CS/SE 2XB3 Lab 2 Report

Wang, Mingzhe
wangm235@mcmaster.ca

Li, Xing
li64@mcmaster.ca

Moon, Hyosik
moonh8@mcmaster.ca

January 29, 2021

This report concludes the main observations that we found in this week's lab, with the analysis that we did for the experiments of each methods.

1 Timing Data

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

1.1 Timing $f(n)$ data

For the data set of $f(n)$, the trend line looks like 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.

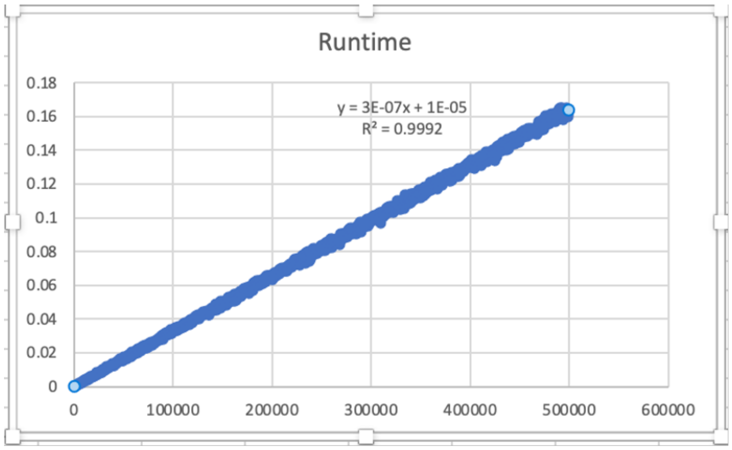

Figure 1: linear fitting for $f(n)$

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

1.2 Timing $g(n)$ data

For the data set of $g(n)$, the trend line looks not linear. From the chart below we can see that the R^2 is only 0.7974 for the linear model.

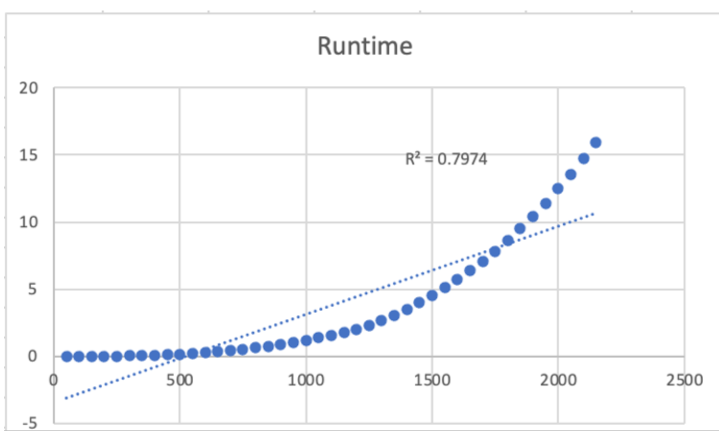

Figure 2: linear fitting for $g(n)$

With Polynomial model, R^2 is 0.9883 for quadratic and 0.9999 for cubic.

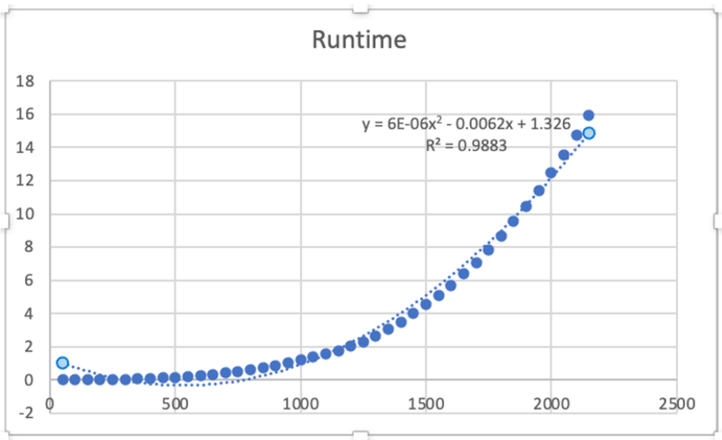

Figure 3: quadratic fitting for $g(n)$

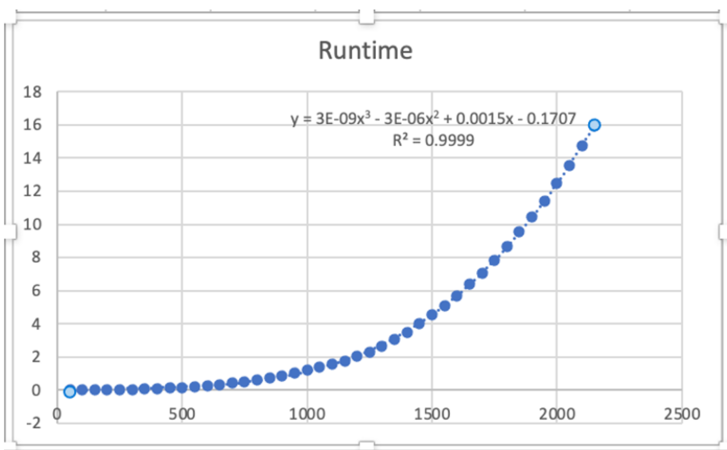

Figure 4: cubic fitting for $g(n)$

Therefore, we can conclude $g(n) \sim O(n^3)$

1.3 Timing $h(n)$ data

For the data set of $h(n)$, the trend line looks like linear, but it starts to deviate when the n is greater than 400,000.

