15-110 Check3 - Written Portion

Name:

AndrewID:

#1 - Aliasing and Mutability - 20pts

The following code creates and modifies lists. Determine each list's values after the code has run.

```
a = [ "apple", "banana", "carrot", "donut" ]
b = a
b.remove("apple")
c = a + [ "eclair" ]
d = c[1:]
d.insert(2, "fig")
```

Variable	List Values
a	
b	
c	
d	

Select all of the pairs of lists that are **aliased** at the end of the code.

- ☐ **a** and **b**
- ☐ **a** and **c**
- ☐ **a** and **d**
- ☐ **b** and **c**
- ☐ **b** and **d**
- ☐ **c** and **d**
- ☐ None of the lists are aliased

#2 - Base Cases and Recursive Cases - 16pts

Assume you want to write a function `recursiveSum` that takes a positive integer, `n`, and **recursively** computes the sum from one to `n`.

For example, the result when calling the function on n=5 is 5+4+3+2+1 = 15.

What condition do you need to check for your **base case**?

What do you return in the **base case**?

What is the recursive call on a smaller problem in the **recursive case**?

How do you use the recursive call's result to solve the whole problem for `n` in the **recursive case**?

#3 - Tracing Towers of Hanoi - 10pts

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?

#4 - Binary Search - 14pts

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.

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	

15-110 Check3 - Programming Portion

Each of these problems should be solved in the starter file available on the course website. Submit your code to the Gradescope assignment Check3 - Programming for autograding.

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

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

Write a **non-destructive** function `onlyOdds(lst)` that takes a list of integers and returns a **new** list containing only the odd elements of `lst`. Note that this should not return the odd indexes- it should return the odd **elements**!

For example, `onlyOdds([1, 2, 3, 4, 5, 6])` returns `[1, 3, 5]`, and `onlyOdds([4, 1, 70, 35, -9])` returns `[1, 35, -9]`

#2 - `removeEvens(lst)` - 10pts

Write a **destructive** function `removeEvens(lst)` that takes a list of integers and destructively removes the even elements of the provided list so that it contains only the original odd elements at the end of the function. This function should return `None` instead of the list; we'll test it by checking whether the input list was modified properly.

For example, `removeEvens([1, 2, 3, 4, 5, 6])` modifies the list to be `[1, 3, 5]`, while `removeEvens([4, 1, 70, 35, -9])` modifies the list to be `[1, 35, -9]`.

Hint: this is tricky because `lst` will change as the function runs. You should use an appropriate loop to account for this - see the course slides! Also, make sure to check for aliasing issues.

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

Write a function `recursiveReverse(lst)` that takes a list as input and returns a **new** list which has the same elements, but in reverse order. This function must use **recursion** in a meaningful way; a solution that uses a loop, built-in reverse functions, or a slice with a negative step will receive no points.

For example, `recursiveReverse([1, 2, 3])` should return `[3, 2, 1]`.

#4 - `recursiveCount(lst, item)` - 10pts

Write a function `recursiveCount(lst, item)` that takes a list and a value as input and returns a count of the number of times that item occurs in the list. This function must use **recursion** in a meaningful way; a solution that uses a loop or built-in count functions will receive no points.

For example, `recursiveCount([2, 4, 6, 8, 10], 6)` returns `1`,
`recursiveCount([4, 4, 8, 4], 4)` returns `3`, and
`recursiveCount([1, 2, 3, 4], 5)` returns `0`.