CS/SE 2XC3 Lab 2 Report

Glotov, Oleg
L03, 400174037
glotovo@mcmaster.ca

Willson, Emma
L02, 400309856
willsone@mcmaster.ca

January 30, 2022

This report includes the main observations that we found in this week's lab, along with the analysis of our results.

1 Timing Data

In this section, we analyze the test results of three functions and give our best judgement of how each of these functions is growing in n.

1.1 $f(n)$

For the data set of $f(n)$, the trend line appears to be linear. From the chart below we can see that the R^2 is 0.9992 for the linear equation. It is already a very good result.

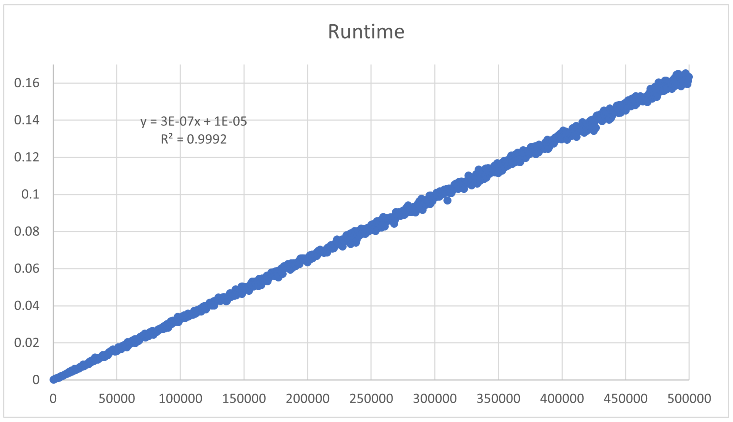

Figure 1: linear fitting for $f(n)$

Therefore, we can conclude that $f(n) = O(n)$.

1.2 $g(n)$

When we graph the data set for $g(n)$, the trend line appears to be polynomial. From the chart below we can see that the R^2 is 0.9883 for the quadratic equation.

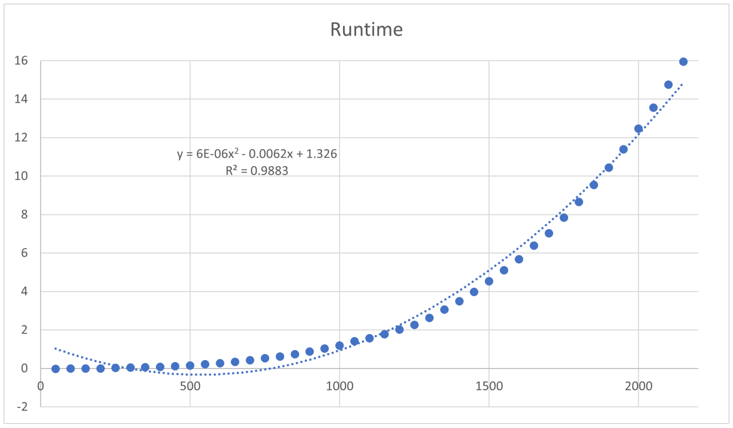

Figure 2: polynomial fitting for $g(n)$

We will try to improve the R^2 by finding the value of k in $T(n) = cn^k$. We do so by taking the logarithm of both sides of this equation: $\log T = \log c + k \log n$ and plotting $\log T$ against $\log n$.

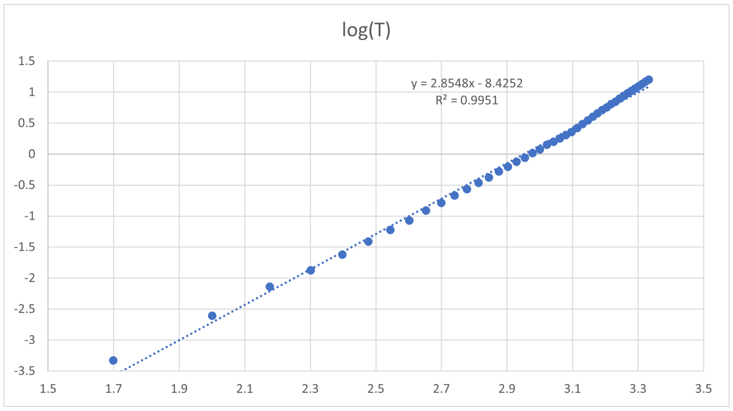

Figure 3: polynomial fitting for $\log T$

When we choose a linear trend line for this relation, we see that k, the slope, is 2.8548, which is closer to 3. When we recalculate the polynomial trend line for the original data with $k = 3$, we see that this is a better fit.

