Problem 1—Short answer (10 points)

1a) Suppose that the function `conundrum` is defined as follows:

```python
def conundrum():
    array = []
    for i in range(13):
        array.append(0)
        for j in range(i, 0, -1):
            array[j] += j
    return array
```

If you look at the code, you’ll see that the function appends a new value onto the array for every value between 0 and 12, which means that the array will eventually contain 13 elements with indices ranging from 0 to 12. Work through the function carefully and indicate the value of each of the elements in the array returned by a call to `conundrum`:

```
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
```

1b) What value is printed if you call the function `example` in the following code:

```python
class MyClass():
    def __init__(self, x):
        def myFunction(y):
            return 2 * x + y
        self.myFunction = myFunction
        def test(self, x):
            return self.myFunction(x + 6)

    def example():
        value = MyClass(14)
        print(value.test(8))
```
Problem 2—Simple graphics (10 points)

Programs like Microsoft Word make it possible to insert text inside a rectangular frame called a *text box*. Once you have created the text box, you can position it anywhere in the document so that it stays together as a unit. You can build much the same facility for the Portable Graphics Library by creating a `GCompound` that represents a text box.

Your task in this problem is to write a function `createTextBox(lines, font)` that takes an array of lines and a font specification and then returns a `GCompound` that displays the lines in the specified font enclosed in a rectangular frame. As an example, you could use the following code to create a text box containing a useful bit of wisdom drawn from Dr. Seuss’s *Oh, the Places You’ll Go!*

```plaintext
SEUSS = [
    "You have brains in your head. You",
    "have feet in your shoes. You can steer",
    "yourself any direction you choose."
]
gw.add(createTextBox(SEUSS, "24px 'Sans-Serif'"), 50, 50)
```

This code draws the following text box with its upper left corner at (50, 50):

```
You have brains in your head. You have feet in your shoes. You can steer yourself any direction you choose.
```

The quote is presented over three lines, where each line is a `GLabel` set in the specified font. The height of the text box scales with the number of lines, and the width of the rectangle is chosen so that the widest of three lines fits within it. The text is separated from the enclosing frame by a margin of 10 pixels on each side, which is given by the constant `TEXT_BOX_MARGiN`.

As another example, the following code snippet:

```plaintext
MANDELA = [
    "No one is born hating another person because of the color",
    "of his skin, or his background, or his religion. People",
    "must learn to hate, and if they can learn to hate, they can",
    "be taught to love, for love comes more naturally to the",
    "human heart than its opposite."
]
gw.add(createTextBox(MANDELA, "12px bold 'Monospaced'"), 50, 250)
```

would wrap a larger text box around Nelson Mandela’s famous quote from his 1994 autobiography, *Long Walk to Freedom*, like this:

```
No one is born hating another person because of the color of his skin, or his background, or his religion. People must learn to hate, and if they can learn to hate, they can be taught to love, for love comes more naturally to the human heart than its opposite.
```
Problem 3—Interactive graphics (15 points)

One of the earliest electronic arcade games was Whac-A-Mole, which was released in 1976 by Creative Engineering, Inc. in Japan. In the game, the surface of the display was covered with an array of circular holes. From time to time, a mole would rise up out of the hole, and the player’s job was to pound a hammer on that hole before the mole disappeared again below the surface.

In this problem, your job is to create a simplified version of Whac-A-Mole, for which the implementation can be broken down into the following steps:

Step 1.

Write a program WhacAMole.py that displays a set of circles on the screen arranged to form a square matrix with $N_{CIRCLES\_PER\_ROW}$ in each row and column. The diameter of each circle is given by the constant CIRCLE_SIZE and the space between each circle is given by the constant CIRCLE_SEP. The constants GWINDOW_WIDTH and GWINDOW_HEIGHT are set so that there is just enough room for the circles with a margin that is half the value of CIRCLE_SEP on all four sides. Thus, with the values of these constants shown on page 7, the initial display should look like this:

![Initial display of WhacAMole](image)

Step 2.

Set up a timer process that runs every TIME_STEP milliseconds. In each time step, your program should pick a random $(x, y)$ point somewhere on the graphics window. If this point is inside an unfilled circle, you should set the circle to be filled and set its fill color to Black, indicating the appearance of a mole. For example, if the random point is inside the circle just above the lower left circle, that circle should turn black, as follows:

![Display with mole](image)
Whether or not the point is inside a circle, the code for each time step should change the previously selected circle to be unfilled, assuming that it still exists on the screen.

