2XC3 Lab – 2

Contact Member:

Name: Swaleha Jasmine

Student Number: 400348553

MacID: jasmines

Email: jasmines@mcmaster.ca

Lab Section: 2

Member:

Name: Neha Asthana

Student Number: 400329655

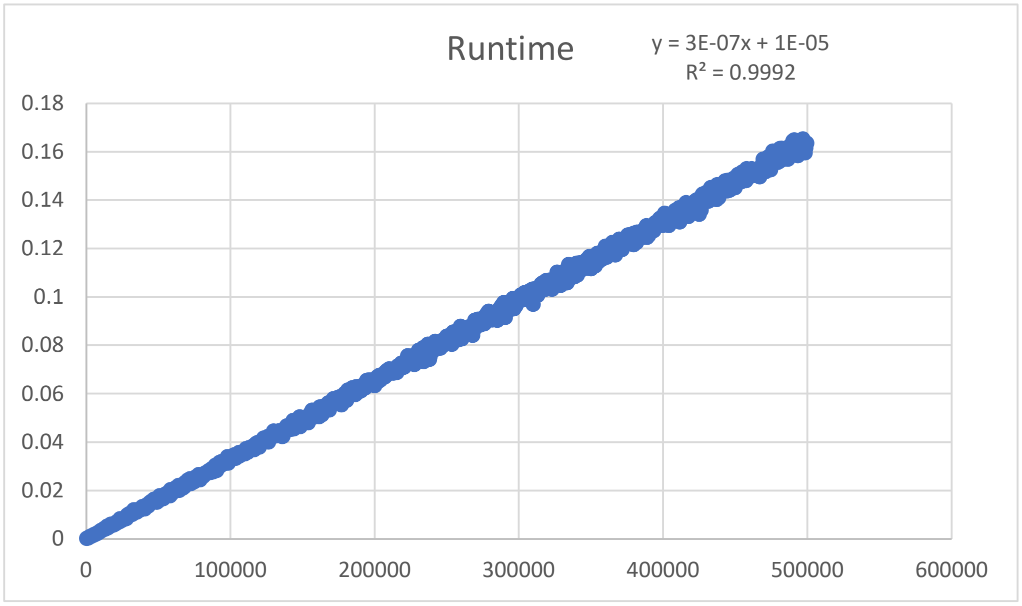
MacID: asthann

Email: asthann@mcmaster.ca

Lab Section: 2

Timing Data:

- To determine how f(n) grows in n, we first plot n and runtime on a scatter graph and try out different functions to see which trendline it fits best.



We see that it fits the linear function graph the best and we display the equation and the R^2 value. The R^2 value represents how accurately we can predict the graph, the closer it is to 1, the more accurate we are.

To make sure it is a linear function, we plot the log values of n against the log values of runtime.

