DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Python Efficiency

submitted to:
Professor Joseph Picone
ECE 3822: Software Tools for Engineers
Temple University
College of Engineering
1947 North 12th Street
Philadelphia, Pennsylvania 19122

April 6, 2018

prepared by:
Von Kaukeano, Chad Martin
Email: tuh42003@temple.edu, tug96858@temple.edu

Table of Contents

Table of Figures

Table of Tables

SUMMARY

In this assignment, we examine six different python algorithms. These algorithms each produce a total count of how many total integers occur more than once. For benchmarking we increasing the total number of integers by a factor of ten, while also increasing the range of numbers by a factor of ten for scaling purposes. Each of the six cases are designed differently yet acquire the exact same output. After adding python's memory_profile library, we used its tools to compare each algorithm by the amount of memory each uses. After, we used the timer function from the standard python library to approximate the amount of time it takes to complete each algorithm. The time plotted was the average of time of ten runs per algorithm, this was repeated for each input size. Algorithm #1 is the most efficient solution because as we increased the input size it remained the fastest to count duplicates, while also using approximately the same or less memory as the other algorithms.

INTRODUCTION

This assignment begins with the six-different algorithms and learning how to use various python library tools to profile the algorithms. We used two libraries for profiling these algorithms. The first being a standard library, and the second being a non-standard called memory_profile. The memory_profiling library must be downloaded. You can download this library by following this hyperlink to memory_profiler. This tool gives a line by line memory usage that is sampled at a specified interval, the default used for this project was a tenth of a second. Each algorithm is executed ten times for each input size and then averaged. To preserve the data that would be output to the screen, it was appended to a text file that was specific to that algorithm and its input size. For the approximate run time, we used a timer function in python's library, then repeated the process of appending it to a text file specific to that algorithm and input size.

ANALYSIS

When the data is formatted as shown in Table 1 and Table 2. Then graphed like Figure 11 and Figure 10. We can correctly estimate the increase in CPU time with any input size as shown in Table 3. There was no significant data to support the that there was a difference in memory allocation for each of the algorithms. This conflicts with the norm of more memory being used the faster you make the algorithm. While using other primitive method and Figure 12 to better understand the why one algorithm is more efficient than the others we perceive that the algorithms have complexities shown in Table 4.

CONCLUSION

There may be some volatility in our data from the fact that we used a personal computer to perform the runs for memory and the Neuronix cluster to perform the time trials. We were not able to run the less efficient algorithms on the personal computer due to the time required to complete the run, we deemed these runs impractical. From the trend shown from Table 1 we estimated the memory usage to be the same or very close to the other algorithms.

APPENDIX

1.1. Tables

Algorithm Memory Usage in MiB					
Size: / **Algorithm:**	**10^2**	**10^3**	**10^4**	**10^5**	**10^6**
1	~10.35	~10.38	~10.95	~16.54	~76.33
2	~10.30	~10.44	-	-	-
3	~10.30	~10.37	~10.71	~16.55	~75.5
4	~10.36	~10.40	~10.75	~16.37	-
5	~10.43	~10.43	~10.76	~17.19	-
6	~10.36	~10.41	~10.78	~17.33	-

Table 1. Algorithm Memory Usage

Algorithm CPU Time in seconds					
Size: / **Algorithm:**	**10^2**	**10^3**	**10^4**	**10^5**	**10^6**
1	1.88E-05	1.44E-04	1.54E-03	1.96E-02	2.43E-01
2	3.31E-04	2.75E-02	2.71	3.04E+02	3.82E+04
3	1.13E-05	7.48E-04	7.35E-02	7.74	8.14E+02
4	9.20E-06	7.44E-04	7.34E-02	7.75	8.07E+02
5	2.72E-05	3.13E-04	3.68E-03	5.20E-02	6.46E-01
6	1.40E-04	1.59E-02	2.74E+00	1.81E+02	4.29E+04