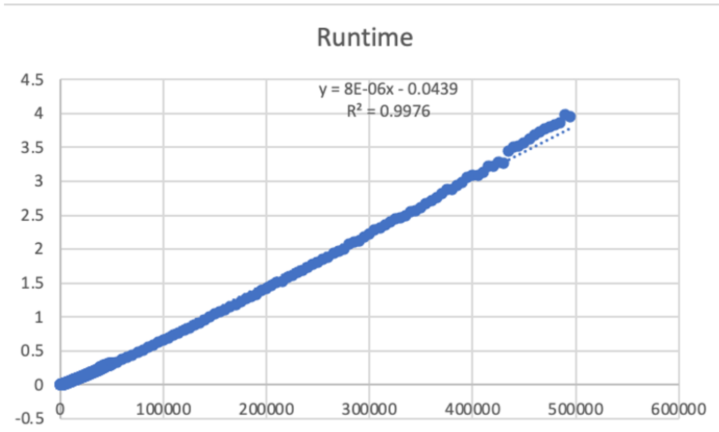

Figure 5: linear fitting for $h(n)$

Is it possible that it is actually $O(nlogn)$? We created a new series by dividing the runtime $h(n)$ by n, multiplying by 500,000, and fitting in a logarithmic model. The R^2 is 0.9612. If $h(n) \sim O(n)$, $h(n)/n$ should be a constant; if $h(n) \sim O(nlogn)$, $h(n)/n$ should be logarithmic. It seems $O(nlogn)$ is better to describe $h(n)$.

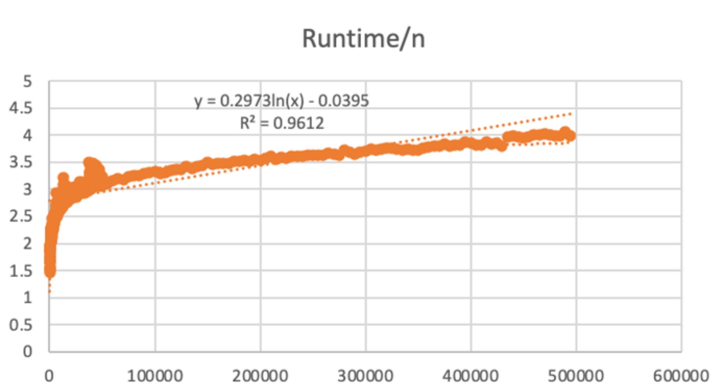

Figure 6: logarithmic fitting for $h(n)/n$

Therefore, we conclude that the best estimate with the given data is $h(n) \sim O(nlogn)$.

2 Python Lists

In this part, we will analyze the performance of three Python list methods, and give our conclusion about their time complexity and the corresponding reasoning.

2.1 Copy

(1) **Description of test method**

To test the `copy()` method, first we define a `create_random_list(n, upper)` method that creates a list of length `n` with random integers in the range of `[0, upper)` as elements. For each test, i.e. each possible value of `n`, there are runs with the number of `runs`. This design is to exclude possible distraction to show a more concise trending. For each run, a list `ls` is created first, then the time counter starts to record the actual running time of `ls.copy`. When all runs for each possible `n` value are finished, the average running time of `ls.copy` will be returned.

We also implement `plot_copy_test` to plot or get the data of the test result, the work principle of the methods are returning a `y` representing the y-coordinates by a given `x` representing the x-coordinates to plot the scatter diagram or return the data set. It should be noted that we use the `xlsxwriter` package to automatically write the data to excel files.

The source code for this part is as the following:

```
import matplotlib.pyplot as plt
import random
import timeit
import xlsxwriter

data_path = r'/Users/kidsama/Documents/COMPSCI
   2XB3/2xb3_lab2/list_data.xlsx'
workbook = xlsxwriter.Workbook(data_path)

def create_random_list(n, upper):
    return [random.randint(0, upper) for _ in range(n)]

# def copy test
def copy_test(runs, n, upper):
    total = 0
    for _ in range(runs):
        ls = create_random_list(n, upper)
```

```
        start = timeit.default_timer()
        ls.copy()
        end = timeit.default_timer()
        total += end - start
    return total/runs

# plot copy test
def plot_copy_test():
    x = [_ * 100 for _ in range(100)]
    y = []
    for _ in x:
        y.append(copy_test(100, _, 500))
    plt.scatter(x, y, marker='.')
    plt.xlabel('N ')
    plt.ylabel('T (s)')
    plt.title('Time complexity of copy())')
    copy_test_data = workbook.add_worksheet("copy_test_data")
    copy_test_data.write(0, 0, "N")
    copy_test_data.write_column(1, 0, x)
    copy_test_data.write(0, 1, "T")
    copy_test_data.write_column(1, 1, y)
```

(2) **Observations and conclusion**

From the scatter diagram derived from the test data, we conclude that the time complexity of `copy()` is $O(n)$, where n is the input size – the length of the argument list.

