Data Structures (810:052) Lab 1 Name:_____

Objective: To get a feel for big-oh notation by analyzing algorithms as well as timing them.

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 xrange(n):
    for c in xrange(n):
        for d in xrange(n/2):
            result = result + d
        # 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). If we know that T(10,000) = 1 second, then we can predict what T(1,000,000). First approximate c as $c \approx T(n) / n^3$ = 1 second / $10{,}000^3$ = 1 second / 10^{12} = 10^{-12} seconds. Since we are running the algorithm on the same machine c is unchanged for the larger problem, so $T(1{,}000{,}000) \approx c\ 1{,}000{,}000^3 = c\ 10^{18} = 10^{-12}$ 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 lab1.zip from
http://www.cs.uni.edu/~fienup/cs052s11/labs/lab1.zip

Part A: In the folder lab1\partA, 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 xrange(10000000):
            result = result + i
```

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

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

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

Algorithm 2:

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

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

Algorithm 4:

```
result = 0
for r in xrange(n):
    for c in xrange(n):
        for d in xrange(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 = 0L
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.

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.

EXTRA CREDIT -- Part B: In the folder lab1\partB run the timeSearches.py program which takes a minute or two. While its executing, study the code. Obverse that it creates a list, evenList, that holds 10,000 sorted, even values (e.g., evenList = [0, 2, 4, 6, 8, ..., 19996, 19998]). It then times several linear searching algorithms repeatedly by searching for target values from 0, 1, 2, 3, 4, ..., 19998, 19999 so half of the searches are successful and half are unsuccessful. The search algorithms are:

- linearSearch (imported from linearSearch.py) that performs an iterative (uses a loop) sequential search assuming an **unsorted list**.
- linearSearchOfSortedListA that performs an iterative sequential search on a **sorted list in ascending order**, so it can stop when it sees a list item bigger than the target. It uses a for-loop with an if-statement to break out of the loop when the outcome of the search can be determined.
- linearSearchOfSortedListB that performs an iterative sequential search on a **sorted list** so it can stop when it sees a list item bigger than the target. Uses a for-loop and returns from inside the loop if the outcome of the search can be determined.
- linearSearchOfSortedListC that performs an iterative sequential search on a **sorted list in ascending order**, so it can stop when it sees a list item bigger than the target. Uses a while-loop with an if-elif-else statement.
- linearSearchRecursiveA that performs a recursive sequential search on an **unsorted list**. Recursive calls pass a slice of the list with the last item removed.
- linearSearchRecursiveB that performs a recursive sequential search on an **unsorted list**. Recursive calls pass the original list and the index of the end item to be considered in the search.

	linearSearch	linearSearchOfSortedList			linearSearchRecursive	
		A	B	C	A	B
Time in seconds						

Answer the following questions about the search algorithms:

a) What is the big-oh notation for a single sequential search? (Let "n" be the number of items in the list)

b) Why is the linearSearch of the evenList slower than the linearSearchOfSortedListA of the evenList?

c) Why is the linearSearchOfSortedListB of the evenList slower than the linearSearchOfSortedListA of the evenList?

d) Why is linearSearchOfSortedListC slower than linearSearchOfSortedListA, but faster than linearSearchOfSortedListB?

e) Why is linearSearchRecursiveB slower than linearSearch?

f) Why is linearSearchRecursiveA so much slower than linearSearchRecursiveB?

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