CSC505-001 [Dr. Jennings]
Project 1 REPORT, Spring 2022

This document is a template for your Project 1 report, to be turned in on Gradescope by one team member. When submitting, be sure to list the other team members on Gradescope.

Refer to the Project 1 assignment document for additional information, including detailed requirements. See also the discussions on Piazza tagged as "project1".

CHECK THE DOCUMENT INDENTATION

Team members

1. Print the name and Unity ID (email ID) of each team member here:

   a. Bhuwan Bhatt, brbhatt (brbhatt@ncsu.edu)

   b. Aditya Tewari, adtewari (adtewari@ncsu.edu)

   c. Shubham Dua, sdua2 (sdua2@ncsu.edu)

Implementation

2. Give the URL to the NCSU GitHub Repository that contains your code and data:

   https://github.ncsu.edu/adtewari/CSC_505_Project_1

3. Add the following members of the teaching staff as collaborators to your repository so that we can examine your code and data:

   - ☑ ~~Professor: Dr. Jamie A. Jennings — Unity ID: jajenni3~~
   - ☑ ~~TA: Lauren Alvarez — Unity ID: lalvare~~
   - ☑ ~~TA: Qinjin Jia — Unity ID: qjia3~~
   - ☑ ~~TA: Yusuf Satici — Unity ID: msatici~~

4. What programming language did you use? Give name and version.

   Name - Python
   Version - 3.7

5. What was your main development platform(s)? Give OS name and version.

Linux (Ubuntu 20.04)
MacOS big Sur version 11.4

6. Give the origin of each sorting algorithm you used. Tell us whether it was built-in to the programming language (or part of its standard library), or it came from a library that you imported/loaded, or if you obtained source code for it, or if you wrote it yourselves. For each sort, also give the name of the sort function itself.

a. Insertion sort (an in-place sort)

We used this GeekforGeeks article with minor changes :
https://www.geeksforgeeks.org/insertion-sort/
The name of the function used in this project is <insertionSort>

b. Merge sort (the traditional top-down version, which allocates extra storage)

We used this GeekforGeeks article with minor changes :
https://www.geeksforgeeks.org/merge-sort/
The name of the function used in the project is <merge>

c. An *adaptive* sort algorithm

We used this GeekforGeeks article with minor changes :
https://www.geeksforgeeks.org/timsort/
The name of the function used in this project is <timSort>

7. Does your repository README contain instructions for how to build and run your code on the test data? YES or NO

YES - the README document contains instructions to run the code on all environments

8. Does your repository contain data files with the raw data that was generated by your programs and used to create the plots you will include below in this report? YES or NO

YES. There are two files - one for VCL and one for any other environment. After running the files, we will get the sorted log files. The plots generated after running all three algorithms on all three datasets are also added to the repository.

(Note that neither your plots/graphs nor your written report need to be in the repository. They can be there if you wish.)

9. The most straightforward approach to the sorting part of this project as follows:
   - The input is read into an array (or list) of strings, one for each line of the file.
   - The input array (of strings) is passed directly to the sort algorithm, along with a function that compares two lines, A and B, and returns -1 if A<B, 0 if A=B, and 1 if A>B (or something similar).
   - The comparison function, since it operates on strings, first extracts the substring corresponding to the timestamp. It does this for both inputs, A and B, and then converts the string representation into a native time representation. That conversion, and comparing the resulting times are done by library or built-in functions that vary across programming languages.
   - The sort returns a sorted version of the array of strings (or it modifies the array that was passed in as an argument). In either case, at this point it is easy to verify that the result is sorted by iterating through the array comparing consecutive elements.

   Did your approach differ from the outline above? If so, then describe how.

The approach is very similar to what is described above. The input is read into a list of strings. Then the list of strings is iterated through, and we split the log files and extracted the date time part from the log files. Then the extracted part was converted to date-time format using Python's inbuilt library <datetime>.

Then according to the user, a particular sort algorithm (one of merge, insertion, or tim sort) will run on a particular dataset (A, B, or C) according to the approach explained above. Finally, we obtain a graph that is made by plotting the number of lines in the logfiles against the time elapsed while running the algorithm.