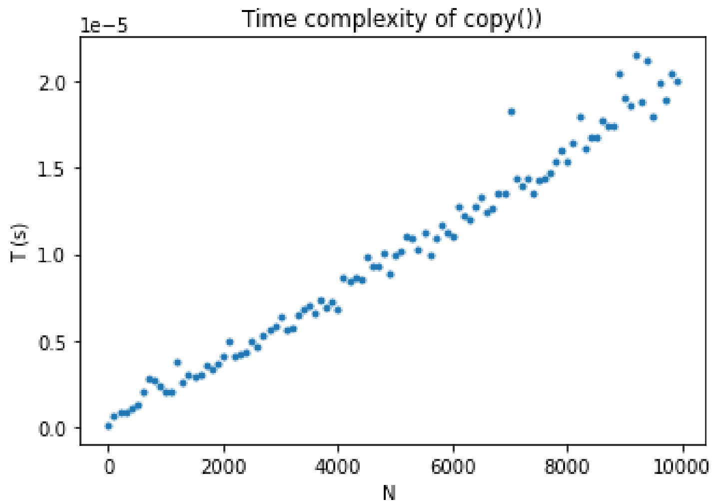

Figure 7: scatter plot of copy() test

(3) **Evidence**
We back up our observation by plotting the trend line of this scatter diagram and determine which one is best fitted using R^2.

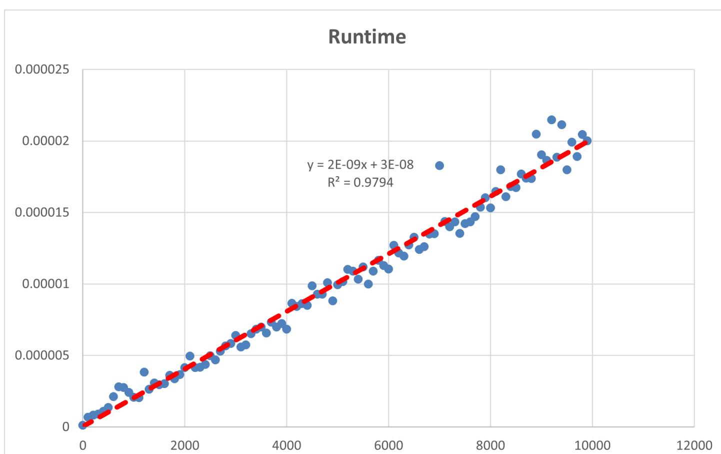

Figure 8: linear fitting for copy() test

Apparently, for linear trending line, R^2 is equal 0.9794, which is enough for us to verify the conclusion.

(4) **Explanation**
Through the research, we find that `copy()` in Python is actually a shallow copy.

A shallow copy creates a new list object, but it does not create new list elements. Instead, it simply copies the references to these objects. In other word, Python goes over all elements in the list and adds a copy of the object reference to the new list (copy by reference). Thus, the time complexity of `copy()` is $O(n)$.

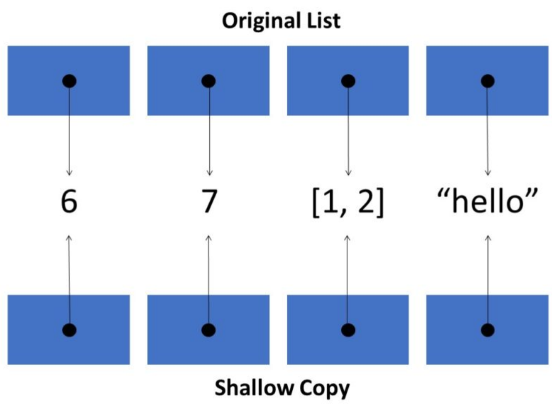

Figure 9: example of Python shallow copy

2.2 Lookups

(1) **Predictions**

Before we starting the test, our prediction is that the time complexity of `lookups`, i.e. `L[i]` is $O(n)$, where n is the max index of the item we looking for. Because in our theory courses, we are taught that the time complexity of traversing a linked list of length `n` to find a value is $O(n)$ in the worst case.

