COMPSCI 2XB3 L04 Lab Report #2

Mahad Aziz - azizm17 - 400250379

Talha Amjad - amjadt1 - 400203592

Logan Brown - brownl33 - 400263889

McMaster University

COMPSCI/SFWRENG 2XB3: Binding Theory To Practice

Dr. Vincent Maccio

TA: Amir Afzali

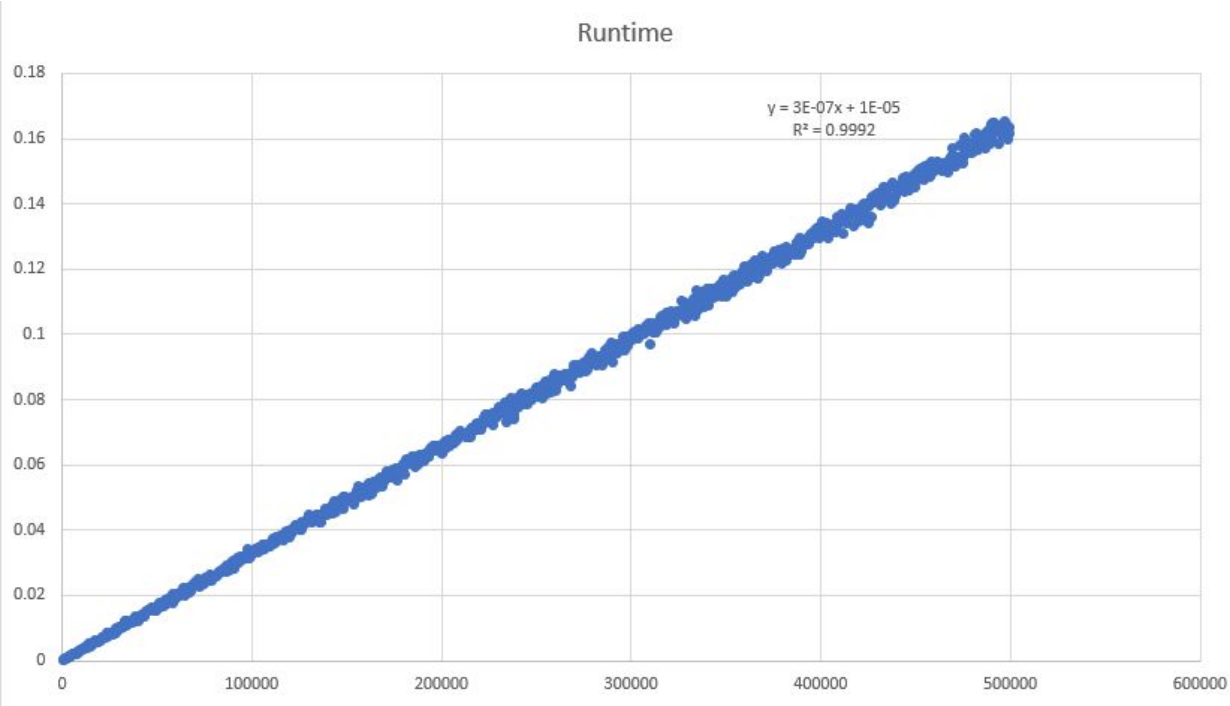
January 29, 2021

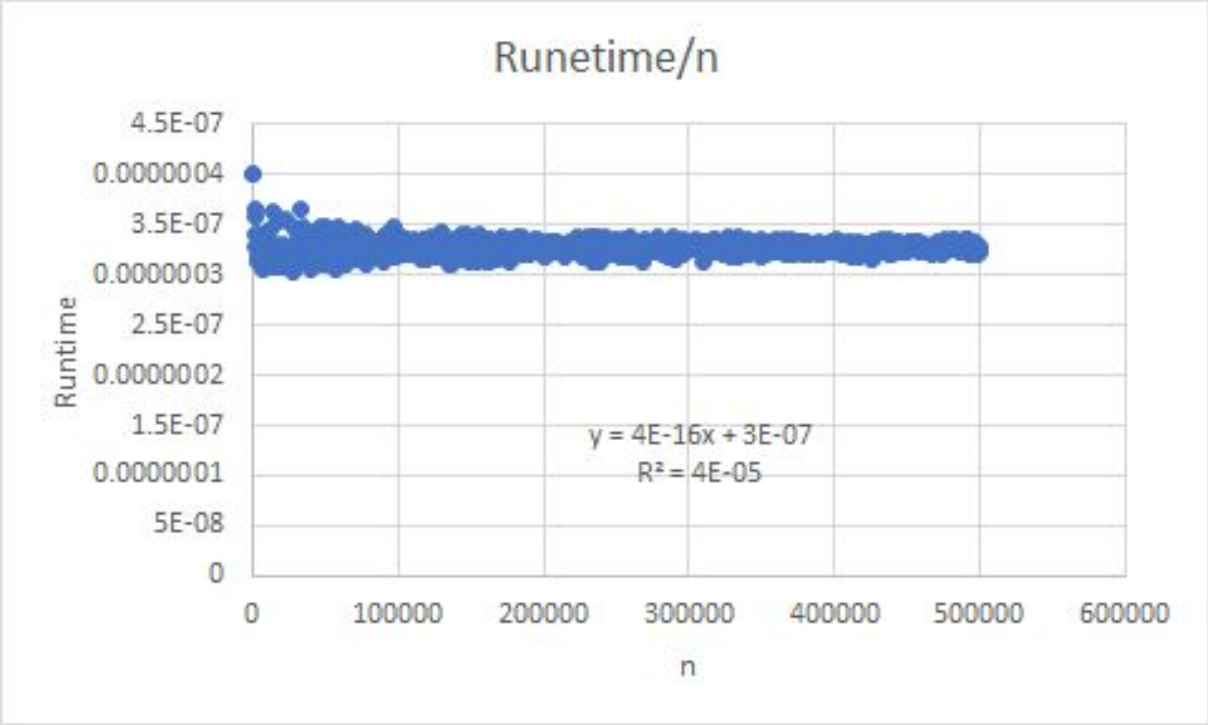
Timing Data:

