**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 n3 /2 times, so this algorithm is “big-oh” of n-cubed, *O*( n3 ). Thus, the ***execution-time formula***, T(n), with-respect-to n is:

T(n) = c n3 + (slower growing terms).

For large values of n, the execution time as a function of n, T(n) c n3, 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). If we know that T(10,000) = 1 second = c x 10,0003 , then we can calculate c and use it to predict what T(1,000,000). First approximate c as c T(n) / n3 = 1 second / 10,0003 = 1 second / 1012 = 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) c 1,000,0003 = c 1018 = 10-12 seconds \* 1018 = 106 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 eLearning.

# **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? O(1)

Algorithm 0:

result = 0

for i in range(10000000):

result = result + i

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

Algorithm 1:

result = 0

for i in range(n):

result = result + i

# end for

c) What is the big-oh of Algorithm 2? O(n\*log2n)

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? O(n2)

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? O(n5)

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? O(2n)

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 | 0.390917 | 0.378491 | 0.304343 | 0.280271 | 0.291648 | 0.292715 |
| Algorithm 1 | 0.000001 | 0.000002 | 0.000002 | 0.000002 | 0.000001 | 0.000001 |
| Algorithm 2 | 0.000000 | 0.000003 | 0.000004 | 0.000005 | 0.000007 | 0.000009 |
| Algorithm 3 | 0.000000 | 0.000003 | 0.000010 | 0.000020 | 0.000035 | 0.000053 |
| Algorithm 4 | 0.000000 | 0.002275 | 0.086456 | 0.717325 | 2.832067 | 9.100464 |
| Algorithm 5 | 0.000002 | 0.000059 | 0.079394 | 102.365534 | These take too long! | |

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

T(n)=c\*2n

T(20)=c\*220=0.079394

c=0.079394/220

c=7.57160187\*10-8 seconds

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

T(30)=(7.57160187\*10-8)\*230

T(30)=81.299456 seconds

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

Prediction is slightly lower, presumably because of background processes on the device

**(NOTE: Part B is on the backside of this sheet)**

# **Part B**: The lab2.zip file also contains:

1. A simple Die class (in the simple\_die.py module) for a six-sided die.
2. 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.
3. An averageOutcome.py program that computes the average outcome (i.e., average total on the pair of dice) on a pair of 10-side dice over 1,000 rolls. Unfortunately, it uses randint(1,10)+randint(1,10).

a) Modify the averageOutcome.py program so that it uses the AdvancedDie class by:

1. creating two 10-sided AdvancedDie objects (remember to “from advanced\_die import AdvancedDie”)
2. rolls the pair of dice 1,000 times to compute the average outcome

(Note: most of the program will remain unchanged. **See Python Summary Page 11 in the course packet (or** lab2/pythonSummary.pdf**) to see examples of the AdvancedDie class usage**)

# **Part C**: (NOTE: Part C does not need to use averageOutcome.py program)

**a) Implement a new subclass MoreAdvancedDie** **in a new file more\_advanced\_die.py** which inherits from the AdvancedDie class, and **includes only** a new setRoll method. The setRoll method can be used for testing certain dice games, because the new method setRoll takes as a parameter a roll value that is used to set a die’s current roll to a specified value. We can test the new **MoreAdvancedDie** class after loading its definition in the Python shell by using the Run | Run Module IDLE menu selection:

>>> myDie = MoreAdvancedDie(10)# create a 10 sided MoreAdvancedDie object

>>> print(myDie) # print the current roll

>>> myDie.setRoll(3) # sets myDie’s current roll to 3

>>> print(myDie) # print the new current roll which should be 3

When implementing the MoreAdvancedDie class you must:

1. Include documentation with the MoreAdvancedDie class (comments right below it’s class line)
2. Include documentation with the setRoll method including preconditions and postconditions
3. Enforce the setRoll method’s preconditions by raising appropriate exceptions.

(**see the AdvancedDie class’s \_\_init\_\_ method as an example**s of raising exceptions)

We can test that exceptions are raised by setRoll method after loading the **MoreAdvancedDie** class into the Python shell by:

>>> myDie = MoreAdvancedDie(10)# create a 10 sided MoreAdvancedDie object

>>> myDie.setRoll(3) # sets myDie’s current roll to 3

>>> myDie.setRoll(0) # should raise a ValueError

>>> myDie.setRoll(11) # should raise a ValueError

>>> myDie.setRoll(3.14) # should raise a TypeError

>>> myDie.setRoll(“six”) # should raise a TypeError

b) View your 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 working code, zip the lab1 folder and submit it on eLearning.**

**If you do not get done today, then submit it by next week’s lab period. Today save your lab 2 files (USB drive, etc.) and remember to log off.**

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