Experimental Setup

10. On what platform were the timed experiments performed?
    a. Give the OS (name and version)?
       Mac , 8 gb ram,, 128 GB ssd
       MacOS big Sur version 11.4

    b. CPU type and speed?
       1.6 GHz dual core intel i5, Turbo boost upto 3.5 GHz

    c. The amount of memory (RAM)?
       RAM is 8 GB

    d. Was it a laptop, desktop, or vm?
       Laptop (no VM)

e. If a vm, was the vm running on a laptop, desktop, or server?
We did not use a Virtual Machine

f. Anything else you want to tell us about the experimental environment/platform?
N/A

11. Answer these questions about how you did your timing.

a. What framework, library, or function call was used to obtain the clock time?

We got clock time from the Python library “Time”. The inbuilt function used is <process_time>. We start a clock or stopwatch, run a sort algorithm on a dataset (A, B, or C) and then stop the clock. The difference between the two clocks gives us the floating point value of the runtime in seconds.

b. What is the unit of the clock value, e.g. milliseconds, microseconds, nanoseconds?

The unit of clock value we used for this project is seconds.

c. What time unit did your program output? I.e. did you convert to a different unit?

We did not convert time to a different unit, we used “seconds” as the time unit for this project.

12. How many warm-up executions were done? How many timed executions were done? Did you use the mean or median of the timed executions in your plots? Did you do any other processing of the timing data (e.g. discard “outliers”)?

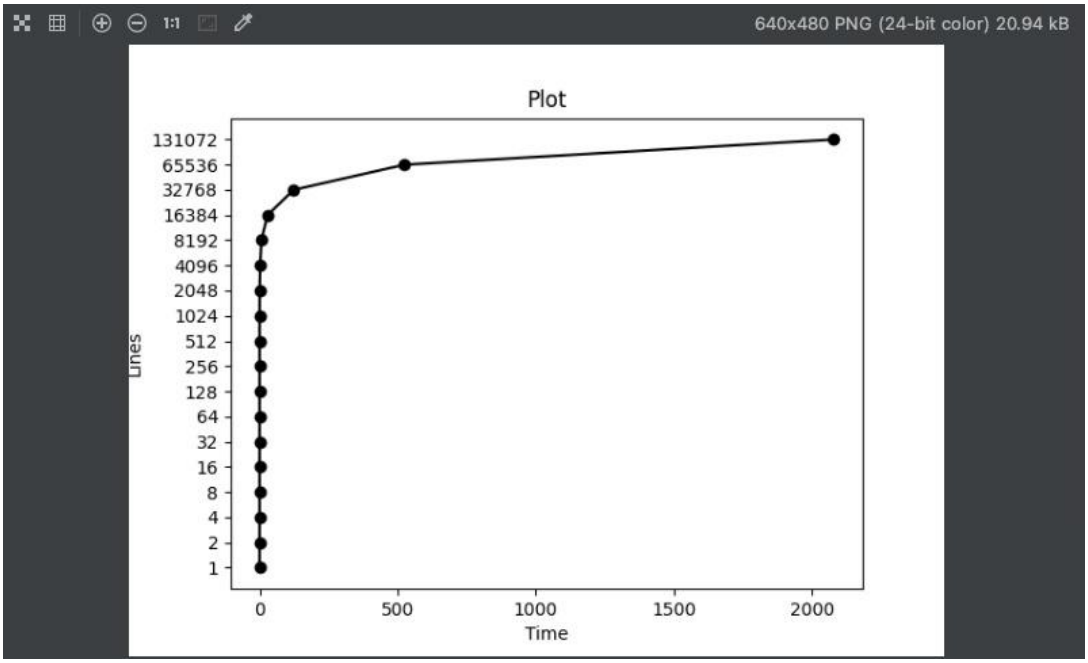
We used 5 warm up executions for all log files with entries less than 32,000 logs. We used 2 warm up executions for files with entries less than 250,000 logs. For files with entries above 250,000 logs, we did not perform any warm-up executions because it was taking too long for the files to run for the larger datasets. After the warm-up executions, we only did one timed execution so we did not take mean or median.

No, we did not do any other processing of the timing data.

