Assignment 1 Question 6 Written Report
Question 1

Running our Code:

To run our code, please use the sort files in the following format: *./sort1.py a1.small* or *./sort1.py a1.large*

Please ensure to use *chmod +x sort1.py* on python files to ensure they run.
As a last resort, please compile or code using python3

The python files correspond to our sorting algorithms as follows:
- Insertion Sort: sort1.py
- Heap Sort: sort2.py
- Bubble Sort: sort3.py
- Bubble Sort Optimized: sort4.py

Thank you for understanding :)

(a) Experimentally identify the run time complexity and identify any constraints that determine the performance of your algorithm. You will need to run your experiments for various input sizes to obtain good approximations for the growth functions that characterize your algorithms' runtimes, but also to obtain a good estimate for the constants.

For run time we ignored the time taken to load the csv file, since a csv with a larger data set will obviously take longer to run, and this is not relevant to the algorithm run time itself. We ran tests on the following 3 algorithms: heapsort, insertion sort and bubble sort. For each algorithm we used a sample size of 10 spanning from 1000 data points to 28,000 data points. The time.time() function was used to measure the start and stop time of the code. Below are the four tables that show our relative run time with a given Data size.

Data for Heapsort

Data Size	Time (miliseconds)
100	0.678
1000	11
3000	37.5
6000	82.5
9000	132
12000	176
15000	249
18,000	311

21,000	341
24,000	413
28,000	491.9

Data for Insertion Sort

Data Size	Time (miliseconds)
100	0.862
1000	71.423
3000	606.287
6000	2564.516
9000	5662.060
12000	10242.235
15000	16718.236
18,000	25235.660
21,000	36082.630
24,000	44971.343
28,000	62185.019

Data for Bubble Sort

Data Size	Time (miliseconds)
100	1.756
1000	150.229
3000	1353.074
6000	5801.861
9000	12922.871
12000	23928.294
15000	34245.157

18,000	48842.257
21,000	77893.228
24,000	98147.401
28,000	157328.505

Data for Optimized Bubble Sort

Data Size	Time (miliseconds)
100	1.730
1000	151.9
3000	1425.908
6000	6186.506
9000	14460.108
12000	25722.939
15000	40094.014
18,000	57005.939
21,000	78742.259
24,000	108729.574
28,000	139419.205

(b) Provide one graph that demonstrates how you determined the constants for each of your implementations.

Desmos Graph for Insertion Sort (https://www.desmos.com/calculator/uuqfxcd7oy) is shown in Figure 1. Its run time equation was estimated as the following in the form of f(n)=cn^2 +bn+a where c = 0.000078, b=0.014, a=0. This was done through best fitting the data points as shown below. Numerous values were tried for constants abc, and the closest were chosen.

