15-110 Hw3 - Written Portion

Name:

AndrewID:

Complete the following problems in the fillable PDF, or print out the PDF, write your answers by hand, and scan the results. Also complete the programming problems in the starter file hw3.py from the course website.

When you are finished, upload your hw3.pdf to **Hw3 - Written** on Gradescope, and upload your hw3.py file to **Hw3 - Programming** on Gradescope. Make sure to check the autograder feedback after you submit!

Written Problems

#1 - Hw2 Code Review - 5pts

It isn't always good enough just to write code that works. It's also important to write code that is **clear** and **robust** - easy to understand and ready to handle a variety of inputs.

To help you learn how to write good code, we will have a few **code reviews** this semester where you will meet with a course TA to go over the code you wrote for a previous assignment. The TA will point out things you're doing well and areas where your code can be cleaner (even if it works already!).

To receive five points for the Hw2 code review, sign up for and attend a code review session with a TA the weekend after Week 6 of classes. We'll release more details about how to sign up for and attend these sessions via Piazza.

#2 - 2D Lists - 5pts

Can attempt after Lists and Methods lecture

Fill the below table with the values in the 2D list returned by `mysteryFunction`, with each value placed in the appropriate row and column. Write an X in squares that are outside the bounds of the list.

```
def mysteryFunction(x, y):
    myList = []
    for i in range(x):
        innerList = []
        j = 0
        while j < y:
            if j <= i:
                innerList.append("o")
            else:
                innerList.append("-")
            j = j + 1
        myList.append(innerList)
    return myList

result = mysteryFunction(5, 4)
```

	result[0][0]	result[0][1]	result[0][2]	result[0][3]	result[0][4]	result[0][5]
result[0]						
	result[1][0]	result[1][1]	result[1][2]	result[1][3]	result[1][4]	result[1][5]
result[1]						
	result[2][0]	result[2][1]	result[2][2]	result[2][3]	result[2][4]	result[2][5]
result[2]						
	result[3][0]	result[3][1]	result[3][2]	result[3][3]	result[3][4]	result[3][5]
result[3]						
	result[4][0]	result[4][1]	result[4][2]	result[4][3]	result[4][4]	result[4][5]
result[4]						
	result[5][0]	result[5][1]	result[5][2]	result[5][3]	result[5][4]	result[5][5]
result[5]						

#3 - Recursion Tracing - 5pts

Can attempt after Recursion lecture

Trace the following code, then fill out the table below to indicate all the **recursive function calls** that are made and which **value** is returned by each function call. You may not need all of the rows.

```
def gcd(x, y):
    if y == 0:
        return x
    else:
        result = gcd(y, x % y)
        return result

print(gcd(20, 12))
```

Note: in the second column, make sure to indicate the actual returned **value**, not a set of arguments, a function call, or an expression.

Function Call	Returned Value
gcd(20, 12)	

#4 - Tracing Towers of Hanoi - 10pts

Can attempt after Recursion II & Search Algorithms lecture

Recall the algorithm we discussed in class to solve the Towers of Hanoi problem. Use that algorithm to fill out all the steps needed to move three discs from Peg A to Peg C in the table below. You might not need to use all the rows.

The three discs are called 1, 2, 3 (where 1 is the smallest and the disc on top). So the algorithm starts with the discs 1, 2, 3 on Peg A, and should end with 1, 2, 3 on Peg C. We've done the first step for you.

	Peg A	**Peg B**	**Peg C**
Start	1, 2, 3		
Step 1	2, 3		1
Step 2			
Step 3			
Step 4			
Step 5			
Step 6			
Step 7			
Step 8			
Step 9			
Step 10			
Step 11			

How many steps would it take to move 4 discs instead of 3?

#5 - Linear Search Debugging - 10pts

Can attempt after Recursion II & Search Algorithms lecture

The following three functions all attempt to implement the algorithm linear search, but with a twist: instead of identifying whether or not the target occurs in the list, each function returns the first index where the item occurs, or -1 if it never shows up. However, only one of the three implementations is fully correct.

Identify which of the three functions is correct, then explain on the next page both what is wrong with the other two and how they can both be fixed.

Note: for this code-reading problem, you are allowed (and encouraged!) to run the code directly to see how it works. Try using the debugging approaches you learned before!

```
def linearSearchA(lst, target):
    for i in range(len(lst)):
        if lst[i] == target:
            return i
        return -1

def linearSearchB(lst, target):
    i = 0
    while i < len(lst):
        if lst[i] == target:
            return i
        i = i + 1
    return -1

# Initially called with index = 0
def linearSearchC(lst, target, index):
    if lst[0] == target:
        return index
    elif len(lst) == 0:
        return -1
    else:
        return linearSearchC(lst[1:], target, index + 1)
```

[continued on next page]

[continued from previous page]

Which implementation is correct?