f(n):

Predictions:

We predicted that the graph would have a linear trendline from observing the shape of the graph.





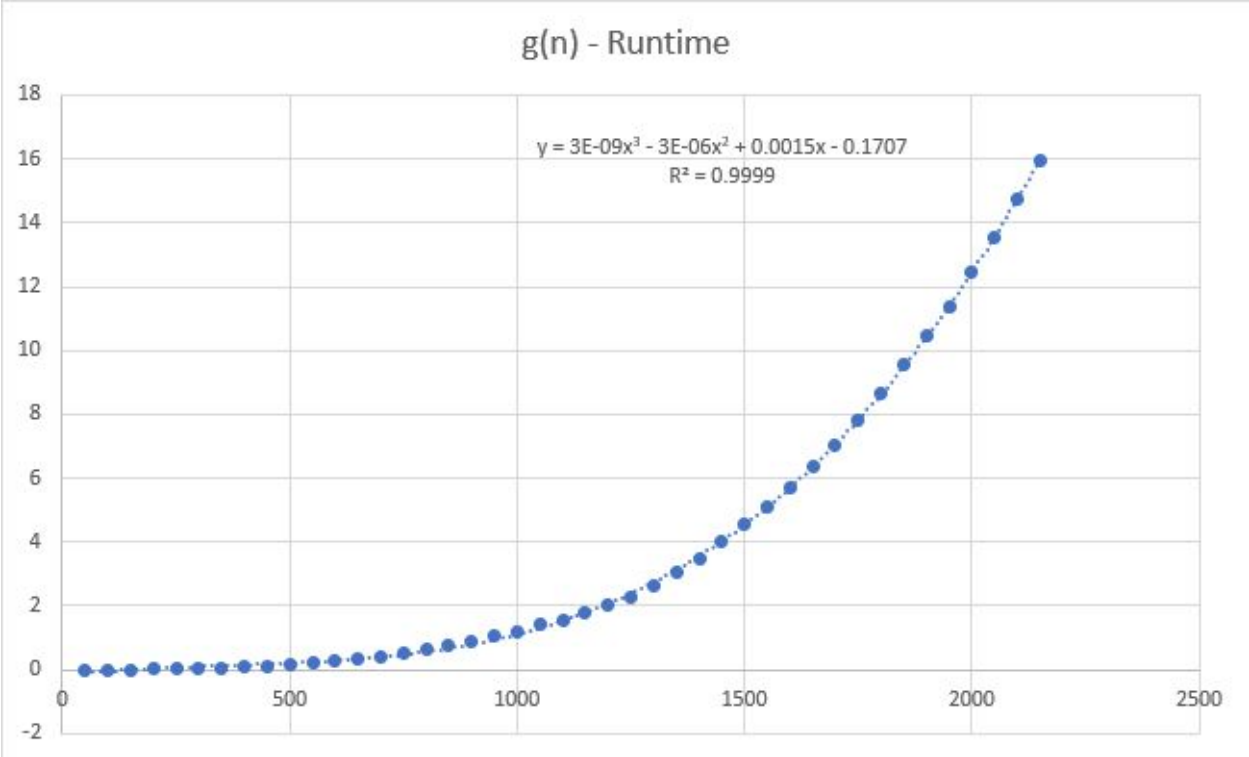
Observations and Conclusions:

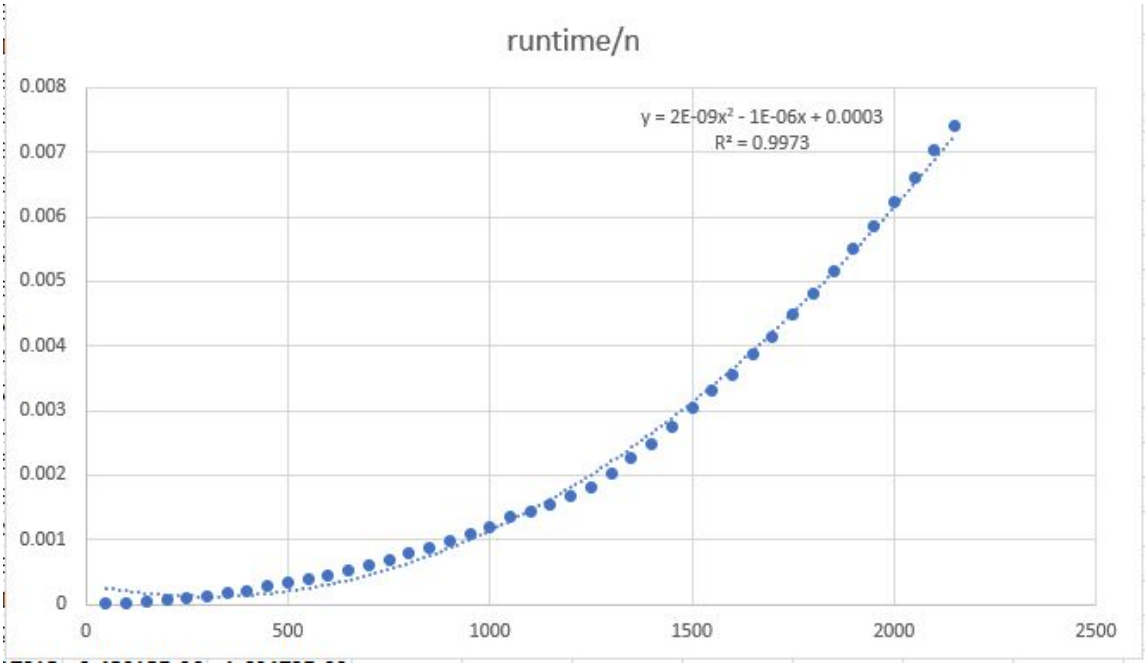
The function f(n) is growing at a rate of O(n) as the number of runs for the function grows. From the graph, the trendline is a linear function. The trendline has an R2 value of 0.992 which further proves that the data points are located very close to the trendline of the data. We had initially predicted that the graph would generate a linear trendline and that is the result that we got. We further analyzed the graph by dividing the runtime values by n. Using this analysis we got a constant graph in the end which further proves that the original function grows linearly.

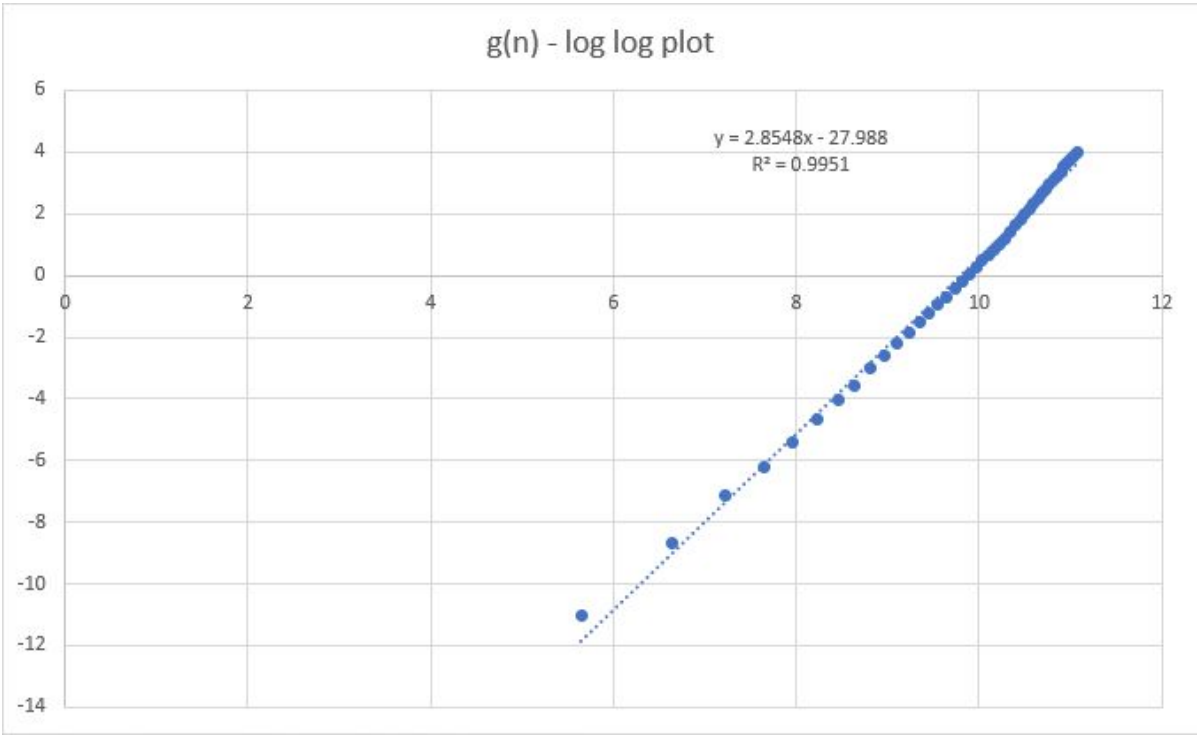
g(n):

