Data Structures (CS 1520) Lab 2 Name:_____

Objective: To get a feel for big-oh notation by analyzing algorithms as well as timing them (Part A), and gain some experience writing Python classes.

Informal Big-oh (and Big-Theta) Definition: As the size of a computational problem grows (i.e., more data), we expect our program to run longer, but this run-time growth is not necessarily linear. Big-oh notation gives us an idea how our program's run-time will grow with respect to its problem size on larger data.

This might seem like a lot of mathematical mumbo-jumbo, but knowing an algorithms big-oh notation can help us predict its run-time on large problem sizes. While running a large size problem, we might want to know if we have time for a quick lunch, a long lunch, a long nap, go home for the day, take a week of vacation, pack-up the desk because the boss will fire you for a slow algorithm, etc.

For example, consider the following algorithm:

```
result = 0
for r in range(n):                          ← loops n times
    for c in range(n):                          ← executes a total of n*n times
        for d in range(n//2):                       ← executes a total of n*n*n/2 times
            result = result + d                         ← executes a total of n*n*n/2 times
        # end for
    # end for
# end for
```

Clearly, the body of the inner-most loop (the "`result = result + d`" statement) will execute $n^3/2$ times, so this algorithm is "big-oh" of n-cubed, $O(n^3)$. Thus, the execution-time formula with-respect-to n is:

$$T(n) = c\,n^3 + \text{(slower growing terms)}.$$

For large values of n, $T(n) \approx c\,n^3$, where c is the *constant of proportionality* on the fastest growing term (the machine dependent time related to how long it takes to execute the inner-most loop once) and T(n) represents execution time as a function of n. If we know that T(10,000) = 1 second = c x $10{,}000^3$, then we can predict what T(1,000,000). First approximate c as $c \approx T(n) / n^3 = 1 \text{ second} / 10{,}000^3 = 1 \text{ second} / 10^{12} = 10^{-12}$ seconds. Since we are running the algorithm on the same machine, c is unchanged for the larger problem. Thus, $T(1{,}000{,}000) \approx c\ 1{,}000{,}000^3 = c\ 10^{18} = 10^{-12}$ seconds * $10^{18} = 10^6$ seconds or about 11.6 days. (A couple weeks of vacation is appropriate!)

To start the lab: Download and unzip the file lab2.zip from

http://www.cs.uni.edu/~fienup/cs1520s13/labs/lab2.zip

Part A: In the folder lab2, open the timeStuff.py program in IDLE. (Right-click on timeStuff.py | Edit with IDLE) **Start it running** in IDLE by selecting Run | Run Module from the menu. **While it is running**, answer the following questions about each of the algorithms in timeStuff.py.

a) What is the big-oh of Algorithm 0?

```
Algorithm 0:
        result = 0
        for i in range(10000000):
            result = result + i
```

b) What is the big-oh of Algorithm 1?

```
Algorithm 1:
        result = 0
        for i in range(n):
            result = result + i
        # end for
```

c) What is the big-oh of Algorithm 2?

Algorithm 2:

```
result = 0
for r in range(n):
    c = n
    while c > 1:
        result = result + c
        c = c // 2
    # end while
# end for
```

d) What is the big-oh of Algorithm 3?

Algorithm 3:

```
result = 0
for r in range(n):
    for c in range(n):
        result = result + c
    # end for
# end for
```

e) What is the big-oh of Algorithm 4?

Algorithm 4:

```
result = 0
for r in range(n):
    for c in range(n):
        for d in range(n*n*n):
            result = result + d
        # end for
    # end for
# end for
```

f) What is the big-oh of Algorithm 5?

Algorithm 5:

```
result = 0
i = 0
while i < 2**n:
    result = result + i
    i += 1
# end while
```

g) Complete the following timing table from the output of timeStuff.py.

Algorithm	Execution Time in Seconds					
	n = 0	**n = 10**	**n = 20**	**n = 30**	**n = 40**	**n = 50**
Algorithm 0						
Algorithm 1						
Algorithm 2						
Algorithm 3						
Algorithm 4						
Algorithm 5				(work on h & i while waiting)		

h) For Algorithm 5, use the timing for n = 20 to compute the *constant of proportionality* on the fastest growing term.

i) Using the constant of proportionality computed in (h), predict the run-time of Algorithm 5 for n = 30.

j) How does your prediction in (i) compare to the actual time from (g)?

After you have answered the above questions, raise your hand and explain your answers.

Part B: The lab2.zip file also contains:
- a simple Die class (in the simple_die.py module) for a six-sided die, and
- an AdvancedDie class (in the module advanced_die.py module) for a die which can be constructed with any number of sides. The AdvancedDie class inherits from the Die class.

a) Write a new program that used AdvancedDie class ("`from simple_die import AdvancedDie`") to:
- create two 10-sided AdvancedDie objects
- roll the pair of dice 1000 times
- determine and print the average outcome (i.e., average total on the pair of dice) on the pair of dice during the 1000 rolls.

b) For testing certain dice games, suppose we want to extend the AdvancedDie class to include a new method `setRoll` which takes as a parameter a roll value that is used to set a die's roll to a specified value. We might invoke this method as:

```
myDie.setRoll(3)   # sets myDie to a current roll of 3
```

Implement a new subclass MoreAdvancedDie (in a new file `more_advanced_die.py`) which inherits from the AdvancedDie class, and includes the new `setRoll` method. When implementing the MoreAdvancedDie class: (**see the AdvancedDie class and its methods, especially the `__init__` method as an example**)

- Include documentation with the MoreAdvancedDie class (right below it's `class` line)
- Include documentation with the `setRoll` method including preconditions and postconditions
- Enforce preconditions by raising appropriate exceptions.

c) View the programmer-authored documentation for the MoreAdvancedDie class by typing `help(MoreAdvanceDie)` at the IDLE shell prompt ("**>>>**") after selecting Run | Run Module in the file `more_advanced_die.py` which contains the MoreAdvancedDie class.

After you have implemented AND fully tested your MoreAdvancedDie class, raise you hand and demonstrate it.

If you complete all parts of the lab, nothing needs to be turned in for this lab. If you do not get done today, then show me the completed lab in next week's lab period. When done, remember to log off and take your USB drive.

If you have extra time, this would be a good chance to work on Homework #1!