1. Draw the graph for \( \text{sumList} \ (O(n)) \) and \( \text{someLoops} \ (O(n^2)) \) from the previous lecture.

2. Consider the following \text{sumSomeListItems} function.

```python
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 {} of 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 - 1
    return total

main()
```

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

b) If we input \( n \) of 10,000 and \text{sumSomeListItems} takes 10 seconds, how long would you expect \text{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?)

\[ 0(\log n) \]
\[ \times = \log_2 n \iff 2^x = n \]

c) What is the big-oh notation for \text{sumSomeListItems}? \( n \)

d) Add the execution-time graph for \text{sumSomeListItems} to the graph.
3.  
   \[ i = 1 \]
   \[ \text{while } i \leq n: \]
   \[ \text{for } j \text{ in range}(n): \]
   \[ \quad \# \text{ something of } O(1) \]
   \[ \text{end for} \]
   \[ i = i \times 2 \]
   \[ \text{end while} \]
   
   a) Analyze the above algorithm to determine its big-on notation, \( T(n) \).
   
   \[ T(n) = C \cdot n \log_2 n \]

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

   \[ T(n) = C \cdot n \log_2 n \]
   
   \[ T(10000) = C \cdot 10000 \log_2 10000 = 10 \text{ sec} \]
   
   \[ C = \frac{10 \text{ sec}}{10000 \log_2 10000} \]

   \[ T(20000) = C \cdot 20000 \log_2 20000 \]

   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:

<table>
<thead>
<tr>
<th>address</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>4000</td>
<td>1</td>
</tr>
<tr>
<td>4004</td>
<td>3</td>
</tr>
<tr>
<td>4008</td>
<td>5</td>
</tr>
<tr>
<td>4012</td>
<td>7</td>
</tr>
<tr>
<td>4016</td>
<td>9</td>
</tr>
<tr>
<td>4020</td>
<td>11</td>
</tr>
<tr>
<td>4024</td>
<td>13</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

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

   \[ \text{address of } X[i] = \text{Start of array} + i \times \text{Size of an item} \]

   b) What is the big-on 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?
\[ T(n) = c \log_2 n + x \]

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

\[ c = \frac{10 \text{ sec}}{\log_2 10000} = \frac{10 \text{ sec}}{13.27} \]

\[ T(20000) = c \log_2 20000 = \frac{10 \text{ sec}}{13.27} \cdot 14.28 = 10.27 \text{ sec} \]

\[ \log_2 x = \log_{10} x \cdot \frac{1}{\log_{10} 2} \]

\[ \log_2 10000 = ? \]

\[ 2^4, 2^3, 2^5, 2^7 \]

\[ 2^0, 2^1, 2^2, 2^3, 2^4, 2^5, 2^6, 2^7, 2^8, 2^9, 2^{10} \]

\[ 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096 \]

[Image 0x0 to 613x792]
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:

![Array Example]

- **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? \(O(1)\)
- **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?

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

6. Consider the following list methods in Python:

<table>
<thead>
<tr>
<th>Method</th>
<th>Usage</th>
<th>Average (O()) for myList containing (n) items</th>
</tr>
</thead>
<tbody>
<tr>
<td>index</td>
<td>itemValue = myList[i]</td>
<td>(O(1))</td>
</tr>
<tr>
<td></td>
<td>myList[i] = newValue</td>
<td>(O(1))</td>
</tr>
<tr>
<td>append</td>
<td>myList.append(item)</td>
<td>(O(1))</td>
</tr>
<tr>
<td>extend</td>
<td>myList.extend(otherList)</td>
<td>(O(n))</td>
</tr>
<tr>
<td>insert</td>
<td>myList.insert(i, item)</td>
<td>(O(n)) or (O(n))</td>
</tr>
<tr>
<td>pop</td>
<td>myList.pop()</td>
<td>(O(n))</td>
</tr>
<tr>
<td>pop(i)</td>
<td>myList.pop(i)</td>
<td>(O(n)) or (O(n))</td>
</tr>
<tr>
<td>del</td>
<td>del myList[i]</td>
<td>(O(2n)) or (O(n))</td>
</tr>
<tr>
<td>remove</td>
<td>myList.remove(item)</td>
<td>(O(n))</td>
</tr>
<tr>
<td>index</td>
<td>myList.index(item)</td>
<td>(O(\frac{n}{2})) or (O(n))</td>
</tr>
<tr>
<td>iteration</td>
<td>for item in myList:</td>
<td>(O(n))</td>
</tr>
<tr>
<td>reverse</td>
<td>myList.reverse()</td>
<td>(O(n))</td>
</tr>
</tbody>
</table>

**Dictionary Operations:**

<table>
<thead>
<tr>
<th>Method</th>
<th>Usage</th>
<th>Explanation</th>
<th>Average (O()) for (n) keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>get item</td>
<td>myDictionary.get(key) = myDictionary[key]</td>
<td>Returns the value associated with myKey; otherwise None</td>
<td>(O(1))</td>
</tr>
<tr>
<td>set item</td>
<td>myDictionary[key] = value</td>
<td>Change or add myKey:value pair</td>
<td>(O(1))</td>
</tr>
<tr>
<td>in</td>
<td>myKey in myDictionary</td>
<td>Returns True if myKey is in myDictionary; otherwise False</td>
<td>(O(1))</td>
</tr>
<tr>
<td>del</td>
<td>del myDictionary[key]</td>
<td>Deletes the mykey:value pair</td>
<td>(O(1))</td>
</tr>
</tbody>
</table>