Let's say we have a polynomial function,

$T(n) = cn^k$

Taking log on both sides,

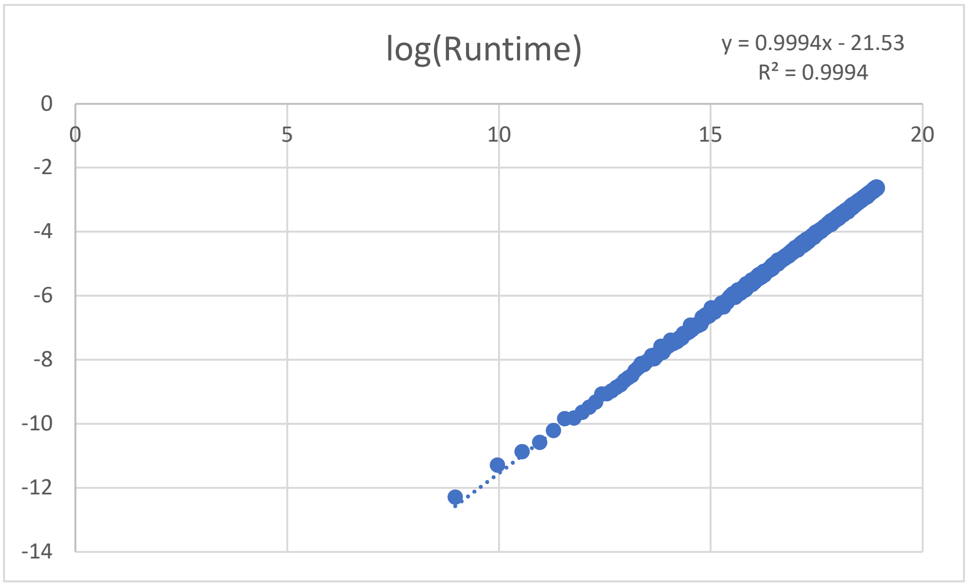
$Log(T) = k*log(n) + log(c)$

This looks like the general equation for a line,

$y = m*x + c$, where m is the slope and c is the intercept.

We can now deduce that from the log log graph, k should be the slope which also should be the exponent in the original function.

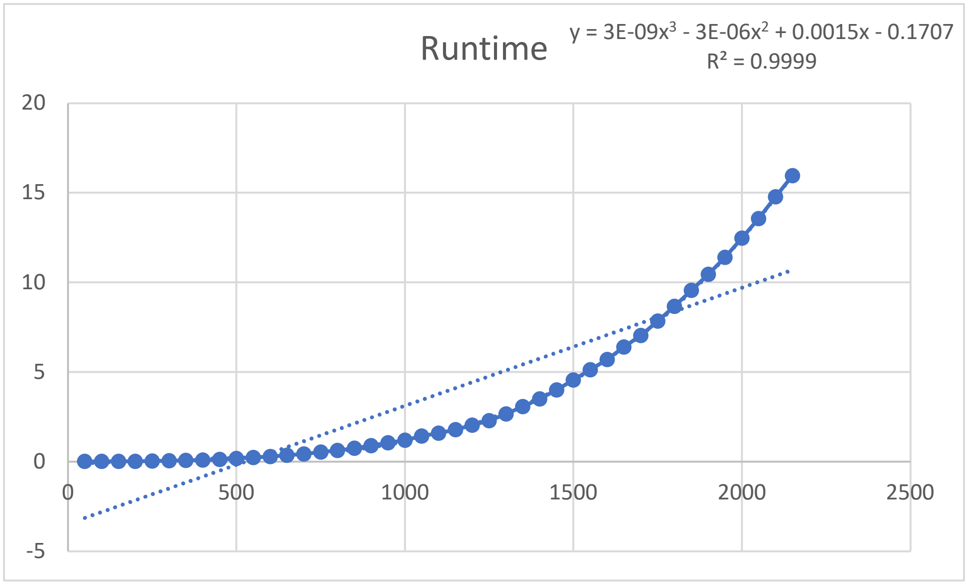
If we want to prove f(n) is a linear function, then the slope of the log log graph should be 1 or close to it.



After plotting the log log graph, we see that it does in fact form a linear and from the equation we see, the slope is close to 1 proving that the f(n) is a linear function.

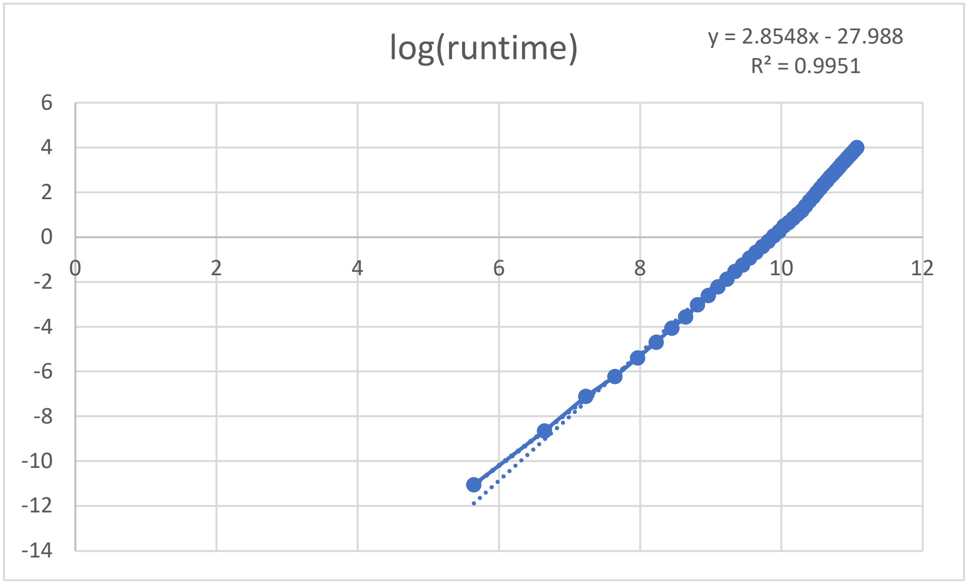
Therefore, n in f(n) grows linearly, T(n) = O(cn).

