# SE 2XB3 Group 4 Report 2

Huang, Kehao
400235182
huangk53@mcmaster.ca
L01

Jiao, Anhao
400251837
jiaoa3@mcmaster.ca
L01

Ye, Xunzhou
400268576
yex33@mcmaster.ca
L01

January 29, 2021

# 1 Timeing Data

## 1.1 $f(n)$ Analysis

$f(n)$ is determined to be growing in the order of $\mathcal{O}(n)$. We started by plotting $f(n)$ on a x-y plane and observed a graph similar to that of a linear function. We formed our speculation on $f(n)$ being linear. A linear regression on the data set was then attempted to further investigate. As shown in Figure 1, the coefficient of determination is 0.9992, indicating that there is a high chance that $f(n)$ is indeed a linear function. To confirm our guesses, we plotted a log-log graph for $f(n)$ in Figure 2, and performed linear regression on the plot. Both the slope of the trend line and the $R^2$ value were approximately 0.9994. This is strong evidence that $f(n)$ grows in the order of $\mathcal{O}(n)$.

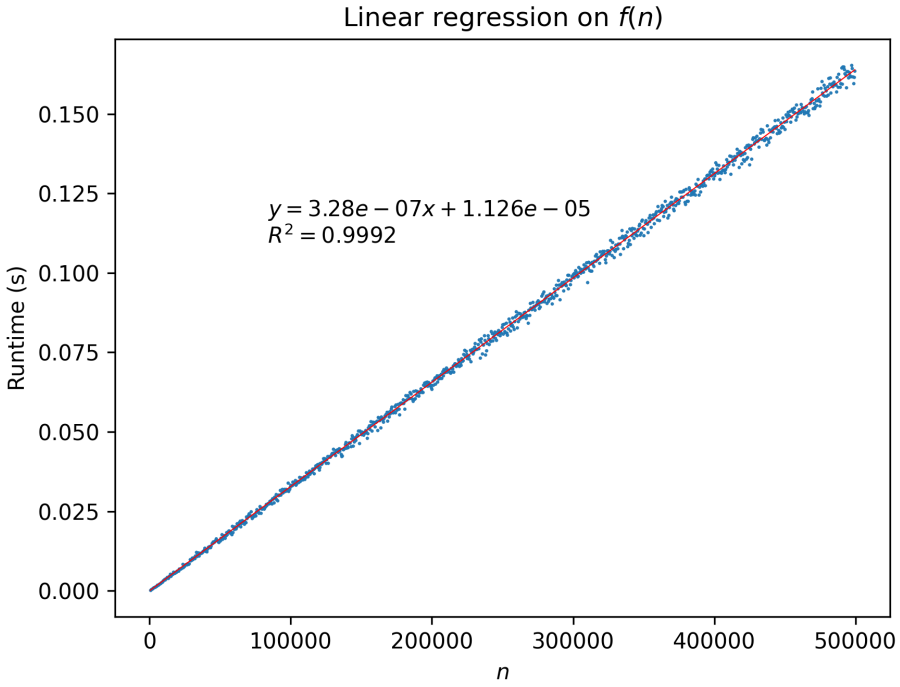


Figure 1: $f(n)$ linear regression

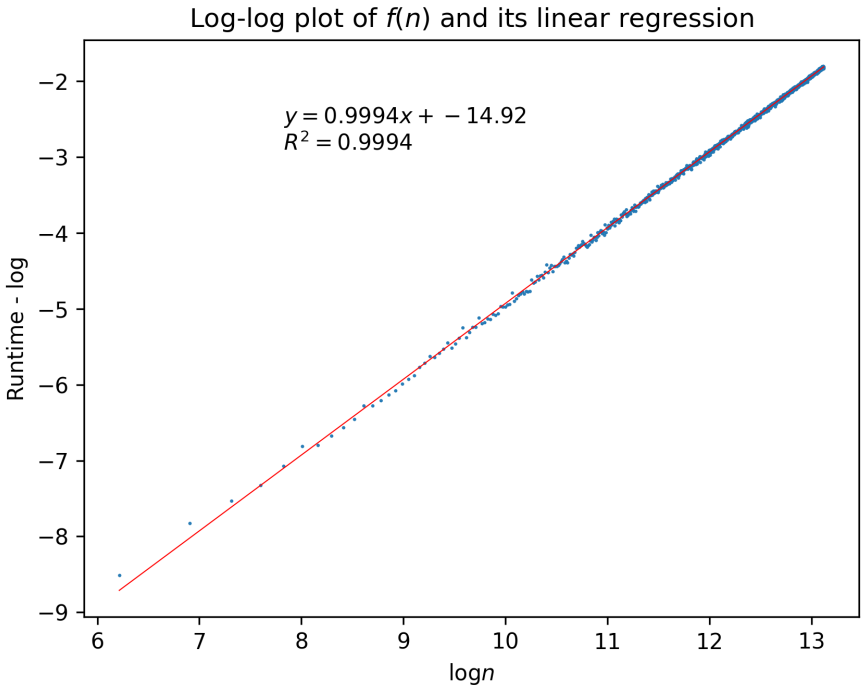


Figure 2: $f(n)$ log-log plot

## 1.2 $g(n)$ Analysis

By observing the graph of $g(n)$, our guess on the order of growth was polynomial or power. After applying the fitting line and constructed $R^2$ value, we found that the polynomial function with power of 3 fit the dots the most. However, the coefficient of $x^3$ is relatively smaller than the coefficient of $x^2$. To confirm our hypothesis on the growth order, we plotted the log graph and resulted in a linear regression with 2.8548 as the coefficient of $x$. Because there were nly 43 pairs of data were given for $g(n)$, so the insufficient amount of data ould cause the difference between 2.8548 and 3(the expected growth rate). Overall, with the finite given date set, we concluded that $g(n)$ has a growth order of $\mathcal{O}(x^3)$