(2) **The original scatter plot**

To follow the professor's instructions strictly, we reduce the abstractions, i.e. the number of subroutines, in our implementation. That means all the statements are written in a single function. Our first version of test function `plot_lookups_test` is as the following:

```
def plot_lookups_test():
    ls = create_random_list(1000000, 1000000)
    x = range(1000000)
    y = []
    for i in x:
        start = timeit.default_timer()
        ls[i]
```

```
        end = timeit.default_timer()
        y.append(end - start)
    plt.scatter(x, y, marker='.')
```

It provides the first version of the scatter plot.

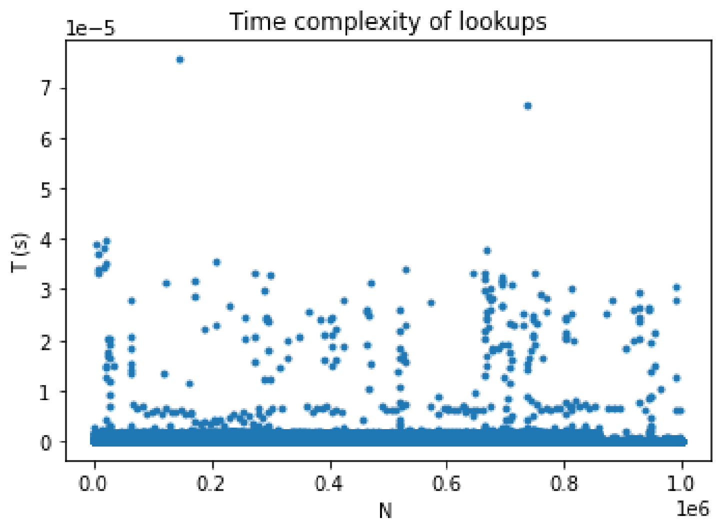

Figure 10: original scatter plot for lookups

(3) **Potential problems and corresponding fix**

Although the scatter plot shows certain trending of the complexity, there are some issues – there are too many outliers in the plot, which could reduce the goodness of fit in our next steps. To fix this issue, we apply a similar strategy as used in the `Copy()` test. For each possible value of `n`, we will run 500 times same tests and calculate the average time, which we expect can reduce the noises from irrelevant factors, like the OS, the concurrent programs, etc. The fixed version of test function `plot_lookups_test` is as the following:

```
def plot_lookups_test_fixed(runs):
    ls = create_random_list(1000000, 1000000)
    x = range(1000000)
    y = []
    for i in x:
        for _ in range(runs):
            total = 0
            start = timeit.default_timer()
            ls[i]
```

```
            end = timeit.default_timer()
            total += end - start
        y.append(total/runs)
    plt.scatter(x, y, marker='.')
    plt.xlabel('N ')
    plt.ylabel('T (s)')
    plt.title('Time complexity of lookups')
    copy_test_data =
       workbook.add_worksheet("lookups_test_data")
    copy_test_data.write(0, 0, "N")
    copy_test_data.write_column(1, 0, x)
    copy_test_data.write(0, 1, "T")
    copy_test_data.write_column(1, 1, y)
```

(4) **The revised scatter plot**
After fixing that issue, we rerun the experiment. As showed in Figure 11, the number of outliers in the scatter diagram is smaller.

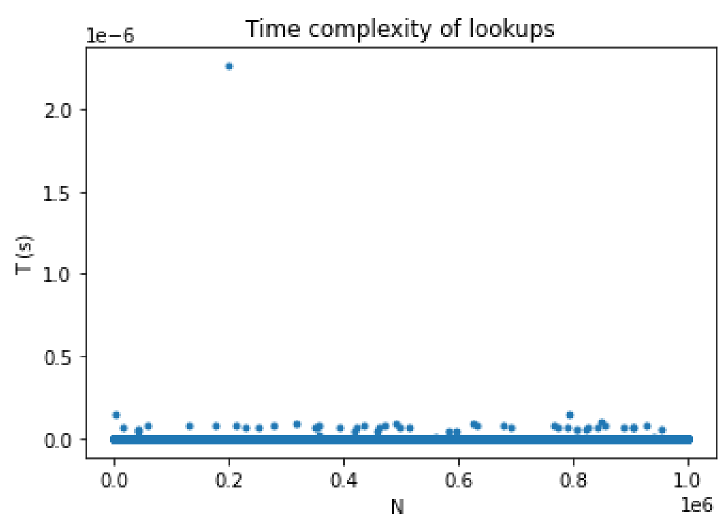

Figure 11: revised scatter plot for lookups

Then we export our data to excel to draw the trending line. Among all possible fitting functions, the constant line is the best.