- To determine how g(n) grows in n, we first plot n and runtime on a scatter graph and try out different functions to see which trendline it fits best.



We noticed that it didn't fit the linear graph, nor logarithmic or exponential. We then tried polynomial functions of order higher than 1. As we reached 3, the R value became as close to 1 as it could. This led us to believe that maybe g(n) is a polynomial function of order 3.

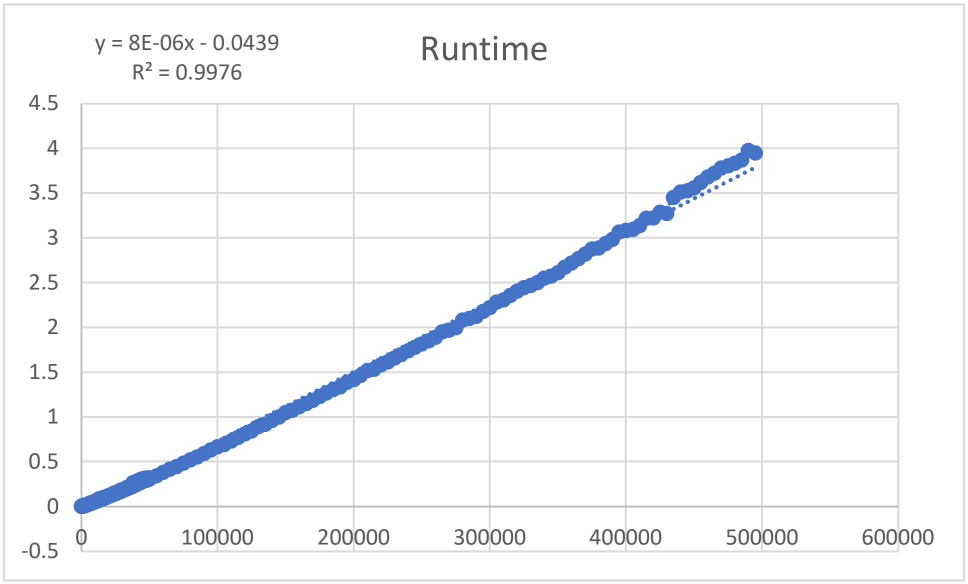
To make sure, we plotted the log log graph which must be linear to prove g(n) is a polynomial function while having a slope close to or 3 (using the logic from the previous f(n)).