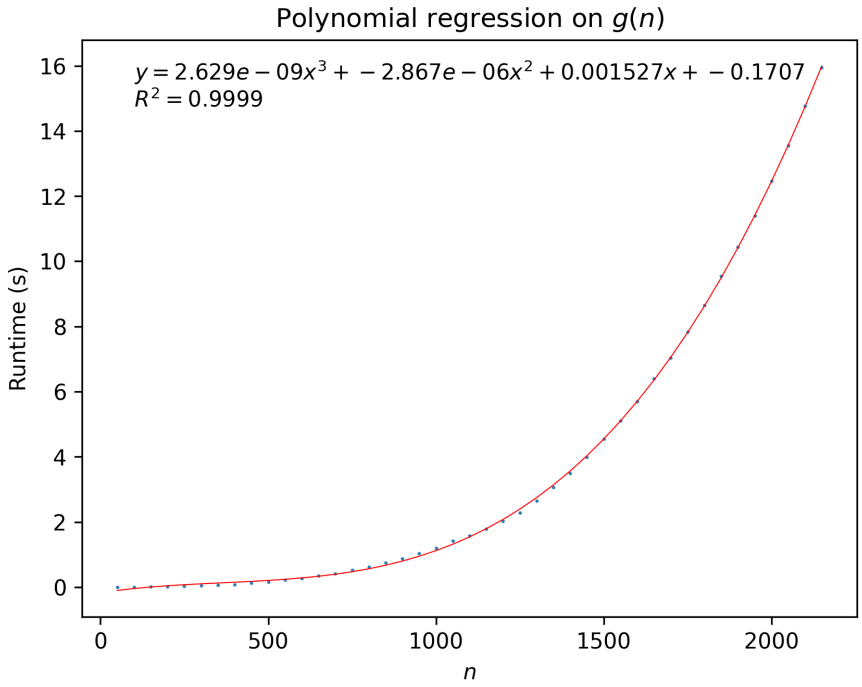


Figure 3: $g(n)$ polynomial regression

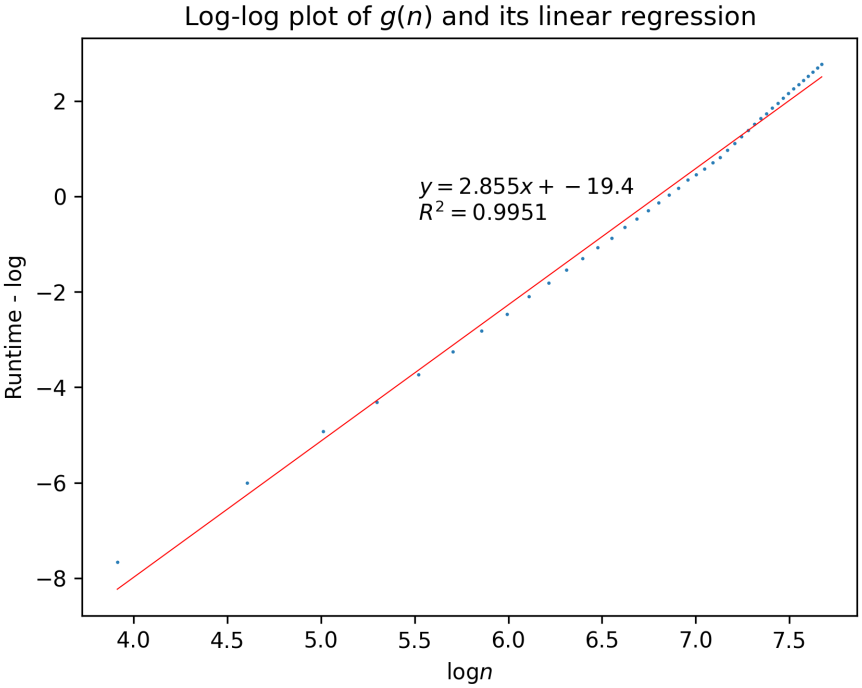


Figure 4: $g(n)$ log-log plot

## 1.3 $h(n)$ Analysis

After plotting the $h(n)$ data set into the graph, our first intuition about the type of growth was linear. However, by the visual contrast with the $h(n)$ in Figure 5, we all thought that the fitting line bended too much as a linear function. Therefore, we made the second graph which plotted with $log(n)$ and $log(runtime)$ in Figure 6, as the x-axis and the y-axis respectively. Then, we analyzed the coefficient of the term that has the highest power($x$). Compared to 1, the coefficient 1.1069 is off by quite a bit. At this point we were uncertain about the intuition we had at the beginning. Because the coefficient is larger than 1 which means it grows faster than linear, however, not as much as polynomial, exponential or power functions. At this point, we doubted that it could be $nlog(n)$ form. Last but not least, we divided the runtime($nlog(n)$) by its number($n$) and we got a $logn$ graph in Figure 7. Our assumption was right and the graph fits as a log function. In conclusion, the $h(n)$ grows in the order of $\mathcal{O}(nlog(n))$

