ANALYSIS REPORT FOR ASSIGNMENT-2

NAME- MAREESH KUMAR ISSAR
NET ID- mi251
COURSE- 16:332:573 DATA STRUCTURE AND ALGORITHMS
DATE OF SUBMISSION- 26TH FEBRUARY,2019

ANSWER 1

SHELL SORT (WITH INCREMENTAL SEQUENCE OF 7,3,1) AND INSERTION SORT

The data collected (no. of comparisons) from the shell sort implementation and the insertion sort implementation program is given in Table 1 along with its plots (Figure:1-For all the data0 files) and (Figure:2- For all the data1 files).

DATA FILE	INSERTION SORT (NO.OF COMPARES)	SHELL SORT (NO.OF COMPARES)
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	265564	46768
data1.2048	1029283	169081
data1.4096	4187899	660673
data1.8192	16936958	2576322
data1.16384	66657566	9950984
data1.32768	267966675	39442505

Table:1

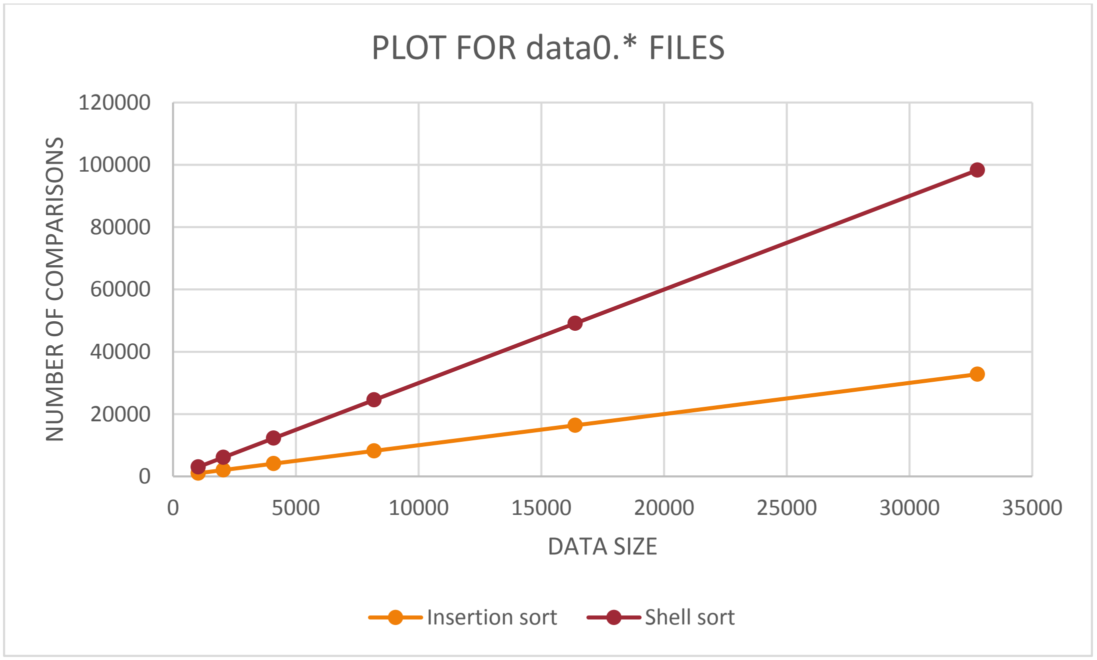

Figure:1

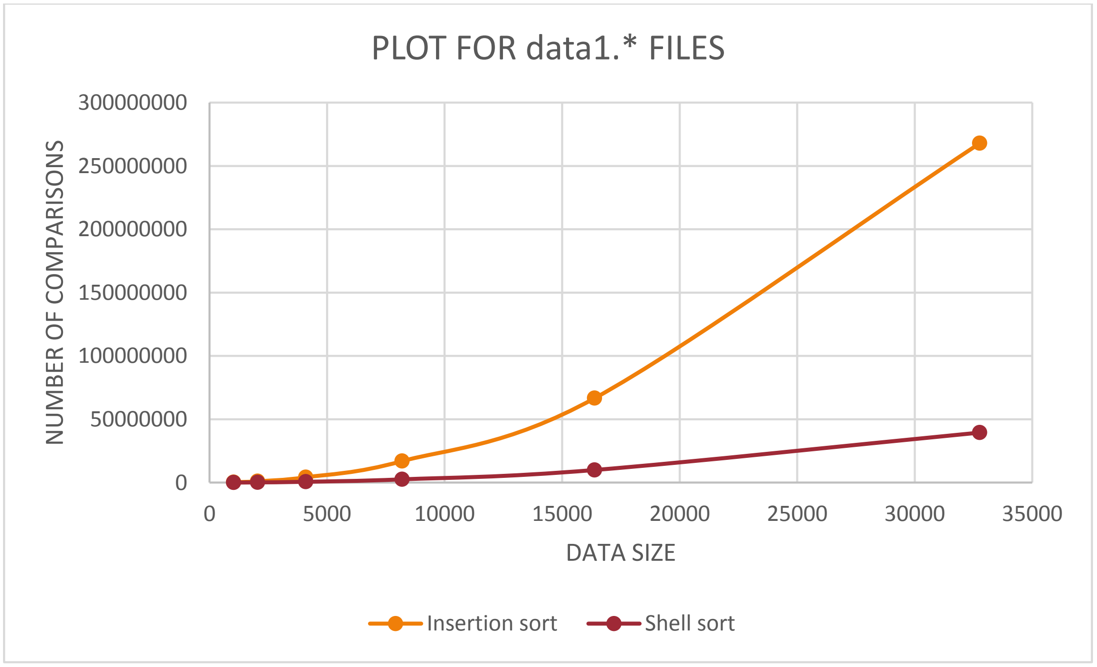

Figure:2

From Figure:1, we find that the number of comparisons done in the case of shell sort is more than those in insertion sort, this is due to the fact that the numbers in these files are already sorted. So, shell sort runs extra comparisons for the sequence number 7 and 3 in addition to 1(which is the same for insertion sort).

In the case of all the data1.* files (Figure:2), we find that the shell sort takes less number of comparisons as compared to insertion sort, i.e. shell sort is more effective than insertion sort. This is due to the fact that we first sort the numbers in the file with the increment sequence of 7 and 3 before doing it for 1(insertion sort). Therefore, by the time shell sort reaches to the 1(insertion sort) stage, the sequence of numbers is already partially sorted and we know that for partially sorted arrays insertion sort runs in linear time. Thus there will be less number of comparisons in this case of shell sort as compared to insertion sort.

ANSWER 2

KENDALL TAU

The data collected from the kendall tau implementation program is given in Table 2. Distance is measured for all the sequences in data1.* files from the sequences present in their corresponding data0.* files.

DATA FILE	DISTANCE	TIME (IN SECONDS)
data1.1024	264541	0.000698
data1.2048	1027236	0.001905
data1.4096	4183804	0.004093
data1.8192	16928767	0.009863
data1.16384	66641183	0.018822
data1.32768	267933908	0.028913