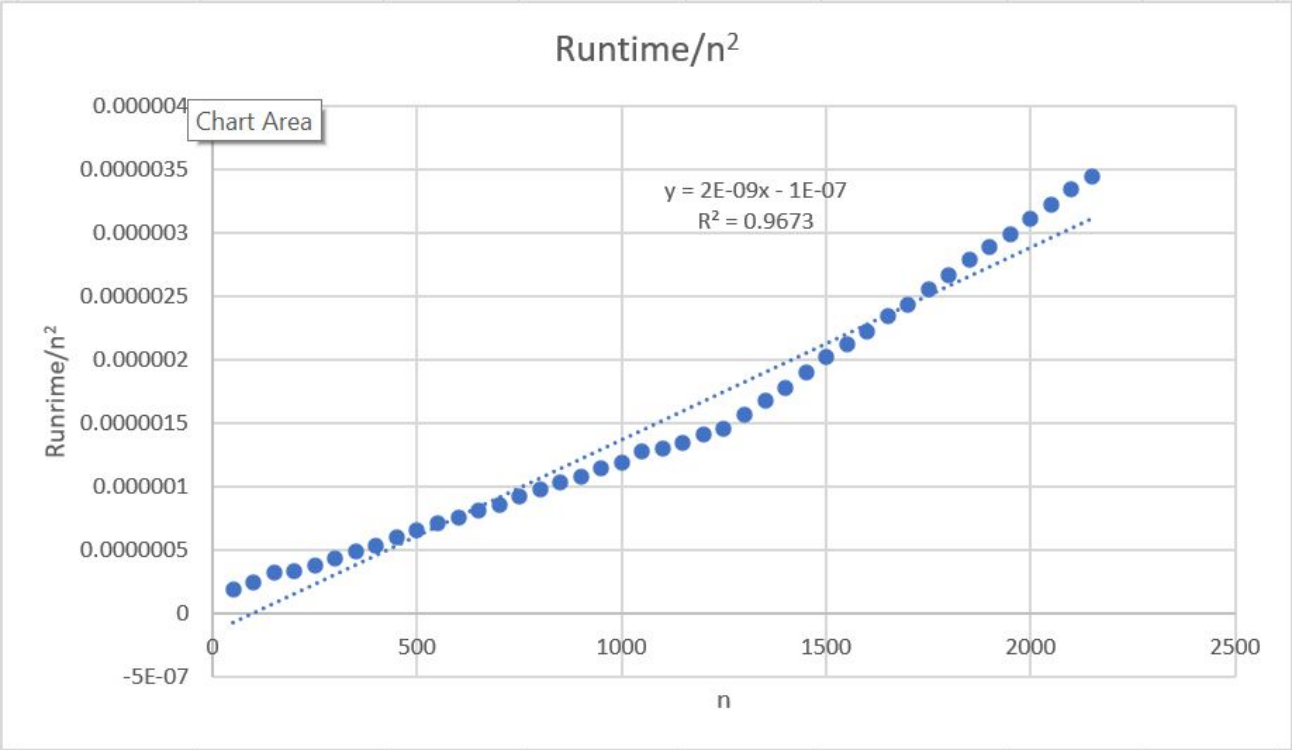
Prediction:

We believed that the g(n) function was going to be quadratic after looking at the shape of the graph.




runtime/n
y = 2E-09x² - 1E-06x + 0.0003
R² = 0.9973
0.008
0.007
0.006
0.005
0.004
0.003
0.002
0.001
0
0
500
1000
1500
2000
2500


g(n) - log log plot
y = 2.8548x - 27.988
R² = 0.9951
6
4
2
0
-2
-4
-6
-8
-10
-12
-14
0
2
4
6
8
10
12