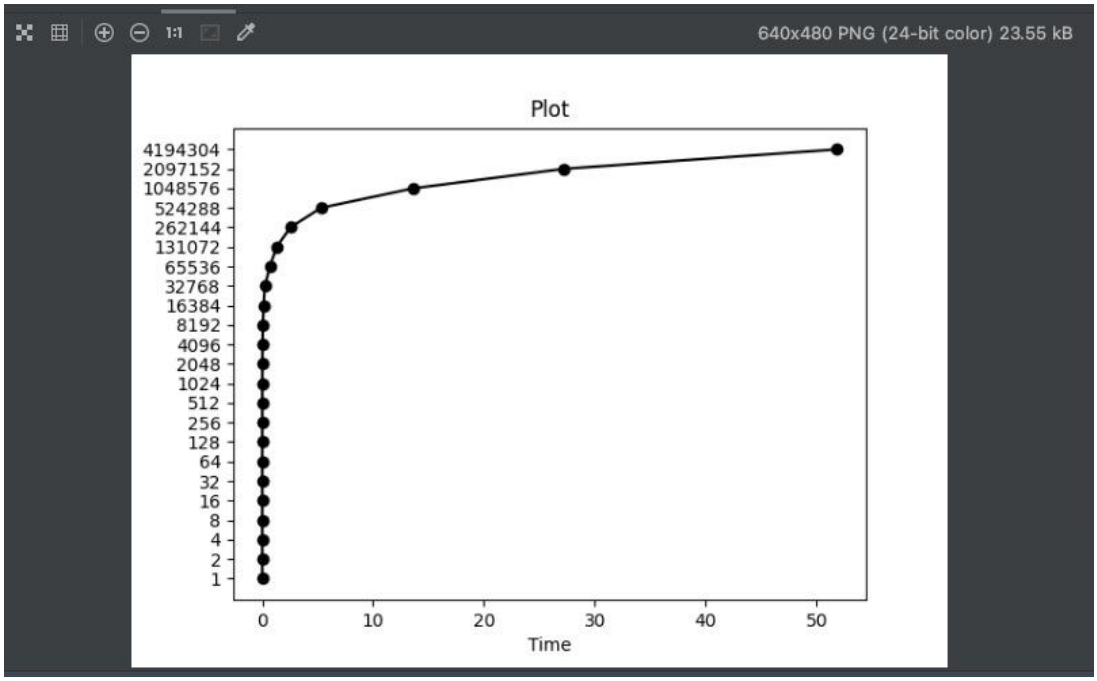
Experimental Results

13. Show plots of time needed (y-axis) versus input size in lines (x-axis) for each sort algorithm. (You'll have 9 plots in total.) Show the curve/line that you fit to the data, ideally on the same plot with the individual data points, but separately if necessary.