Table:2

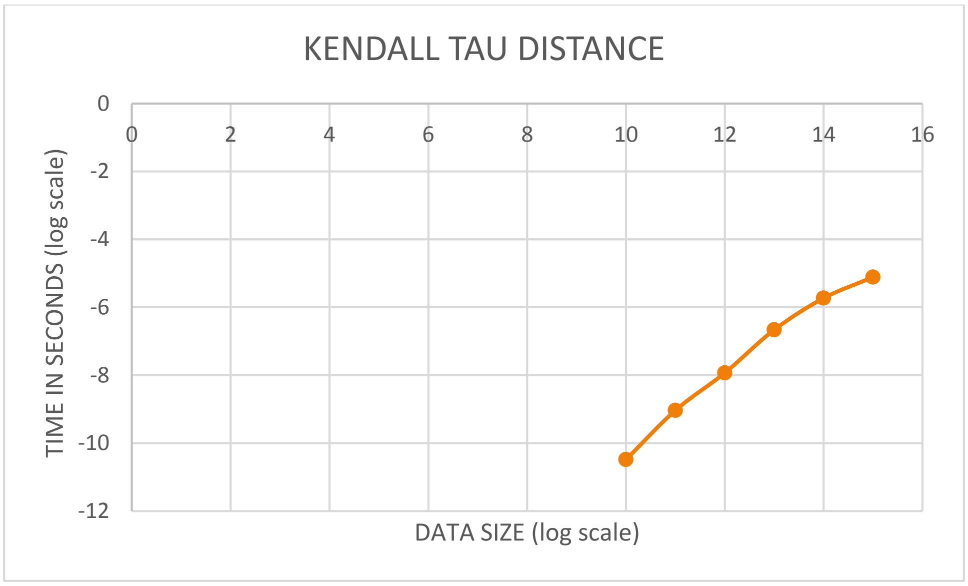

Figure:3

DATA SIZE (LOG SCALE)	TIME IN SECONDS (LOG SCALE)
10	-10.484
11	-9.036
12	-7.933
13	-6.664
14	-5.731
15	-5.112

Table:3

Table:3 shows the data (in log base 2 scale), while Figure:3 represents this data graphically.

To find the runtime cost of the implementation as a function of the input data size we use the following equations:

$$\lg\left(T\left(N\right)\right) = b\lg\left(N\right) + c$$
$$T\left(N\right) = aN^{b} \quad , a = 2^{c}$$

Here T(N) is the running time and N is the input size.

To find the value of a and b, we use the equation of a line passing through two points:-

$$y - y_1 = \frac{y_2 - y_1}{x_2 - x_1}(x - x_1)$$

Here, (x_1,y_1) and (x_2,y_2) are the two points.

For Kendall Tau implementation- using(11,-9.036) and (12,-7.933) we get b=1.103 and c= -21.169.

$$T\left(N\right) = 4.24 \times 10^{-7} N^{1.103}$$

From the above computed T(N) values, we find that the Kendall Tau implementation runs in less than quadratic time since the exponent of N is less than 2.

ANSWER 3

I think that the merge sort(with the practical improvements of insertion sort for smaller subarrays and stopping extra comparisons if the array is already sorted) will be the most effective algorithm for sorting the given data. This is due to the fact that the given sequence of numbers can be grouped into 4 categories, which then need to be merged efficiently. Even if we didn't knew anything about the structure of our initial data, we could use merge sort as it uses at most N X log N comparisons and is also a stable sorting algorithm.

After running the merge sort program (including the practical improvements), I found that the program took 48131 comparisons, which is less than N X logN (106496 for N=8192).

ANSWER 4

MERGE SORT(RECURSIVE) AND MERGE SORT(BOTTOM-UP)

The data collected from the merge sort (recursive as well as bottom-up) implementation program is given in Table 4 along with its plots (Figure:4-For all the data0 files) and (Figure:5- For all the data1 files).

DATA FILE	MERGE SORT (RECURSIVE)	MERGE SORT (BOTTOM UP)
data0.1024	5120	5120
data0.2048	11264	11264
data0.4096	24576	24576
data0.8192	53248	53248
data0.16384	114688	114688
data0.32768	245760	245760
data1.1024	8954	8954
data1.2048	19934	19934
data1.4096	43944	43944
data1.8192	96074	96074
data1.16384	208695	208695
data1.32768	450132	450132

