Lab 4: Formal Methods; "(Enumerate and) Test" Algorithm Design

In this lab, you will use formal verification methods to check the correctness of a not-obviously-correct algorithm, and continue to work on the enumerate-and-test graph coloring algorithm (from Question 4.9 in the course notes).

The only proof technique required for the verifications will be the "direct proof" technique ("substituting equals for equals") combined with the lists of steps given in lecture and the course notes — make sure you complete all the steps needed for each proof (or, if you follow your own organization, make sure all the information is in there somewhere). You are welcome to work together in pairs to produce this proof and answer the related questions, or work on these things on your own.

In the development of the graph coloring algorithm, you will write the "testing" step of the enumerate-and-test design (i.e., the rest of Question 4.9.a) and create a test suite for the enumeration (Question 4.9.b.i). The actual writing of the enumeration algorithm will wait for another week. As in the lecture, the "test" part will be amenable to a combination of top-down design and basic recursive design, and the "enumerate" part will be most concisely handled via advanced recursive design.

Your graph coloring functions must work for an arbitrary graph, but your functions do *not* have to work for an arbitrary set of colors — you are allowed to restrict your answer to just use the colors r, g, and b (for red, green, and blue) if you like (this will not make much difference in this week's work, but it will make things significantly simpler later; you are welcome to have a testing function that works for arbitrary sets of colors but enumerate only r, g, b). If you choose to restrict your answer to three colors, do not modify the parameters for the functions — just add a precondition that rules out cases in which other colors are used.

Pre-Lab Work:

Create a "call tree" `for strange_fib(7)` and for `fib(7)`, including the values of each parameter and the value returned in each function. (Refer to your answer to the previous lab to see a python program for the `fib` function.) If you are willing to spend a bit more time to really make clear why `strange_fib` might be faster, use 10 rather than 7.

Arrange to be able to mark up a copy of a test suite for `is_a_legal_coloring` (you may print it on paper or open it in an editor in which you can make markups by circling things or permantly highlighting text). Your test suite may be from your answer to the previous lab, or an answer key for this if one has been made available. Identify each as either "probably base case" or "probably recursive"; for each of the latter, circle/highlight the parts of the parameters that would constitute a smaller instance of the is_a_legal_coloring problem whose answer would help you to find the final answer for that test.

Lab Work:

1. In this lab exercise, you will explore the use of mathematical proof techniques to reason about

a strange algorithm for computing elements of the Fibonacci Sequence:

```
#  This is considerably faster than the obvious recursive approach,
#  though, as we'll see later, there are ways that are faster still
def strange_fib(n):
   precondition(is_integer(n) and n>0)
   # postcondition:  return fib(n), where
   #      fib(1)=1, fib(2)=1, and fib(n)=fib(n-1)+fib(n-2) when n>2
   # alternate postcondition:  returns fib(n), where
   #      fib is the function from the previous lab
   if n <= 2:
       return 1
   elif n == 3:
       return 2
   else:
       # The following line deserves a better comment than this
       return 2*strange_fib(n-1) - strange_fib(n-3)
```

a) Prove that the `strange_fib` function is correct. You may use either of the two choices for the postcondition above.
One of the main challenges here is to make sure you've listed all the steps and connected each to the facts that ensure it must hold. Give a "fully detailed" proof (as defined in Section 5.2.3) for the postcondition steps; you may give a "semi-detailed" proof for the precondition and progress steps of the proof, as long as you address each required element of the proof.)

b) Explain briefly why the `elif n==3:` part of this algorithm is needed. Describe *both* what would go wrong if you ran the function without this part, *and* state specifically what step of the proof would not hold (some step should fail, since it shouldn't be possible to prove that a broken function is correct...)

c) Give a concise but more helpful comment to replace the inadequate comment before the final `return` in the `strange_fib` function.

You must hand in your proof and discussion on paper via the CS 105 bin in the back of the lab (H110). If you would like to include this information in your lab files to make sure you don't lose it, note that you can copy files into folders within our cs105/Workspace (i.e., into your `basic_recursive_design` project as long as you let Eclipse know they're there by right-clicking on the project and choosing "refresh" from the menu).
You may do this work alone or in a team of two students, but make sure the work you submit has everyone's name on it.

2. Design and write the `is_a_legal_coloring` function in `is_legal.py` of the `graph_coloring` project:

a) Write, as a comment before the function, a concise description of a basic recursive design for this function (following the steps of Section 4.4 of the notes, but you can be very terse about anything that's obvious). If your definition requires the solution to some sub-problem (i.e., involves top-down design), add a test suite for this problem, and give design comments for it too.

b) Edit the `my_is_a_legal_coloring` function, to implement your design as a recursive function (with additional functions as needed by your design). Also set the `MODE` variable to `'mine'`

so your function will be called (as in the previous lab). **Include preconditions** describing the kinds of parameters that are acceptable to each function. You may use comments if the precondition can't be easily stated in Python. **Extra credit:** Add functions to test the preconditions (you don't have to write preconditions for these precondition-checking functions, though).

c) Test and debug as neccesary to make sure your algorithm works. Remember that there are several approaches to debugging that you can use:

- Try to find the smallest example that demonstrates a problem: Choose a test case that shows a problem, and add simplified versions of it (possibly simplifying in the way your recursive design does), until you find the simplest problematic case. This is usually a good way to start debugging — sometimes it provides insight by itself, and in other cases it simply reduces the amount of work for other approaches.
- Use the PyDev debugger: copy a (hopefully short!) troublesome instance into the `A_file_for_debugging.py`, double-click in the left margin to insert a "thumb tack" to the left of the instance you want to debug. Right-click on `A_file_for_debugging.py` and choose "Debug As→Python Run" to start the debugger, and then try out the steps that were demonstrated in lecture.
  When you're done, click on the "red box" icon to stop the program if it hasn't finished already, and then switch back to the PyDev perspective to keep working.
- Add print statements: Like the PyDev debugger, this approach does not fit will with the DocTest system, so start by copying a problematic example into `A_file_for_debugging.py`. Then strategically add lines to your algorithm to use `print` to display important values (possibly with a message about where in the program the print is occurring).
  When you're done, remove the `print` statements you added, so that they don't distract anyone who is reading the program.

If your work on this part of the problem makes you think of interesting "corner cases" for the test suite, go back and add them to the test suite. If your work suggests that you'll need to write additional Python functions, add them along with a complete test suite for each.

When you are done, remember to commit your final answer. You should, as always, also commit intermediate forms (these can be used for reference if you want to look back at your work; precise comments when you do the Team→Commit will help you find specific versions).