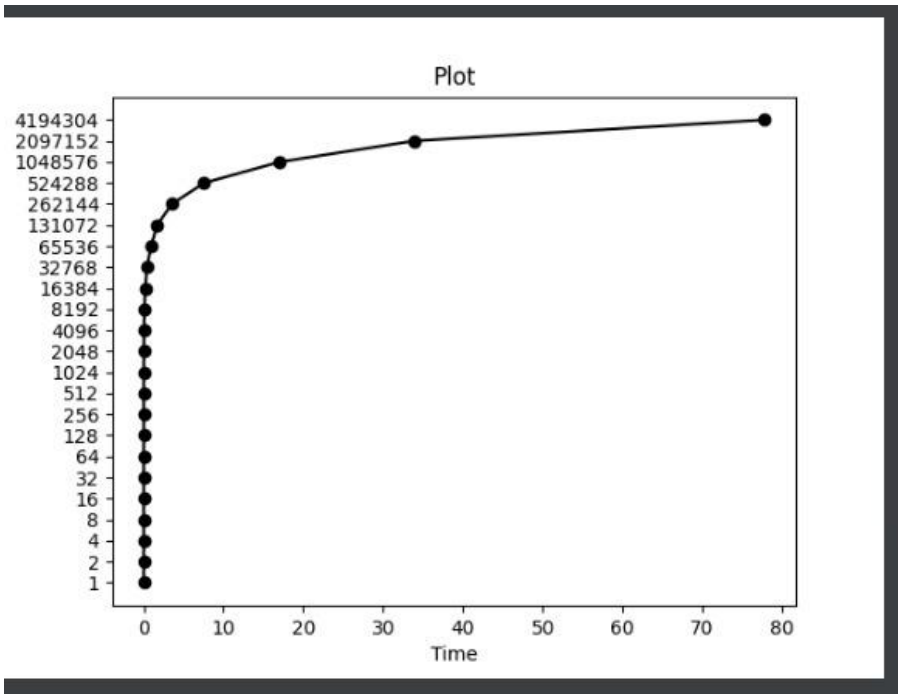
a. Data set A



Insertion sort

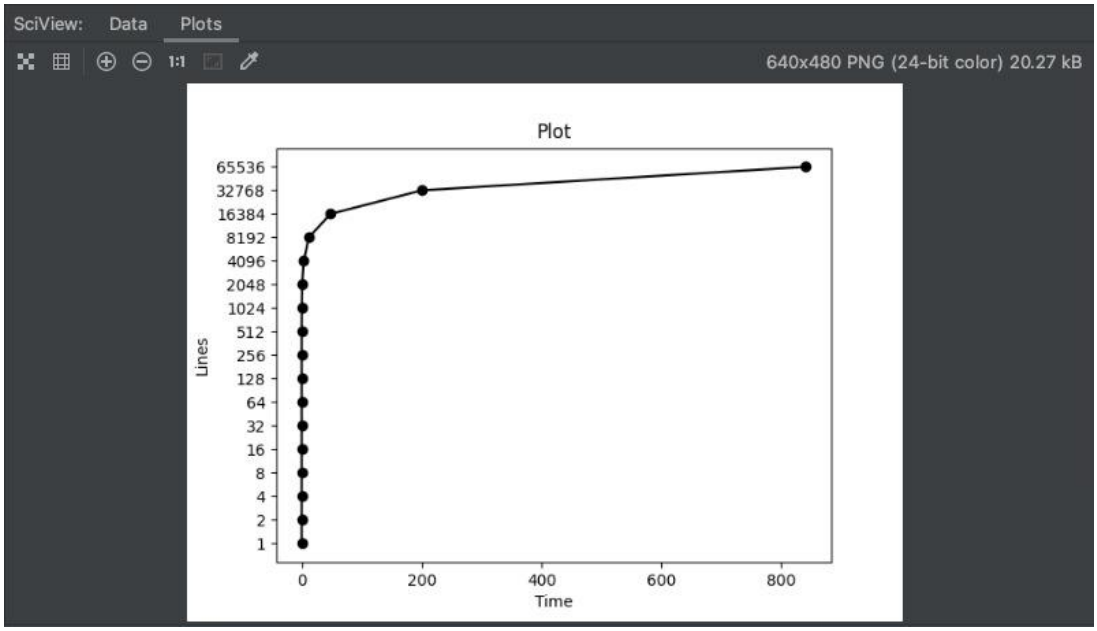


Merge sort

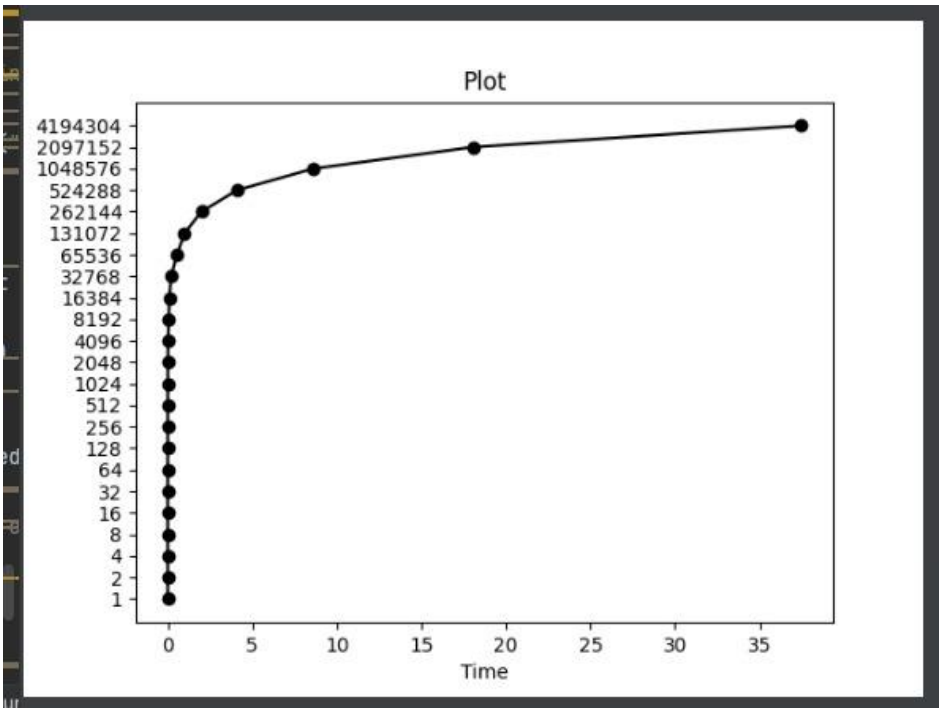


Tim sort

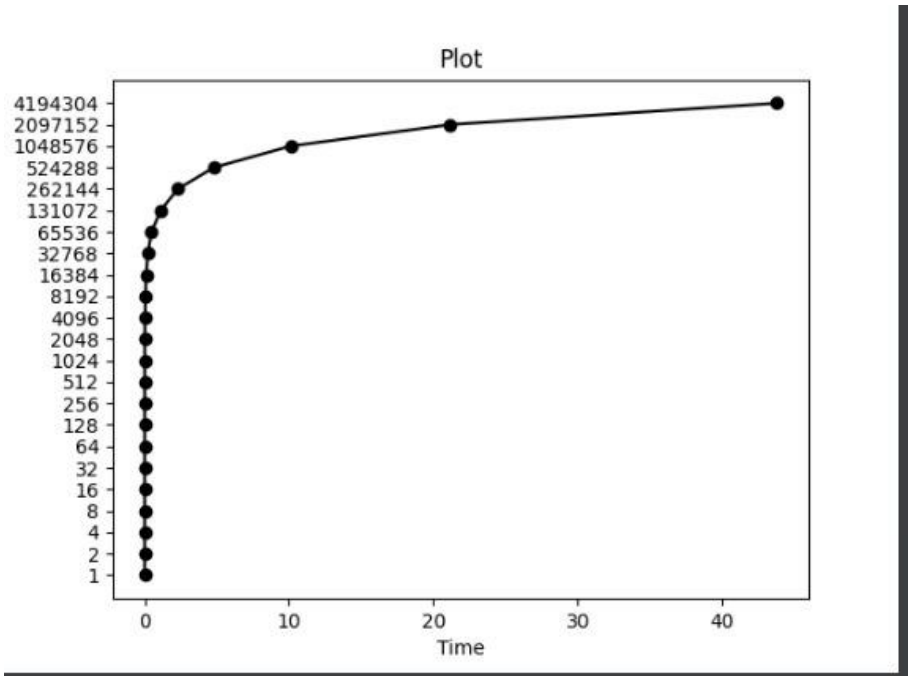
b. Data set B



Insertion sort