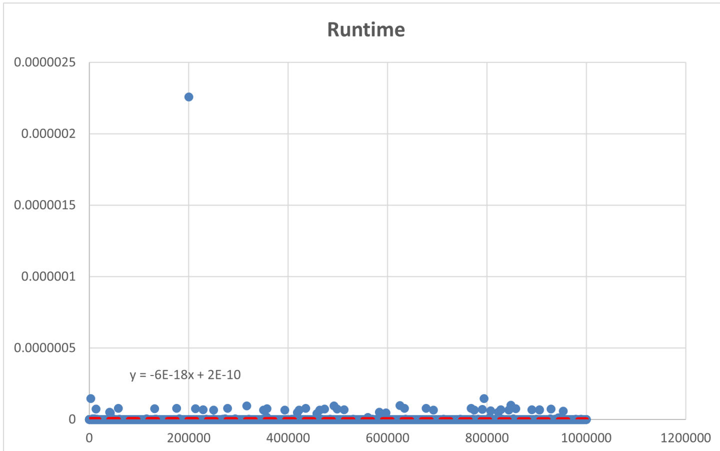

Figure 12: constant fitting for lookups

(5) **Conclusion**
The observation of our tests does not support our prediction that the run time of `lookups`, i.e. `L[i]` is $O(n)$ in the worst case. Instead, it shows that whatever the i is, the corresponding runtime is always a constant.

Our explanation is that the `list` type of Python is actually an array rather than a linked list. Thus, we can randomly access the data in the Python lists using their indexes in a constant time, which means the time complexity of this process is $O(n)$.

2.3 Append

(1) **Prediction**
Before we starting the test, again we create our prediction - the time complexity of `append()`, i.e. `list1.append(a)` is $O(1)$. The reason is that `append()` is a mutator that will mutate the current object. That means to achieve the 'append' behavior, it simply add a reference of the argument to the end of the array, which takes a constant time.

(2) **The implementation**
Following the design principle of lookups tests, we design our test function as the following:

```
def plot_append_test_fixed(runs):
    x = range(1000000)
    y = []
```

```
ls = []
for i in x:
    for _ in range(runs):
        total = 0
        value = random.randint(0, 1000000)
        start = timeit.default_timer()
        ls.append(value)
        end = timeit.default_timer()
        total += end - start
    y.append(total/runs)
plt.scatter(x, y, marker='.')
plt.xlabel('N ')
plt.ylabel('T (s)')
plt.title('Time complexity of append()')
copy_test_data =
   workbook.add_worksheet("append_test_data1")
copy_test_data.write(0, 0, "N")
copy_test_data.write_column(1, 0, x)
copy_test_data.write(0, 1, "T")
copy_test_data.write_column(1, 1, y)
```

The run times of the same tests for each possible value of `x` is set to be 100. As mentioned in the instruction, we instantiate the current list with one million random integers in the range of $[0, 1000000)$.

(3) **Scatter plot and trend line**
This time the output scatter plot seems fine, because we follow the previous design principle. We plot the scatter plot using `matplotlib.pyplot` and also convert our data to excel to find the best fit trend line.

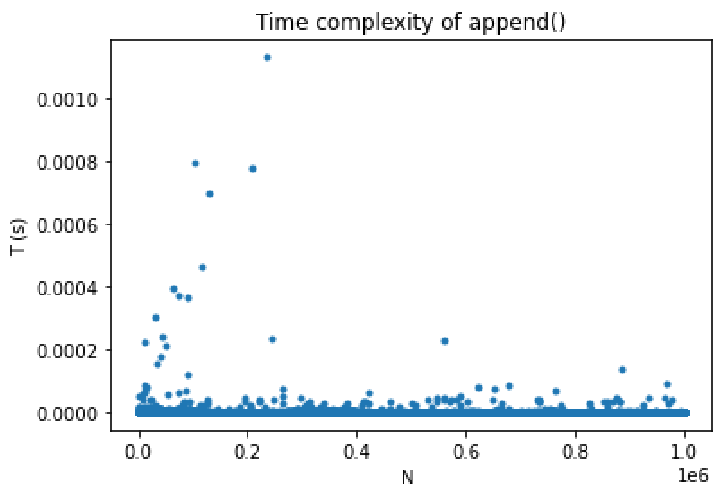

Figure 13: scatter plot for append() a single value

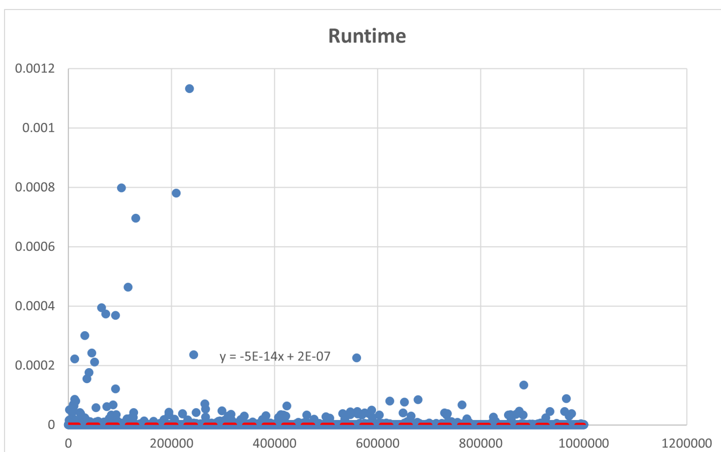

Figure 14: constant fitting for append() a single value

As showed in Figure 14, the best fit pattern of trending line is the constant line, which demonstrates our prediction that the time complexity of `append()`, i.e. `list1.append(a)` is $O(1)$.