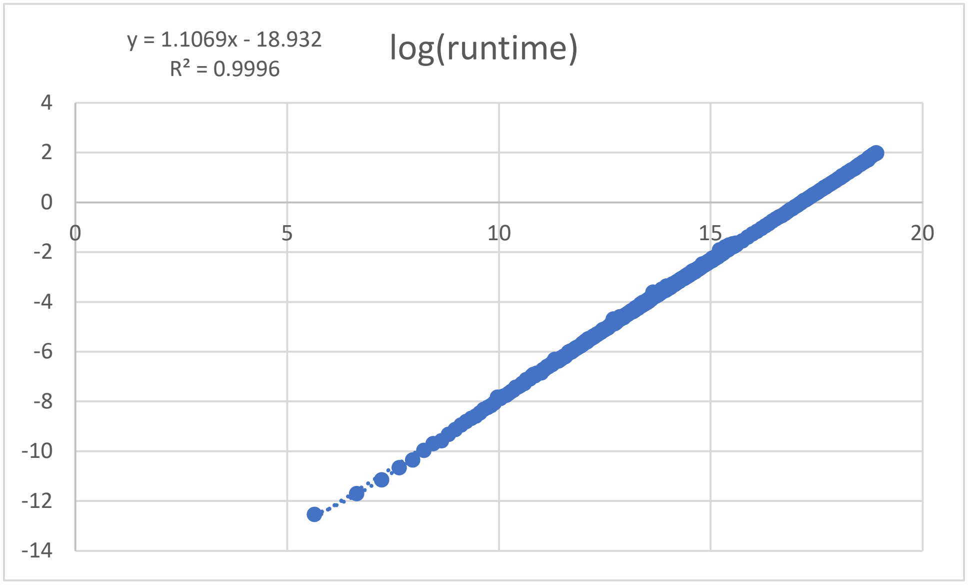
The slope of the log log graph is 2.8548 which is very close to 3, which proves our assumptions.

Therefore, g(n) is a cubic function, T(n) = O(cn^3)

- To determine how h(n) grows in n, we first plot n and runtime on a scatter graph and try out different functions to see which trendline it fits best.



It looks mostly like a linear graph but it doesn't coincide completely which means we have to prove some way else that this is a linear function.
If we want to prove f(n) is a linear function, then the slope of the log log graph should be 1 or close to it.



Seeing the equation of the log log graph, the slope is 1.1069 which is close to 1 which shows that our assumptions are right.

Therefore, n in h(n) grows linearly, T(n) = O(cn).

Python Lists

- Copy()

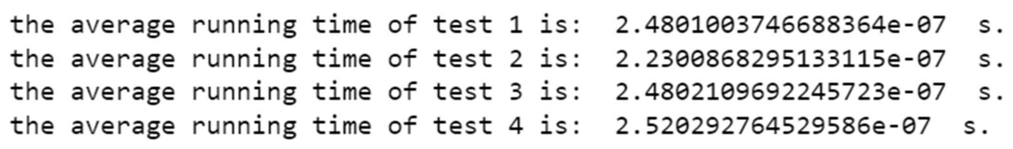
Reference: Code in copy.py

To test the complexity of copy(), we first wondered if different datatypes would affect our experiments.

To test this out, we created 4 different test cases of lists of same length with one being an empty list. Integers, strings, floating point numbers and an empty list. Would the data type affect the runtime? Using the timeit function, and running the test cases once, we noticed differences in each of the test cases.

In the spirit of this lab, testing anything once seemed a little too easy, so we decided to run each test case at least 100 times and taking the average of the runtime to test out our assumptions.

To our shock, we found out that the datatypes don't actually matter in the copy function, even an empty list takes almost the same time.



