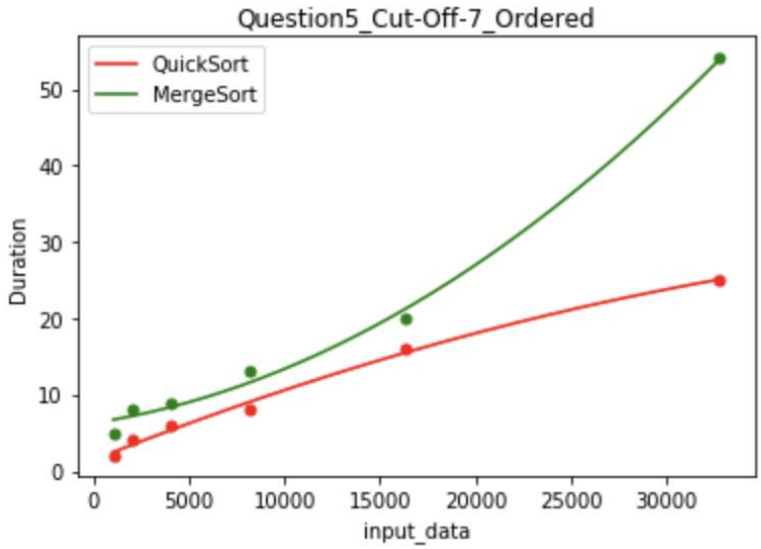


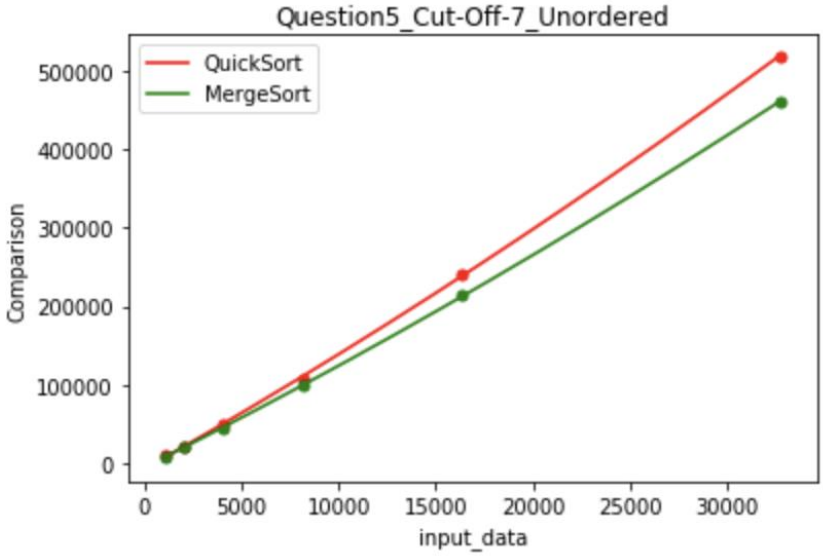


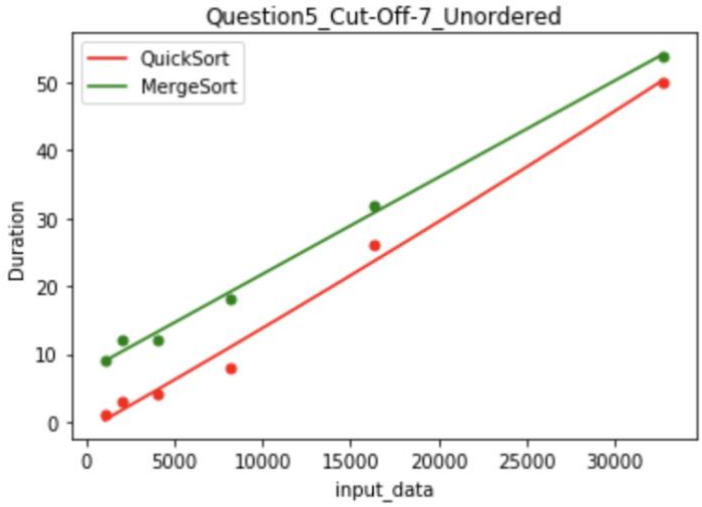
Cut-off:
When cut-off is set to be 7, we can plot the following four diagrams. Since the sorted datasets can't reflect the actual performance of algorithms, I'll only test on unsorted datasets for other cut-off values, and here the sorted results are only for reference.

Putting our focus on the unsorted result, the main trends are the same as the analysis above, but the difference between two algorithms are quite small. Quick sort is still faster than merge sort with even bigger number of comparisons.



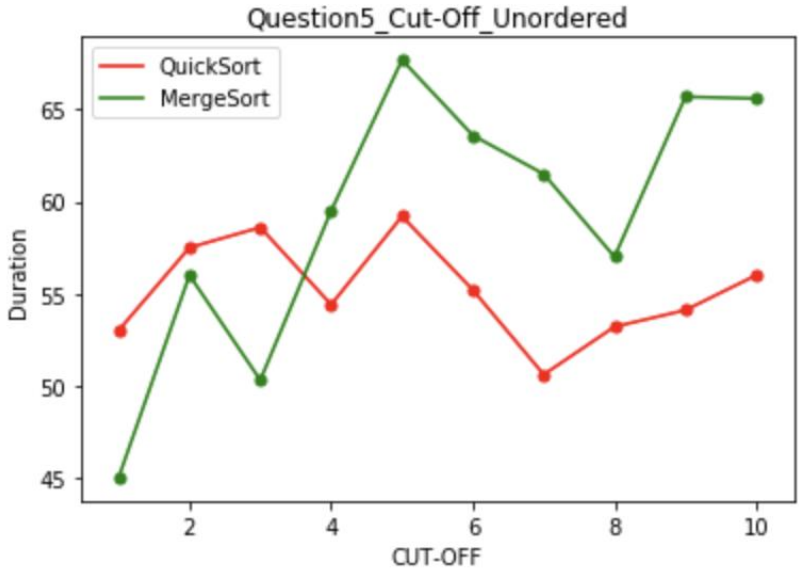

Question5_Cut-Off-7_Ordered
QuickSort
MergeSort
Duration
input_data
0
10
20
30
40
50
0
5000
10000
15000
20000
25000
30000


Question5_Cut-Off-7_Unordered
QuickSort
MergeSort
Comparison
input_data
0
100000
200000
300000
400000
500000
0
5000
10000
15000
20000
25000
30000


Question5_Cut-Off-7_Unordered
QuickSort
MergeSort
Duration
input_data
0
10
20
30
40
50
0
5000
10000
15000
20000
25000
30000

To experiment on the value that makes the result invert, I used the unordered dataset with 32768 data, and tested the CUT-OFF from 1 to 10. The performance is valued by the runtime, and to avoid noise, each value is tested for 8 to 10 times and calculate the mean.

As can be seen from the diagram below, the overall performance is the best at around 7 as CUT-OFF, and starting from 4, the performance of quick sort is better than merge sort. But when CUT-OFF is smaller than 3, we can see from the table that the previous result inverts and merge sort become the faster one, but we can keep our suspicion since the 32768 dataset is still small and I'm not 100% sure this result can remove the noise factors completely.



Q6:
(5), (6), (1), (4), (3), (8), (2), (7)