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.

PROCEDURE

```
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 Using Get Function and Dictionary

```
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

Script to Approximate Run Time	Script to Approximate Memory
#!/usr/bin/env python	#!/usr/bin/env python
import random	import random
from timeit import default_timer as timer	import cProfile
def FindDuplicates1(numbers):	from memory_profiler import profile
d = {}	@profile
for val in numbers:	def FindDuplicates1(numbers):
d[val] = d.get(val, 0) + 1	
return sum(d[i] > 1 for i in d)	d = {}
	for val in numbers:
def FindDuplicates2(numbers):	d[val] = d.get(val, 0) + 1
dupVals = []	return sum(d[i] > 1 for i in d)
for i in range(0, len(numbers)):	
for j in range(i+1, len(numbers)):	@profile
if numbers[j] == numbers[i] and	def FindDuplicates2(numbers):
numbers[j] not in dupVals:	dupVals = []
dupVals.append(numbers[j])	for i in range(0, len(numbers)):
return len(dupVals)	for j in range(i+1,
	len(numbers)):
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 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
	```

```
#
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)
```

```
        # 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
```

Table 1. Scripts for Timing and Memory Calculations

ANALYSIS

Algorithm Memory Usage in MiB					
Size:	**10^2**	**10^3**	**10^4**	**10^5**	**10^6**
Algorithm:					
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	75.50
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 2. Algorithm Memory Usage

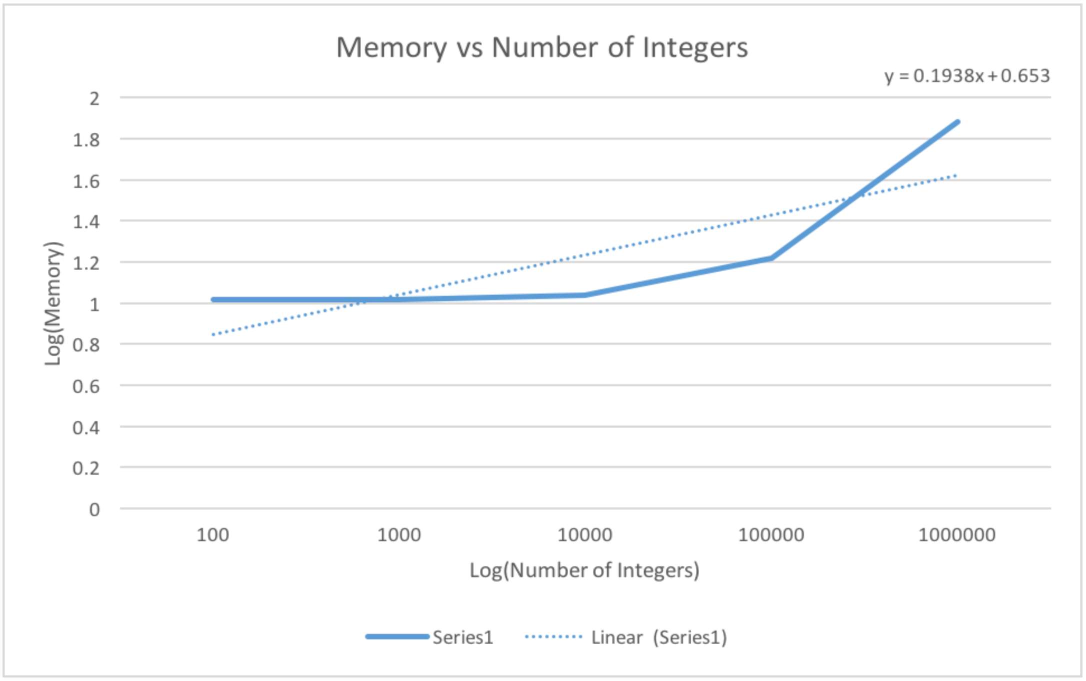

Figure 7. Algorithm Memory Response versus Amount of Random Numbers Sorted

As you can see from Table 2 above, the memory usage for each algorithm remains roughly the same. For this reason, we have plotted the average memory usage between the Algorithms. Because of the very small difference in these algorithms if they were to all be plotted together it would be six lines close to on top of each other. Another very noteworthy thing to take from this data is that there is no correlation speed and memory usage. Normally he expectation would be that the faster you go, the more memory you consume. This is not the case for our project. If you refer to the table, you will see that the fastest algorithm number one remained in the median for memory consumption. Out profiling method for memory gave a output in the exact format shown in Figure 8 on the next page. It is our interpretation of this report that the reason that all the algorithms have approximately the same amount of memory because none of them allocate memory, they simply manipulate existing data.

