3. **General "Algorithmic-Complexity Analysis" terminology:**

<u>problem</u> - question we seek an answer for, e.g., "What is the sum of all the items in a list/array?"

<u>parameters</u> - variables with unspecified values

<u>problem instance</u> - assignment of values to parameters, i.e., the specific input to the problem

	0	1	2	3	4	5	6
myList:	5	10	2	15	20	1	11

sum: ?

(number of elements) n: 7

<u>algorithm</u> - step-by-step procedure for producing a solution

<u>basic operation</u> - fundamental operation in the algorithm (i.e., operation done the most) Generally, we want to derive a function for the number of times that the basic operation is performed related to the *problem size.*

<u>problem size</u> - input size. For algorithms involving lists/arrays, the problem size is the number of elements ("n").

<u>Big-oh notation ($O(\,)$)</u> - As the size of a problem grows (i.e., more data), how will our program's run-time grow.

Consider the following sumList function.

```
def sumList(myList):
    """Returns the sum of all items in myList"""
    total = 0
    for item in myList:
        total = total + item
    return total
```

a) What is the basic operation of sumList (i.e., operation done the most) ? addition

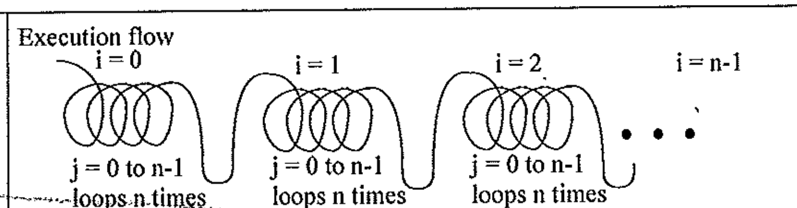
b) What is the problem size of sumList? length of myList, "n"

c) If n is 10000 and sumList takes 10 seconds, how long would you expect sumList to take for n of 20000?
20 sec since we sum two chunks of 10000

d) What is the big-oh notation for sumList? $O(n)$ "linear"

4. Consider the following someLoops function.

```
def someLoops(n):
    total = 0
    for i in range(n):
        for j in range(n):
            total = total + i + j
    return total
```



a) What is the basic operation of someLoops (i.e., operation done the most)? addition

b) How many times will the basic operation execute as a function of n? n^2

c) What is the big-oh notation for someLoops? $O(n^2)$ "quadratic"

d) If we input n of 10000 and someLoops takes 10 seconds, how long would you expect someLoops to take for n of 20000?

$T(n) = c_1 n^2 + \cancel{c_2 n} + \cancel{c_3 1}$ $T(n) \approx c_1 n^2$ $T(10000) = c\,10000^2 = 10\text{sec}$