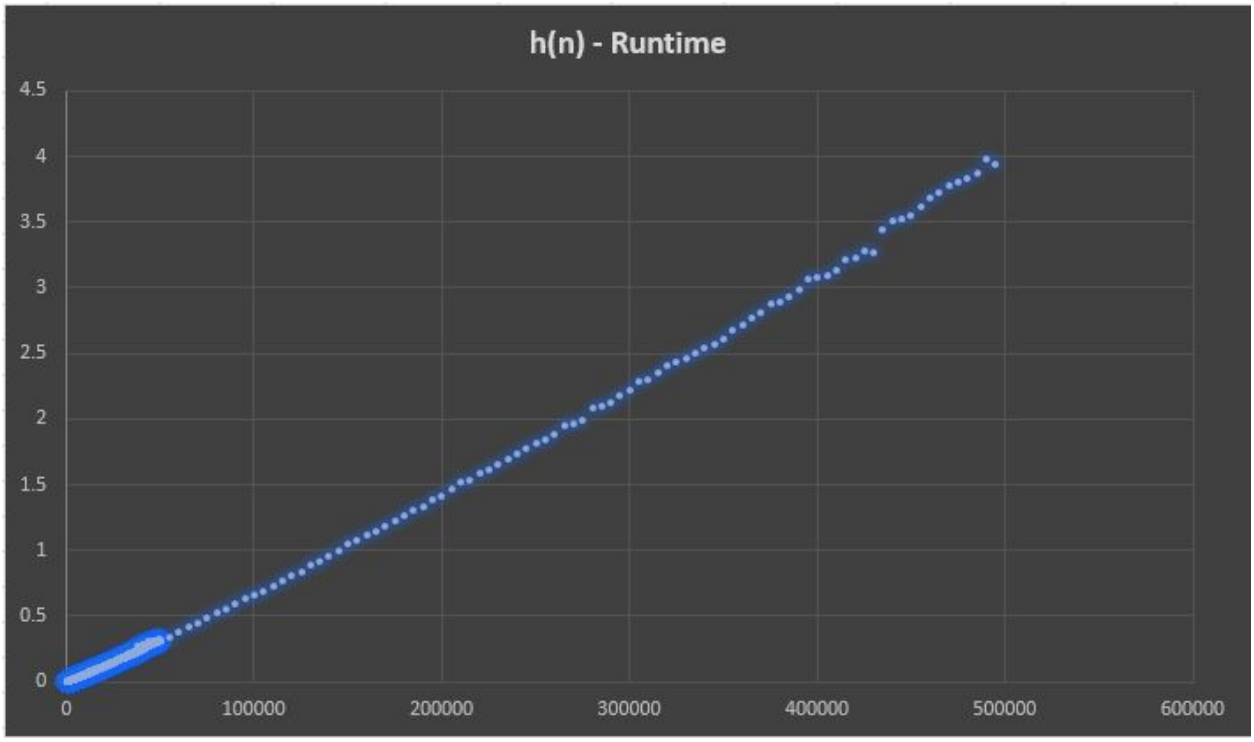
Observations and Conclusions:

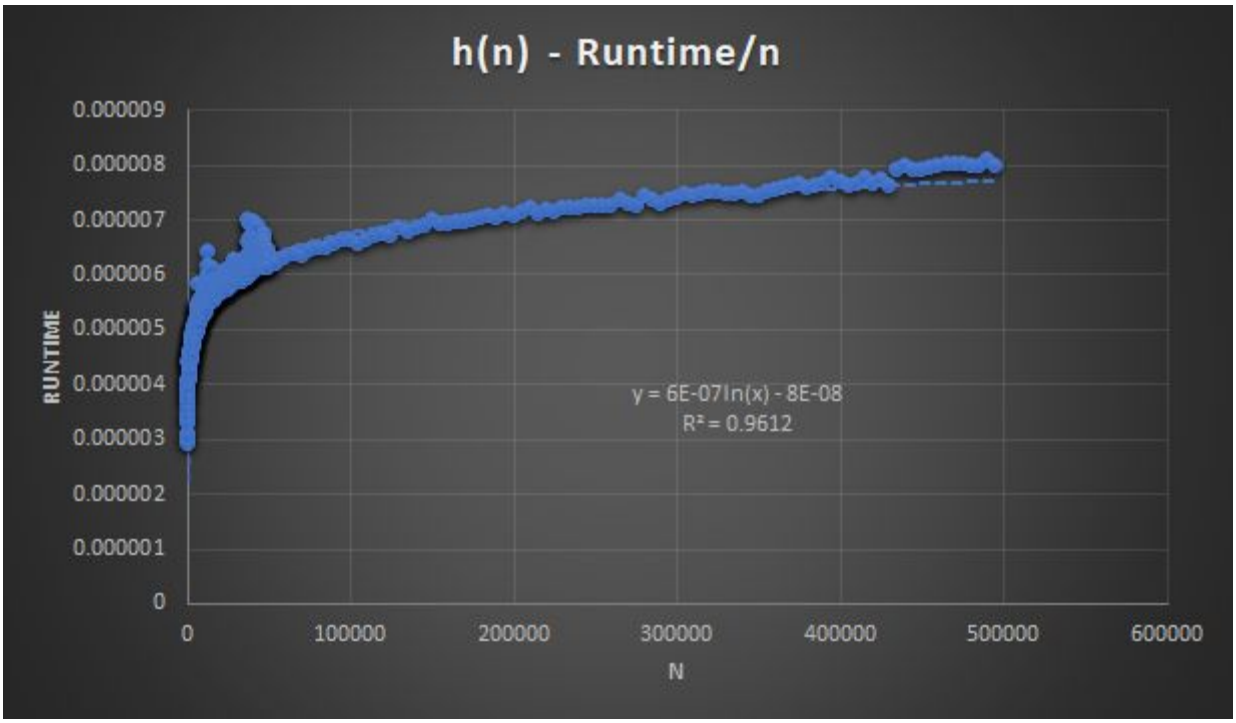
The function g(n) is growing at a rate of $O(n^3)$ as the number of runs for the function grows. We had initially predicted that the trendline would be quadratic by solely looking at the graph however when we added a trendline, the quadratic function gave us a trendline that did not closely represent the data. We tried another polynomial trendline but this time with an order of 3 and it gave us a much more accurate trendline. The trendline has an R2 value of 0.999 which further proves that this data is almost exactly along the cubic trendline. To determine the exact order of growth we used some further tests. We divided the runtime by n twice and looking at the shape of the graphs it gave us a quadratic function rather than a linear function. If the original function had been quadratic we would have seen a linear function when dividing by n the first time which is not the case here. The final and definitive test we used was to make a log-log plot. Looking at the log-log plot, we see that the slope is 2.8548 which is quite close to 3 which lets us conclude that the growth is indeed $O(n^3)$.