Step 3.
Set up a click handler that checks if the user clicks inside a circle during the time that it is filled (you can test whether a G Oval is filled by calling the isFilled method). If the user manages to click in the circle before the timer process sets it back to its unfilled state, you should remove that circle from the graphics window. Thus, if the user clicks on the black circle in the preceding diagram within the TIME_STEP interval, that circle should disappear from the screen, as follows:

For the purposes of this problem, you need not figure out how to get the game to stop, which means that you can simply let the timer run until the user quits the program.
Problem 4—Strings (10 points)

The table of contents for a book typically consists of a list of chapter titles along the left margin of a page and the corresponding page numbers along the right. To make it easier for your eye to match up the chapter and page, the usual approach is to tie the two visually with a line of dots called a leader. Using this style, the entries for the first eight chapters in the Python textbook look like this:

1. Introducing Python .............................................. 1
2. Control Statements ............................................. 41
3. Simple Graphics ................................................ 79
4. Functions .......................................................... 117
5. Writing Interactive Programs .................................... 155
6. Strings .............................................................. 195
7. Lists ........................................................................ 231
8. Algorithmic Analysis ............................................... 269

Write a function

```python
def createTocEntry(title, page)
```

that takes a chapter title (which includes the chapter number in these examples) and the page number on which that chapter begins. Your function should return a string formatted as an entry for the table of contents. Thus, if you were to call

```python
createTocEntry("6. Strings", 195)
```

the function should return the following string:

"6. Strings ............................................................. 195"

In generating this string, your function should adhere to the following guidelines:

- The strings returned by `createTocEntry` should all have the same length, which is given by the constant `TOC_LINE_LENGTH`. In the earlier examples, `TOC_LINE_LENGTH` has the value 60.
- The chapter title must appear at the beginning of the result string and must be separated from the first dot in the leader by at least one space.
- The page number must appear at the end of the result string so that the last character of each page number will line up at the column specified by `TOC_LINE_LENGTH`. Like the title, the page number must be separated from the last dot in the leader by at least one space.
- The leader itself is composed of alternating spaces and dots, indicated by the period character ".". Moreover, the dots must be arranged so that they line up vertically. If you simply start the leader one space after the chapter title, the dots would appear to weave back and forth on the page as illustrated by the following lines:

```plaintext
1. Introducing Python .............................................. 1
2. Control Statements ............................................. 41
3. Simple Graphics ................................................ 79
```
An easy way to ensure that the dots are aligned correctly is to add an extra space after chapter titles—like "3. Simple Graphics"—with an even number of characters but not after those—like "2. Control Statements"—with an odd number.

• You may assume that the chapter title and page number fit in TOC_LINE_LENGTH character positions and need not make your function handle the situation when the title is too long for the line.

Problem 5—Arrays (10 points)
Write a function

```python
def findFirstDuplicate(array):
```

that returns the first element in `array` that appears more than once. If there is no duplicated value in the array, `findFirstDuplicate` should return the value `None`.

The following examples illustrate the values that `findFirstDuplicate` should return:

```python
findFirstDuplicate([ 1, 2, 3, 4, 3, 2 ]) → 2
findFirstDuplicate([ "a", "b", "c", "d", "c" ]) → "c"
findFirstDuplicate([ 1, 2, 3, 4, 5, 6, 7 ]) → None
findFirstDuplicate([ ]) → None
```

Problem 6—Recursive functions (10 points)
The third problem on Problem Set #2 (Handout #6) introduced the hailstone sequence from mathematics. If you start with a positive integer `n`, you can compute the terms in the hailstone sequence by repeatedly executing the following steps:

• If `n` is equal to 1, you’ve reached the end of the sequence and can stop.
• If `n` is even, divide it by two.
• If `n` is odd, multiply it by three and add one.

The hailstone number for a positive integer `n` is the number of steps it takes for this process to reach 1. Thus, the hailstone number for 1 is 0, and the hailstone number for 7 is 16, which is the number of arrows in the following sequence:

7 → 22 → 11 → 34 → 17 → 52 → 26 → 13 → 40 → 20 → 10 → 5 → 16 → 8 → 4 → 2 → 1

Write a recursive function

```python
def hailstoneNumber(n):
```

that returns the hailstone number for `n`. For example, calling `hailstoneNumber(7)` should return the value 16. Similarly, calling `hailstoneNumber(5)` should return 5 because there are five arrows in the sequence

5 → 16 → 8 → 4 → 2 → 1

Calling `hailstoneNumber(1)` should return 0, because it takes that no steps at all to reach 1.
Remember that your function—along with any helper functions you write—must operate recursively and may use no iterative constructs such as `while` or `for`.

