Lab Week 1: (10 points)

A Note about Writing Programs: Experienced programmers know how important it is to understand the problem and develop a solution before writing any code. Why? Because this saves time and effort in the long run. Drawing pictures and writing pseudo code clarify both the problem and how to solve it. If you cannot solve simple versions of the problem by hand using your algorithm, how can you expect to write a program that will solve the problem.

When writing pseudo-code, at first do **not** give a complete program. Give just enough detail that your intentions are clear and unambiguous, but do not provide extraneous details (complex syntax and/or variable declarations) which are **otherwise obvious**. Thinking through examples is especially important. This saves much time later in reducing rework, finding problems, etc.

Lab 1: Recursion and Python practice

- Write an iterative function to find the maximum in a list of integers. Do not submit just demo.
- Write a recursive function to reverse a string. Do not submit just demo.
- Write a recursive function to search a list of integers using binary search along with test cases. The function if the **target** of the search is in the list return its index else return **None**. If the list is empty return **None**

Details:

Download the files: (Read the section below if you are new to Python/CalPoly)

- **Lab1.py**
- **Lab1_test_cases.py**

Test Cases

Many people tend to focus on writing code as the singular activity of a programmer, but testing is one of the most important tasks that one can perform while programming. Proper testing provides a degree of confidence in your solution. Systematic testing helps you to discover and then fix bugs (i.e., debug). Writing high quality test cases can greatly simplify the tasks of both finding and fixing bugs and, as such, **will save you time during development**. However, testing does not guarantee that your program is correct.

For this part of the lab you will practice writing some simple test cases to gain experience with the unittest framework. I recommend watching the first 20 minutes or so of the following video if you are unfamiliar with testing in Python. https://www.youtube.com/watch?v=6tNS--WetLI

Using your editor/IDE of choice, open the *lab1_test_cases.py* file. This file defines, using code that we will treat as a boilerplate for now, a testing class with a single testing function.

In the *test_expressions* function you will see a single test case already provided. You must add additional test cases to verify that you binary search program is correct.

Submission: Submit two files to PolyLearn by Wednesday 9/20 in your lab hours.

lab1.py : Correct and well documented recursive binary search based on the template provided (5 points)

lab1_test_cases.py : A complete set of test cases for your recursive binary search function. (5 points)
Your test cases should test boundary conditions and other possible errors based on the structure of your program. (white box testing) For each test provide a comment that explains what it is testing.

For folks new to Python or who need review

The basic Python Tutorial

https://docs.python.org/3/tutorial/

This site gives the essentials.

Python for Java Programmers: http://python4java.necaiseweb.org/Fundamentals/Fundamentals

Videos on specific topics you will need to know.

Installing Python on Mac/Windows: https://www.youtube.com/watch?v=YYXdXT2l-Gg&list=PL-osiE80TeTt2d9bfVyTiXJA-UTHn6WwU

Strings: https://www.youtube.com/watch?v=k9TUPpGqYTo&list=PL-osiE80TeTt2d9bfVyTiXJA-UTHn6WwU&index=2

Lists, etc. (Lists are what you need for now) https://www.youtube.com/watch?v=W8KRzm-HUcc&index=4&list=PL-osiE80TeTt2d9bfVyTiXJA-UTHn6WwU

Conditionals: https://www.youtube.com/watch?v=DZwmZ8Usvnk&list=PL-osiE80TeTt2d9bfVyTiXJA-UTHn6WwU&index=6

Loops: https://www.youtube.com/watch?v=6iF8Xb7Z3wQ&list=PL-osiE80TeTt2d9bfVyTiXJA-UTHn6WwU&index=7

Functions: https://www.youtube.com/watch?v=9Os0o3wzS_I&list=PL-osiE80TeTt2d9bfVyTiXJA-UTHn6WwU&index=8

Modules: https://www.youtube.com/watch?v=CqvZ3vGoGs0&list=PL-osiE80TeTt2d9bfVyTiXJA-UTHn6WwU&index=9

For folks new to Cal Poly: Unix Environment

Unix

The lab machines run a distribution of the Linux operating system. For simplicity, and to gain experience in a, potentially, new environment, we will do our coursework in this environment.

Open a terminal window. To do so, from the system menu on the desktop toolbar, select **Applications → System Tools → Terminal**. The Terminal program will present a window with a command-line prompt. At this prompt you can type Linux commands to list files, move files, create directories, etc. For this lab you will use only a few commands. Additional commands can be found at:

- Unix Tutorial
  - Tutorials 1 & 2
  - Parts 1-5
- Editors
  - emacs tutorial
  - vi tutorial

In the terminal, type **ls** at the prompt and hit <Enter>. This command will list the files in the current directory. (also know as a folder.) If you type **pwd**, the current directory will be printed (it is often helpful to type **pwd** while you are navigating directories). If you type **tree**, then you will see a tree-like listing of the directory structure rooted at the current directory.

Create a new directory for your coursework by typing **mkdir cpe202**. Use **ls** again to see that the new directory has been created.

Change into this new directory with **cd** by typing **cd cpe202**. To move back "up" one directory, type **cd ..**.

To summarize

- **ls** list files in the current directory
- **cd** change to another directory
- **mkdir** create a new directory
- **pwd** print (the path of) the current directory

Though these basic commands are enough for now, consider working through a Unix tutorial.

Executing a Program

Download the **.py** files for lab1 from polylearn into the **cpe202** directory created above; this can be done via the browser (by selecting the location to save to), via the graphical file manager, or through the use of the **mv** command in the terminal window (e.g., from the **cpe202** directory, after downloading, type **mv ~/Downloads/lab1.zip .**). If you need assistance doing this, please ask.

A Python program is written in a plain text file. The program can be run by using a Python interpreter.

You can see the contents of *lab1.py* by typing **more lab1.py** or **cat lab1.py** at the command-line prompt.

To execute the program, type **python lab1.py** at the command-line prompt.

To summarize

- **mv** move files
- **more or cat** display contents of a file
- **python** python interpreter used to execute a program by specifying name of program file

Editing

There are many options for editing a Python program. On the department machines, you will find vi, emacs/xemacs, nano, gedit, and some others. The editor that one uses is often a matter of taste. You are not required to use a specific editor, but we will offer some advice (and we will try to help with whichever one you choose). There is lots more information here:
http://users.csc.calpoly.edu/~akeen/courses/csc101/handouts/labs/lab1.html

Interactive Interpreter

The Python interpreter can be used in an interactive mode. In this mode, you will be able to type a statement and immediately see the result of its execution. Interactive mode is very useful for experimenting with the language and for testing small pieces of code, but your general development process with be editing and executing a file as discussed previously.

Start the interpreter in interactive mode by typing **python** at the command prompt. You should now see something like the following.

```
Python 2.6.6 (r266:84292, Jan 22 2014, 09:42:36)
[GCC 4.4.7 20120313 (Red Hat 4.4.7-4)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

The >>> is the interpreter's prompt. You can type an expression at the prompt to see what it evaluates to. Type each of the following (hit enter after each one) to see the result. When you are finished, you can exit the interpreter by typing ctrl-D (i.e., hold the control key and hit d).

- 0 + 1
- 2 * 2
- 19 // 3
- 19 / 3
- 19 / 3.0
- 19.0 / 3.0
- 4 * 2 + 27 // 3 + 4
- 4 * (2 + 27) // 3 + 4