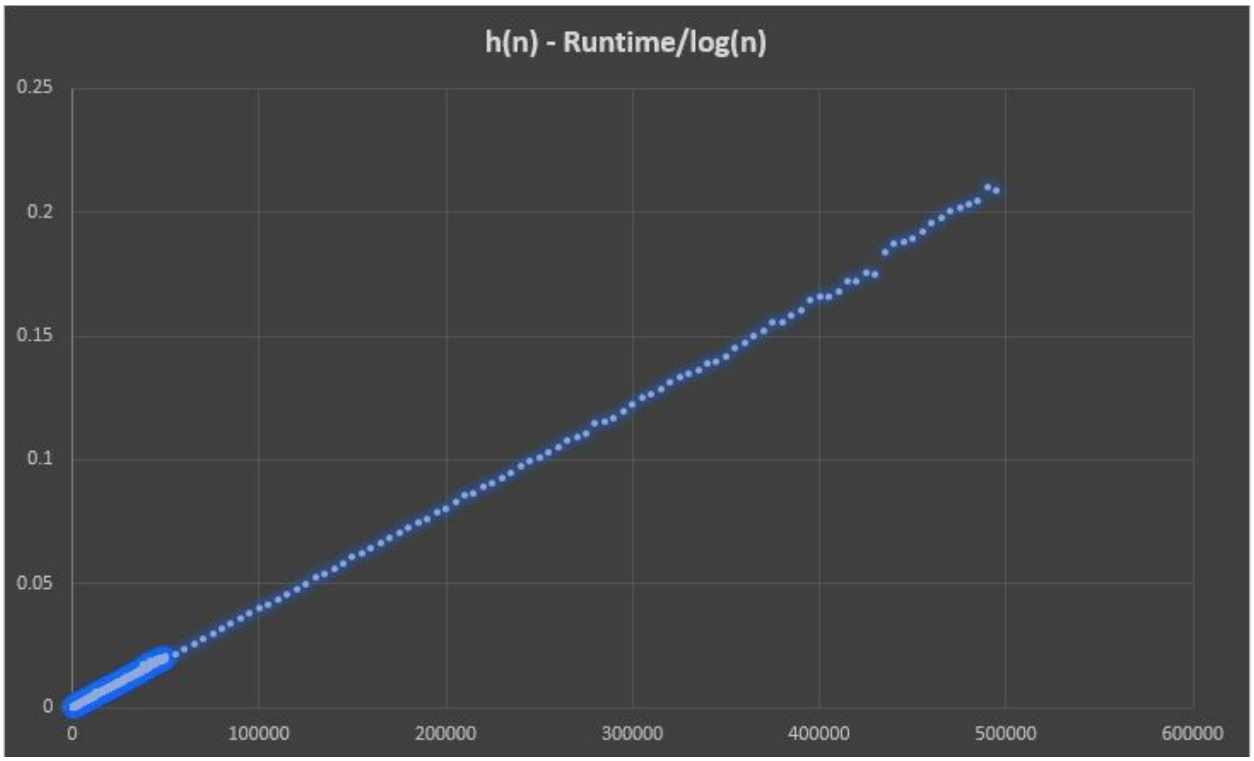
h(n):

Predictions:

We believed that the trendline would be linear because of how similar it looked to f(n) and also when we observed the data points, we thought the values were increasing at a constant rate.







