CSCI 1133
Exercise Set 10

You should complete as many of these problems as you can. Practice is *essential* to learning and reinforcing computational problem solving skills. We encourage you to do these without any outside assistance.

Create a directory named `exercise10` and save each of your problem solutions in this directory. You will need to save your source files in this directory and push it to your repository in order to get help/feedback from the TAs. This requires that you follow a rigid set of naming requirements. Name your individual Python source files using "`ex10`" followed by the letter of the exercise, e.g., "`ex10a.py`", "`ex10b.py`", etc.

For these first problems, include the class definitions in the module.

Automatic testing of your solutions is performed using a GitHub agent, so it is necessary to name your source files precisely as shown and push them to your repository as you work.

A. **Circle Class**

Write the definition for a class named `Circle` that has a *single* private attribute named `__radius` representing the radius of the circle, one mutator method: `setRadius(r)` that will update the radius value, and two accessor (getter) methods: `getCircumference()` and `getArea()` that return the circumference and area of the circle, respectively. The class initializer should take a single optional argument containing the radius (default = 1). Import the `math` library and use the `math.pi` value as appropriate.

Be sure to thoroughly test your class by instantiating, computing circumference/area and modifying several circles using Python in interactive mode.

B. **Bug Class**

For this exercise you need to create a class named `Bug` that models a bug moving along a horizontal line. The bug can move either to the right or left, one step at a time. In each move, its position changes by one unit in the *current direction*. Initially, the current direction is set to move the bug to the right, but it can turn to change its direction. The bug class has two integer instance variables: position (0..*n*) and current direction (+1: right, -1: left). Implement the following five methods:

- An initializer method with a *single argument* to initialize the initial *position* of the bug. It should default to the value 0. The initial direction value should always be +1
- A mutator method named `move` with no arguments that will move the bug one step in the current direction. The `move` method should not allow the bug to move to a location less than zero!
- A mutator method named `turn` with no arguments that will change the direction the bug is moving
- An accessor method named `display` with no arguments that will display (i.e., print) the location and direction of the 'bug'. The display method should output a period ('.') for each position from 0 to the current position of the bug and then output a '>' if the bug direction is to the right or a '<' if the bug is moving to the left. e.g.,:

```
.....<
```

Include a non-pure function named `main` that will first create a `Bug` object at position 10, move the bug, turn the bug, and then (using a loop) move the bug 13 more times. Display the bug's state using the `display` method following each move.

C. **Stopwatch Object**

Construct a class definition for a `Stopwatch` class that will simulate a simple start-stop timer. Include two private instance variables: `startTime` and `endTime` (floats) and the following methods:

- An nitializer method that will initialize both start and end times with the current time of day as returned by the Python `time()` function (from the `time` module). The `time()` function returns the current time of day in seconds as a floating point value with microsecond resolution:

```
from time import time
time()
```

- A mutator method named `start` that will reset the `startTime` to the current time
- A mutator method named `stop` that will set the `endTime` to the current time
- An accessor method named `elapsedTime` that returns the elapsed time for the stop watch as a float
- Accessor methods `getStartTime` and `getEndTime` that return their respective values

Write a non-pure function named `main` that will instantiate and use a `Stopwatch` object to determine and print the amount of time it takes to sort a list with 10,000 randomly generated elements using the Python `.sort()` list method.

D. **Sparse Matrices**

Write the definition for a class named `Smatrix` that will represent sparse matrices. A *sparse* matrix is one in which most of the individual values are zero, so it is more efficient to simply store the non-zero elements and assume all others are zero.

The `Smatrix` class will maintain a *single* private instance variable representing the matrix as a Python *dictionary* in which each key of the dictionary is a tuple: `(`*row*, *column*`)` and the associated object is the value of the matrix at the specified row and column.

Implement the following methods:

- An initializer that will initialize a null-dictionary.
- An accessor method: `get(row,col)` that will return the value of the matrix at row,col. If it is not in the matrix, return zero.
- An accessor method: `set(row,col,value)` that will save/replace the value of the matrix at row,col. If the value is zero, remove the association from the dictionary.
- An overloaded `__repr__` method that will display the complete matrix in matrix form (including the zero elements)
- An overloaded `__add__` operator that will sum two matrices and return a new matrix
- An overloaded `__sub__` operator that will subtract two matrices and return a new matrix
- An overloaded `__mul__` operator that return the product of two matrices

Thoroughly test your `Smatrix` class by instantiating several matrices and displaying their sum, difference and product using arithmetic operations. e.g.:

```
m1 = Smatrix()
m1.set(0,0,2)
m1.set(0,1,2)
m2 = Smatrix()
m2.set(0,0,1)
m2.set(0,1,4)
```

```
m1 + m2
m1 - m2
m1 * m2
```