Table:4

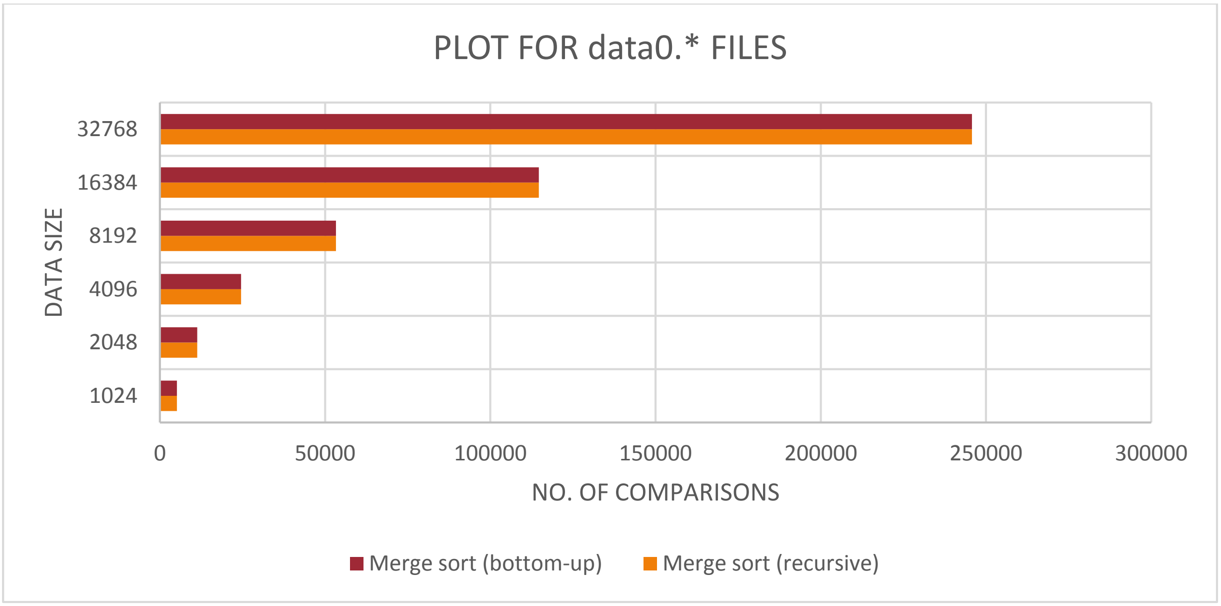

Figure:4

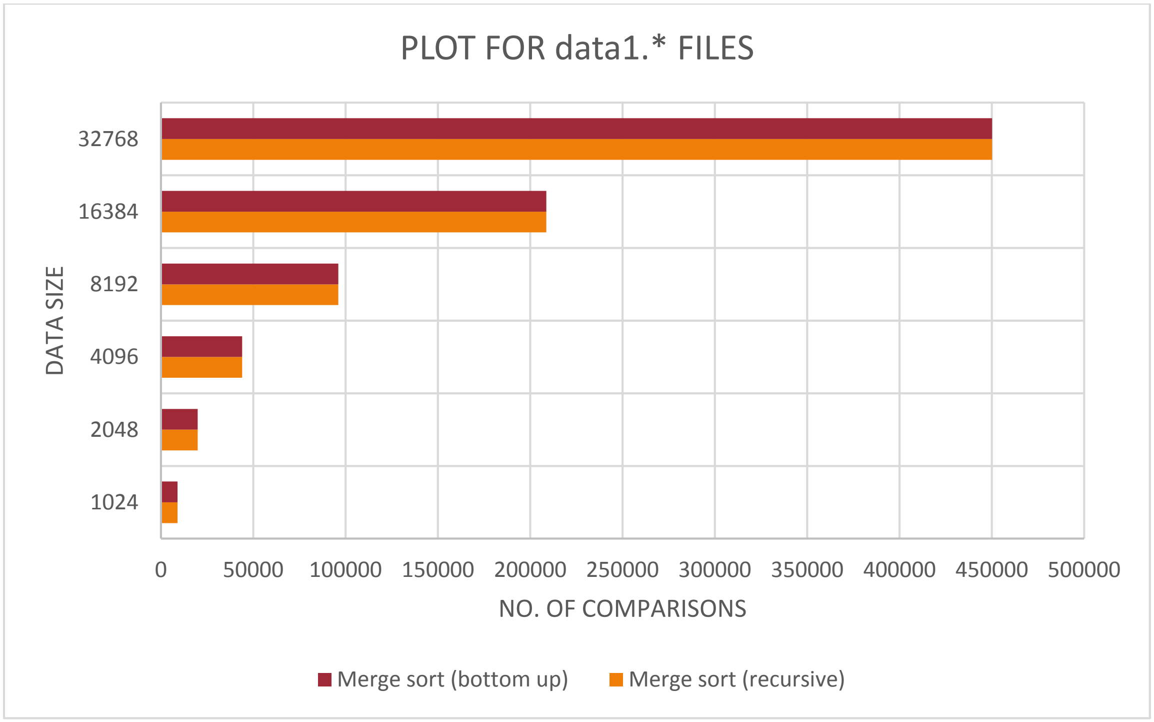

Figure:5

From Figure:4 and Figure:5 we can clearly see that both the approaches take the same number of comparisons for all the corresponding data0 as well as data1 files. This is because the data present in these files is in powers of 2 and both the implementations perform essentially the same merge operations but in different order. The number of comparisons needed for data0 files is less than those needed for corresponding data1 files because the data in the data0 files is already sorted.

ANSWER 5

MERGE SORT(RECURSIVE), QUICK SORT(WITHOUT CUTOFF) AND QUICK SORT(WITH CUTOFF=7)

NOTE: For quick sort implementation, I have not used any shuffling before doing quick sort.

The data collected from the merge sort (recursive), quick sort(without cutoff) and quick sort(with cutoff=7) implementation program is given in Table 5 along with its plots (Figure:6- For the data0.* files) and (Figure:7- For the data1.* files).

DATA FILE	MERGE SORT (TIME IN SECONDS)	QUICK SORT WITHOUT CUTOFF (TIME IN SECONDS)	QUICK SORT WITH CUTOFF = 7 (TIME IN SECONDS)
data0.1024	0.000565	0.000286	0.000214
data0.2048	0.001376	0.000652	0.000468
data0.4096	0.003014	0.001313	0.001071
data0.8192	0.006462	0.002819	0.002217
data0.16384	0.011824	0.005646	0.002736
data0.32768	0.014333	0.009183	0.008584
data1.1024	0.000967	0.000582	0.000579
data1.2048	0.001691	0.001468	0.000934
data1.4096	0.004112	0.003053	0.002797
data1.8192	0.009442	0.006566	0.005977
data1.16384	0.019967	0.013264	0.011283
data1.32768	0.035246	0.021014	0.018916

