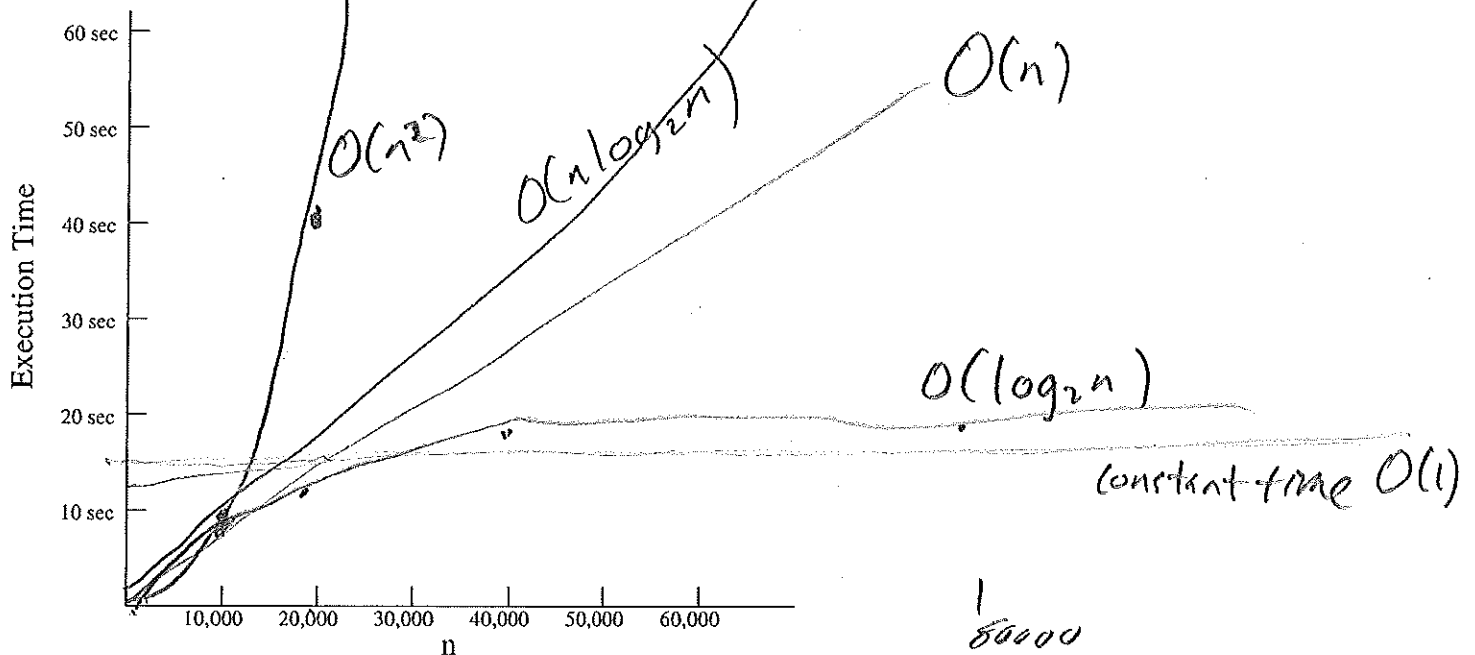


1. Draw the graph for sumList ($O(n)$) and someLoops ($O(n^2)$) from the previous lecture.



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
```

Handwritten notes on the code: $\log_2 n = x \text{ iff } 2^x = n$, $n = 128$, $\text{index} = 128, 64, 32, 16, 8, 4, 2, 1$, and a sequence of powers of 2: $2^7, 2^6, 2^5, 2^4, 2^3, 2^2, 2^1, 2^0$.

- a) What is the problem size of sumSomeListItems? $\text{len}(\text{myList}) = n$
- 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?)