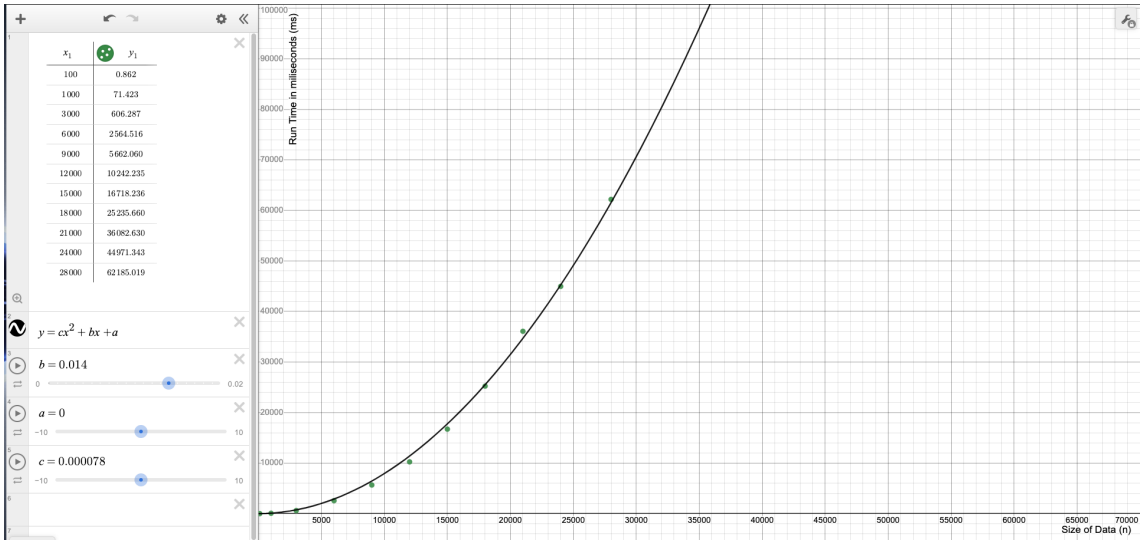

Figure 1. Insertion Sort Graph

Desmos Graph for Bubble Sort (https://www.desmos.com/calculator/4ay8weloy6) is seen in Figure 2. Its equation was f(n)=0.00015n^2 +0.2n+0*a. It was determined in the same fashion as Insertion sort.

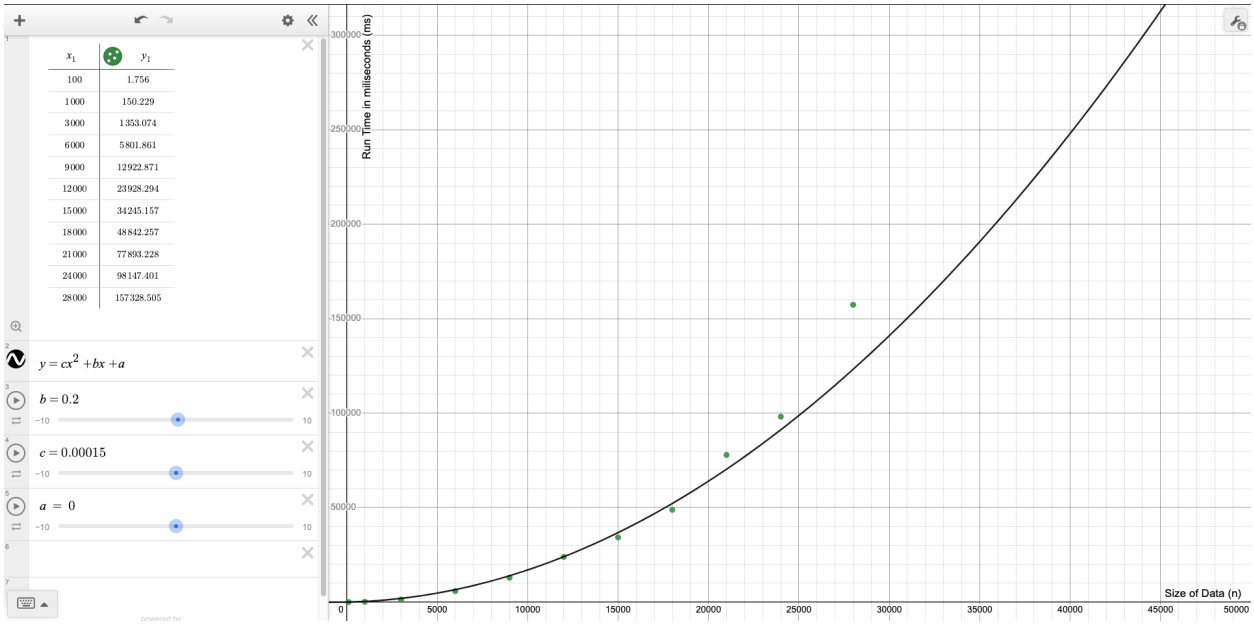

Figure 2. Graph for Bubble Sort

Desmos graph for Heap Sort (https://www.desmos.com/calculator/vwyrg573uc) is shown in Figure 3.

