This assignment may be done in groups of 2 or 3 people.

CSC505 Jennings Homework 2, Spring 2020
Please read, implement, and answer all 5 sections below.

Turn in one copy of the written part of the assignment on Gradescope, and enter the names of each team member during the submission process.

I. Names of team members:

Chandana Veeraneni

Shubham Miglani

II. Implementation:

In this assignment, you will use 3 sorting algorithms: bubble sort, quick sort, and merge sort. You may implement these algorithms yourself, or you may use an existing library, or you may obtain source code. (You may make different choices for each algorithm.)

Please indicate which single programming language you will use for this assignment:

Java 12
Python 3.6.9
C (C11 or ANSI)
C++ 17

Here, provide a citation for each algorithm. Indicate which members of your team implemented it, or which library it came from, or where you obtained source code.

We are using Python 3.6.9 for this assignment which is the default on Google Colab.

Bubble sort: https://www.geeksforgeeks.org/bubble-sort/

Quick sort: https://gist.github.com/JeremieGomez/fd3498e0b9df0980ad56

Merge sort: Python's Numpy library inbuilt function

III. Task:

Using the data files in the HW2 directory of https://github.ncsu.edu/jajenni3/csc505, run each of the sort algorithms, capturing the amount of user process time[1] spent in (1) reading the input data into memory, and (2) performing the sort.

You will sort the lines of each file by the timestamp field, which is the first field of each line. A space separates it from the rest of the line. The sort should be ascending, so the earliest time occurs first in the output.

Put your code and data into a GitHub repository on github.ncsu.edu in a branch called "HW2" (upper case). Give read access to the teaching staff (3 TAs and Dr Jennings – email addresses on Piazza). Put the repo URL here:

Code and data repo: https://github.ncsu.edu/smiglan/csc505

IV. Data collection and presentation:

For each sort algorithm, do the following:

1. Create a data table of user process times with these columns: file name, number of input lines, data load time, sort time, sum of load and sort times. Report all times in micro-seconds. 2. Plot the growth of the data loading time as a function of data size (in lines). 3. Plot the growth of the sorting time as a function of data size (in lines). 4. In another table, the “meta-data” table, indicate (1) how many times the sort was executed before timing data was recorded, (2) how many executions were performed, (3) whether the highest and lowest times were dropped; (4) whether the mean or median time is the one reported in the data table; (5) the operating system and version; (6) the CPU type, speed, and cache sizes; and (7) the type and size of disk holding the data.

Answers:

ATTEMPT 1 - Implemented standard code for the three sorting algorithms. Parsing of the string to datetime object was being done while comparison inside the sorting program. It was taking too long.

ATTEMPT 2 - All the elements parsed before sorting. Used numpy library inbuilt sort functions for merge sort algorithm. Bubble sort and Quicksort were defined from a source. Also timestamps here are parsed before the sorting happens, so that it will not interfere with sorting time. The conversion time for parsing has been included separately below. All time values are in milli seconds.

BUBBLE SORT

Note that the time for loading the data is taken for the complete file in case of 1Ma, 1Mb, 1Mc but the conversion and sorting are done for 50000 lines in case of bubble sort

File Name	Number of Input lines	Load time (ms)	Sort time (ms)	Sum (Load and sort)(ms)
syslog1Ma	1Ma	4701.76387	1490465.591	1495167.355
syslog1Mb	1Mb	4702.38018	1810549.755	1815252.135

syslog1Mc	1Mc	4542.36031	1073279.538	1077821.898
syslog200k	200,000	1284.428	29625310.66	29626595.09
syslog50k	50,000	107.748	1807249.466	1807357.214
syslog20k	20,000	39.083	265119.451	265158.534
syslog10k	10,000	21.948	56464.461	56486.409
syslog5000	5000	13.914	12150.198	12164.112
syslog2500	2500	13.023	2867.2	2880.223

Table with conversion time included is as follows:

File Name	Number of Input lines	Load time (ms)	Conversion time (ms)	Sort time (ms)	Sum(Load+Sort +Conversion) (ms)
syslog1Ma	1Ma	4701.76387	4697.840214	1490465.591	1499865.2
syslog1Mb	1Mb	4702.38018	4631.830454	1810549.755	1819883.97
syslog1Mc	1Mc	4542.36031	4662.544727	1073279.538	1082484.44
syslog200k	200,000	1284.428	20126.46794	29625310.66	29646721.6
syslog50k	50,000	107.748	4643.362999	1807249.466	1812000.58
syslog20k	20,000	39.083	1874.974489	265119.451	267033.508
syslog1	10,000	21.948	927.6373386	56464.461	57414.0

0k					463
syslog5000	5000	13.914	472.8088379	12150.198	12636.9208
syslog2500	2500	13.023	250.9803772	2867.2	3131.20338

QUICK SORT

File Name	Number of Input lines	Load time (ms)	Sort time (ms)	Sum (Load and sort)(ms)
syslog1Ma	1Ma	2691.686	73780.742	76472.428
syslog1Mb	1Mb	3142.543	94090.394	97232.937
syslog1Mc	1Mc	3624.7	74013.736	77638.436
syslog200k	200,000	630.538	16840.959	17471.497
syslog50k	50,000	201.956	3847.407	4049.363
syslog20k	20,000	47.158	1714.705	1761.863
syslog10k	10,000	19.777	873.991	893.768
syslog5000	5000	12.883	407.849	420.732
syslog2500	2500	6.398	199.678	206.076