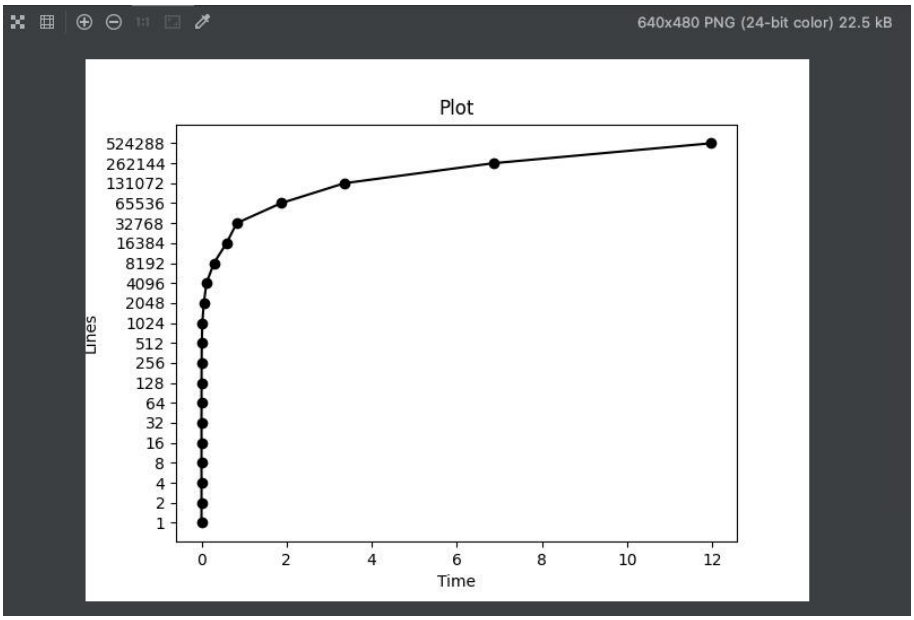


Merge sort

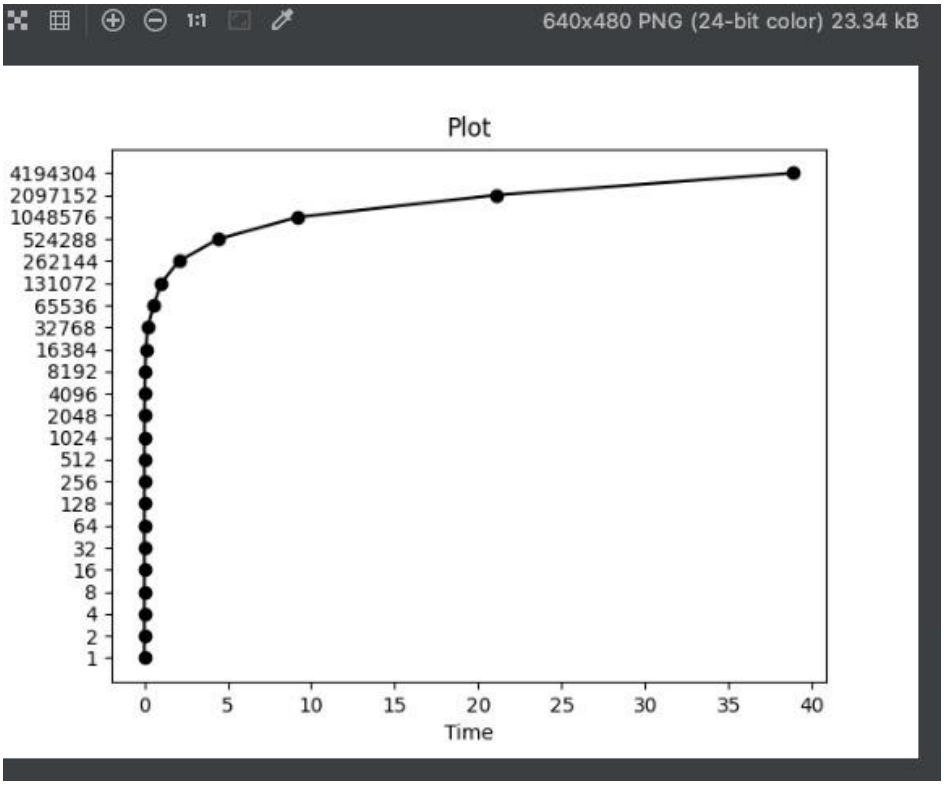


Tim sort

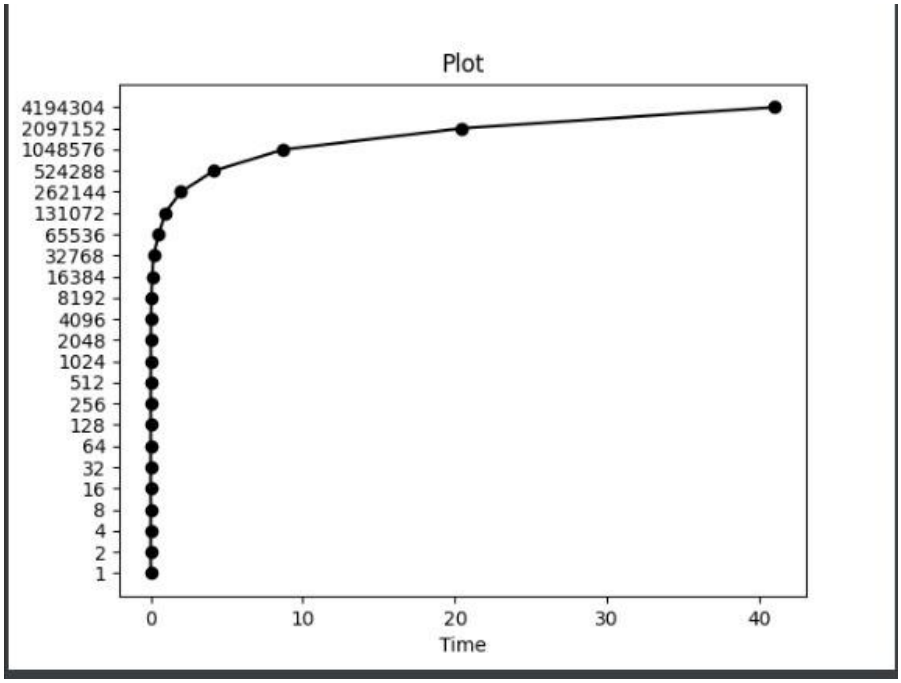
c. Data set C



Insertion sort



Merge sort



Tim sort

14. Were there any limitations in your ability to collect timing data for all 9 scenarios (3 sorts on each of 3 data sets)? If so, then please explain here. If there were any data points you could not collect, please give estimates for the data points you could not measure, and tell us how you produced your estimates.