exec. time function

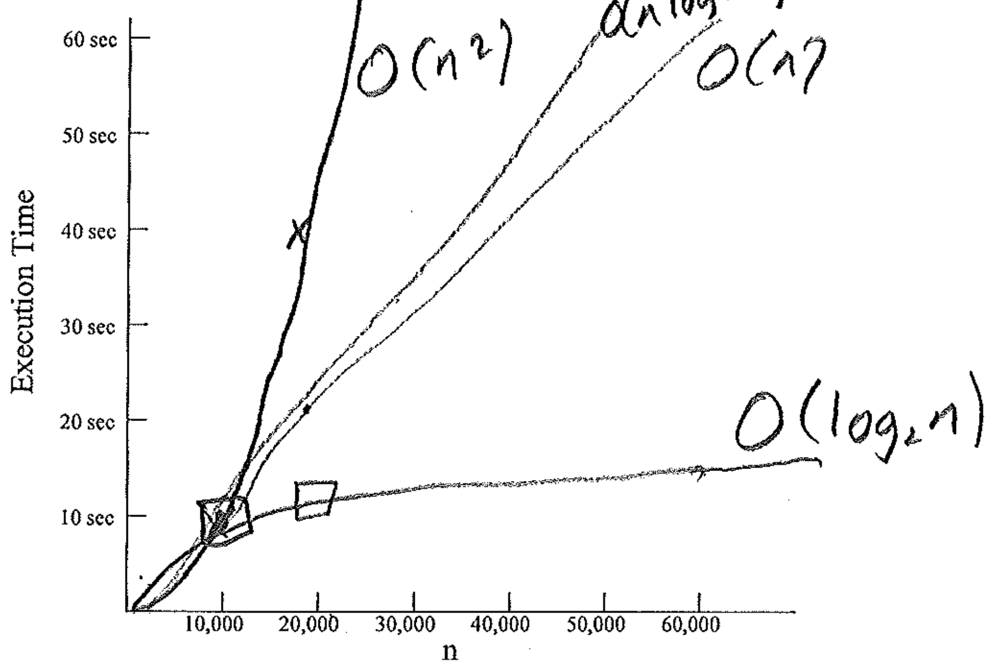
time to do inner-loop once

$c = \frac{10\text{sec}}{10000^2}$

$T(20000) = c\,20000^2 = \left(\frac{10\text{sec}}{10000^2}\right) \cdot 20000^2 = 40\text{sec}$

Data Structures (CS 1520) Lecture 3 Name:

1. Draw the graph for sumList ($O(n)$) and someLoops ($O(n^2)$) from the previous lecture.



2. Consider the following sumSomeListItems function. 8192 4096 2048 1024

```
import time

def main():
    n = eval(input("Enter size of list: "))
    aList = list(range(1, n+1))
    start = time.clock()
    sum = sumSomeListItems(aList)
    end = time.clock()
    print("Time to sum the list was %.9f seconds" % (end-start))

def sumSomeListItems(myList):
    """Returns the sum of some items in myList"""
    total = 0
    index = len(myList) - 1
    while index > 0:
        total = total + myList[index]
        index = index // 2
    return total

main()
```

32×16 $16 \times$ 2^{15} 2^{14} 2^{13} 2^{12} 2^{11} 2^{10}

$14.xx$ $2^{13.x}$

$1024 = 2^{10}$

index $= n, \frac{n}{2}, \frac{n}{4}, \frac{n}{8}, \frac{n}{16}, 8, 4, 2, 1, 0$

$2^{10}, 2^9, 2^8, 2^3, 2^2, 2^1, 2^0$

$\log_2 n = x$ iff $n = 2^x$

a) What is the problem size of sumSomeListItems? length mylist, "n"

b) If we input n of 10,000 and sumSomeListItems takes 10 seconds, how long would you expect sumSomeListItems to take for n of 20,000?

(Hint: For n of 20,000, how many more times would the loop execute than for n of 10,000?)

$O(\log_2 n)$ $T(n) = c \log_2 n$ $T(20000) = c \log_2 20000$

$T(10000) = c \log_2 10000 = 10 sec$ $= \frac{10 sec}{\log_2 10000} \log_2 20000$

$c = \frac{10 sec}{\log_2 10000}$

c) What is the big-oh notation for sumSomeListItems? $O(\log_2 n)$

$\frac{10 sec}{13.xx} 14.xx$

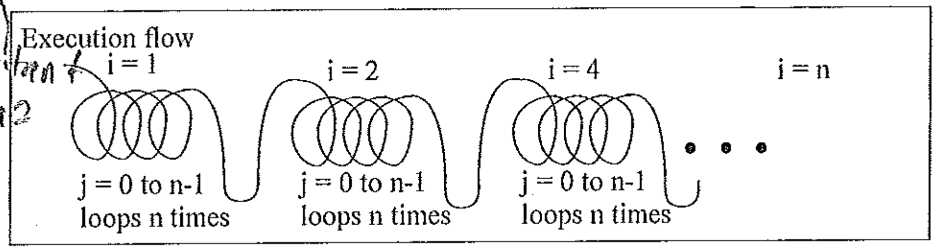
d) Add the execution-time graph for sumSomeListItems to the graph.

$\log_2 x = \frac{\log_{10} x}{\log_{10} 2}$

3.

```
i = 1
while i <= n:
    for j in range(n):
        # something of O(1)
    # end for
    i = i * 2
# end while
```

$\log_2 n$ $n\times$ constant time $O(1)$ time



a) Analyze the above algorithm to determine its big-oh notation, *O*( ).