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

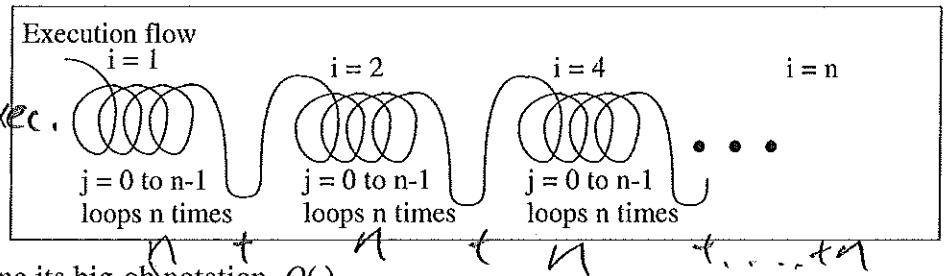
$$T(10000) = c \log_2 10000 = 10 \text{ sec} \quad = \frac{10 \text{ sec}}{13.x} \quad 14.x$$

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

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



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

$O(n \log_2 n)$

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

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

$\approx 20 \times$

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

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

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] |
| ⋮ | | |

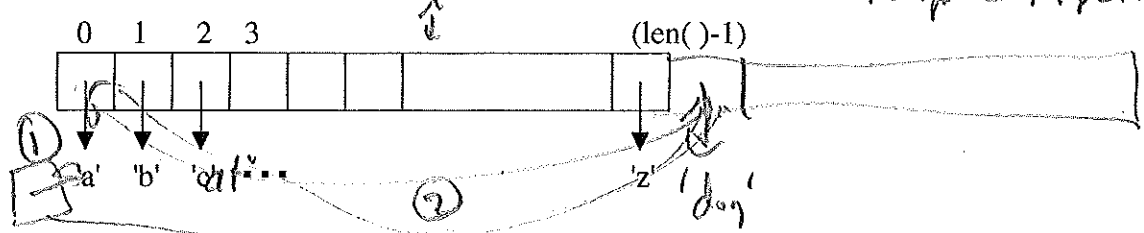
a) Any array element can be accessed randomly by calculating its address. For example, address of $X[5] = 4000 + 5 * 4 = 4020$. What is the general formula for calculating the address of the i th element in an array?

$$\text{addr. } X[i] = \left(\begin{matrix} \text{starting} \\ \text{addr.} \\ \text{of array} \end{matrix} \right) + \left(\begin{matrix} \text{index} \\ i \end{matrix} \right) * \left(\begin{matrix} \text{element} \\ \text{size} \end{matrix} \right)$$

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

$O(1)$

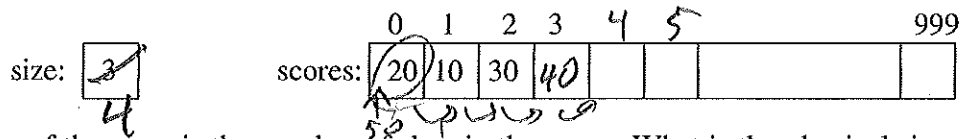
c) A Python list uses an array of references (pointers) to list items in their implementation of a list. For example, a list of strings containing the alphabet:



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

References are all the same size so formula in (a) still works even for 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:



- 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 fraction 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?
- e) What is the $O()$ for adding a new score to the “left end” of the array? $O(1)$
- f) What is the *average* $O()$ for adding a new score to the array? $O(\frac{n}{2}) \approx O(n)$
- 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? *twice the size of old array (2000 slots)*

- h) What is the $O()$ of moving to a larger array? $O(n)$
6. Consider the following list methods in Python: $len(myList) = 'n'$ $len(otherList) = 'm'$

| 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) | $O(\frac{n}{2}) \approx O(n)$ |
| pop | myList.pop() | $O(1)$ |
| pop(i) | myList.pop(i) | $O(n/2), O(n)$ |
| del | del myList[i] | $O(n/2), O(n)$ |
| remove | myList.remove(item) | $O(n)$ |
| index | myList.index(item) | $O(n/2), O(n)$ |
| 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 | $O(1)$ |