(4) **Conclusion**

The time complexity of `append()`, i.e. `list1.append(a)` is $O(1)$. Because this method only adds ONE elements to the end of the current list, which requires only a constant number of operations – no matter the size of the list. To better understand the behavior of `append()`, consider this small example:

```
In [1]: a = [1, 2]
In [2]: a.append([3, 4])
In [3]: a
Out[3]: [1, 2, [3, 4]]
```

In fact, `append()` really only adds one element to the end of the current list, no matter the type of the value you want to add.

2.4 Addition append experiment

In this section, we want to create some additional experiments for testing the performance of `append()`. In the previous test, we did the timing experiment which builds a list with one million values by appending a single value to it one step at a time.

How about appending a single list of length `n` to a current list? Will the length of this appended list `n` affects the running time? To answer this question, we design the following test code:

```
def plot_append_ls2_test(runs):
    x = [_ * 1000 for _ in range(100)]
    y = []
    for n in x:
        for i in range(runs):
            total = 0
            ls = []
            ls2 = create_random_list(n, n)
            start = timeit.default_timer()
            ls.append(ls2)
            end = timeit.default_timer()
            total += end - start
        y.append(total/runs)
    plt.scatter(x, y, marker = '.')
    plt.xlabel('N ')
    plt.ylabel('T (s)')
    plt.title('Time complexity of append()')
    copy_test_data = workbook.add_worksheet("append_test_data2")
    copy_test_data.write(0, 0, "N")
    copy_test_data.write_column(1, 0, x)
    copy_test_data.write(0, 1, "T")
    copy_test_data.write_column(1, 1, y)
```

In this case, the output plots does not have too many outliers. So we simply set the `runs`, which is the run times of the same tests for each possible value of `x`, to be 1 to save the time. Each time, we add a Python list of length `n` in the set of $\{0, 1000, 2000, \ldots, 100000\}$ to an empty list.

Our prediction is still that the scatter diagram plotted will be a horizontal line, which means the running time of `append` is irrelevant to the length of the argument list.

We do the same process as before, plot the scatter diagram and then find the best fit trending line. The result is as the following:

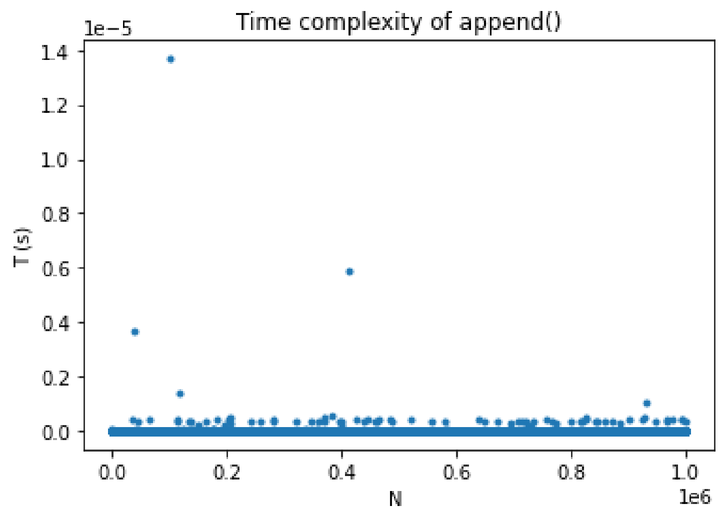

Figure 15: scatter plot for append() a list with the length n

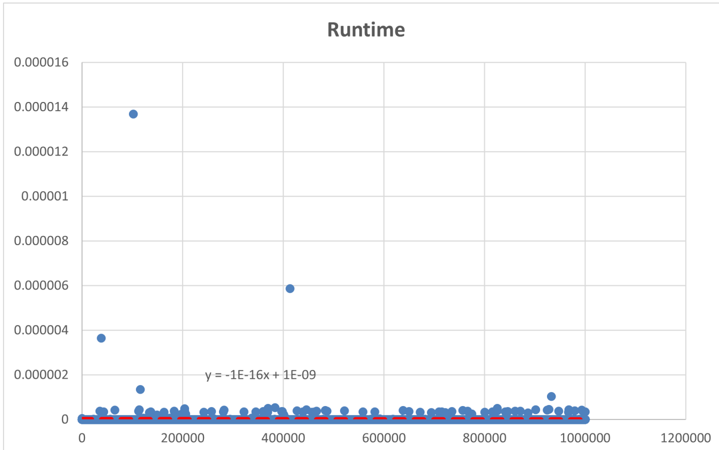

Figure 16: constant fitting for append() a list with the length n

Clearly, this result verifies our prediction that the running time of `append` is irrelevant to the length of the argument list.