Table 2. Algorithm CPU Time

Algorithm	Equation for total time where x input size
1	$T = e^{(1.0356x - 5.8471)}$
2	$T = e^{(2.0168x - 5.5588)}$
3	$T = e^{(1.9732x - 7.0011)}$
4	$T = e^{(1.9904x - 7.0717)}$
5	$T = e^{(1.0974x - 5.6879)}$
6	$T = e^{(2.1026x - 5.9726)}$

Table 3. Equations for Total Time

Algorithm	Complexity
1	O(n log(n))
2	O(n!)
3	O(n^2)
4	O(n^2)
5	O(n log(n))
6	O(2^n)

Table 4. Complexity

1.2. Figures

```
def FindDuplicates(numbers):
	d = {}
	for val in numbers:
		d[val] = d.get(val, 0) + 1
	return sum(d[i] > 1 for i in d)
```

Figure 1. Algorithm 1

```
def FindDuplicates(numbers):
	dupVals = []
	for i in range(0, len(numbers)):
		for j in range(i+1, len(numbers)):
			if numbers[j] == numbers[i] and numbers[j] not in dupVals:
				dupVals.append(numbers[j])
	return len(dupVals)
```

Figure 2. Algorithm 2

```
def FindDuplicates(numbers):
	temp = []
	foo = 0
	numberz = set(numbers)
	for num in numberz:
		temp.append(numbers.count(num))
	for num in temp:
		if num > 1:
			foo += 1
	return foo
```

Figure 3. Algorithm 3

```
def FindDuplicates(numbers):
	x=[]
	for n in set(numbers):
		count = numbers.count(n)
	if count > 1:
		x.append(n)
	return (len(x))
```

Figure 4. Algorithm 4

```
def FindDuplicates(numbers):
    counter = 0
    if len(numbers) < 2:
        return counter
    else:
        numbers.sort()
        dup = 0
        for i in range(1,len(numbers)):
            if ((numbers[i-1] == numbers[i]) & (dup == 0)):
                counter = counter + 1
                dup = 1
            elif numbers[i-1] != numbers[i]:
                dup = 0
    return counter
```

Figure 5. Algorithm 5

```
def FindDuplicates(numbers):
    allDupes = [x for x in numbers if numbers.count(x) >= 2]
    uniqueDupes = list(set(allDupes))
    numberOfDupes = len(uniqueDupes)
    return numberOfDupes
```

Figure 6. Algorithm 6

```
#!/usr/bin/env python
import random
from timeit import default_timer as timer

def FindDuplicates1(numbers):

    d = {}
    for val in numbers:
        d[val] = d.get(val, 0) + 1
    return sum(d[i] > 1 for i in d)

def FindDuplicates2(numbers):
    dupVals = []
    for i in range(0, len(numbers)):
        for j in range(i+1, len(numbers)):
            if numbers[j] == numbers[i] and numbers[j] not in dupVals:
                dupVals.append(numbers[j])
    return len(dupVals)

def FindDuplicates3(numbers):
    temp = []
    foo = 0
    numberz = set(numbers)
    for num in numberz:
        temp.append(numbers.count(num))
    for num in temp:
        if num > 1:
            foo += 1
    return foo

def FindDuplicates4(numbers):
    x=[]
    for n in set(numbers):
        count = numbers.count(n)
    if count > 1:
        x.append(n)
    return (len(x))

def FindDuplicates5(numbers):
    counter = 0
    if len(numbers) < 2:
        return counter
    else:
        numbers.sort()
        dup = 0
        for i in range(1,len(numbers)):
            if ((numbers[i-1] == numbers[i]) & (dup == 0)):
                counter = counter + 1
                dup = 1
            elif numbers[i-1] != numbers[i]:
                dup = 0
    # exit function and return number of unique duplicates
    #
    return counter

def FindDuplicates6(numbers):
    allDupes = [x for x in numbers if numbers.count(x) >= 2]
    uniqueDupes = list(set(allDupes))
    numberOfDupes = len(uniqueDupes)
    return numberOfDupes


# begin gracefully
#
if __name__ == "__main__":

    numbers = []

#chage the value inside of range for each run
    for i in range(100):
        value = random.randint(1, 4) #change the random int range for each run by a decimal place
        numbers.append(value)

    start = timer()
    FindDuplicates1(numbers)
    end = timer()
    print "FindDuplicates1", (end - start)

    start = timer()
    FindDuplicates2(numbers)
    end = timer()
    print "FindDuplicates2", (end - start)

    start = timer()
    FindDuplicates3(numbers)
    end = timer()
    print "FindDuplicates3", (end - start)

    start = timer()
    FindDuplicates4(numbers)
    end = timer()
    print "FindDuplicates4", (end - start)

    start = timer()
    FindDuplicates5(numbers)
    end = timer()
    print "FindDuplicates5", (end - start)

    start = timer()
    FindDuplicates6(numbers)
    end = timer()
    print "FindDuplicates6", (end - start)
```