**Problem 7—Defining classes (10 points)**

In writing the ImageShop project, one of the functions that you needed to perform for several of the operations is creating a pixel array for a given width and height. Although it is easy to write a function for this purpose—and many of you did just that—you can also define a `PixelArray` class that extends the built-in Python class `list` so that it represents a two-dimensional pixel array. The constructor for `PixelArray` should take two parameters in addition to the required `self` parameter that specify the width and height of the pixel array. When the constructor returns, the object should be initialized to a list of lists where the outer list contains the rows and the inner lists contain the individual pixel values. Your class definition does not need to define any methods other than the constructor.

For example, calling `PixelArray(4, 3)` creates a two-dimensional array with the following contents:

```
[ [ 0, 0, 0, 0 ], [ 0, 0, 0, 0 ], [ 0, 0, 0, 0 ] ]
```

Remember that each row must be allocated independently to avoid having the rows be copies of the same row.

The advantage of defining `PixelArray` as a subclass of `list` is that doing so means that objects of type `PixelArray` automatically implement all the standard functions that apply to Python lists.

**Problem 8—Linked structures (10 points)**

The *height* of a tree is defined as the length of the longest path from the root to a leaf. Consider, for example, the fully balanced tree of Disney’s seven dwarves, which looks like this:

```
  Grumpy
   /
  Doc  Sleepy
 /     /
Bashful Dopey Happy Sneezy
```

The height of this tree is 2 because all paths from the root at Grumpy to a leaf (and therefore certainly the longest path) involves two links, as in `Grumpy->Doc->Bashful` contains two links. If you instead create a binary search tree by entering the 13 dwarves who visit Bilbo at the beginning of J. R. R. Tolkien’s *The Hobbit*, the resulting structure has the following form, assuming that the dwarves are added to the tree in the order in which they showed up at Bilbo’s door:
The height of this tree is 5 because the longest path from the root to a leaf is the five-step chain that runs Dwalin→Balin→Dori→Bifur→Bofur→Bombur.

When using this standard definition of height, the height of a tree containing just a single root node is 0 and the height of an empty tree that consists only of the value None is −1.

Assuming that the Python definition for a node in this tree is

```python
class _Node:
    def __init__(self, name, left=None, right=None):
        self._name = name
        self._left = left
        self._right = right
```

write the definition of a height method that returns the height of the tree.

The definition of height is extremely short if you approach this problem recursively, keeping in mind that the height of any tree is always one greater than the maximum height of its left and right subtree.

**Problem 9—Python data structures (15 points)**

At this point in the semester, the data structures you know best are the ones from the Adventure assignment, which drive the operation of the run method in AdvGame. You could, of course, do something else with those same structures. For example, you might want to write a function that would generate a cheat sheet for solving a particular Adventure game by listing all the objects that are available in the game along with where they are located at the beginning of the game and where they are required to traverse a locked passage.
Your job in this problem is to implement the function

```python
    def printCheatSheetForObjects(rooms, objects)
```

that takes two parameters, both of which are dictionaries:

1. The dictionary `rooms`, in which the keys are the room names and the values are the `AdvRoom` objects. As discussed in Milestone #7, you may assume that the `AdvRoom` class exports a method called `getPassages`, which returns a list of tuples containing the direction name, the destination room, and the object (possibly `None`) that unlocks the passage.

2. The dictionary `objects`, in which the keys are the object names and the values are the corresponding `AdvObject` structures.

Your function should display a cheat sheet showing the name of each object, its short description in parentheses, and the short description of its initial location (or the word "PLAYER" for objects that the player is initially holding). After each object in the list, the `printCheatSheetForObjects` function should go through the rooms data structure and print out a line for each entry in which that object acts as a key to a locked passage.

For example, if you call the function for the rooms and objects for the `Small` adventure, your implementation of `printCheatSheetForObjects` should generate the following output:

```
| KEYS (a set of keys): Inside building |
| Needed for DOWN from Outside grate |
| LAMP (a brightly shining brass lamp): Beneath grate |
| Needed for XYZZY from Inside building |
| Needed for WEST from Cobble crawl |
| ROD (a black rod with a rusty star): Debris room |
| WATER (a bottle of water): PLAYER |
```