By now, we have another question, does the length of the current object list matters, i.e. the length of `a` in `a.append(b)`? To answer this question, we design the following two tests:

```
def plot_append_ls3_test(runs):
    x = [_ * 1000 for _ in range(100)]
    y = []
    for n in x:
        for i in range(runs):
            total = 0
            ls = create_random_list(2000, 2000)
            ls2 = create_random_list(n, n)
            start = timeit.default_timer()
            ls.append(ls2)
            end = timeit.default_timer()
            total += end - start
        y.append(total/runs)
    plt.scatter(x, y, marker = '.')
    plt.xlabel('N ')
    plt.ylabel('T (s)')
    plt.title('Time complexity of append()')
    copy_test_data = workbook.add_worksheet("append_test_data3")
```

```
    copy_test_data.write(0, 0, "N")
    copy_test_data.write_column(1, 0, x)
    copy_test_data.write(0, 1, "T")
    copy_test_data.write_column(1, 1, y)

def plot_append_ls4_test(runs):
    x = [_ * 1000 for _ in range(100)]
    y = []
    for n in x:
        for i in range(runs):
            total = 0
            ls = create_random_list(n, n)
            ls2 = create_random_list(n, n)
            start = timeit.default_timer()
            ls.append(ls2)
            end = timeit.default_timer()
            total += end - start
        y.append(total/runs)
    plt.scatter(x, y, marker = '.')
    plt.xlabel('N ')
    plt.ylabel('T (s)')
    plt.title('Time complexity of append()')
    copy_test_data = workbook.add_worksheet("append_test_data4")
    copy_test_data.write(0, 0, "N")
    copy_test_data.write_column(1, 0, x)
    copy_test_data.write(0, 1, "T")
    copy_test_data.write_column(1, 1, y)
```

In the first test, a list with length n is appended to a list of length 2000; in the second test, a list with length n is appended to a list of length n. Their corresponding results are showed in the following scatter diagrams.

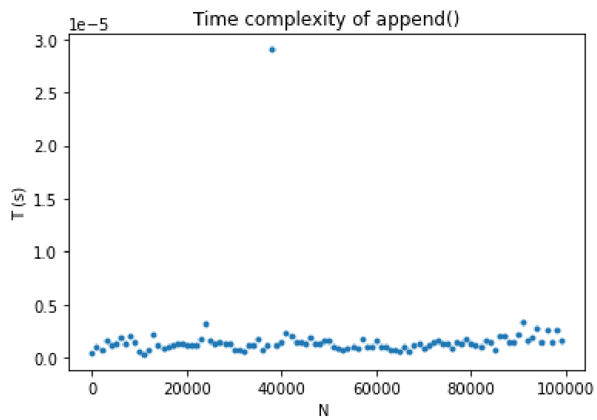

Figure 17: scatter plot for append() a n-length list to a 2000-length list

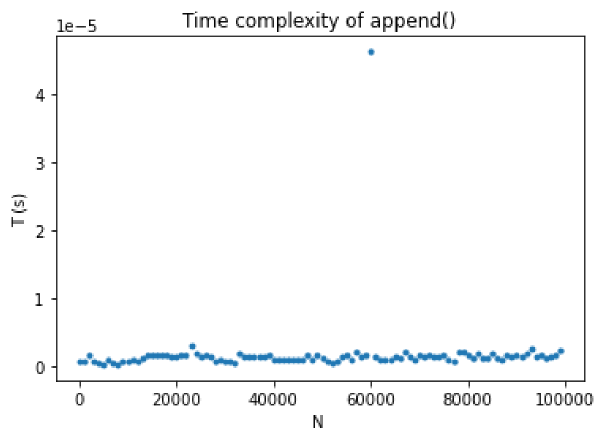

Figure 18: scatter plot for append() a n-length list to a n-length list

Apparently, both the length of the current list or the argument list does not affect the runtime of `append`. This observation answer our previous questions.

Appendix: source code

Below is all our test code for this lab, to test a single method, simply call the corresponding function, all the plot or data will be generated automatically. Note: you could need to change the path to make it work.

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Thu Jan 28 11:52:22 2021

@author: kidsama
"""
import matplotlib.pyplot as plt
import random
import timeit
import xlsxwriter

data_path = r'/Users/kidsama/Documents/COMPSCI 2XB3/2xb3_lab2/list_data.xlsx'
workbook = xlsxwriter.Workbook(data_path)

def create_random_list(n, upper):
    return [random.randint(0, upper) for _ in range(n)]

# def copy test
def copy_test(runs, n, upper):
    total = 0
    for _ in range(runs):
        ls = create_random_list(n, upper)
        start = timeit.default_timer()
        ls.copy()
        end = timeit.default_timer()
        total += end - start
    return total/runs