- ☐ linearSearchA
- ☐ linearSearchB
- ☐ linearSearchC

Why are the other two incorrect, and how can they be fixed?

#6 - Binary Search - 10pts

Can attempt after Recursion II & Search Algorithms lecture

In the following table, write out the recursive calls that our implementation of `binarySearch` from lecture would make while searching the given list for the given item. Make sure to write out the **function call**, not the result. You might not need to use all the rows.

For example, if you were to trace `binarySearch([1, 2, 3, 4, 5], 1)`, you'd get:
Recursive Call 1: `binarySearch([1, 2], 1)`
Recursive Call 2: `binarySearch([1], 1)`

Q1: Search for 5

Original Call	`binarySearch([3, 5, 6, 7, 9, 11, 11, 15, 15, 19], 5)`
Recursive Call 1	
Recursive Call 2	
Recursive Call 3	
Recursive Call 4	
Recursive Call 5	

Q2: Search for 14

Original Call	`binarySearch([3, 5, 6, 7, 9, 11, 11, 15, 15, 19], 14)`
Recursive Call 1	
Recursive Call 2	
Recursive Call 3	
Recursive Call 4	
Recursive Call 5	

#7 - Dictionary Keys and Values - 5pts

Can attempt after Dictionaries lecture

Given the following set of code:

```
d = { "snow" : 7, "rain" : 4, "sun" : 10 }
d["fog"] = d["rain"] - 1
for k in d:
    d[k] = d[k] * 2
d.pop("sun")
```

What are the **keys** of the dictionary `d` after this code has run?

For each of the keys you listed above, what is its **value** associated with that key after the code has run?

What line of code would you write to add the **key-value pair** `("hail", 1)` to the code? Assume the line will be added at the *end* of the current code.

Programming Problems

For each of these problems (unless otherwise specified), write the needed code directly in the Python file in the corresponding function definition.

All programming problems may also be checked by running 'Run current script' on the starter file, which calls the function `testAll()` to run test cases on all programs.

#1 - `onlyPositive(lst)` - 10pts

Can attempt after Lists and Methods lecture

Write a function `onlyPositive(lst)` that takes as input a **2D list** and returns a new 1D list that contains only the positive elements of the original list, in the order they originally occurred. You may assume the list only has numbers in it.

Example: `onlyPositive([[1, 2, 3], [4, 5, 6]])` returns `[1, 2, 3, 4, 5, 6]`, `onlyPositive([[0, 1, 2], [-2, -1, 0], [10, 9, -9]]` returns `[1, 2, 10, 9]`, and `onlyPositive([[-4, -3], [-2, -1]])` returns `[ ]`.

#2 - `addToEach(lst, s)` - 5pts

Can attempt after References and Memory lecture

Write the function `addToEach(lst, s)` which takes a list of strings and a string `s` and **destructively** modifies the list so that every element has `s` concatenated to it, returning `None` once done. For example, if `a = ["how", "are", "you"]`, calling the function `addToEach(a, "yah")` will evaluate to `None`, but will also change `a` to hold `["howyah", "areyah", "youyah"]`.

#3 - `recursiveLongestString(lst)` - 10pts

Can attempt after Recursion lecture

Write a function `recursiveLongestString(lst)` that takes a list of strings as input and returns the longest string in the list. You may assume the list contains at least one element and there will not be a tie. This function must use **recursion** in a meaningful way; a solution that uses a loop or the built-in max function will receive no points.

For example, `recursiveLongestString(["a", "bb", "ccc"])` returns `"ccc"`, and `recursiveLongestString(["hi", "its", "fantastic", "here"])` returns `"fantastic"`.

Hint: what properties does the recursive result have if the function works as expected?

Another hint: consider what the **base case** for this algorithm should be. It isn't the usual list base case where the list is empty, because an empty list can't have a longest string. What should it be instead?

#4 - `generateBubbles(canvas, bubbleList)` - 10pts

Can attempt after Dictionaries lecture

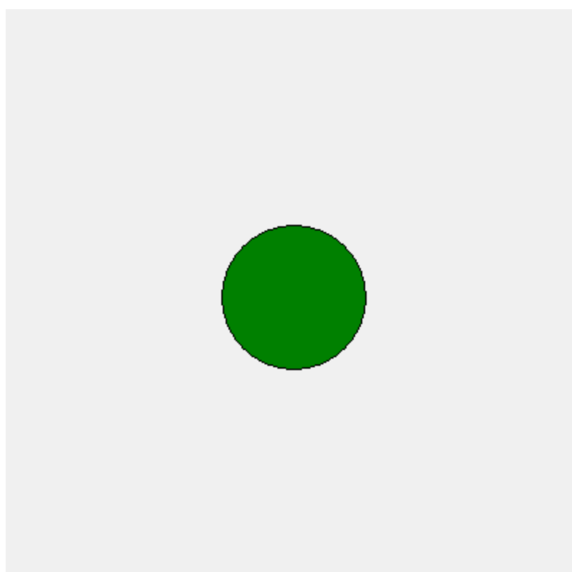
Write the tkinter function `generateBubbles(canvas, bubbleList)` which takes a tkinter canvas and a **list of dictionaries**, `bubbleList`, and draws bubbles as described in `bubbleList`.