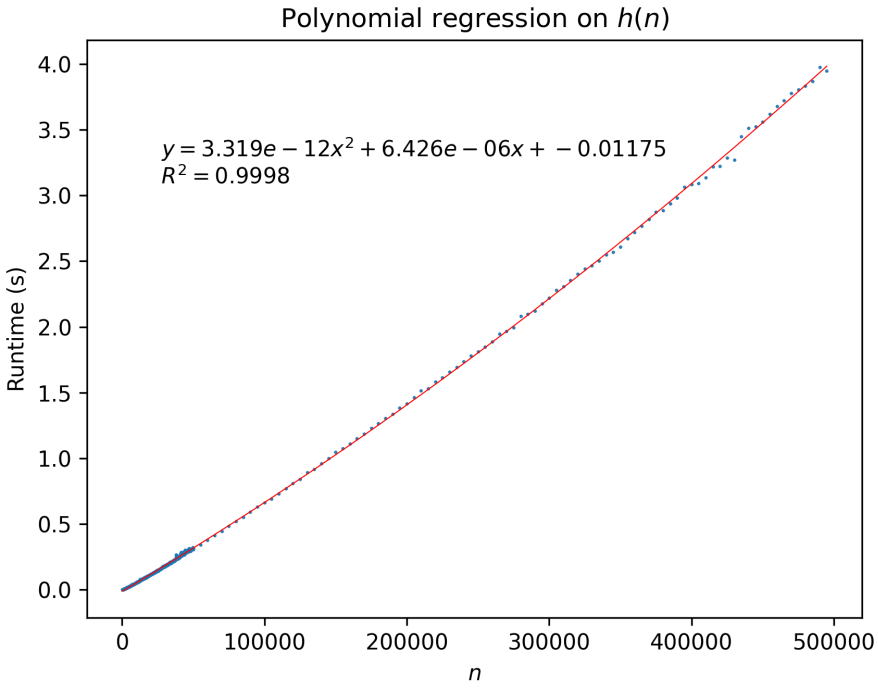


Figure 5: $h(n)$ polynomial regression

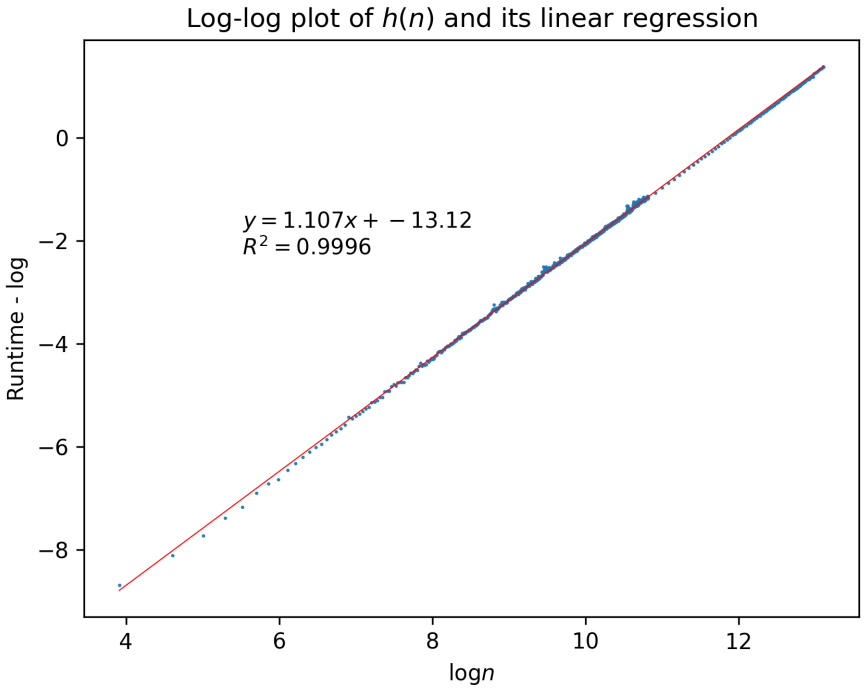


Figure 6: $h(n)$ log-log plot

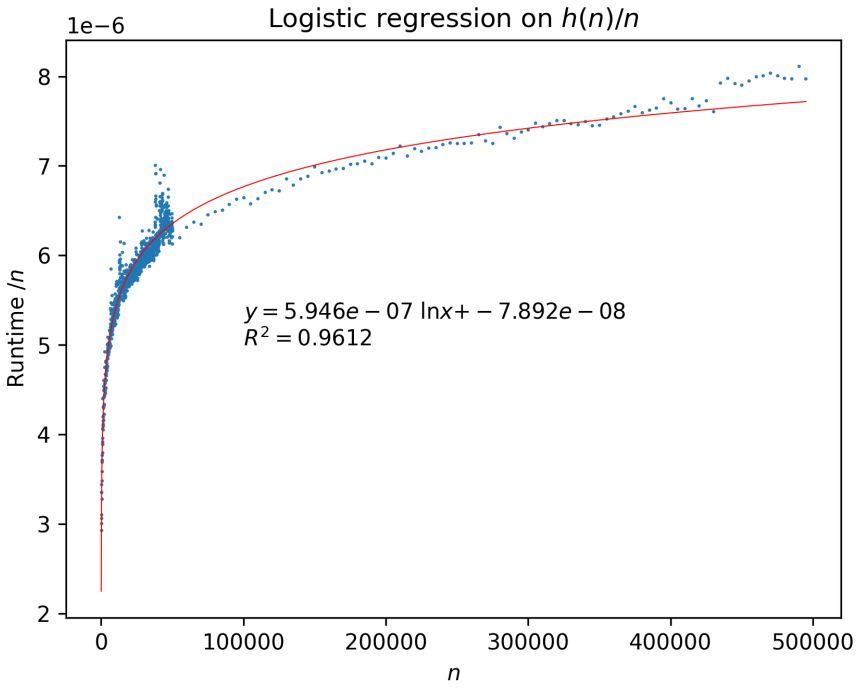


Figure 7: $h(n)/n$ logarithmic regression

# 2 Python Lists

## 2.1 `list.copy`

### 2.1.1 Methodology

We tested list.copy on lists of different lengths, ranging from 100 to 10000 with a step size of 100. For each round of the test, list.copy is called 100 times. The runtime of a round is the time required to copy one specific array 100 times. The reason for analyzing 100 list.copy calls, or in a sense normalizing each test, instead of making a single list.copy call is because a longer runtime helps to avoid floating point arithmetic errors from processing extremely small float numbers. For each given list length, n, 5 rounds of tests are carried out, and the minimum runtime out of the 5 is assigned as the runtime of the given length. Each round of test is performed with a fresh list of n random numbers. Inspired by the documentation for the timeit library, calculating mean and standard deviation from multiple rounds of tests is not very useful. In a typical case, the lowest value gives a lower bound for how fast a machine can run a test; longer runtimes are typically not caused by variability in Python's speed, but by other processes interfering with the timing accuracy. So the minimum runtime out of the 5 rounds is sufficient to represent the other runtimes.