# plot copy test
def plot_copy_test():
    x = [_ * 100 for _ in range(100)]
    y = []
    for _ in x:
        y.append(copy_test(100, _, 500))
    plt.scatter(x, y, marker='.')
    plt.xlabel('N ')
    plt.ylabel('T (s)')
    plt.title('Time complexity of copy()')
    copy_test_data = workbook.add_worksheet("copy_test_data")
    copy_test_data.write(0, 0, "N")
    copy_test_data.write_column(1, 0, x)
    copy_test_data.write(0, 1, "T")
    copy_test_data.write_column(1, 1, y)

# lookups test
def plot_lookups_test():
    ls = create_random_list(1000000, 1000000)
    x = range(1000000)
    y = []
    for i in x:
        start = timeit.default_timer()
        ls[i]
        end = timeit.default_timer()
        y.append(end - start)
    plt.scatter(x, y, marker='.')
    plt.xlabel('N ')
    plt.ylabel('T (s)')
    plt.title('Time complexity of lookups')

def plot_lookups_test_fixed(runs):
    ls = create_random_list(1000000, 1000000)
    x = range(1000000)
    y = []
    for i in x:
        for _ in range(runs):
            total = 0
            start = timeit.default_timer()
            ls[i]
            end = timeit.default_timer()
```

```python
            total += end - start
        y.append(total/runs)
    plt.scatter(x, y, marker='.')
    plt.xlabel('N ')
    plt.ylabel('T (s)')
    plt.title('Time complexity of lookups')
    copy_test_data = workbook.add_worksheet("lookups_test_data")
    copy_test_data.write(0, 0, "N")
    copy_test_data.write_column(1, 0, x)
    copy_test_data.write(0, 1, "T")
    copy_test_data.write_column(1, 1, y)

def plot_append_test_fixed(runs):
    x = range(1000000)
    y = []
    ls = []
    for i in x:
        for _ in range(runs):
            total = 0
            value = random.randint(0, 1000000)
            start = timeit.default_timer()
            ls.append(value)
            end = timeit.default_timer()
            total += end - start
        y.append(total/runs)
    plt.scatter(x, y, marker='.')
    plt.xlabel('N ')
    plt.ylabel('T (s)')
    plt.title('Time complexity of append()')
    copy_test_data = workbook.add_worksheet("append_test_data1")
    copy_test_data.write(0, 0, "N")
    copy_test_data.write_column(1, 0, x)
    copy_test_data.write(0, 1, "T")
    copy_test_data.write_column(1, 1, y)

def plot_append_ls2_test(runs):
    x = [_ * 1000 for _ in range(100)]
    y = []
    for n in x:
        for i in range(runs):
            total = 0
            ls = []
            ls2 = create_random_list(n, n)
            start = timeit.default_timer()
            ls.append(ls2)
            end = timeit.default_timer()
            total += end - start
        y.append(total/runs)
    plt.scatter(x, y, marker = '.')
    plt.xlabel('N ')
    plt.ylabel('T (s)')
    plt.title('Time complexity of append()')
    copy_test_data = workbook.add_worksheet("append_test_data2")
    copy_test_data.write(0, 0, "N")
    copy_test_data.write_column(1, 0, x)
    copy_test_data.write(0, 1, "T")
    copy_test_data.write_column(1, 1, y)

def plot_append_ls3_test(runs):
    x = [_ * 1000 for _ in range(100)]
    y = []
    for n in x:
        for i in range(runs):
            total = 0
            ls = create_random_list(2000, 2000)
            ls2 = create_random_list(n, n)
            start = timeit.default_timer()
            ls.append(ls2)
            end = timeit.default_timer()
            total += end - start
        y.append(total/runs)
    plt.scatter(x, y, marker = '.')
    plt.xlabel('N ')
    plt.ylabel('T (s)')
    plt.title('Time complexity of append()')
    copy_test_data = workbook.add_worksheet("append_test_data3")
    copy_test_data.write(0, 0, "N")
    copy_test_data.write_column(1, 0, x)
    copy_test_data.write(0, 1, "T")
    copy_test_data.write_column(1, 1, y)
```

```
def plot_append_ls4_test(runs):
    x = [_ * 1000 for _ in range(100)]
    y = []
    for n in x:
        for i in range(runs):
            total = 0
            ls = create_random_list(n, n)
            ls2 = create_random_list(n, n)
            start = timeit.default_timer()
            ls.append(ls2)
            end = timeit.default_timer()
            total += end - start
        y.append(total/runs)
    plt.scatter(x, y, marker = '.')
    plt.xlabel('N ')
    plt.ylabel('T (s)')
    plt.title('Time complexity of append()')
    copy_test_data = workbook.add_worksheet("append_test_data4")
    copy_test_data.write(0, 0, "N")
    copy_test_data.write_column(1, 0, x)
    copy_test_data.write(0, 1, "T")
    copy_test_data.write_column(1, 1, y)

# test
# plot_copy_test()
# plot_lookups_test()
# plot_lookups_test_fixed(500)
# plot_append_test_fixed(1)
# plot_append_test_fixed(100)
# plot_append_ls2_test(1)
# plot_append_ls3_test(1)
# plot_append_ls4_test(1)

workbook.close()
```