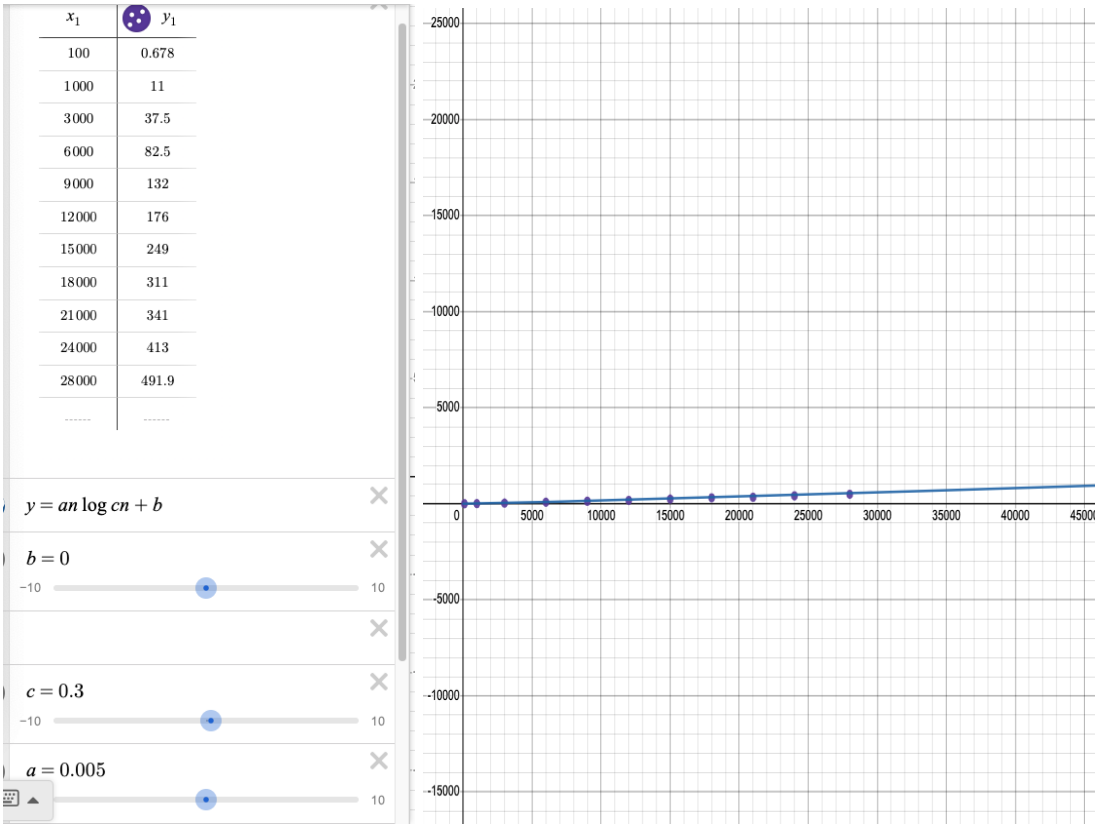

Figure 3. Heapsort Desmos Graph

Equation for heapsort is 0.005nlog(0.3n). Derived this equation by playing with sliders until the equation fit.

(c) Provide a table that lists row-wise the algorithm implemented, its growth function, and its constant.

Algorithms	**Runtime Complexity**	**Calculated Function**
Insertion Sort (Joey)	O(n^2)	f(n) = 0.000078n^2+0.014n +0*a
Heap Sort [Chris]	O(nlogn)	0.005nlog(0.3n)
Bubble Sort (Joey)	O(n^2)	f(n)=0.00015n^2 +0.2n+0*a
Optimized Bubble Sort	O(n^2)	f(n)=0.00017n^2 +0.2n+0*a

(d)Pick any one of your algorithms and realize an optimization of your choice; re-run experiments and report the outcome in your graph and table as well.

Desmos graph for Bubble Sort Optimized (https://www.desmos.com/calculator/lj1mffcf1h) is shown in Figure 4.