Table with conversion time included is as follows:

File Name	Number of Input lines	Load time (ms)	Conversion time (ms)	Sort time (ms)	Sum(Load+Sort+Conversion) (ms)
syslog 1Ma	1Ma	2691.686	94785.81071	73780.742	171258.2387
syslog 1Mb	1Mb	3142.543	93723.52171	94090.394	190956.4587
syslog 1Mc	1Mc	3624.7	94218.24169	74013.736	171856.6777
syslog 200k	200,000	630.538	18955.15847	16840.959	36426.65547
syslog 50k	50,000	201.956	4789.223433	3847.407	8838.586433
syslog 20k	20,000	47.158	1846.100092	1714.705	3607.963092
syslog 10k	10,000	19.777	951.3478279	873.991	1845.115828
syslog 5000	5000	12.883	470.0260162	407.849	890.7580162
syslog 2500	2500	6.398	250.7231236	199.678	456.7991236

MERGE SORT

File Name	Number of Input lines	Load time (ms)	Sort time (ms)	Sum (Load and sort)(ms)

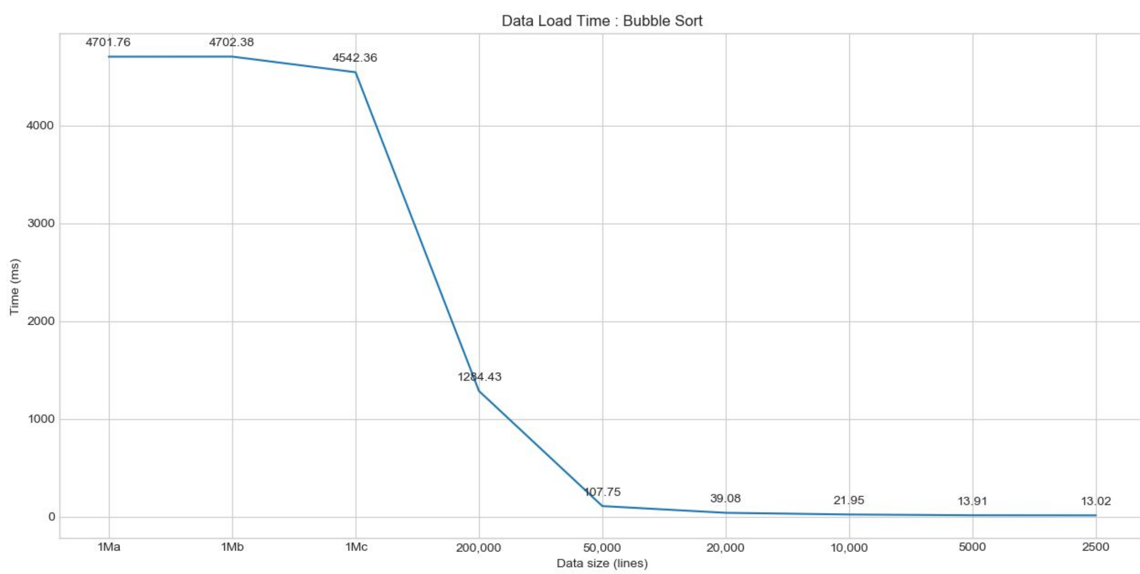
syslog1Ma	1Ma	2830.129	17871.122	20701.251
syslog1Mb	1Mb	3648.963	46797.095	50446.058
syslog1Mc	1Mc	2770.978	3483.818	6254.796
syslog200k	200,000	653.768	7866.815	8520.583
syslog50k	50,000	196.271	1719.295	1915.566
syslog20k	20,000	47.331	608.815	656.146
syslog10k	10,000	20.898	287.839	308.737
syslog5000	5000	16.388	125.405	141.793
syslog2500	2500	11.59	57.725	69.315

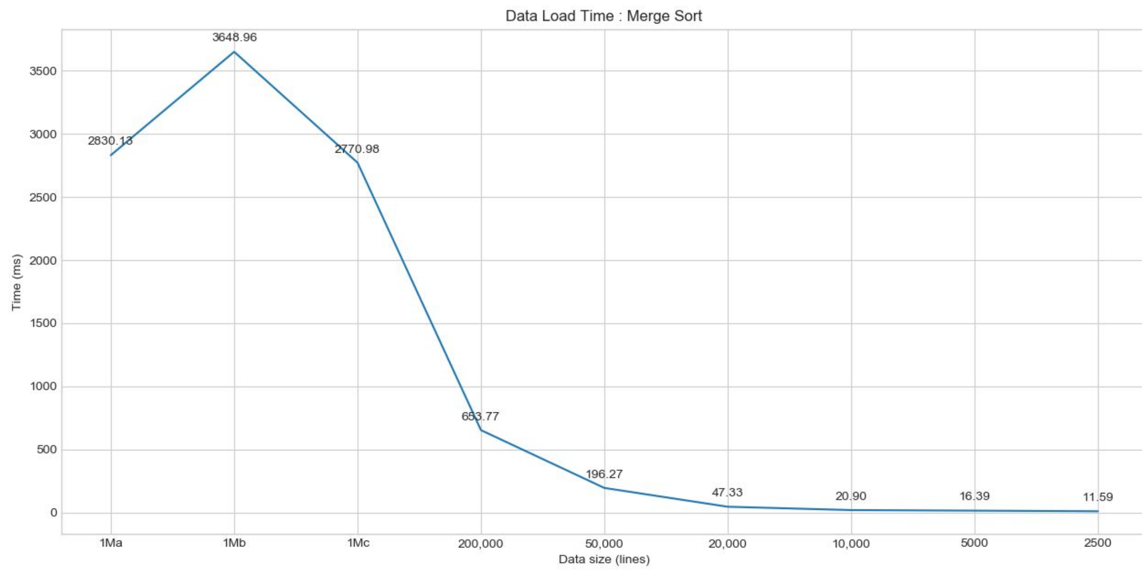
Table with conversion time included is as follows:

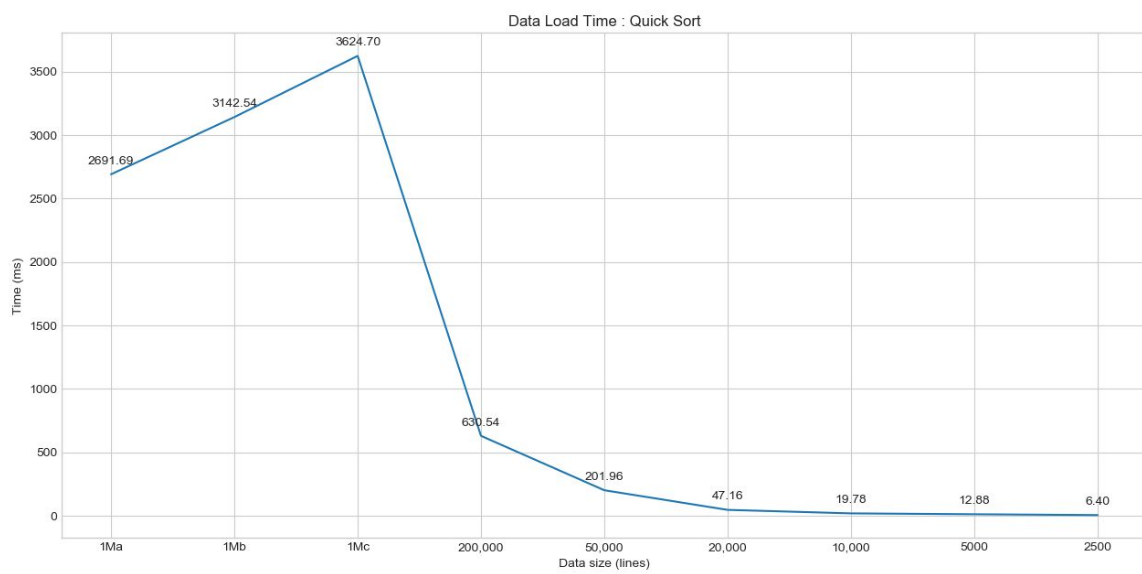
File Name	Number of Input lines	Load time (ms)	Conversion time (ms)	Sort time (ms)	Sum(Load+Sort+Conversion) (ms)
syslog1Ma	1Ma	2830.129	93923.2848	17871.122	114624.5358
syslog1Mb	1Mb	3648.963	94308.4309	46797.095	144754.4889
syslog1Mc	1Mc	2770.978	95527.3921	3483.818	101782.1881
syslog200k	200,000	653.768	18939.1954	7866.815	27459.77839

syslog50k	50,000	196.271	4847.15128	1719.295	6762.717279
syslog20k	20,000	47.331	1879.81892	608.815	2535.964916
syslog10k	10,000	20.898	966.694832	287.839	1275.431832
syslog5000	5000	16.388	467.103243	125.405	608.8962429
syslog2500	2500	11.59	242.733002	57.725	312.0480017