### 2.1.2 Observations

The regression line of the plotted dots grows in the order of $\mathcal{O}(n)$. The $R^2$ value is 0.9911[1] which is close enough to 1, as shown in Figure 8. In order to back up our expectations, we created a log-log plot for the x and y values, in Figure . The resulting regression line has a slope close to 1, suggesting a linear relationship between the input size $n$ and the function runtime. We therefore conclude that `list.copy` grows in the order of $\mathcal{O}(n)$

[1]The exact value may not be matching with the one on the figure because the figure is generated using a new set of data in each run of the script.

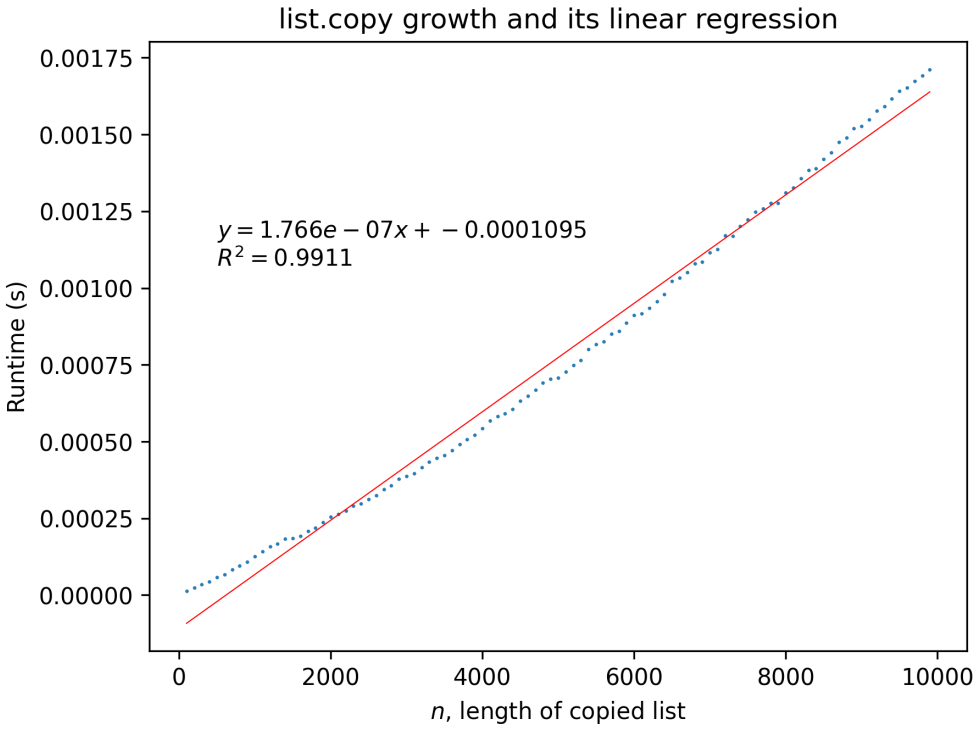


Figure 8: `list.copy` runtime

### 2.1.3 Explanation

`list.copy` copies the elements in a list one by one to a new location in the memory. The only factor that affects the runtime is $n$, the length of the list, or, the amount of data. The more data needed to be copied, naturally the longer the time `list.copy` needs.

## 2.2 lookup

### 2.2.1 Prediction

By the typical definition of arrays in common programming languages, random array access is expected to have constant time complexity. Since the builtin/native representation of array in Python is `list`, we expect lookups by value for `list` in Python to also have constant time complexity.

### 2.2.2 Potential Problem

Due to the expected huge amount of data generated during the experiment, it's impossible to copy and paste the data into other tools like Excel and generate the plot[2]. Therefore, we used a community Python library, `matplotlib` to plot the graphs we needed. The timing data generated by our experiment is directly fed and processed to the library within our Python module.