Observations and Conclusions

The function h(n) is growing at a rate of O(nlog(n)) as the number of runs for the function increases. From the graph, our initial thoughts were that the trendline would be linear, but after further testing, we found that the growth is O(nlog(n)). We used two tests to determine this. The first test we used was to divide the runtime by n. We found that the graph shows a logarithmic shape with an R2 value of 0.9612. The second test we used was to divide runtime by log(n) and the graph shows a linear shape with an R2 value of 0.9995. This is a strong indication that the original graph grows as O(nlog(n)).

Python Lists:

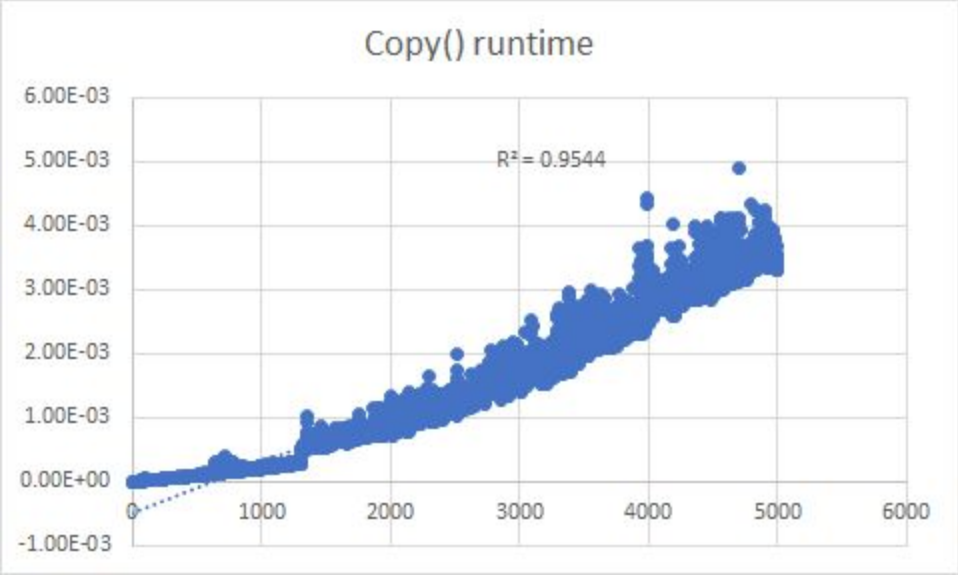
Copy

Description:

We chose to test the copy function using progressively increasing lists up to a length of 5000. The runtime value we received every time one run was completed was an average of 500 runtimes. Furthermore, we used the random library in python to generate random lists of increasing length to ensure more accurate results.

Predictions:

We predicted that the copy function in python would return an O(n) result because the list would increase by 1 index everytime the copy function is called.



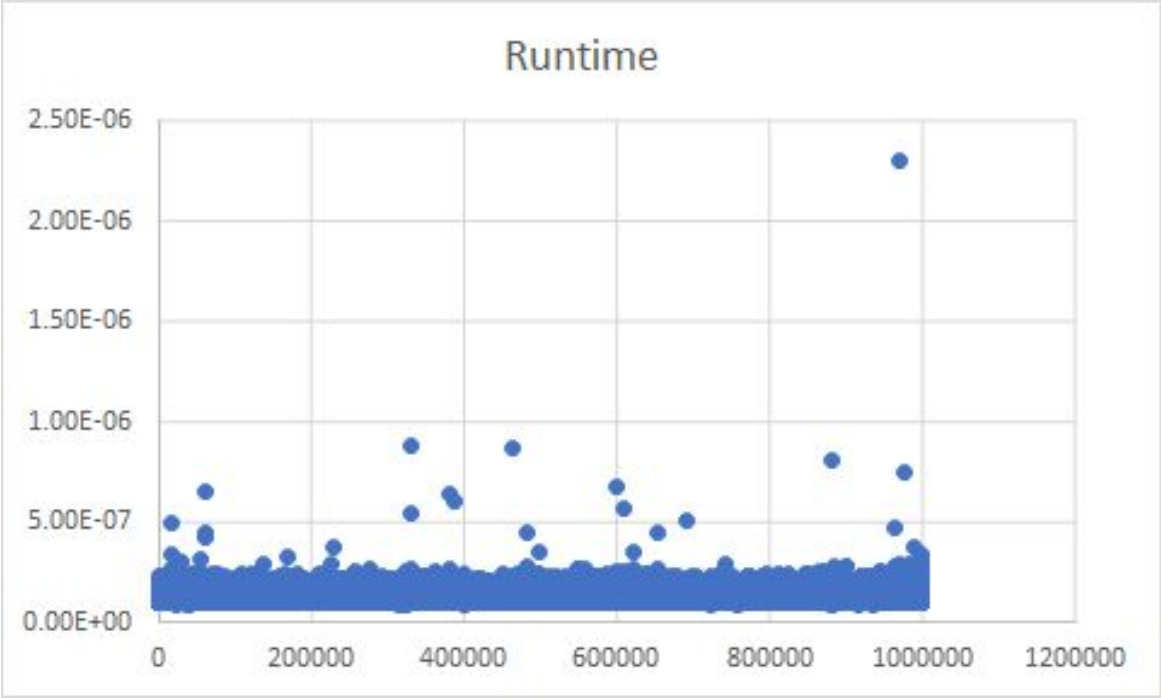
Observations and conclusions:

We observed that the longer the list was at the end, the longer it took to get results. However, the data points collected at the end were getting more accurate. As a result, we decided to stop at a list sized 5000 as it was taking too long to run the tests for longer lists and we believed that these data points to be sufficient enough to prove that the copy function is O(n). Furthermore, we looked at the claim made by the blog, and agree that the time complexity of copy should be O(n).

Lookups

Predictions:

It would make sense if the complexity is the constant time since a lookup of a list only requires finding a single address in memory. There may be fluctuations in the time it takes to find an element since we don't believe python uses sequential addresses to store lists.



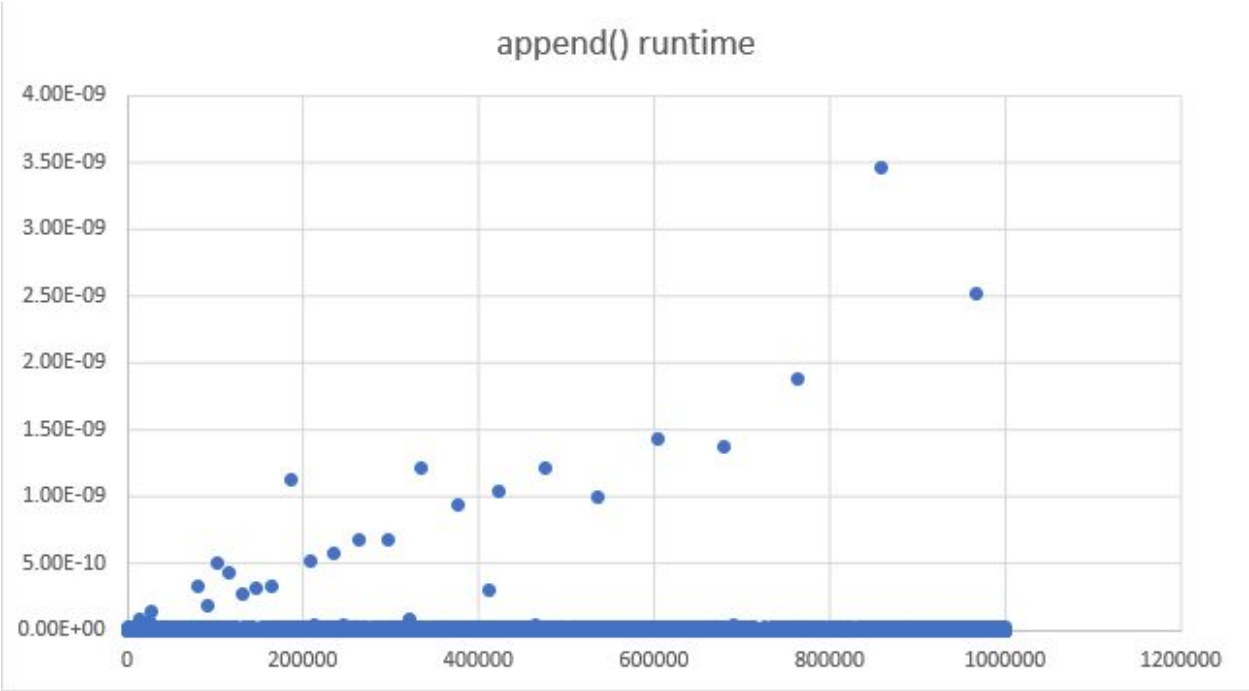
Observations and Conclusions:

After running our code for the runtime of lookups and graphing it we noticed that the results came out quite close to what we predicted them to be. Looking at the graph we were able to conclude that the lookups function did end up taking constant run time excluding a few outliers. After running our code 1000000 times, we were able to conclude that the runtime for the lookups function for lists has an O(1) runtime. Furthermore, when we checked the blog to see if our results matched up with the actual runtime of the lookups method, we found that they had also said the runtime was O(1) which allowed us to conclude that the runtime for lookups is constant regardless of the number of times it is run.

Append

Predictions:

We believed that the time complexity of running the Append function was going to be O(1) which is constant. We believed this because you would be appending a value to the list one at a time and the amount of memory required to append that new value has already been allocated.



Observations and Conclusions:

We observed that the testing for the append function was the fastest compared to lookups and copy. Furthermore, we believe that the anomalies you see in the graph above are data points for when the memory allocation changes because of the need for more memory as the list size has increased significantly larger. Looking at the explanation from the blog post and their runtime of O(1) for append, we concluded that our prediction was correct.