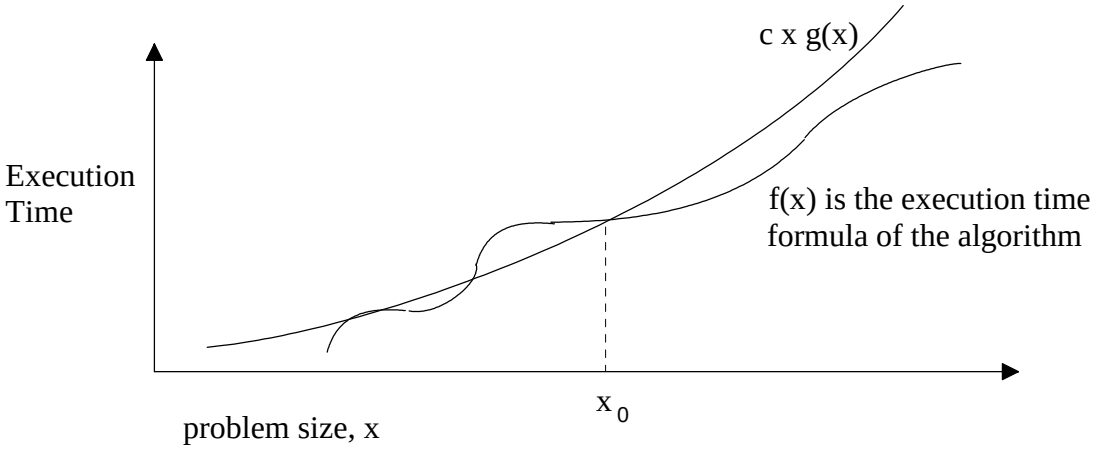
Data Structures (810:052) Lab 1 Name:_____

Objective: To experiment with searching and sorting to get a feel for big-oh notation.

The Assignment Overview

Big-oh notation gives an asymptotic upper bound on execution time within a constant factor.

Mathematical Big-oh Definition: A function $f(x)$ is "Big-oh of $g(x)$" or "$f(x)$ is $O(g(x))$" or "$f(x) = O(g(x))$" if there are positive, real constants c and x_0 such that for all values of $x \geq x_0$, $f(x) \leq c \times g(x)$.



Suppose that $T(n) = c_1 + c_2 n = 100 + 10 n$ which I claim is $O(n)$.

"Proof": Pick $c = 110$ and $x_0 = 1$.
Then $100 + 10 n \leq 110 n$ for all $n \geq 1$ since
$100 + 10 n \leq 110 n$
$100 \leq 100 n$
$1 \leq n$ which is CLEARLY true for all $n \geq 1$.

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:

```
for r in xrange(n):
    for c in xrange(n):
        for d in xrange(n/2):
            result = result ^ d  # bit-wise XOR
        # 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:
$T(n) = c n^3 +$ (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 inorder!)

To start the lab: Download and unzip the file at: www.cs.uni.edu/~fienup/cs052sum09/labs/lab1.zip

Activity 0: In the folder lab1\ACTIVITY_0\, print the timeStuff.py program. Start it running. 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?

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

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

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

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

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

g. Complete the following timing table.

	Execution Time in Seconds					
Algorithm	**n = 0**	**n = 10**	**n = 20**	**n = 30**	**n = 40**	**n = 50**
Algorithm 0						
Algorithm 1						
Algorithm 2						
Algorithm 3						
Algorithm 4						
Algorithm 5						

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)?

Activity 1: In the folder lab1\ACTIVITY_0\ 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 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:

- orderedSequentialSearch (imported from orderedSequentialSearch.py) that performs an iterative (uses loops) sequential search on a sorted list, so it can stop when it sees a value bigger than the target. It returns a Boolean result indicating whether or not a specified target was found
- binarySearch (imported from binarySearchRecursive.py) that performs a recursive binary search algorithm and returns a Boolean result indicating whether or not a specified target was found
- binarySearch (imported from binarySearchIterative.py) that performs an iterative (uses loops) binary search algorithm and returns a Boolean result indicating whether or not a specified target was found
- binarySearch (imported from binarySearchRecursiveLocation.py) that performs a recursive binary search algorithm and returns an index location indicating where a specified target was found; unsuccessful searches return -1
- binarySearch (imported from binarySearchIterativeLocation.py) that performs an iterative (uses loops) binary search algorithm and returns an index location indicating where a specified target was found; unsuccessful searches return -1

Answer the following questions about the search algorithms:

a. What is the big-oh notation for a single sequential search?

a. What is the big-oh notation for a single binary search?

b. What is the big-oh notation for the timed section of code that calls sequential search?

c. What is the big-oh notation for the timed section of code that calls any of the binary searches?

d. Why does the recursive binarySearch imported from binarySearchRecursive.py run so much slower than the recursive binarySearch imported from binarySearchIterativeLocation.py?

Activity 2: In the folder lab1\ACTIVITY_2\, run the timeBinarySearch.py program that only times the binarySearch algorithm imported from binarySearchIterativeLocation.py. Currently, this program searches a list of 10,000 items.

a. How long does it take to search for target values from 0, 1, 2, 3, 4, ..., 19998, 19999?

b. Before running the program on 100,000 items in the list, let's use big-oh notation to predict how long it will take. The section of code being timed is $O(n \log_2 n)$ since we are looping 2n times and calling an $O(\log_2 n)$ algorithm each time. If we let T(n) be the actual execution-time formula for this code, then the definition of big-oh tells us that $T(n) = c * n \log_2 n +$ (slower growing terms), where c is some constant value that's machine dependent. For large values of n, the slower growing terms should be relatively small with respect to the $c * n \log_2 n$ term. Therefore, $T(n) \approx c * n \log_2 n$. Using your timing in part (a) where n = 10,000, calculate the value of c on your lab machine? (recall that $\log_2 x = (\log_{10} x / \log_{10} 2) = (\ln x / \ln 2) = (\log_b x / \log_b 2)$

c. Predict the execution time of the program on 1,000,000 items in the list.

d. Modify and run the timeBinarySearch.py program using a 1,000,000 item list. How long did it take?

e. Predict the execution time for 10,000,000 items the list.

f. How long did 10,000,000 items in the list take?