Insertion sort has a worst case complexity of n^2
Even though insertion sort is efficient in terms of space, it is not efficient in terms of time. As the number of data entries start becoming larger, insertion sort will start performing more inefficiently (tending towards n^2). If the entry files are partially or fully sorted, then insertion sort works well. But there is some noise in all the datasets, so the algorithm is unable to find large subsequences of sorted data and is not able to sort the log files efficiently. Because of this reason, we were unable to run insertion sort on the three biggest data files.

Analysis and reflection questions:

15. Comment on the experimental results. How do they compare to what you expected, in terms of asymptotic complexity and actual time values (i.e. real performance)? Be sure to note any surprises.

a. Insertion sort

Dataset C is almost sorted. From the output graph, we can see it takes slightly more than 12 seconds to sort 524,288 log files. The previous two entries (262,144 and 131,072 log files) take just more than 6 and 3 seconds respectively. So we can see that as the number of data entries doubles, the time elapsed also doubles (approximately). This is because in its best case (where the file is almost sorted), Insertion sort has a complexity of **n**.

Dataset B is sorted in reverse order. From the output graph, we can see it takes more than 800 seconds to sort 65,536 log files. The previous two entries (262,144 and 131,072 log files) take just more than 200 and 50 seconds respectively. So we can see that as the number of data entries doubles, the time elapsed quadruples (approximately). In short, when the number of log files doubles (increase by a factor of 2), time elapsed quadruples (increases by a factor of $2^2 = 4$) This is because in its worst case, Insertion sort has a complexity of $\mathbf{n^2}$.

Dataset A has randomly ordered log files. From the output graph, we can see it takes around 2400 seconds to sort 131,072 log files. The previous two entries (65,536 and 32,768 log files) take around 600 and 150 seconds respectively. So we can see that as the number of data entries doubles, the time elapsed quadruples (approximately). In short, when the number of log files doubles (increase by a factor of 2), time elapsed quadruples (increases by a factor of $2^2 = 4$) This is because in its average case , Insertion sort has a complexity of $\mathbf{n^2}$.

b. Merge sort

Dataset C is almost sorted. From the output graph, we can see it takes around 45 seconds to sort 4,194,304 log files. The previous two entries (2,097,152 and 1,048,576 log files) take just more than 22 and 10 seconds respectively. So we can see that as the number of data entries doubles, the time elapsed is slightly more than double (approximately). In short, when the number of log files doubles (increase by a factor of 2), time elapsed is slightly more than double (increases by a factor of nlog(n)) This is because in its best case (where the file is almost sorted), Merge sort has a complexity of **nlog(n)**.

Dataset B is sorted in reverse order. From the output graph, we can see it takes slightly less than 40 seconds to sort 4,194,304 log files. The previous two entries (2,097,152 and 1,048,576 log files) take around 20 and 9 seconds respectively. So we can see that as the number of data entries doubles, the time elapsed is slightly more than double (approximately). In short, when the number of log files doubles (increase by a factor of 2), time elapsed is slightly more than double (increases by a factor of nlog(n)) This is because in its worst case, Merge sort has a complexity of **nlog(n)**.

Dataset A has randomly ordered log files. From the output graph, we can see it takes around 55 seconds to sort 4,194,304 log files. The previous two entries (2,097,152 and 1,048,576 log files) take around 27 and 13 seconds respectively. So we can see that as the number of data entries doubles, the time elapsed is slightly more than double (approximately). In short, when the number of log files doubles (increase by a factor of 2), time elapsed is slightly more than double (increases by a factor of nlog(n)) This is because in its average case, Merge sort has a complexity of **nlog(n)**.

c. Adaptive sort - Tim sort

Dataset C is almost sorted. From the output graph, we can see it takes around 42 seconds to sort 4,194,304 log files. The previous two entries (2,097,152 and 1,048,576 log files) take just more than 22 and 10 seconds respectively. So we can see that as the number of data entries doubles, the time elapsed also doubles (approximately). This is because in its best case (where the file is almost sorted), Tim sort has a complexity of **n**.