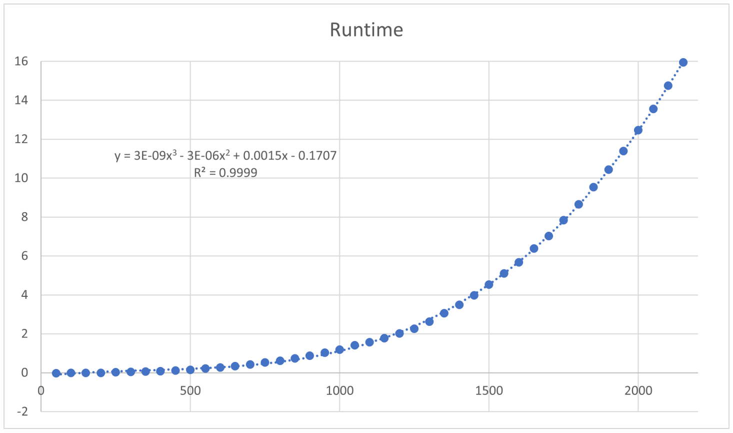

Figure 4: linear fitting for $g(n)$

Here, the R^2 is 0.9999, which is a very good result. Therefore we can conclude that $g(n) = O(n^3)$.

1.3 $h(n)$

When we graph the data set for $h(n)$, the trend line appears to be linear. From the chart below we can see that the R^2 is 0.9976 for the linear equation. This is already pretty good, but we might be able to improve it.

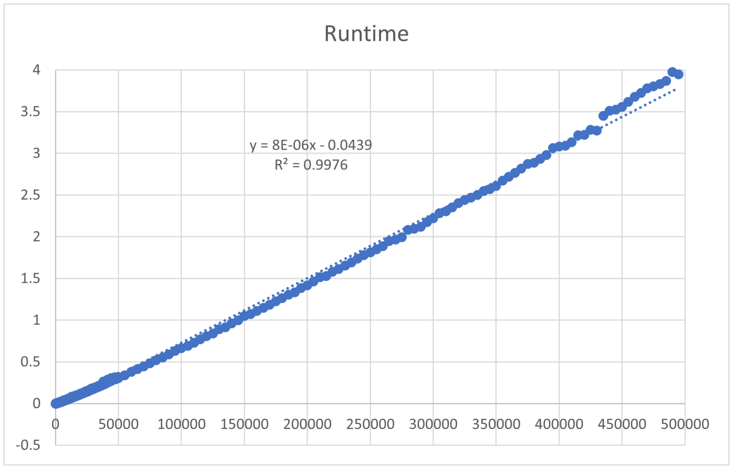

Figure 5: linear fitting for $h(n)$

First, we will check the value of k in $T(n) = cn^k$. We do so by taking the logarithm of both sides of this equation: $\log T = \log c + k \log n$ and plotting $\log T$ against $\log n$.

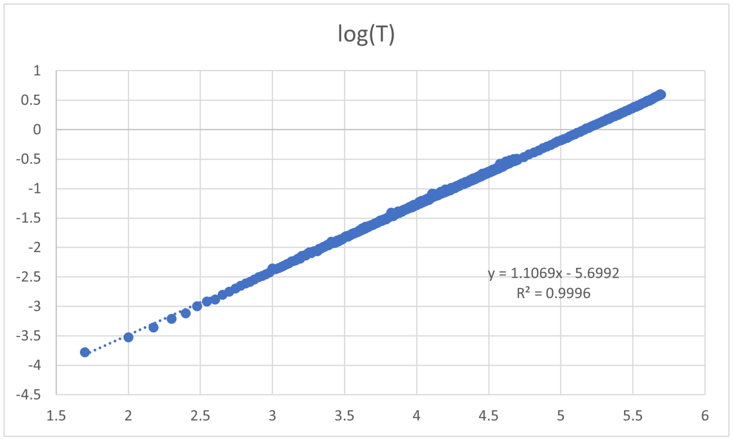

Figure 6: linear fitting for $\log T$

When we choose a linear trend line for this relation, we see that k, the slope, is 1.1069. Since k represents the exponent on n in the original data set, this makes the time complexity almost linear. However, $O(n \log n)$ may be a better fit than $O(n)$. We can check which is better by plotting $\frac{T(n)}{n}$ against n. If $T(n) = cn$, then the resulting graph $\frac{T(n)}{n} = c$, will be linear. If $T(n) = cn \log n$, then the resulting graph $\frac{T(n)}{n} = c \log n$, will be logarithmic.

