1. An alternative to functional-decomposition design is to use object-oriented design (OOD). For the following program, what objects would be useful and what methods (operations on the objects) should each support?

```
Objects: Die, data attributes: currentRoll
methods/operations: roll, getRoll

Tally Sheet: data attributes tallyDictionary
incrementAValue
addLabel
```

2. Consider the Die and AdvancedDie classes from the Python Summary handout.

a) What data attributes of AdvancedDie are inherited from the parent Die class?
   - currentRoll

b) What new data attributes are added as part of the subclass AdvancedDie?
   - numSides

b) Which Die class methods are used directly for an AdvancedDie object?
   - roll

d) Which Die class methods are redefined/overridden by the AdvancedDie object?
   - __init__, __str__, roll

e) Which methods are new to the AdvancedDie class and not in the Die class?
   - eq, add, getSides

f) If die1 and die2 are AdvancedDie objects, then the statement "if die1 == die2:" invokes the __eq__ method of AdvancedDie with die1 "passed" as self and die2 passed as rhs_Die.

```
def __eq__(self, rhs_Die):
    ""
    """Overides default 'eq' operator to allow for deep comparison of dice""
    return self.currentRoll == rhs_Die.currentRoll
```

What would the code be for AdvancedDie __le__ method to allow for the "if die1 <= die2:" statement?

```
def __le__(self, rhs_Die):
    return self.currentRoll <= rhs_Die.currentRoll
```

g) Good software engineering practice is to include precondition and postcondition comments on each method/function where the:

- **precondition** - indicates what must be true for the method to work correctly. Typically, the precondition describes the valid values of the parameters. If the precondition is not satisfied, the method does not need to work correctly!
- **postcondition** - describes the expected state after the method has executed

Consider the AdvancedDie constructor:

```
class AdvancedDie(Die):
    """Advanced die class that allows for any number of sides""
    def __init__(self, sides = 6):
        """Constructor for any sided Die that takes an the number of sides
        as a parameter; if no parameter given then default is 6-sided.""
        Die.__init__(self) # call Die parent class constructor
        self.numSides = sides
        self.currentRoll = randint(1, self.numSides)
```

What precondition and postcondition comments should we add?

- **precondition**: sides is positive integer
- **postcondition**: current roll is random value between 1 and # sides

h) If a method/function has a precondition that is not met when invoked (e.g., die1 = AdvancedDie("six")), why should the method raise an error?

**To raise an error immediately when it occurs**
class AdvancedDie(Die):
    def __init__(self, sides=6):
        # precondition: sides is a positive integer
        # post cond: current roll is a random value between 1 and # sides.

        if not isinstance(sides, int):
            raise (TypeError, "AdvancedDie sides
                must be an integer.")

        if sides <= 0:
            raise (ValueError, "AdvancedDie sides
                must be a positive integer.")

        self._num_sides = sides
        self._current_roll = randint(1, sides)
3. General “Algorithmic-Complexity Analysis” terminology:

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

**parameters** - variables with unspecified values

**problem instance** - assignment of values to parameters, i.e., the specific input to the problem

<table>
<thead>
<tr>
<th>myList:</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>5</td>
<td>10</td>
<td>2</td>
<td>15</td>
<td>20</td>
<td>1</td>
<td>11</td>
</tr>
</tbody>
</table>

(number of elements) $n$: 7

**algorithm** - step-by-step procedure for producing a solution

**basic operation** - 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**.

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

**Big-oh notation ($O(n)$)** - As the size of a problem grows (i.e., more data), how will our program’s run-time grow.

Consider the following `sumList` function.

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

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

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

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

4. Consider the following `someLoops` function.

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

<table>
<thead>
<tr>
<th>Execution flow</th>
</tr>
</thead>
<tbody>
<tr>
<td>$i=0$</td>
</tr>
<tr>
<td>$i-1$</td>
</tr>
<tr>
<td>$i=2$</td>
</tr>
<tr>
<td>$i=n-1$</td>
</tr>
</tbody>
</table>

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

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

$$n + n + ... + n = n^2$$

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

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

$O(n^2)$

$$T(n) = \frac{2n}{20000^2} \approx 10 \text{ sec}$$

Lecture 2 Page 2
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.time()
    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()
```

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

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?)

\[
\text{index} = \begin{cases} 0, 3, 8, 16, \ldots & O\left(\log_2(n^4)\right) \\ 2^0, 2^5, 2^{10}, 2^{15}, \ldots & O\left(\log_2(n)\right) \end{cases}
\]  \( \log_2 n = x \) if \( f(x) = n \)

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

d) Add the execution-time graph for \text{sumSomeListItems} to the graph.
3. $i = 1$
   while $i <= n$: $\log_2 n$
   for $j$ in range($n$): $\leq n$ times
     # something of $O(1)$
   # end for
   $i = i \times 2$
 # end while

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

$$O(n \log_2 n)$$

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

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

$$T(n) = cn \log_2 n$$

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

$$c = \frac{10 \text{sec}}{10000 \log_2 10000} = 19.3$$

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

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

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

$$= \frac{10 \text{sec}}{19.3}$$

$$= 0.513 \text{ sec}$$

3.2.b) 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?

$$X[i] = \text{(Starting address of array)} + i \times \text{(element size in bytes)}$$

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

$$O(1) \text{ constant time}$$

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:

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

$$O(1) \text{ constant time access to any index}$$
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:

<table>
<thead>
<tr>
<th>size:</th>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>999</th>
</tr>
</thead>
<tbody>
<tr>
<td>scores:</td>
<td>20</td>
<td>10</td>
<td>30</td>
<td>25</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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? $O(n)$ $n \leq \text{logical size}$
f) What is the average $O()$ for adding a new score to the array? $O\left(\frac{n}{2}\right) = O\left(\frac{T(n)}{2}\right) = \frac{c_1}{2} = d$ $n$
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 size over the old array

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></td>
</tr>
<tr>
<td>append</td>
<td>myList[i] = newValue</td>
<td></td>
</tr>
<tr>
<td>extend</td>
<td>myList.extend(otherList)</td>
<td></td>
</tr>
<tr>
<td>insert</td>
<td>myList.insert(i, item)</td>
<td></td>
</tr>
<tr>
<td>pop</td>
<td>myList.pop()</td>
<td></td>
</tr>
<tr>
<td>pop(i)</td>
<td>myList.pop(i)</td>
<td></td>
</tr>
<tr>
<td>del</td>
<td>del myList[i]</td>
<td></td>
</tr>
<tr>
<td>remove</td>
<td>myList.remove(item)</td>
<td></td>
</tr>
<tr>
<td>index</td>
<td>myList.index(item)</td>
<td></td>
</tr>
<tr>
<td>iteration</td>
<td>for item in myList:</td>
<td></td>
</tr>
<tr>
<td>reverse</td>
<td>myList.reverse()</td>
<td></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(myKey)</td>
<td>Returns the value associated with myKey; otherwise None</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>set item</td>
<td>myDictionary[myKey] = 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[myKey]</td>
<td>Deletes the mykey: value pair</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>