Lab 6: Recursion

Labs are graded for participation rather than correctness. Keep all your lab code in your course GitHub repo to receive credit for your work. We'll be looking to see that you have at least partially completed all problems in each lab.

If you finish the lab assignment early, you may get started on the homework.

Getting started

In your local repo, create a folder for Lab 6. Create new .py files for this week's lab problems and save them in the Lab 6 folder.

Documentation and format guidelines

The guidelines from previous labs still apply. No new guidelines this week!

Problem 1: Tracing recursive functions

In this problem, we're asking you to trace variables and returns through example recursive functions. You do not need to write any code or turn anything in for this problem. There will definitely be a question like this in the midterm so don't skip it!

Example One -- Tracing -- Reversing a list
This is a classic list problem (and one that might show up on one of your interviews someday). This is a recursive function that, given a list, returns a copy of the list with its elements in reverse order. You saw this in lecture yesterday.

If the original list is:

[1, 1, 2, 3, 5, 8, 13]

Then this function should return the following list:

[13, 8, 5, 3, 2, 1, 1]

There is no need to modify the original list. This example is similar to summing the elements of a list, because we can break down the original list into a smaller subproblem by removing one element each time, until eventually we end up with a list of size 1 (or an empty list) -- that's the base case.

Here's a function to do the reversing

```
1  def reverse_list(lst):
2      if len(lst) <= 1:
3         return lst
4      else:
5         return [lst[-1]] + revervse_list(lst[:-1])
```

Suppose our initial list is: [1, 1, 2, 3, 5]. Fill out the table below to trace the recursion (use paper or a PDF editor if you have one...you don't need to turn this in). In each box, indicate (1) what's contained in the variable lst, (2) whether we're in the base case, and (3) what gets returned. Remember this is the call only! None of the returns are completed until the base case is reached. The first two rows have been completed for you.

Call #	**lst**	**Base Case?**	**Returned**
1	[1, 1, 2, 3, 5]	N	[5] + rev_list([1,1,2,3])
2	[1,1,2,3]	N	[3] + rev_list([1,1,2])
3			
4			
5			

Remember, only after all those calls are executed do we start returning. In the table below, fill in the value of lst *(not* lst[1:]) at line 5*,* and what gets returned to the previous call at line 5. The first line is what's returned by the base case, line 5 in the table above. Don't worry too much about how you format the return, the important thing is that you understand conceptually what is being returned. The first two rows have been completed for you.

Return #	**Lst (line 5)**	**Returned**
1	[1]	[1]
2	[1,1]	[1] + [1] = [1,1]
3		
4		
5		

Example 2 -- Tracing a Mystery Recursive Function
This is where tracing recursion is a super useful skill. If you don’t already know what a function does, you need to be able to figure it out.

Here’s a recursive function (its input is a string):

```
1  def spam(s):
2      if len(s) <= 1:
3          return True
4      else:
5          return s[0] == s[-1] and spam(s[1:-1])
```

Suppose our initial call to the function is **`spam('abceba').`** Fill out the table below to trace the recursion. In each box, indicate (1) what’s contained in the variable s, (2) whether we’re in the base case, and (3) what gets returned, if we’re in the base case. (Note that we’ve left some extra space here, the size of the table isn’t directly correlated with the number of recursive calls.)

Call #	**s**	**Base Case?**	**Returned**
1	**`'abceba'`**	N	True and spam(‘**`bceb’)`**

Remember, only after all those calls are executed do we start returning. In the table below, fill in the value of s *(not* s[1:-1]) at line 5*,* and what gets returned.

Return #	**s (line 6)**	**Returned (line 6)**

Problem 2: Greatest common divisor

Part 1. Write a recursive function to determine the greatest common divisor (GCD) of two non-zero positive integers. The GCD is the largest positive integer that divides two or more numbers without any remainder. For example, the GCD of 12 and 8 is 4, the GCD of 15 and 30 is 15, and the GCD of 10 and 3 is 1.

Hint: for this problem, you will find the % operator useful. The Euclidean algorithm for computing GCD is covered in 5002. If you haven't got there yet, here's an explanation of how it works: https://www.khanacademy.org/computing/computer-science/cryptography/modarithmetic/a/the-euclidean-algorithm

Part 2. Write a recursive function to determine the GCD of *n* positive integers, where $n \geq 1$. Your new function can call the function you wrote in part 1

Problem 3: More recursion practice

Redo the following lab problems using recursive functions instead of iteration:

- Lab 4, problem 1 (logarithm)
- Lab 5, problem 2 (binary to decimal)