Dataset B is sorted in reverse order. From the output graph, we can see it takes around 45 seconds to sort 4,194,304 log files. The previous two entries (2,097,152 and 1,048,576 log files) take just more than 22 and 10 seconds respectively. So we can see that as the number of data entries doubles, the time elapsed is slightly more than double (approximately). In short, when the number of log files doubles (increase by a factor of 2), time elapsed is slightly more than double (increases by a factor of nlog(n)) This is because in its worst case, Tim sort has a complexity of **nlog(n)**.

Dataset A has randomly ordered log files. From the output graph, we can see it takes around 80 seconds to sort 4,194,304 log files. The previous two entries (2,097,152 and 1,048,576 log files) take just less than 40 and 20 seconds respectively. So we can see that as the number of data entries doubles, the time elapsed is slightly more than double (approximately). In short, when the number of log files doubles (increase by a factor of 2), time elapsed is slightly more than double (increases by a factor of nlog(n)) This is because in its average case, Tim sort has a complexity of **nlog(n)**.

NOTE: For insertion sort, the time complexity goes up to n^2. Because of this, we were unable to run the insertion sort algorithm on the largest few files because they were taking too long. Based on the graphs we plotted and the trends between time elapsed and number of data entries in the log files, it would have taken the algorithm more than 16 hours to run the file with 4 million log entries.

16. What was the most challenging part of this project, and why?

Dealing with the file encodings for reading the log files, skipping the lines with no timestamp and encoding into latin-1

Some of the most challenging parts of this project were :

- Dealing with the file encodings to read the log files
- Skipping the lines with no time stamps
- Encoding into latin-1
- Running the sort algorithms on the larger data files (we did not realize they would take so much time to run)
- Making a general script and code so that it can run on all environments

17. How did your team divide up the work? For each team member, what were their main contributions?

Bhuwan and Aditya did most of the coding part of the project, Shubham was responsible for building the project documentation. Even though we met virtually, all three of us are happy with one another as a team and the work was done efficiently.

Bonus questions:

18. Did you collect any other data (timing of other operations aside from sorting, or memory usage, for example)? If so, please describe what you measured, and what you observed. Is it what you expected? Were there surprises? You should consider presenting this extra data here (graphs are great).

N/A

19. After an analysis like the one you did in this project, an engineer might proceed directly to choosing which sort algorithm to use in their product. But more likely, the choice of sort algorithm would be accompanied by another analysis, in which the engineer looks for opportunities to optimize, i.e. to further increase performance beyond simply choosing the best sort algorithm. Here, you may speculate on steps you might take to get a better performing sort than any of the ones you implemented. You may talk about other sorts, or ways of pre-processing the data, or any other technique that you suspect might improve overall performance of the task. (Consider "the task" to be sorting the kinds of files found in all 3 data sets.)

For **dataset C** (that is almost already sorted), we should ideally use insertion sort because insertion sort has a best case time complexity of order **n**. However, this will only be possible if the dataset is sorted already. To remove the noise (data points that deviate from the trend), we can simply remove them from the file. Removing these data points can be done in linear time, and then we will have a perfectly sorted dataset (dataset B ideally should be ordered according to the lecture in class). Even if we do want to implement a sort algorithm on this dataset, we will use insertion sort that only takes linear time. Insertion sort is also really good for datasets with lesser number of data points.

Dataset A has randomly ordered log file entries. One of the best sorting algorithms for average run time complexity is the Quick sort algorithm. Quick sort algorithm is a comparison based algorithm that does not require a lot of memory or overhead, so it is very popular. Quicksort also performs well on large datasets so that is a very big benefit of using it. One major drawback is that if the pivot isn't chosen smartly, it will have a time complexity of order n^2. This also happens when the dataset is already sorted or reverse-sorted.

Dataset B is sorted in a reverse order. So we need to find an algorithm with the best worst-case complexity. Of these three algorithms, Merge sort and Tim sort have a worst case complexity of **nlog(n)**. In that case, Tim sort is better than Merge sort due to the fact that it is a hybrid sorting algorithm. For smaller datasets, tim sort is based on insertion sort (that is faster than merge sort for small datasets). And when the dataset is very large, tim sort is based on merge sort. Tim sort attempts to figure out whether to base its sorting algorithm on Insertion sort or Merge sort. This does add overhead, but it is generally easier to manage this overhead than time for real world data. Another sorting algorithm that can be used is the Heap sort algorithm. Merge sort is generally a little faster than Heap sort for very large datasets, but it also takes twice the amount of memory.