2. Growth of data loading as a function of data size

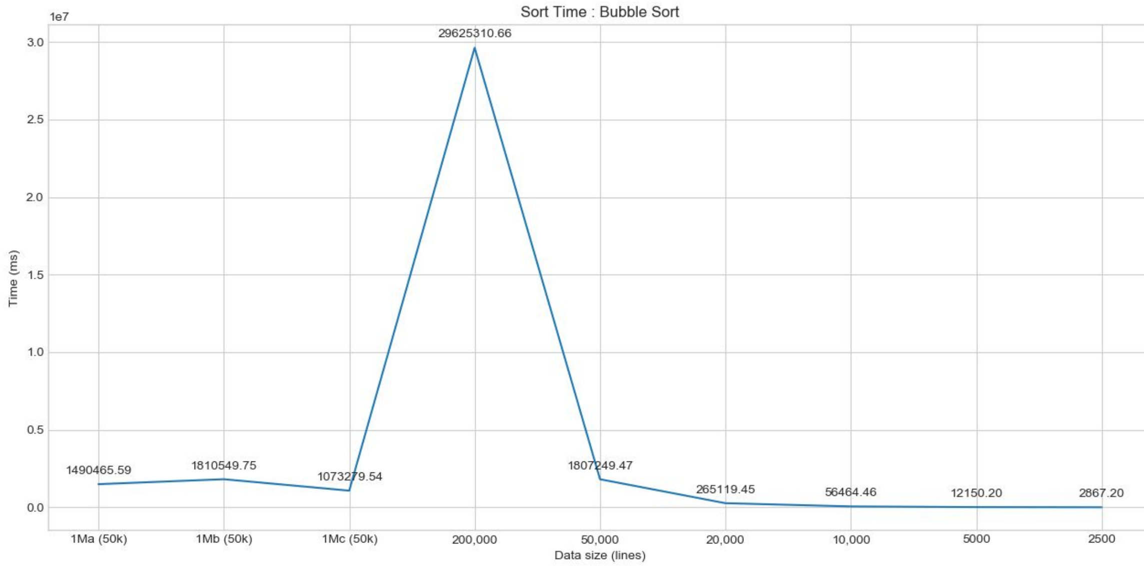


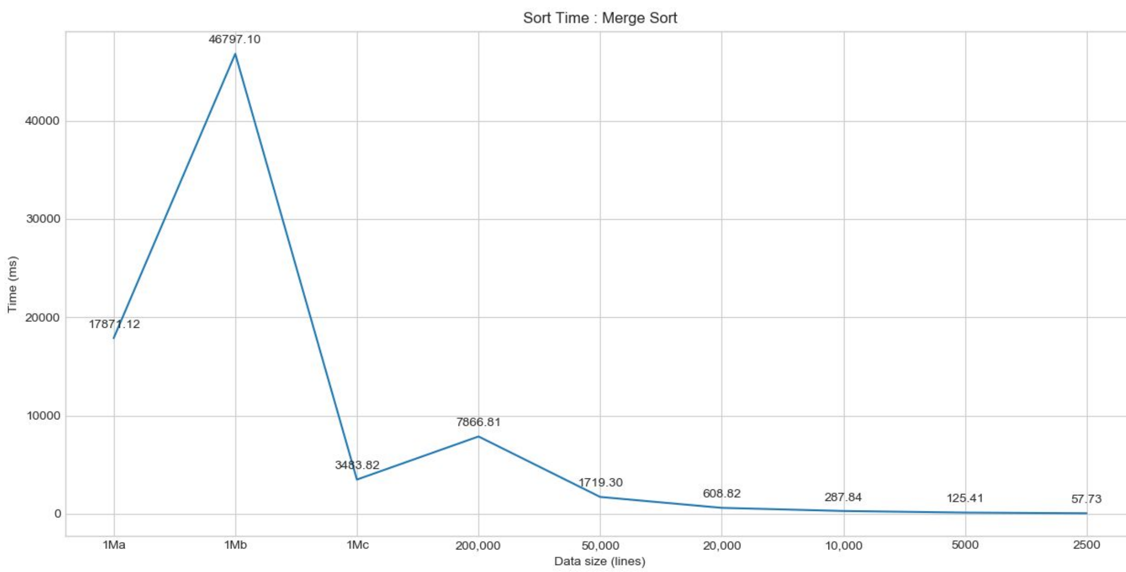


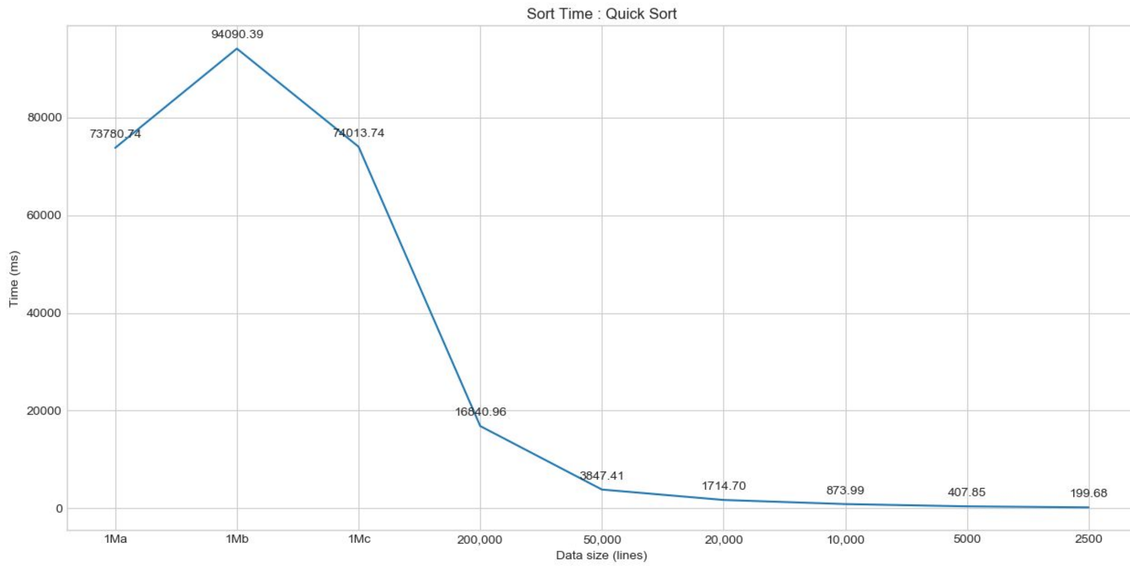


3. Growth of sorting time as a function of data size

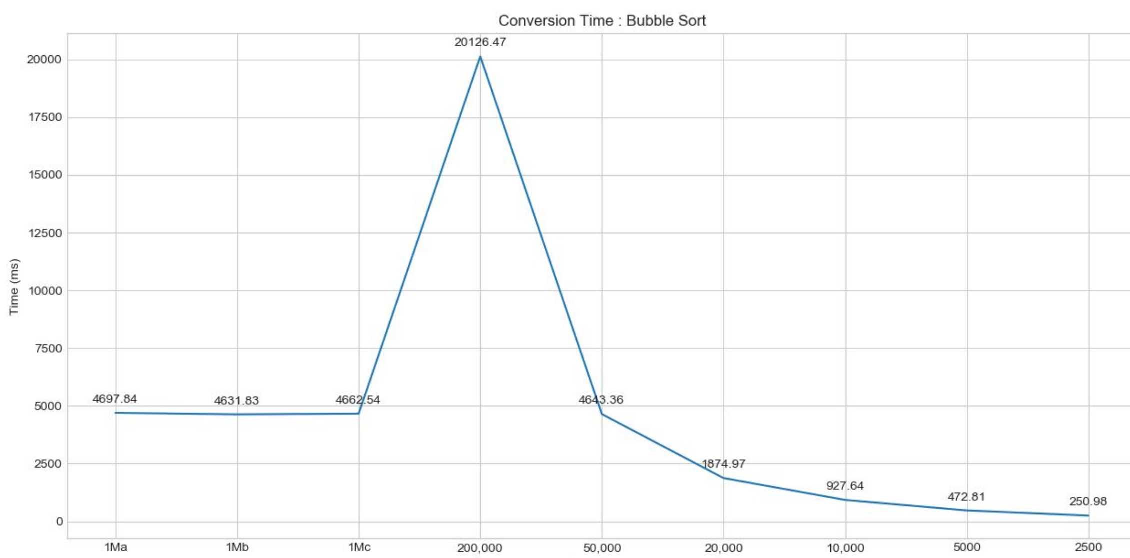
Note, that in bubble sort, for 1Ma, 1Mb, 1Mc, only 50000 lines are used due to excessive time requirements