Table:5

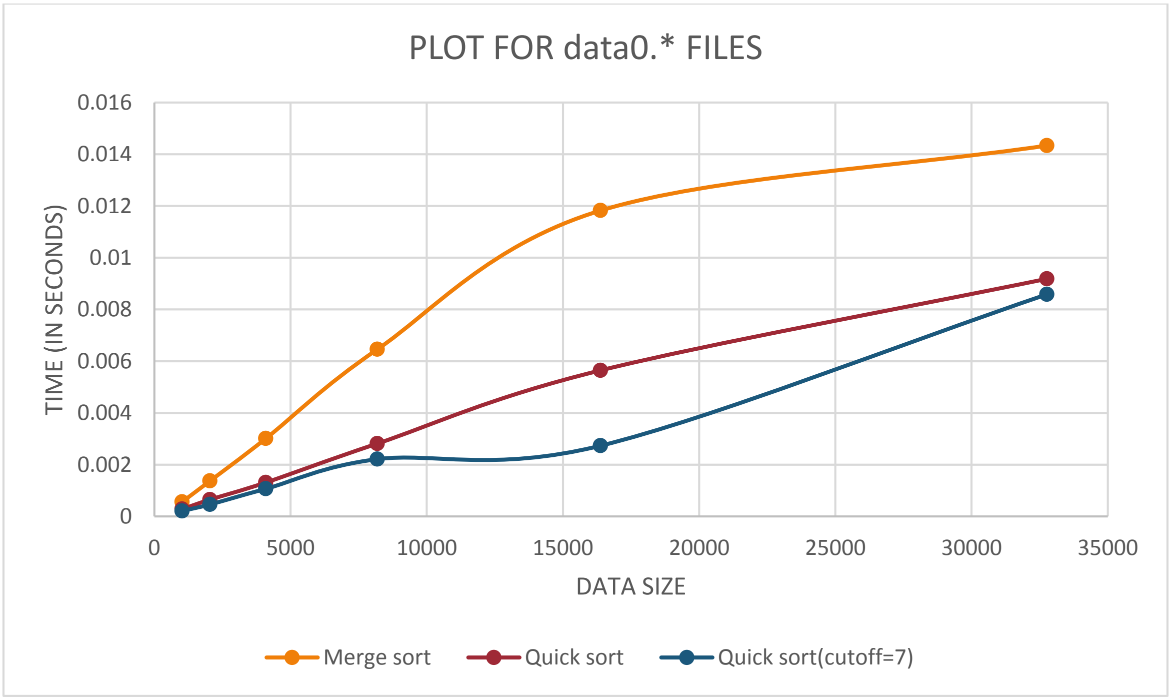

Figure:6

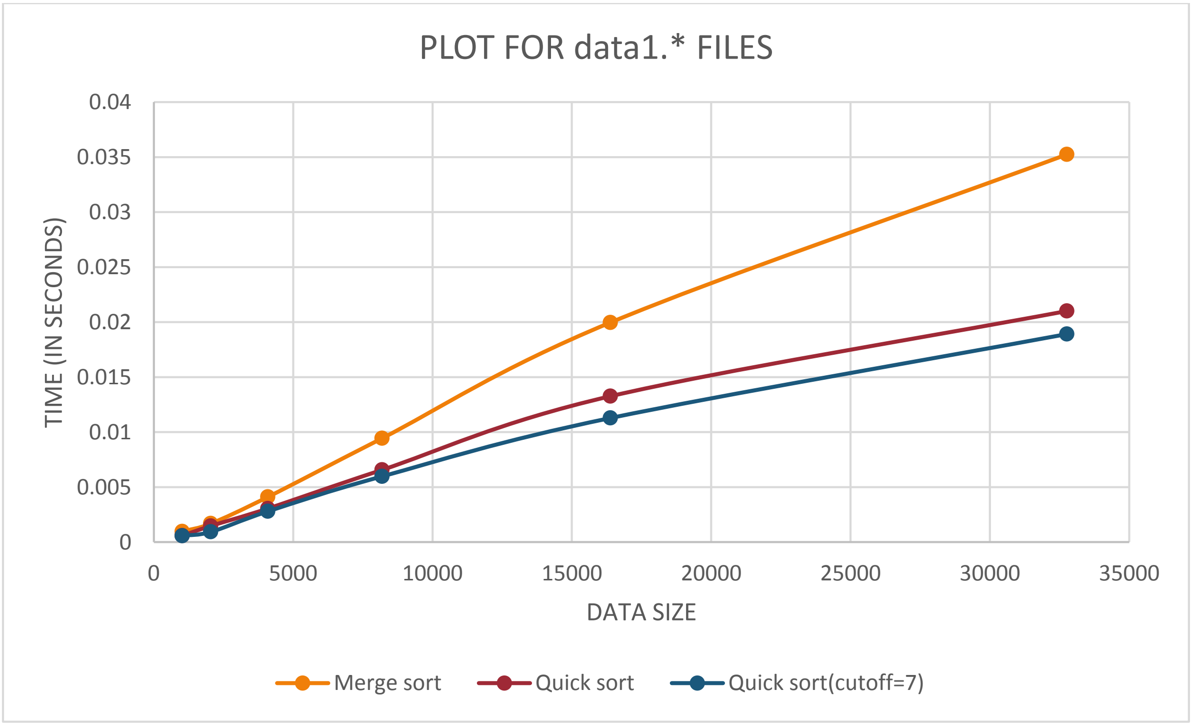

Figure:7

From the Figure:6 and Figure:7, we find that the quick sort (with cutoff=7) gives the best performance (shortest running time). Also, we find that quick sort, in general, performs better than merge sort. This is due to the fact that it does not uses a separate array as used in merge sort (quick sort does in-place sorting).

CUTOFF VALUE