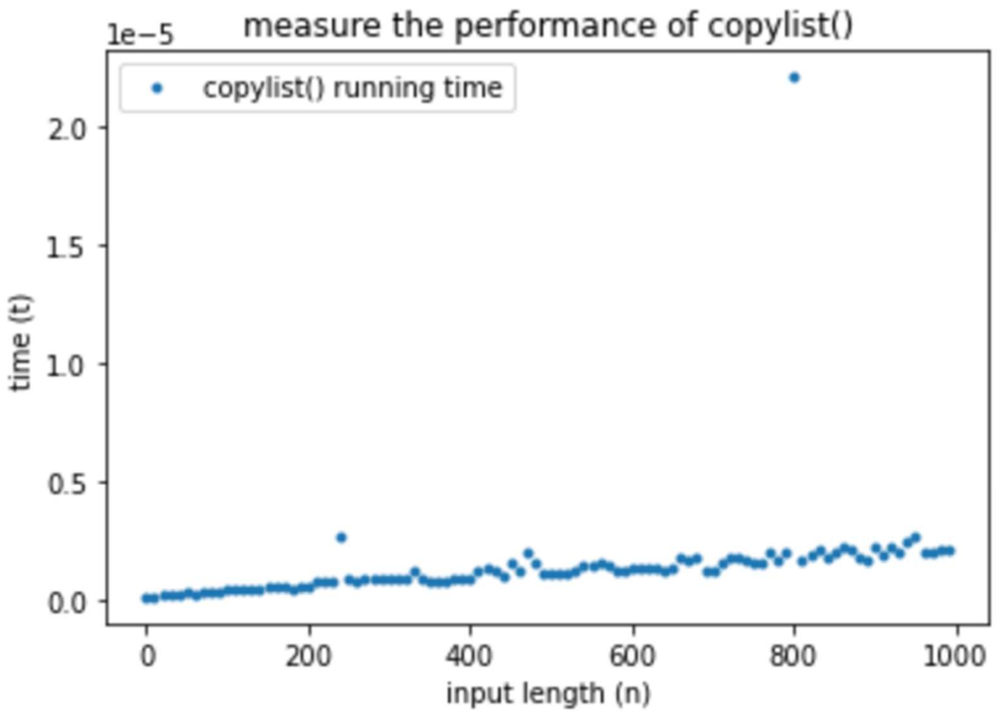
```
the average running time of test 1 is:  2.4801003746688364e-07  s.
the average running time of test 2 is:  2.2300868295133115e-07  s.
the average running time of test 3 is:  2.4802109692245723e-07  s.
the average running time of test 4 is:  2.520292764529586e-07  s.
```

We then decided to stick to integers and test the complexity of the copy() function.

We have a create random list function that takes the length of the list as a parameter and an upper limit of random integers. We used this function to plot a graph of the runtime against the length of the list.

The code for plotting gradually increases the input length of the list and calculates the average runtime for the copy() function by running it a 100 times for each length. When you run this code, to nobody's shock we get a linear looking graph. This suggests that the bigger the list the more time it will take to copy the list.

The function goes over every element i.e. n elements and makes a copy of it therefore, the complexity of copy() is O(n).



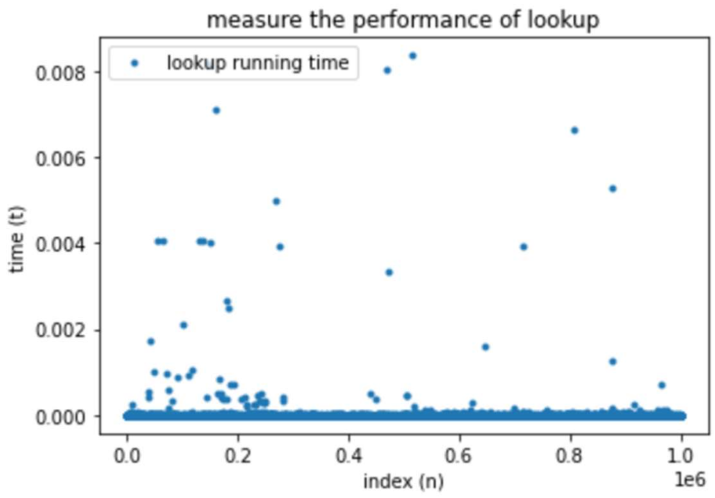
- Lookup

Reference: lookup.py

Prediction: Considering how arrays work, where the first element has the base address for the array and each element is referenced from the base address, we predicted that farther the element we look up, the more time it should take. Based on our predictions, the complexity of lookup should be O(n).

We then created a list of random million numbers, and calculated time taken for a single lookup and plotted runtime for each lookup.

```
time to look up a single index 0.00010270101483911276
```