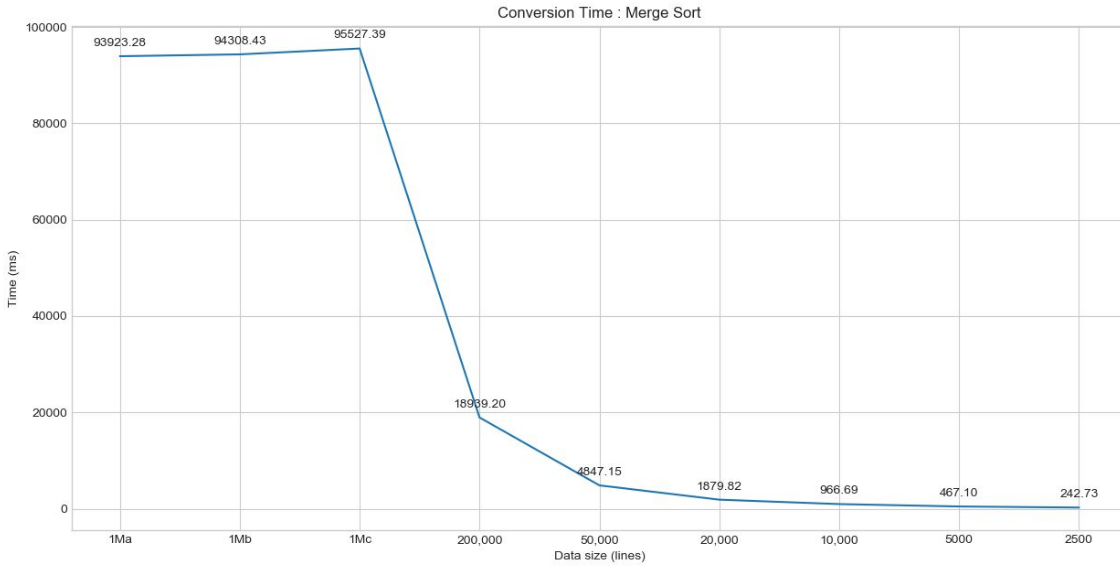

Sort Time : Bubble Sort
1e7
3.0
2.5
2.0
1.5
1.0
0.5
0.0
Time (ms)
29625310.66
1490465.59
1810549.75
1073279.54
1807249.47
265119.45
56464.46
12150.20
2867.20
1Ma (50k)
1Mb (50k)
1Mc (50k)
200,000
50,000
20,000
10,000
5000
2500
Data size (lines)