Each dictionary in the bubble list contains exactly four keys: `"left"`, `"top"`, `"size"`, and `"color"`. The first three all map to integers (the left coordinate, top coordinate, and diameter size of the bubble), and the fourth maps to a string (its color). Use this information to draw the bubble (with `canvas.create_oval`) in the appropriate location, with the correct size and color.

For example, if we make run the function with the bubble list from the first test:

```
bubbleList1 = [ {"left":150, "top":150, "size":100, "color":"green"} ]
```

We'll get:



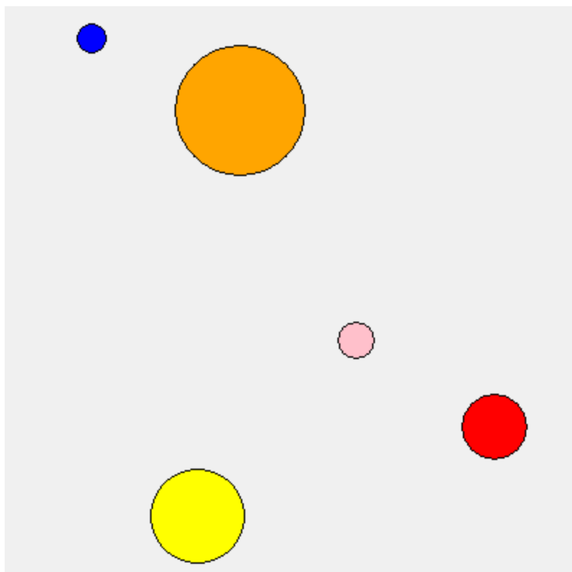
[continued on next page]

[continued from previous page]

And the second test, which has:

```
bubbleList2 = [
     {'left': 317, 'top': 269, 'size': 45, 'color': 'red' },
     {'left': 118, 'top': 27, 'size': 90, 'color': 'orange'},
     {'left': 101, 'top': 321, 'size': 65, 'color': 'yellow'},
     {'left': 231, 'top': 219, 'size': 25, 'color': 'pink' },
     {'left': 50, 'top': 12, 'size': 20, 'color': 'blue' } ]
```

Should produce this:



The third test randomly generates 10 bubbles using the provided `makeNBubbles(n)` function. Try changing the size of `n` to generate more or less bubbles and see how it looks! Your bubbles will be different every time.

Hint: a list of dictionaries might sound intimidating at first, but it's not so bad! Just loop over the list, access the dictionary using the loop control variable, then key-index into the dictionary with the four known keys to get the needed values.

#5 - `getBookByAuthor(bookInfo, author)` - 5pts

Can attempt after Dictionaries lecture

Dictionaries are very good at searching for keys, but not so good at searching for values. Write the function `getBookByAuthor(bookInfo, author)` which takes a dictionary mapping book titles (strings) to author names (also strings), and an author name (a string), and returns the book associated with that author, or `None` if the author does not appear in the dataset. You are guaranteed that no author will appear more than once in the dictionary.

For example, calling the function on `{ "The Hobbit" : "JRR Tolkein", "Harry Potter and the Sorcerer's Stone" : "JK Rowling", "A Game of Thrones" : "George RR Martin" }` and `"JK Rowling"` would return `"Harry Potter and the Sorcerer's Stone"`.

Hint: you basically want to implement **linear search** over a dictionary instead of a list. Make sure you use the right kind of loop for a dictionary!

#6 - `makeIMDB(actorList, movieList)` - 10pts

Can attempt after Dictionaries lecture

Write the function `makeIMDB(actorList, movieList)` that takes two lists, a list of actor names and a list of movie names (both lists of strings), and returns a dictionary mapping names of actors to names of movies. You may assume that the two lists match up, i.e., each actor is at the same index as their movie.

You should use a **loop** to construct the dictionary. You'll need to loop over both the `actorList` and the `movieList` *at the same time* to access the key and value together. To do this, use the **same loop control variable** on both lists in each iteration. Which type of loop will allow you to do this?

If a person occurs in `actorList` multiple times (in other words, if they were in multiple movies), you should map their name to the **first** movie they were paired with. For example, given the names `["Ni Ni", "Sofia Vergara", "Ni Ni"]` and the movies `["Suddenly Seventeen", "Hot Pursuit", "Love Will Tear Us Apart"]`, the function would return the dictionary `{"Ni Ni" : "Suddenly Seventeen", "Sofia Vergara" : "Hot Pursuit" }`.