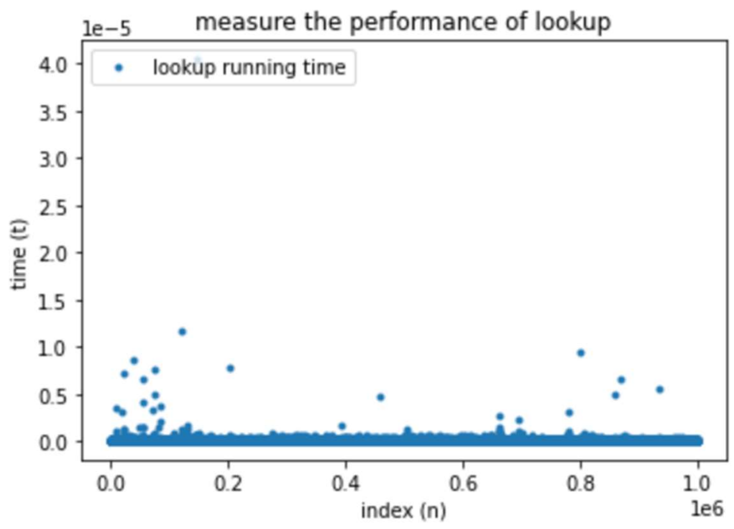
To our surprise, it’s a flat line. This implies that it takes constant time to lookup any value, no matter the length of the list. This is actually very impressive.
Therefore, complexity of lookup is O(1).

Upon further research, we found out since lists in python can have different data types, it keeps a list of pointers to the values which is why it can have constant time complexity unlike arrays.

To further improve our code and reduce the anomalies, we changed the code to take an average of looking up the same element 100 times and then taking its average to plot a more accurate graph.

```
rt =[]
n = range(1000000)
for i in n:
    time = 0
    for _ in range(100):
        start = timeit.default_timer()
        value = test[i]
        end = timeit.default_timer()
        time += (end - start)
    avgtime = time/100
    rt.append(avgtime)

plt.plot(n, rt, '.', label='lookup running time')
plt.xlabel("index (n)")
plt.ylabel("time (t)")
plt.title("measure the performance of lookup")
plt.legend(loc = "upper left")
plt.show()
```

```
time to look up a single index 6.206892430782318e-05
```

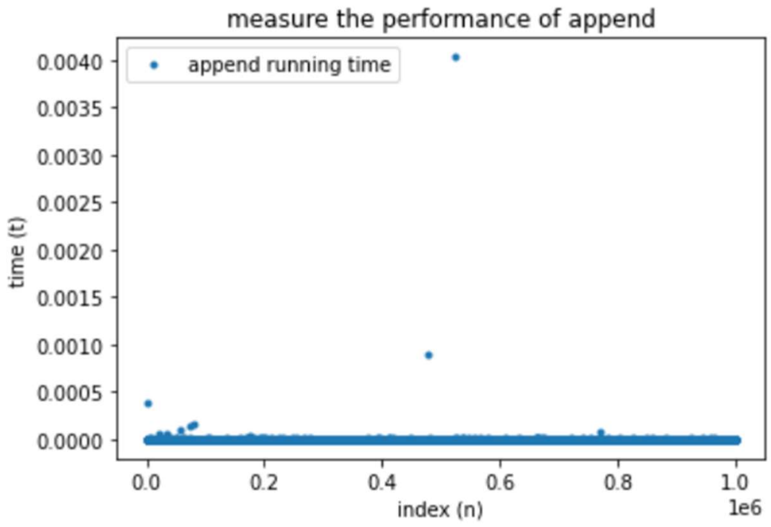


- Append
  Reference: append.py

Prediction: learning from our lookup experiments, with the knowledge we gained, we predicted that the time complexity for append should be constant since python doesn't need to work through the list to reach its end. Our predictions say the graph should look flat.

We wrote code to append a million elements one by one and timed it to plot a scatter plot.

```
time to add a single element 3.589503467082977e-05
```



Our predictions were proven correct.

The complexity of append is O(1).