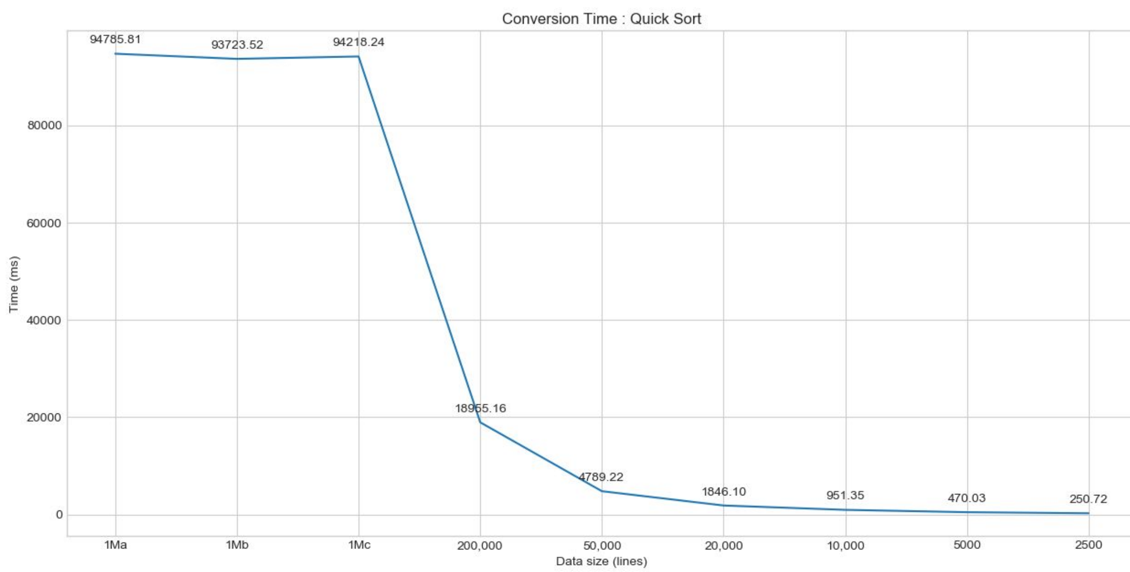

Sort Time : Merge Sort
40000
30000
20000
10000
0
Time (ms)
46797.10
17871.12
3483.82
7866.81
1719.30
608.82
287.84
125.41
57.73
1Ma
1Mb
1Mc
200,000
50,000
20,000
10,000
5000
2500
Data size (lines)



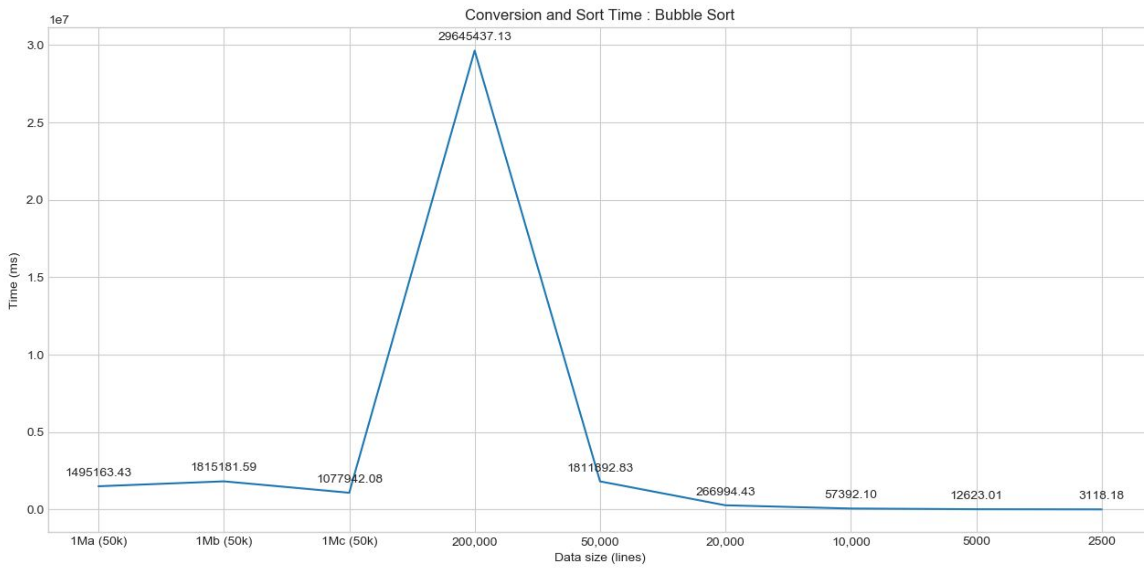
The plots for Conversion time are as follows:

