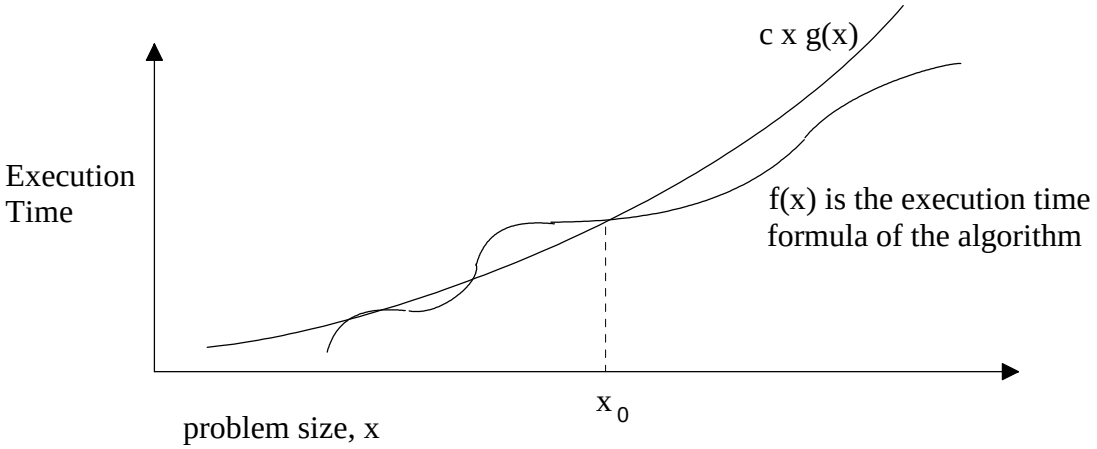
Objective: To experiment with searching and sorting to get a feel for big-oh notation.

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 weeks vacation, pack-up the desk because the boss will fire you for a slow algorithm, etc.

Activity 1: Copy the folder P:\810-063-CSIII\LABS\LAB_6\ACTIVITY_1\ and 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?

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

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

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

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

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

Activity 2: Copy the folder P:\810-063-CSIII\LABS\LAB_6\ACTIVITY_2\ and 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?

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

Activity 3: Copy the folder P:\810-063-CSIII\LABS\LAB_6\ACTIVITY_3\ and run the timeHashSearch.py program that times repeated searches of a hash table using linear probing. Currently, this program searches a hash table containing 10,000 items and has a load factor of 0.2.

a. How long does it take to search for target values from 0, 1, 2, 3, 4, ..., 19998, 19999 in this hash table containing 10,000 items?

b. Experiment with changing the load factor between 0.2 and 0.9 by completing the following table:

	Load Factor							
	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
Execution time with 10,000 items in hash table (seconds)								

c. Explain why the performance of the hash table with linear probing degrades so badly at high load factors.

d. Change the load factor back to 0.2 and experiment with performance as the number of items in the hash table grows by completing the following table:

	Items in the Hash Table			
	10,000	100,000	1,000,000	10,000,000
Execution time with with load factor 0.2 (seconds)				

e. What type of growth rate did you observe as the number of items grew?

f. Explain why you would expect this growth rate after studying the hashfunction method of the HashTable.

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

NOTE: If you complete all of the activities within the lab period, you do not need to hand anything it. However, if you need to finish activities outside of the lab period, then hand in written questions to activities not completed during the lab period.