[2]Later in the experiment we did try and fail to do this

In addition, we believed that there will be a small portion of the data that deffer from the majority. These outliers are caused by other programs running in the background, different resource allocations by the operating system and so on.

### 2.2.3 Result

The plotted graph matches our prediction. As shown in Figure 9, the runtime of each list access is independent of which element is accessed.

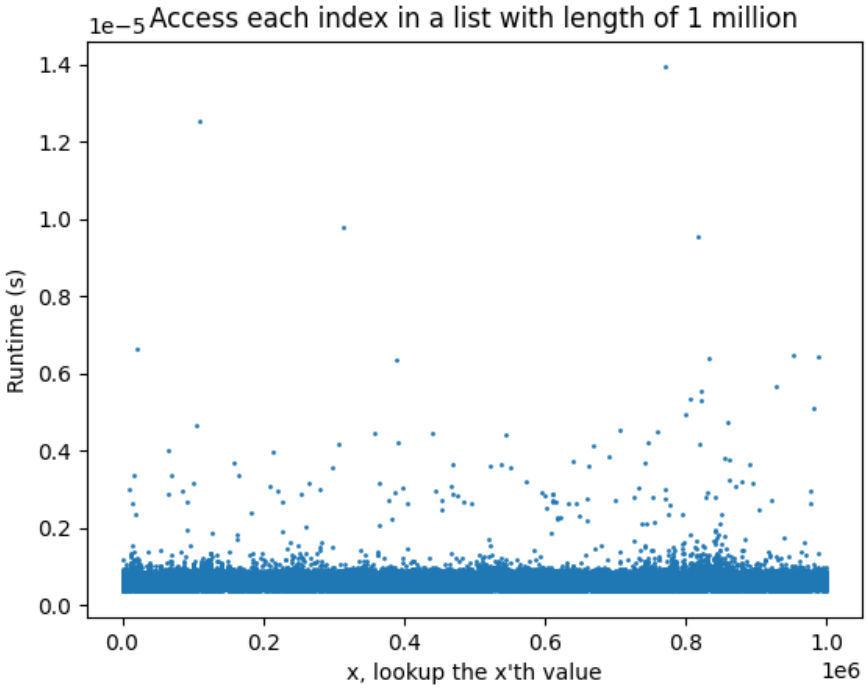


Figure 9: `list` access runtime

## 2.3 `list.append`

### 2.3.1 Prediction

The append method takes a constant amount of time to append each value to the list, the running time stays the same throughout the one million `list.append` calls. Similar to the lookups method, the `list.append` method appends each element to the tail of the list, it does not need to access any of the list or perform any operation. Therefore, we predict the trend of the plotted dots to be constant, which would be reflected as a horizontal line in the graph.

### 2.3.2 Potential Problem

As the same as the lookups, we will be dealing with a million data point at a time. Transferring data in and out excel and the huge workload on the computer would be our potential

problems. Based on the method we summarized on the previous test, we modified the code to fit our needs to get the plotted dot graph for the Append method. Also, we believed that there must be a small portion of the data that acts differently from the majority. Because there are always other programs running in the background, it may affect the performance of our operating system. As long as the incompatible data is relatively less than the majority, the result is still reliable.

### 2.3.3 Result

The result graph is as neat as our prediction. Figure 10 overall shows a straight horizontal line along with a few out of line data points. It proves our guesses that the `list.append` method has a constant growth order.

