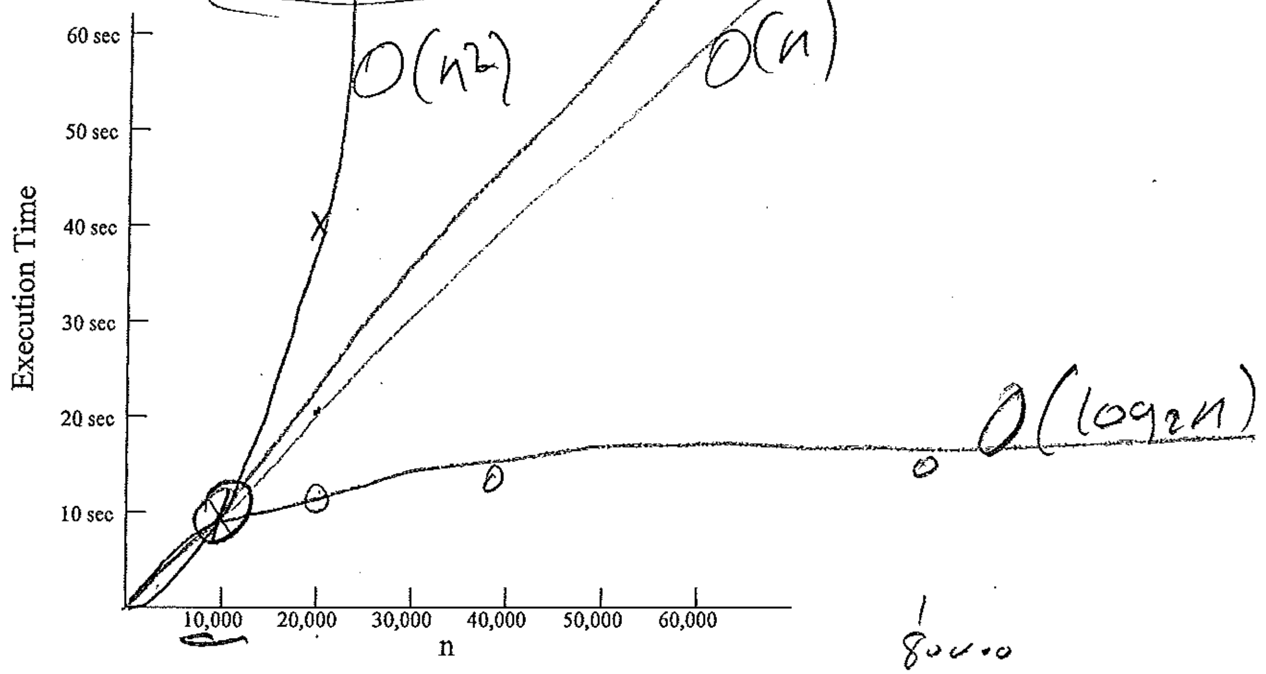
$10(n \log_2 n)$

Data Structures (CS 1520) Lecture 3 Name: Mark F.

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



2. Consider the following `sumSomeListItems` function.

```
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()
```

len(myList) 1 2 4 8 16 32 64

a) What is the problem size of `sumSomeListItems`? len(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?)

$O(\log_2 n)$

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

2^7 2^6 2^5 2^4 2^3 2^2 2^1 2^0

128 64 32 16 8 4 2 1

c) What is the big-oh notation for `sumSomeListItems`? n

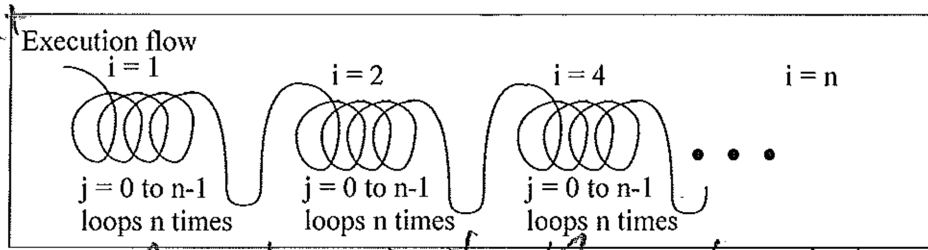
d) Add the execution-time graph for `sumSomeListItems` to the graph.

3.

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

log₂n

constant time



n + n + n problem size n + ... + n

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

$= n \times \log_2 n$

$O(n \log_2 n)$

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

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

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

$c = \frac{10\text{ sec}}{10000 \log_2 10000} = \frac{10\text{ sec}}{10000 \times 13.x}$

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

$\frac{10}{10000 \times 13.x} \times 20000 \times 14.x$

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:

32 bit integer

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]
⋮		

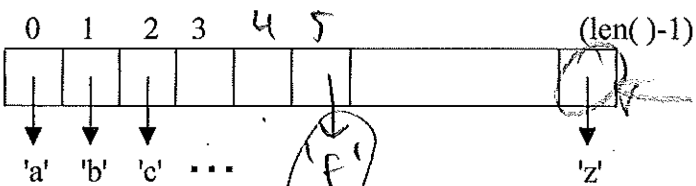
a) Any array element can be accessed randomly by calculating its address. For example, address of X[5] = 4000 + 5 * 4 = 4020. What is the general formula for calculating the address of the ith element in an array?

addr X[i] = (start of addr. of array) + i * (size of an item)

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

$O(1)$

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:



memory addr. of ~~string~~ items

addrs are all same size

$O(1)$

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

$O(\log_2 n)$ $T(n) = c \log_2 n + \cancel{X}$

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

$c = \dfrac{10 sec}{\log_2 10000} = \dfrac{10 sec}{13.x}$

$T(20000) = c \log_2 20000 = \dfrac{10 sec}{13.x} 14.x = 10.x sec$

$\log_2 \cancel{x} = \dfrac{\log_{10} x}{\log_{10} 2}$

$\log_2 10000 =$

$2^? = 10000 = 13.x$?

$20000 = 14.x$?

2^{14} 2^{13} 2^{12} 2^{11} 2^{10} 2^2 2^1 2^0

bi 8192 4096 2048 1024 512 256 128 64 32 16 8 4 2 1

L3 ①

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~~: 1M

size: ~~3~~ 4

	0	1	2	3	4			999
scores:	20	10	30	~~76~~				

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? 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 — double size of array
- 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?

h) What is the $O(\,)$ of moving to a larger array? $O(n)$ copy all n items to bigger array

6. Consider the following list methods in Python:

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)` m items	$O(m)$
insert	`myList.insert(i, item)`	$O(n/2)$ or $O(n)$
pop	`myList.pop( )`	$O(1)$
pop(i)	`myList.pop(i)`	$O(\frac{n}{2})$ or $O(n)$
del	`del myList[i]`	$O(n/2)$ or $O(n)$
remove	`myList.remove(item)`	$O(n)$
index	`myList.index(item)`	$O(n/2)$ or $O(n)$
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	$O(1)$