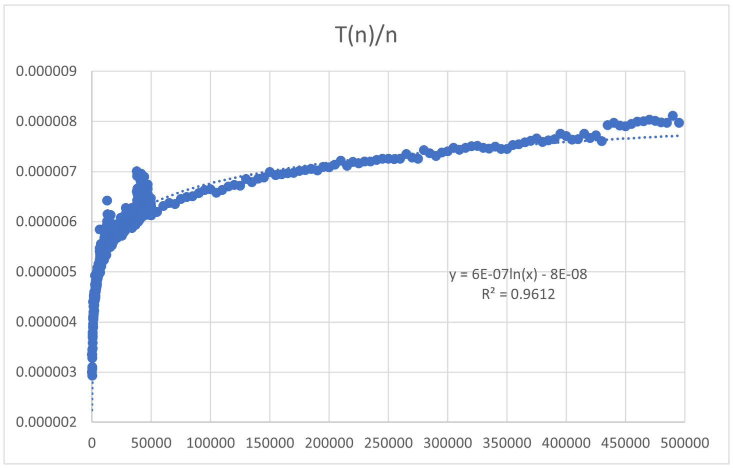

Figure 7: logarithmic fitting for $h(n)$

The resulting graph appears to be logarithmic. The R^2 is 0.9612 for the logarithmic trend line. Therefore we can conclude that $h(n) = O(n \log n)$.

2 Python Lists

To start, I would like to explain how each of the list methods is tested and evaluated. As specified in the .copy() instructions, the "timeit" library must be used. For the purpose of keeping the results consistent, I employed the library throughout the experiments. I used the function timeit.repeat() with parameters generating a total of 100 data points for a single input value. This data is then averaged and exported to a .csv file which is then analyzed in excel to determine the trend line and function complexity. The code to the "timer" is provided below and only changes slightly for each list method being analyzed.

```
def timer(index):
  if __name__ == '__main__':
    import timeit
    print("timing for {} runs".format(index))
    return timeit.repeat("arrayGenApp({})".format(index), setup="from __main__ import arrayGenApp", repeat=10, number=5)
```

2.1 Copy

For all three methods in question, very similar testing methods were selected. Since complexity is measured in the "big O" notation, where N is the number of inputs, the overall trend is seen by testing a relatively small number of inputs.

I decided to start at an array size of 1 and multiply the subsequent array by 10 each time until the array is 1 million in size. Later I added additional data points starting at 3 and following similar steps to make my data more complete.

```
timing for 1 runs
timing for 3 runs
timing for 10 runs
timing for 30 runs
timing for 100 runs
...
```

The actual array was different for each method tested. In the case of .copy() the array consisted of random integers between 0 and the size of the array being generated:

```
import random

def arrayGenApp(upperLimit):
  j = 0
  arr = []

  while (j < upperLimit):
    arr.append(random.randint(0,upperLimit))
    j += 1

  return arr
```

After generating the arrays and measuring the time it took to .copy() each one of them I received data which I plotted on the graph below. The data was plotted with a log scale to properly show the trend.

The initial flat section is consistent with the similar experiments found online and can be disregarded when analyzing the latter data points.

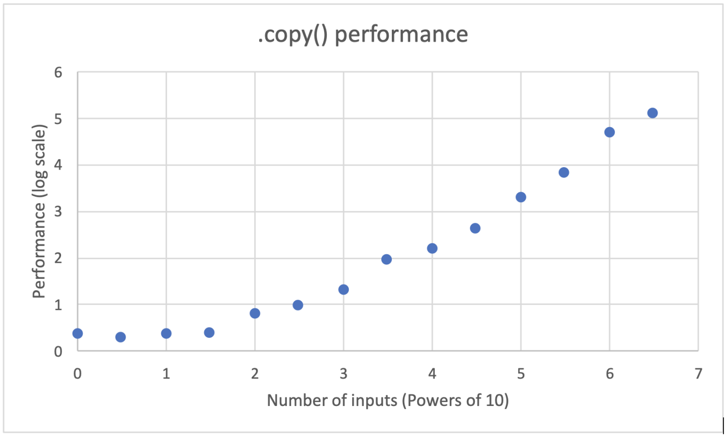

Figure 8: .copy() performance

Based on official documentation, python's .copy() method runs in O(n) time which is verified by the linear time complexity seen in the graph. This means that if we double an array size, it would take, on average, double the time to make a copy of it.

```
def copy(array):

  arrCpy = array.copy()

  return arrCpy
```

Since array.copy() iterates over each object in an array, adding references to a new array, it is expected that the runtime would be linear.