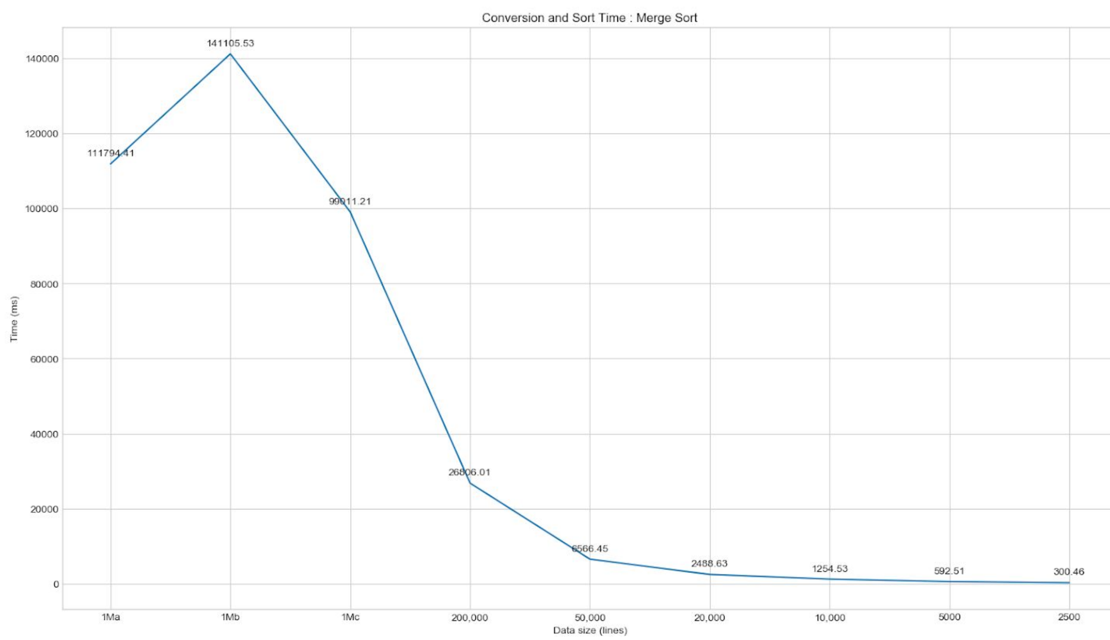


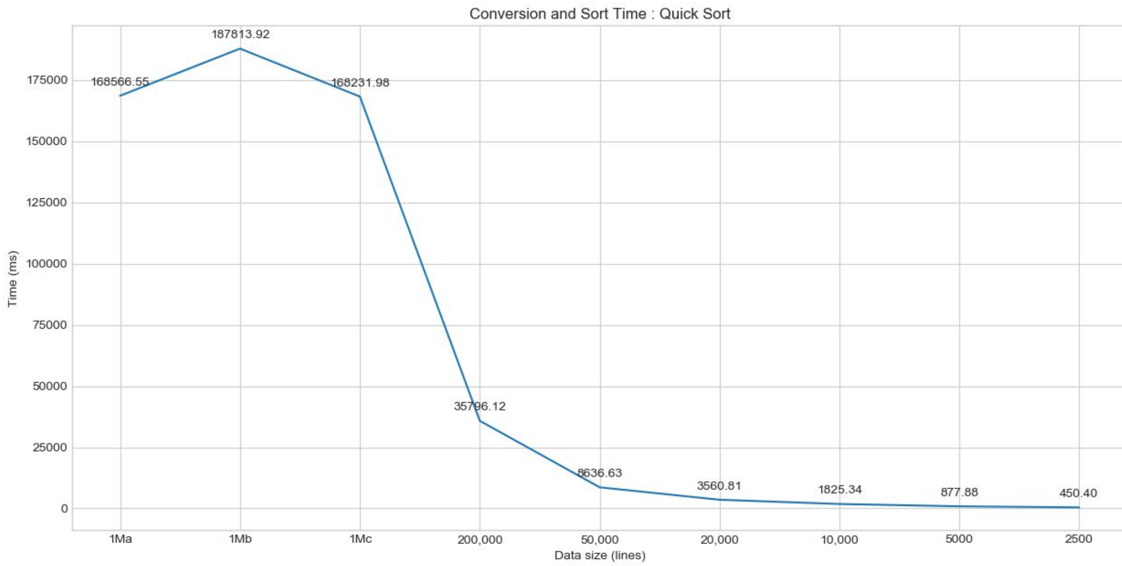




The plots for sum of Conversion and sorting time are as follows:


Conversion and Sort Time : Bubble Sort
1e7
3.0
2.5
2.0
1.5
1.0
0.5
0.0
Time (ms)
29645437.13
1495163.43
1815181.59
1077942.08
1811892.83
266994.43
57392.10
12623.01
3118.18
1Ma (50k)
1Mb (50k)
1Mc (50k)
200,000
50,000
20,000
10,000
5000
2500
Data size (lines)


Conversion and Sort Time : Merge Sort
140000
120000
100000
80000
60000
40000
20000
0
Time (ms)
111794.41
141105.53
99011.21
26806.01
6566.45
2488.63
1254.53
592.51
300.46
1Ma
1Mb
1Mc
200,000
50,000
20,000
10,000
5000
2500
Data size (lines)



4.

how many times the sort was executed before timing data was recorded	Each sort was performed two times before timings were recorded except Bubble sort for 20k files and above(0 for them)
how many executions were performed	Executions were performed 3 times for each sorting algorithm except Bubble sort for 20k files and above(1 for them)
whether the highest and lowest times were dropped	We have taken the last value (third) irrespective of highest and lowest
whether the mean or median time is the one reported in the data table	We taken the last value (third). In most cases, it's the median value
the operating system and version	Google Colab, CPU: 1xsingle core hyper threaded i.e(1 core, 2 threads)

	RAM~ 12.6 GB
the CPU type, speed, and cache sizes	Xeon Processors @2.3Ghz (No Turbo Boost) , 45MB Cache
the type and size of disk holding the data	Disk: ~107.77 GB

V. Analysis questions:

1. Describe the data structure you used to hold the data in memory. 2. Reproduce here the comparison function you used (show your code if you implemented it, or used existing source; otherwise, show the parameters passed to the sort routine and explain what they do). 3. For each algorithm, what order of growth did you expect to see? Compare to what was observed. 4. Comment on the relative amounts of time needed to load data versus sort data. 5. Three data files contained the same number of input lines (1 million = 10). Were any differences observed in execution times across these 3 files? Explain why or why not, for each algorithm. 6. How does your quicksort choose a pivot value? 7. Which of your algorithms sorts in place, and which needs additional memory? 8. Which of your algorithms is implemented recursively, and which iteratively? 9. Which of your algorithms implements a stable sort? How do you know? 10. Suppose the data were much larger, and did not fit easily into memory.

a. Which of the 3 algorithms being studied would you use to implement a solution? b. Describe (in just a few sentences) how such a solution would work. c. Based on your measurements, estimate how long (in user process time) your

proposed solution would require to sort *one trillion* (= 10) lines of data like the sample data. Assume that at most 10 lines will fit in memory at once, in total (input and working memory combined). State any other assumptions. Show how you obtained your answer.

1. Python built-in Data structure: **Lists** were used to hold the data in memory
2. **Bubble sort**

```
def bubblesort(inputdata):
  n = len(inputdata)
  for i in range(n):
    for j in range(0,n-i-1):
      if inputdata[j][0] > inputdata[j+1][0]:
        inputdata[j],inputdata[j+1] = inputdata[j+1],inputdata[j]
  return inputdata
```

The input given to the function is the content of the file. Every line of the file is split using space. The first element after the split is converted to a datetime object using parse for all the lines. So for comparing two lines, only their first elements are compared which are datetime objects.