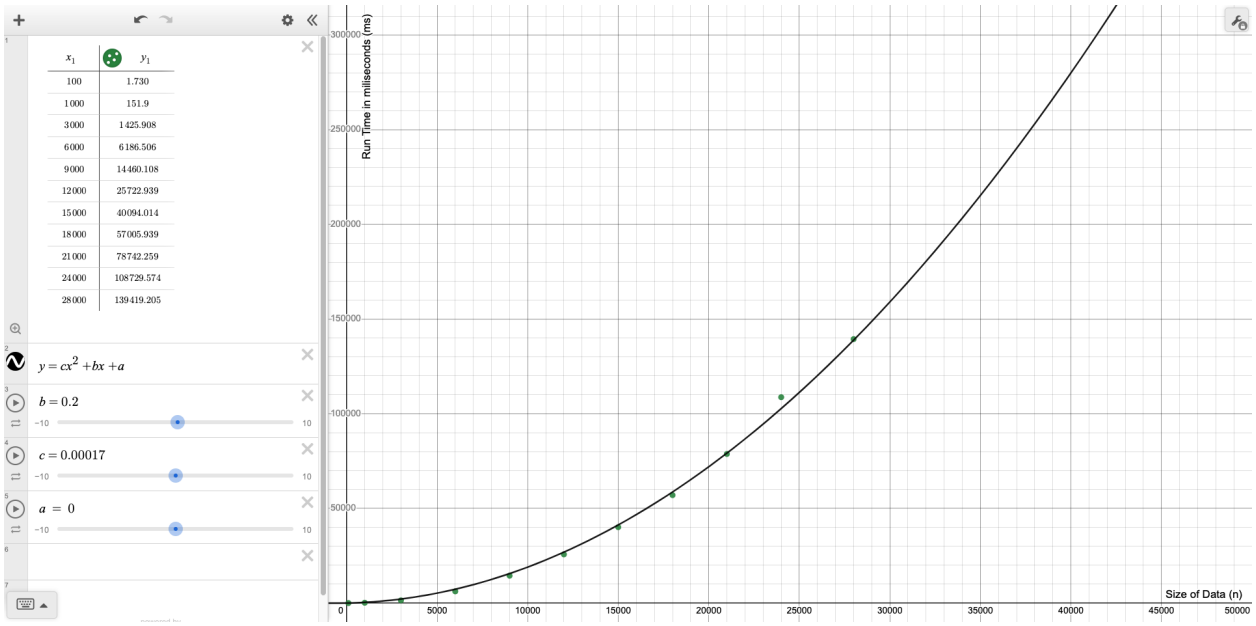

Figure 4. Desmos Graph for Optimized Bubble sort

The growth function of the Optimized Bubble sort is shown in above table in comparison with the other sorts.

(e) What do you conclude about the performance of the four implementations (three different algorithms, one with additional optimization)?

In general, the heapsort algorithm was the fastest, insertion sort was second fastest and bubble sort was the slowest. Heapsort is the fastest because it is a divide and conquer algorithm, it uses the recursive heapify function. For heapsort you only need to go through half the array since you are swapping O(n) and for heapify it is O(logn), which is smaller than O(n). When multiplying O(n) with O(logn) you get a much smaller function than O(n^2) for insertion and bubble sort.

Bubble sort was the least efficient algorithm as shown in our graphs and tables. This is most likely due to it having to encounter so many more cache misses and perform so many writes to memory. Although both Insertion sort and Bubble sort have the running complexity of O(n^2) in general, Bubble Sort has much higher probabilities in encountering cache misses and performing frequent swaps than compared to Insertion Sort.

(f) Diligently list all sources that you used and discuss the optimization you applied.
Here is a list of sources that were used for the assignment:
[1] Video used for heapsort
https://www.youtube.com/watch?v=2DmK_H7IdTo&ab_channel=MichaelSambol
[2] Heapsort Week 2 lecture slides

[3] Insertion sort: https://www.geeksforgeeks.org/insertion-sort/
[4] Bubble Sort and Optimization: https://www.geeksforgeeks.org/python-program-for-bubble-sort/
[5] Optimization Bubble sort: https://www.programiz.com/dsa/bubble-sort
[6] Sorting algorithms visualizer: https://visualgo.net/en/sorting?slide=1
[7] Python csv file opener: https://realpython.com/python-csv/
[8] Sort comparison: https://www.baeldung.com/cs/insertion-vs-bubble-sort

- The flag optimization used in bubble sort can allow it to exit the double for loop earlier when detecting that a given array is already sorted at a given outer loop iteration. This will reduce its complexity from O(n^2) to O(n) given that an input array is already sorted.
- However, this result could not be accurately reflected in our case as it is very unlikely that any of the csv files are sorted.
- Thus, Instead of making our algorithm faster, it actually made it slower as reflected in the graph above and the table for its time. This is due to us adding an additional checking condition each loop that would result in additional constant time operations which in this case slowed our algorithm down instead of speeding it up.

Notes for submission:

- .tar.gz just means we compress our code in to a zipped folder
- Source code must include a full build environment can be done by including this in 1st line of python file `#!/usr/bin/env python3`
- A make file that generates executables when “make” is invoked
    - We only need this for C++ but we are using python so we don’t need it
- Use git pull -r in terminal to pull his new code
- Git checkout -f
- Makes more sense to move every row with the sort key in one motion instead of doing it twice.
- Use ictrl and applications → mate terminal

Look at github and push code