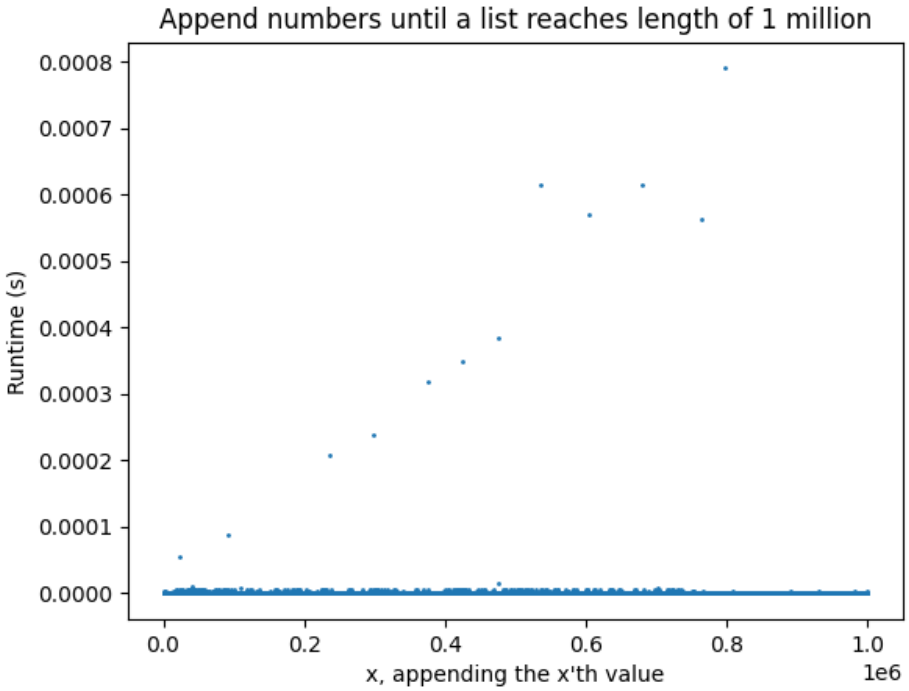


Figure 10: `list.append` runtime

### 2.3.4 Extra Experiment plan

In the previous experiment for `list.append`, we used the method to append one million integers in an increasing order into a `list`. To further investigate the behaviours and runtime of `list.append`, we decided to instead append a fixed string into a `list` 1 million times. We predicted the outcome to be the same as that of the previous experiment, because the runtime of an algorithm is usually independent of the data type on which the algorithm is used.

### 2.3.5 Extra Experiment Result

After appending one million strings to the list and generating the scatter plot of the running time, we proved our guess that the runtime of append does not depend on the data type, as shown in Figure 11. To conclude, the appended data consumes the same time, no matter what the data type is. The running time of append grows in $\mathcal{O}(1)$.

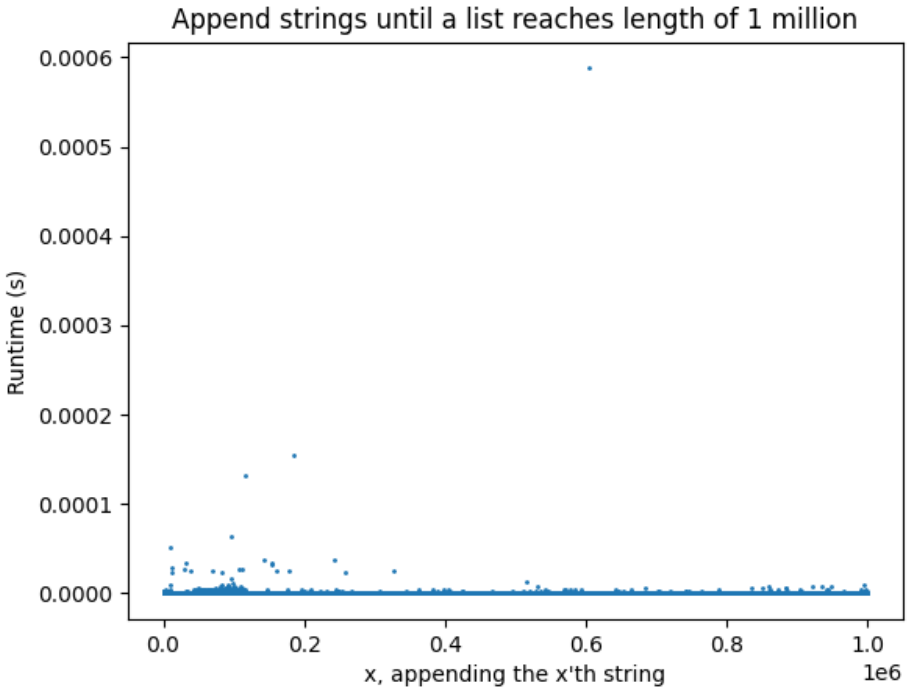


Figure 11: `list.append` runtime, operated on strings

### 2.3.6 Comparison to Official Runtime Claims

All of our experiment results are consistent with the Python complexity claims, namely, `list.copy` has a complexity of $\mathcal{O}(n)$, lookups (index) has a complexity of $\mathcal{O}(1)$, and `list.append` has a complexity of $\mathcal{O}(1)$.