$(\log_2 n) \times n = O(n \log_2 n)$

b) If n of 10,000, takes 10 seconds, how long would you expect the above code to take for n of 20,000?

slightly more than 20000

$T(n) = c\, n \log_2 n$

$T(10000) = c\, 10000 \log_2 10000 = 10\,sec$

$c = \frac{10\,sec}{10000 \log_2 10000}$

$T(20000) = c\, 20000 \log_2 20000$

$= \frac{10\,sec}{10000 \log_2 10000}\, 20000 \log_2 20000$

$= 10\,sec \times 2 \frac{14.x}{13.x}$

c) Add the execution-time graph for the above code to the graph.

4. Most programming languages have a built-in array data structure to store a collection of same-type items. Arrays are implemented in RAM memory as a contiguous block of memory locations. Consider an array X that contains the odd integers:

address	Memory	
4000	1	X[0]
4004	3	X[1]
4008	5	X[2]
4012	7	X[3]
4016	9	X[4]
4020	11	X[5]
4024	13	X[6]
⋮		

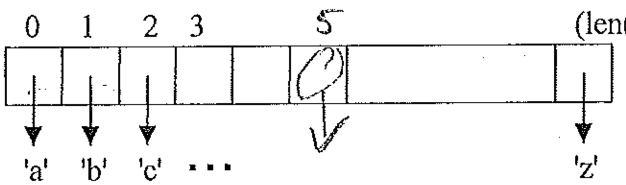
a) Any array element can be accessed randomly by calculating its address. For example, address of X[5] = 4000 + 5 * 4 = 4020. What is the general formula for calculating the address of the ith element in an array?

addr. X[i] = (start addr. of X) + i × (element size)

b) What is the big-oh notation for accessing the ith element?

$O(1)$ constant time

c) A Python list uses an array of references (pointers) to list items in their implementation of a list. For example, a list of strings containing the alphabet:



← references the same to any size item/object.

Since a Python list can contain heterogeneous data, how does storing references in the list aid implementation?

still allows for constant time access by index with heterogeneous data

5. Arrays in most HLLs are static in size (i.e., cannot grow at run-time), so arrays are constructed to hold the "maximum" number of items. For example, an array with 1,000 slots might only contain 3 items:

size: 3

	0	1	2	3					999
scores:	20	10	30						

a) The *physical size* of the array is the number of slots in the array. What is the physical size of scores? 1000

b) The *logical size* of the array is the number of items actually in the array. What is the logical size of scores? 3

c) The *load factor* is faction of the array being used. What is the load factor of scores? $\frac{3}{1000}$

d) What is the *O*( ) for "appending" a new score to the "right end" of the array? $O(1)$

e) What is the *O*( ) for adding a new score to the "left end" of the array? $O(n)$

f) What is the *average O*( ) for adding a new score to the array? $O(\frac{n}{2}) = O(n)$

g) During run-time if an array fills up and we want to add another item, the program can usually:
- Create a bigger array than the one that filled up
- Copy all the items from the old array to the bigger array
- Add the new item
- Delete the smaller array to free up its memory

When creating the bigger array, how much bigger than the old array should it be? double physical size

h) What is the *O*( ) of moving to a larger array? $O(n)$

6. Consider the following list methods in Python: mylist = n items otherList = m items

Method	Usage	Average *O*( ) for myList containing n items
index []	`itemValue = myList[i]`	$O(1)$
	`myList[i] = newValue`	$O(1)$
append	`myList.append(item)`	$O(1)$
extend	`myList.extend(otherList)`	$O(m)$
insert	`myList.insert(i, item)`	$O(\frac{n}{2}) = O(n)$
pop	`myList.pop( )`	$O(1)$
pop(i)	`myList.pop(i)`	$O(\frac{n}{2}) = O(n)$
del	`del myList[i]`	" "
remove	`myList.remove(item)`	$O(n)$
index	`myList.index(item)`	$O(n)$
iteration	`for item in myList:`	$O(n)$
reverse	`myList.reverse( )`	$O(n)$

Dictionary Operations:

Method	Usage	Explanation	Average *O*( ) for n keys
get item	`myDictionary.get(myKey)` `value = myDictionary[myKey]`	Returns the value associated with myKey; otherwise *None*	*O*(1)
set item	`myDictionary[myKey]=value`	Change or add myKey:value pair	*O*(1)
in	`myKey in myDictionary`	Returns True if myKey is in myDictionary; otherwise False	*O*(1)
del	`del myDictionary[myKey]`	Deletes the mykey:value pair	*O*(1)