Figure 7. Script to Approximate Run Time

```
#!/usr/bin/env python
import random
import cProfile

from memory_profiler import profile

@profile
def FindDuplicates1(numbers):

	d = {}
	for val in numbers:
		d[val] = d.get(val, 0) + 1
	return sum(d[i] > 1 for i in d)

@profile
def FindDuplicates2(numbers):
	dupVals = []
	for i in range(0, len(numbers)):
		for j in range(i+1, len(numbers)):
			if numbers[j] == numbers[i] and numbers[j] not in dupVals:
			 dupVals.append(numbers[j])
	return len(dupVals)

@profile
def FindDuplicates3(numbers):
	temp = []
	foo = 0
	numberz = set(numbers)
	for num in numberz:
		temp.append(numbers.count(num))
	for num in temp:
		if num > 1:
			foo += 1
	return foo

@profile
def FindDuplicates4(numbers):
	x=[]
	for n in set(numbers):
		count = numbers.count(n)
	if count > 1:
		x.append(n)
	return (len(x))

@profile
def FindDuplicates5(numbers):
	counter = 0
	if len(numbers) < 2:
		return counter
	else:
		numbers.sort()
		dup = 0
		for i in range(1,len(numbers)):
			if ((numbers[i-1] == numbers[i]) & (dup == 0)):
				counter = counter + 1
				dup = 1
			elif numbers[i-1] != numbers[i]:
				dup = 0
	# exit function and return number of unique duplicates
	#
	return counter

@profile
def FindDuplicates6(numbers):
	allDupes = [x for x in numbers if numbers.count(x) >= 2]
	uniqueDupes = list(set(allDupes))
	numberOfDupes = len(uniqueDupes)
	return numberOfDupes




# begin gracefully
#
if __name__ == "__main__":

	numbers = []

	for i in range(100):
		value = random.randint(1, 4)
		numbers.append(value)

	FindDuplicates1(numbers)
	FindDuplicates2(numbers)
	FindDuplicates3(numbers)
	FindDuplicates4(numbers)
	FindDuplicates5(numbers)
	FindDuplicates6(numbers)
#
# end of file
```

Figure 8. Script to Approximate Memory

```
#!/bin/bash
for i in {1..10}; do
echo `date` >> 10^2.txt
python python_loop_100.py >> 10^2.txt;
echo `date` >> 10^2.txt
done
cat 10^2.txt | mail -s "10^2.txt is done!!!" tug96858@temple.edu
cat 10^2.txt | mail -s "10^2.txt is done!!!" tuh42003@temple.edu
```