2.2 Lookups

Lookups used a similar testing method to the one described in .copy(). This time, each array consisted of values from 0 to the upper limit in a natural order. For example, an array of size 10 had the values 0,1,2,...8,9. After consulting with the TA's, this was the method I decided to use as it provided a better estimate of lookups for python lists.

```
def arrayGenApp(upperLimit): #values in array increasing
  j = 0
  arr = []

  while (j < upperLimit):
      arr.append(j)
      j += 1

  return arr
```

My prediction for the runtime was that the method would not depend on the size of the array, meaning it would run in constant time O(1). For each lookup the algorithm does not need to traverse the array, instead, it would directly access the element with the given index.

```
def arrayLook(array):
  i = 0
  while (i < len(array)):
    array[i]
    i += 1
```

I had a problem with the original formulation of the question because the given instructions would not properly test the lookup method. The correct approach in my opinion was to measure the time it would take to look up each value in different size arrays which is the approach I implemented.

Not only would this determine if the lookup method is dependent on array size, but also whether it is affected by the length of the variable being accessed. Since with my testing strategy, a larger array would have larger values stored within it.

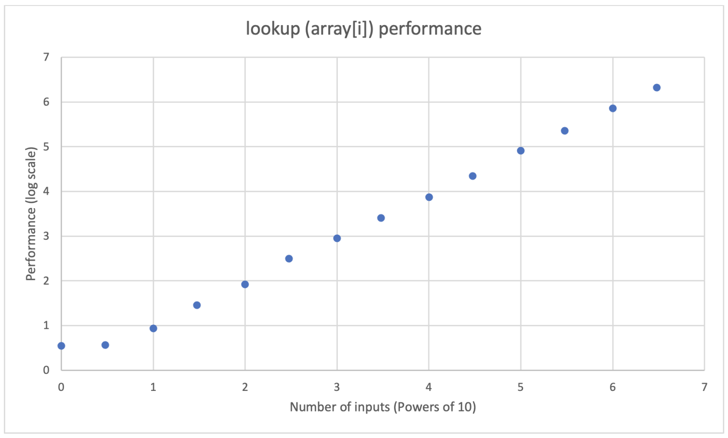

Figure 9: array[i] (lookup) performance

2.3 Append

Similar to the lookup method, the procedure described in the lab document would not produce an accurate result. By measuring the time it will take to append single values, we will inevitably have inaccurate data due to the nature of lists in general.

Since python lists are dynamic, meaning that they grow in size as more elements are added to them, some append operations would take significantly more time than others. For example, If we started with a dynamic array of size 4, we could append 4 elements to it, and each operation would take constant time. Yet appending a fifth element onto that array would take longer as the array would have to create a new array of double the current size (8), copy the old elements onto the new array, and then add the new element.

```
def arrayGenApp(upperLimit): #values in array increasing
  j = 0
  arr = []

  while (j < upperLimit):
      arr.append(1)
      j += 1

  return arr
```

Based on this evidence and a consultation with a TA I used the method I described when working on Lookups here. This method involves measuring the time it would take to create arrays of different sizes using the .append() method and then plotting the data. The generated arrays would progressively get bigger following the pattern seen in the .copy() method to ensure an accurate evaluation.

```
i = 1
j = 3
  while (i < 1000001):
    logtocsv(i, average(timer(i)))
    logtocsv(j, average(timer(j)))

    i *= 10
    j *= 10
```

The method employed here allows us to take care of the amortized analysis efficiently and elegantly, arriving at the correct conclusion.

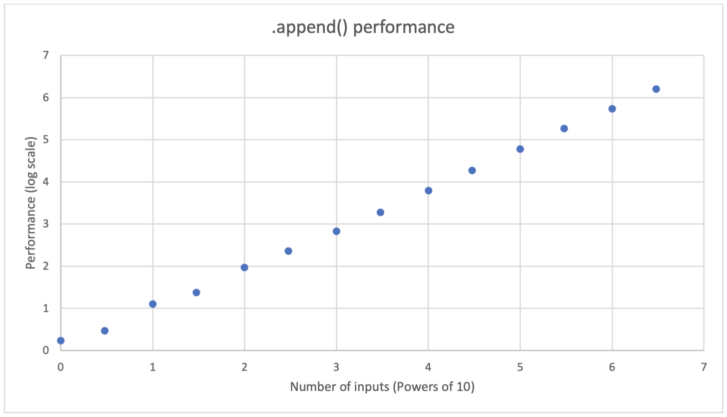

Figure 10: .append() performance

Based on the data seen in the graph above we can conclude that the complexity is linear which is consistent with the previously studied algorithms and the data available online.

3 Conclusion

For each method being tested the analyzed data is consistent with the provided reference, as well as other studies found online. The complexity of O(n) for .append() and .copy() methods are demonstrated by the linear model graphed in the figures 8 and 10. The general case for the lookup method is similarly explained in figure 9 and its complexity is determined.