The process is as follows: The first two elements are compared. If the first element is greater, then the lines are swapped. This process is continued with each element and its next element until the largest element bubbles up to the rightmost index of the list. After each iteration, the larger number ends up in the right. This is done till the bubble sort function sweeps the whole list without any swapping of lines.

Quick sort

```
import random
def quicksort(arr, start, end, pivot_mode='random'):
  if start < end:
    split = partition(arr, start, end, pivot_mode)
    quicksort(arr, start, split-1, pivot_mode)
    quicksort(arr, split+1, end, pivot_mode)
  return arr
def partition(arr, start, end, pivot_mode):
  if pivot_mode == 'first':
    pivot = arr[start]
  else:
    pivot_index = random.randrange(start,end)
    pivot = arr[pivot_index]
    arr[pivot_index], arr[start] = arr[start], arr[pivot_index] # place the pivot at the beginning
  i = start + 1
  for j in range(start+1,end+1):
    if arr[j][0] < pivot[0]:
      arr[i], arr[j] = arr[j], arr[i]
      i += 1
  arr[start], arr[i-1] = arr[i-1], arr[start]
  return i-1
```

quicksort(inputdata, start, end, pivot)

The arguments are the input data, start of the quick sort is from 0, end is upto the length of the file which is the end and the pivot is chosen as random. Instead of picking a first or last element, the pivot is chosen randomly as it makes it more likely the worst case complexity is not encountered

Merge sort

np.sort(file_content, kind = ‘ merge sort)

Numpy library has an inbuilt function ‘sort’ which supports quick sort, merge sort and heap sort. Np.sort is the function being used. Two parameters are passed to it, first is the content of the file which needs to be sorted and second, the kind of the sorting algorithm used.

The parsing for the string to convert to datetime was done before passing to the sort command.

3. **Bubble sort**

   Expected - Best Case O(n), Worst case - O(n^2)

   What we observed - The time for sorting was increasing exponentially as with increase in data. So, the expected and observed are same.

   Quick sort

   Expected - Best Case O(n log n),Worst case - O(n^2)

   What we observed - Quick sort had similar times to merge sort which is nlogn and was even faster in some cases. The worst case complexity for quicksort is high, but based on how its implemented, it can run faster than merge sort on average case so O(n^2) was not really observed for quick sort.

   Merge sort

   Expected - Best Case O(n log n), Worst case - O(n log n)

   What we observed - Merge sort followed the O(n logn) complexity and the results were as expected. It didn't grow exponentially with data but grew slower than that due to the log n factor.

4. Data loading time almost increases linearly with increase in sorting data. The time for loading 200k file was around 1.2s whereas for loading 1M file was around 4s.

5. Difference in execution of 1Ma, 1Mb, 1Mc

   Bubble sort - 1Ma took the average time of 1Mb (highest time) and 1Mc (lowest time). It might be due to the fact that 1Mc represents the best case, 1Mb represents the worst case and 1Ma is the average case situations.

   Quick sort - Syslog1Mb file had greater sort time compared to syslog1Ma and syslog1Mc files which had similar times

   Merge sort - Syslog1Mb file had greater sort time compared to syslog1Ma and syslog1Mc files. Syslog1Mc takes very less time as compared to the 1Ma file also.

6. The pivot value in quicksort is chosen randomly

7. Bubble sort sorts in place. It only needs a single additional memory space to store a temporary variable.

   Quick sort does not sort in place and is unstable. It uses a constant additional space before making each recursive call. The space efficiency of quick sort is log(n). If log(n) is considered okay for qualifying as in place algorithm as then it can be called in place sort depending on the definition of in-place

   Merge sort does not sort in place but is stable. It requires the memory size of the input to be allocated for the sorted output to be stored.

8. Bubble sort is implemented iteratively. Quick sort is implemented recursively. For Merge Sort, numpy library function is used. Looking at the source code, its also done recursively.

9. Bubble sort and merge sort implement stable sort. They keep the items with the same sorting key in order. A sorting algorithm is said to be stable if two objects with equal keys appear in the same order in sorted output as they appear in the input array to be sorted. Some sorting algorithms are stable by nature like Insertion sort, Merge Sort, Bubble Sort, etc. And some sorting algorithms are not, like Heap Sort, Quick Sort, etc.

10. a. The data can be sorted using Merge sort by using external sorting which uses merging technique

b. First divide the data into groups that can fit into memory. Then sort each of these groups and write them to disk. Then load the starting items from each group into the memory and output the smallest one to the disk and repeat these steps.

c. <Quick sort for 1 trillion files calculation>

Divide 10^{14} lines of data to 10^{6} blocks of size 10^{8} each and sort these groups.

Based on our calculation from merge sort, 17.8s(average case) is used

$17.8 = c * n * log(n)$ where n = 10^{6}

Using this time for n = 10^{8} is calculated = 2373.33s

This 2373.33 will be multiplied by 10^{6} blocks.

Total time = $2373.33 * 10^{6}s$

Then the time for number of passes it takes to merge and sort from these sorted lists will be proportional to the above time.

So total time = $C * 2373.33 * 10^{6}s$ where C is some constant depending on number of passes