Figure 9. Command Line Code for Collecting the 10 Samples

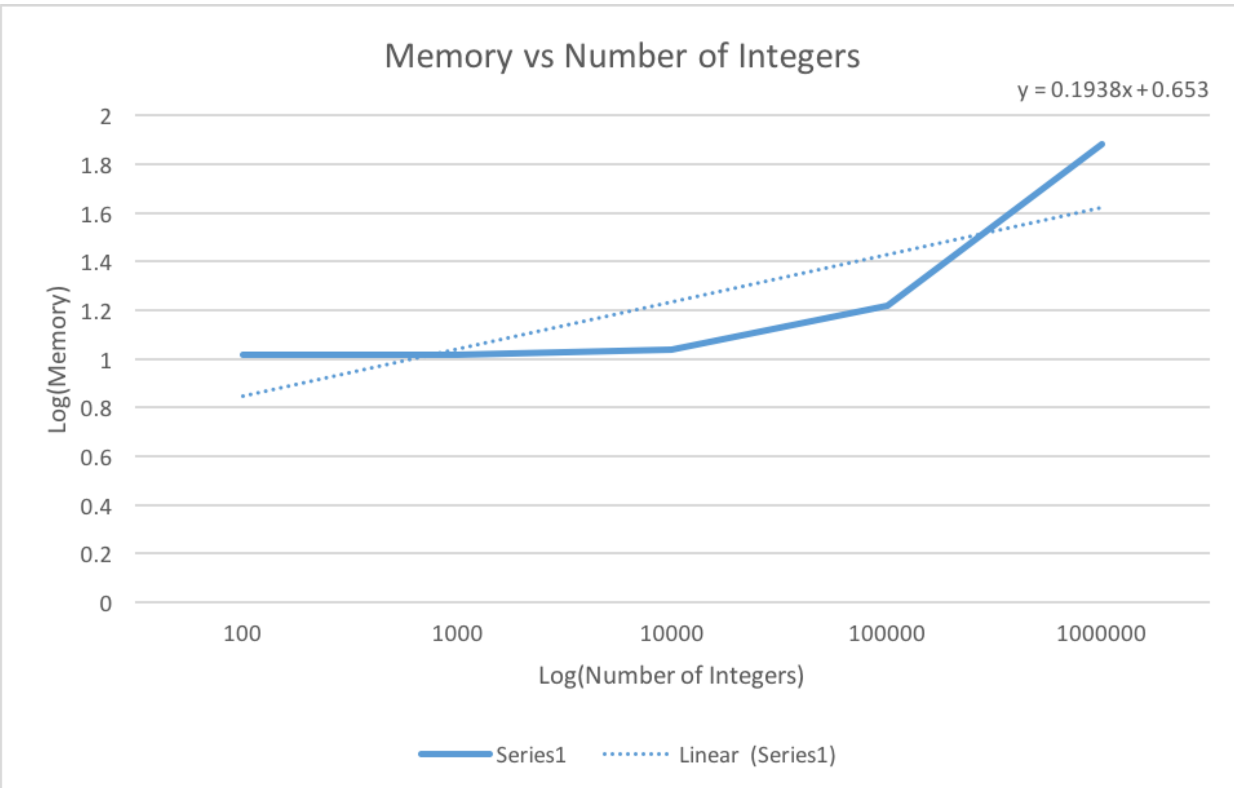

Figure 10. Algorithm Memory Response versus Amount of Random Numbers Sorted