The cutoff value data collected from the quick sort implementation program (using the file data1.16384) for different values of the cutoff is given in Table 6 along with its plot (Figure:8).

Table:6

CUTOFF VALUE	TIME (IN SECONDS)
5	0.012091
6	0.012754
7	0.011283
8	0.009182
9	0.012168
10	0.012165
11	0.009325
12	0.012405
13	0.012106
14	0.011626
15	0.010821

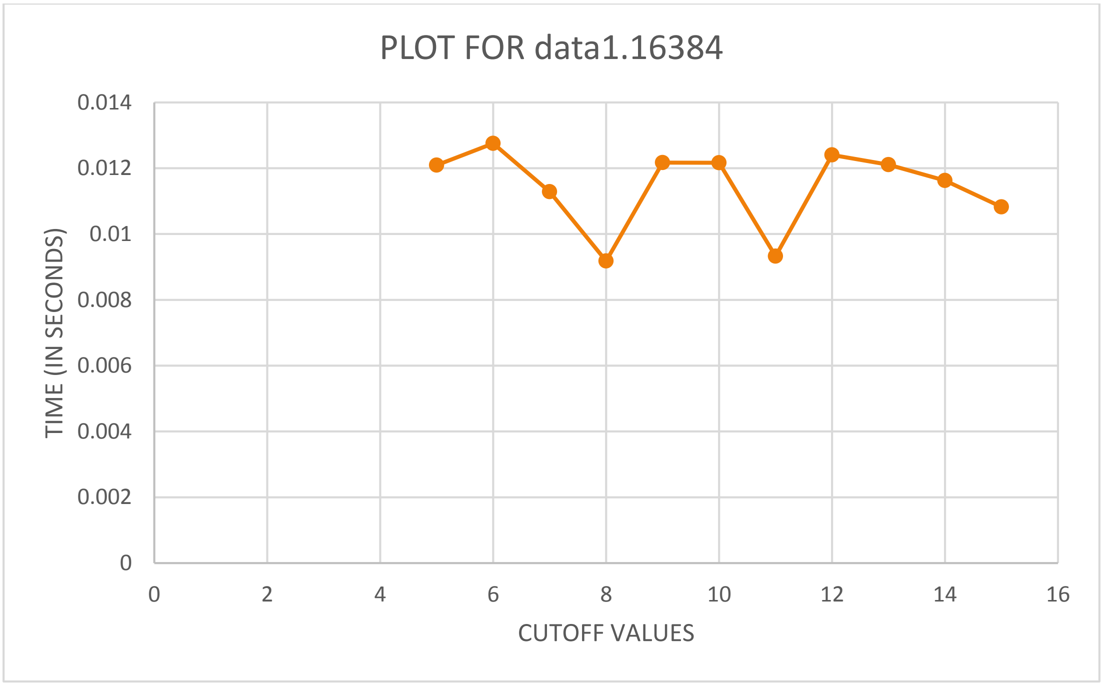

Figure:8

From, Figure:8 we find that at cutoff=8 and cutoff=11 there is a performance inversion.