```
Filename: solution1.py

Line #    Mem usage    Increment   Line Contents
================================================
     7     10.1 MiB     10.1 MiB   @profile
     8                             def FindDuplicates(numbers):
     9
    10     10.1 MiB      0.0 MiB       d = {}
    11     10.1 MiB      0.0 MiB       for val in numbers:
    12     10.1 MiB      0.0 MiB           d[val] = d.get(val, 0) + 1
    13     10.1 MiB      0.0 MiB       return sum(d[i] > 1 for i in d)
mprof: Sampling memory every 0.1s
```

Figure 8. Memory Analysis for a Single Run

Algorithm CPU Time in Seconds					
Size:	**10^2**	**10^3**	**10^4**	**10^5**	**10^6**
Algorithm:					
1	.0000188	.000144	0.00154	0.0196	0.243
2	.0003310	.027500	2.71000	304.0000	38200.000
3	.0000113	.000748	0.07350	7.7400	814.000
4	.0000092	.000744	0.07340	7.7500	807.000
5	.0000272	.000313	0.00368	0.0520	0.646
6	.0001400	.015900	2.74000	181.0000	42900.000

Table 3. Algorithm CPU Time

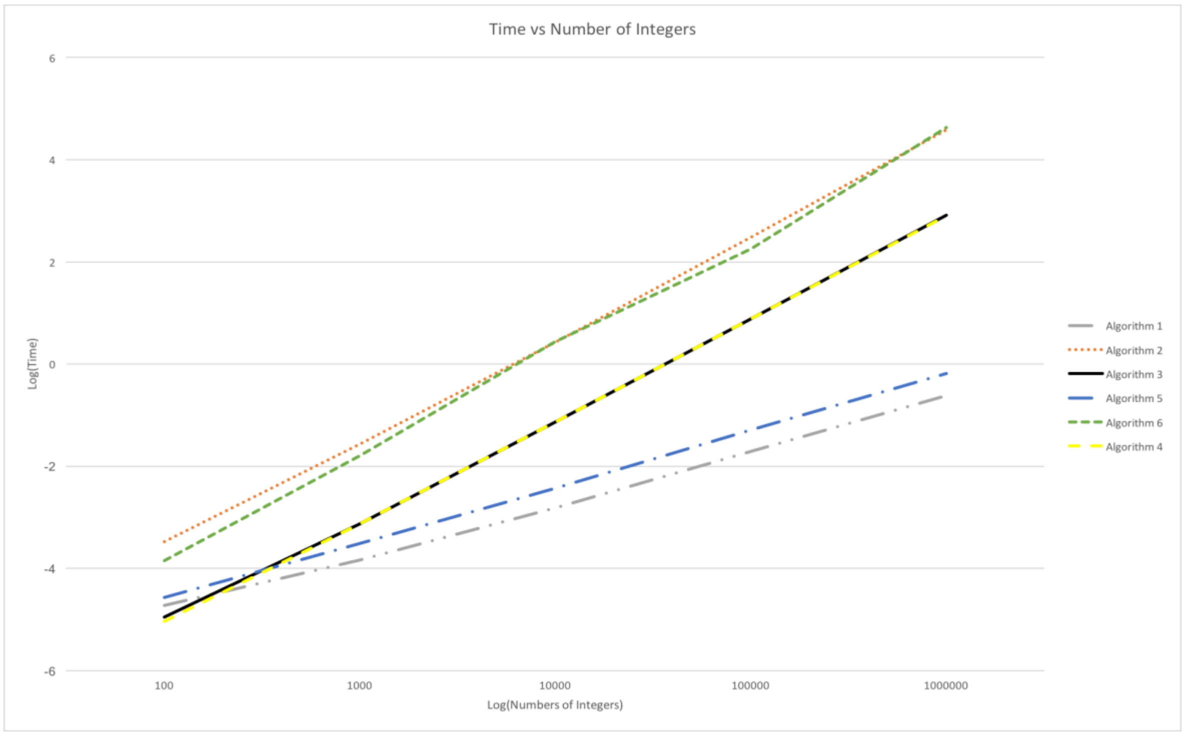

Figure 9. Algorithm Time Response versus Amount of Random Numbers Sorted

The data to support the claim that Algorithm number on is the most efficient is shown above. By analyzing the values shown in Table 2 we have concluded that the
Algorithms have the Complexity shown in Table 3. This was based on the understanding that the more functions that are preformed, the longer the script takes to complete. By inspection it is obvious that algorithm 1 only increases in run time by an approximate power of as we increase the input size. This leads us to believe that the complexity is a O(log(n)). Although the fifth algorithm is a touch slower than the first, it also follows this trend making it the same complexity. Given the exponential increase in run time of the third and fourth algorithm, we believe those to be at a minimum O(n^2). As we approach the least efficient algorithms we see that they have a greater exponential trend in increased run time while also sharing a trend. This would make them at least an O(2^n) complexity and possibly even an O(n!). Our results are summarized in Table 4. A sample of what the output of the runtime approximation script is shown on the next page in Figure 8.

```
FindDuplicates1 0.000152111053467
FindDuplicates2 0.0268650054932
FindDuplicates3 0.000741004943848
FindDuplicates4 0.000731945037842
FindDuplicates5 0.000306844711304
FindDuplicates6 0.0156071186066
```

Figure 10. Time Analysis for a Single Run

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

Table 4. Complexity

Working our way backwards in the algorithms to better understand why one algorithm is more efficient than another we use a function to report back the number of function calls. We noticed that this does in fact have a direct correlation to runtime. The number of function calls grew exponentially in the second and sixth algorithms. The second one grows like this because of the use of nested for loops. Python is like MATLAB in the sense that For loops make the script very slow. Although this method may be more intuitive to most it is shown to be very inefficient. The sixth seems to be inefficient because it is nesting functions inside of functions. The third algorithm is also using a series of nested for loops but much more efficiently, this is the reason for it ranking slower than one but faster than two. Although four uses one less iteration we can see that the it is still less efficient. This is probably due to the indexing over the array to count each element. The two fastest algorithms use iterations but limit the function calls, we believe this to be the defining factor that makes them more efficient than the other methods.

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 **Error! Reference source not found.** we estimated the memory usage to be the same or very close to the other algorithms.