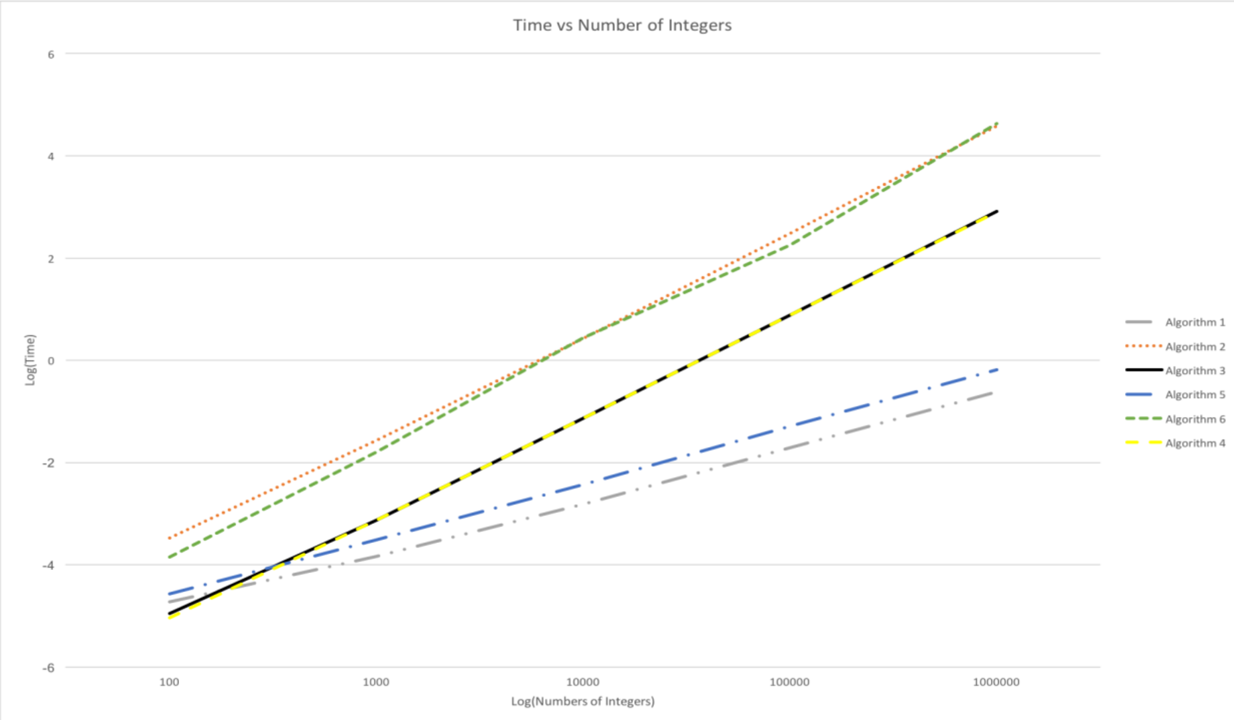

Figure 11. Algorithm Time Response versus Amount of Random Numbers Sorted

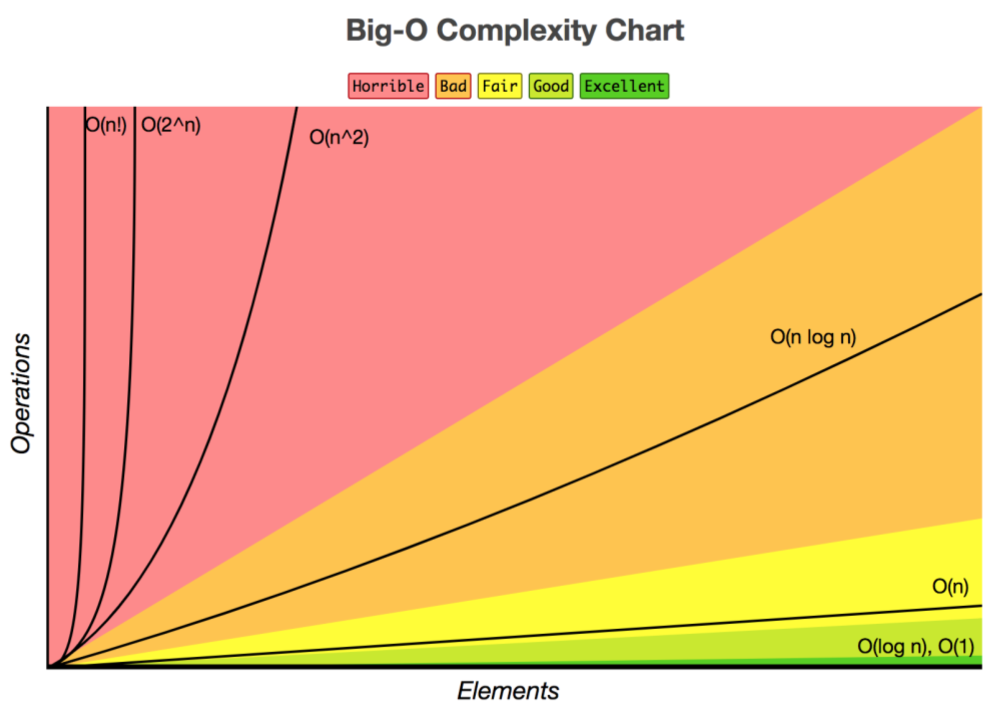

Figure 12. Algorithm complexity chart