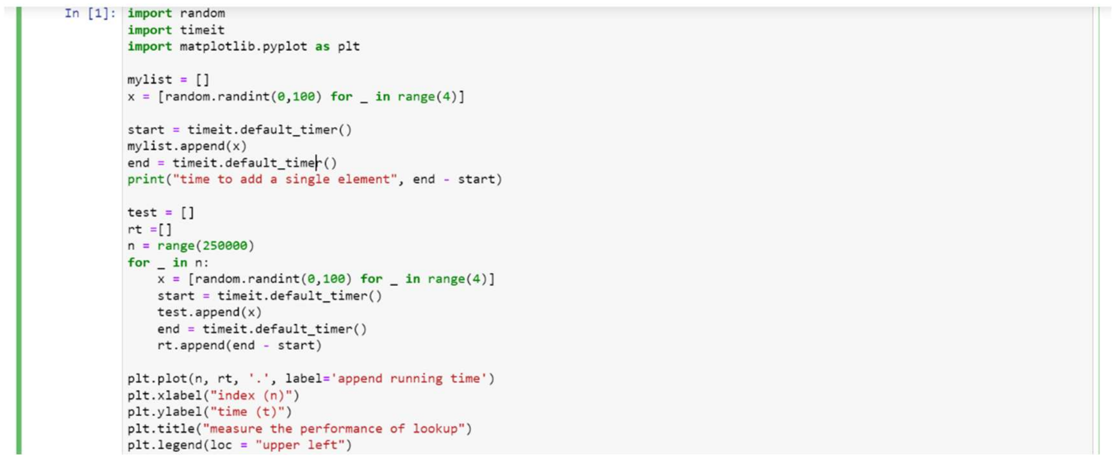
Potential problems:

Potentials problems for both lookup and append could be that all optimized processors have their memory access on predefined normal alignment. Sometimes, when the code runs accessing processors with wrong alignment can cause issues and give exceptions.

Experiments with append:

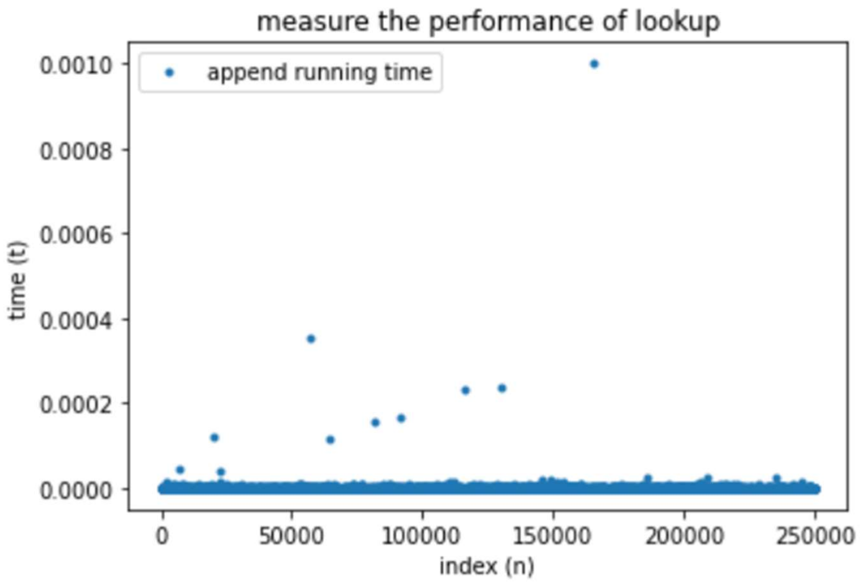
We tried seeing what happens if we append lists into a list, if that makes any difference even though the total elements will still be 1000000. We predicted that the running time would be lesser since we are not adding values individually.

Unfortunately, there were no perceptible changes regardless we did the experiment.



```
In [1]: import random
        import timeit
        import matplotlib.pyplot as plt

        mylist = []
        x = [random.randint(0,100) for _ in range(4)]

        start = timeit.default_timer()
        mylist.append(x)
        end = timeit.default_timer()
        print("time to add a single element", end - start)

        test = []
        rt =[]
        n = range(250000)
        for _ in n:
            x = [random.randint(0,100) for _ in range(4)]
            start = timeit.default_timer()
            test.append(x)
            end = timeit.default_timer()
            rt.append(end - start)

        plt.plot(n, rt, '.', label='append running time')
        plt.xlabel("index (n)")
        plt.ylabel("time (t)")
        plt.title("measure the performance of lookup")
        plt.legend(loc = "upper left")
```

```
time to add a single element 4.058191552758217e-05
```


measure the performance of lookup
● append running time
time (t)
index (n)