Lab 5: Advanced Recursive Design; "Enumerate and Test" Algorithm Design

In this lab, you will complete the `graph_coloring` project, first building a function to enumerate all possible (legal or illegal) colorings of a graph, and then integrating your test function into this new function to create a full "enumerate and test" algorithm (as in Question 4.9 of the course notes). Remember that 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).

Note that we strongly recommend working first on the enumeration by itself (thinking about enumerating all colorings for a set of islands), using the advanced recursive design approach discussed in Sections 4.5 and 4.6 of the course notes. Then, once that works, connect the enumeration to your previous lab's `is_a_legal_coloring` function.

Also note that there are extra-credit options for this lab (ask your lab instructor for details if you are interested). Some options involve making use of different features of Python that we'll see later in the course; some involve variations on the algorithm. But all should be attempted only after you have completed and submitted the three-color version of this problem described here ... this lab is one of the most challenging of the course, even without extra credit options.

Pre-Lab Work:

Before you do any programming for this lab, write out solutions to the following "no-borders" examples of `collect_legal_colorings` (you may use paper, the whiteboard, or with a text editor in which you can cut-and-paste).

```
>>> collect_legal_colorings('A', 'rgb', '')   # one island state
>>> collect_legal_colorings('AB', 'rgb', '')  # two islands
>>> collect_legal_colorings('ABC', 'rgb', '') # three islands
```

Then practice the "break down the answer rather than the question" design approach by dividing the answer to the "three islands" example above into several smaller pieces that you would *like* to have, for use in constructing the complete "three islands" solution. Are these pieces readily available as the solution to any simpler instance of the `collect_legal_colorings` problem? (Hint: the answer should be "no".) Then describe a *slightly different* problem for which there are instances that elicit those smaller pieces of the answer you want ... give this problem a name, think about what parameters are needed to specify an instance, and create a test suite for it.

Lab Work:

1. Add a test suite for the file `graph_coloring.py`. Start with the tests we develop together in lecture, and those in the pre-lab work given above, and then add your own if you can think of any other interesting ones. If possible, your test suite should not rely on the order in which the colorings are listed (though you may, if you wish, assume that each coloring will list the states in order), so you may want to make use of our `count_solutions` and `has_this_coloring` functions, or write additional functions of your own. You may use any graphs you like, but just

use the colors r, g, and b. As you add tests, watch to see how many of our sample answers you can break.

2. Write, as a comment before the `my_collect_legal_colorings` function in the file `graph_coloring.py`, a description of a recursive design for this function. You are allowed to use a combination of basic recursive design and top-down design (which can produce a correct, if somewhat tedious, answer), but we strongly recommend the more elegant "break down the solution" approach to advanced recursive design that is described in the course notes — this will produce a much more concise and elegant function.
   Add to the test suite as necessary — in particular, if your design suggests you will need to write additional functions, include a complete test suite for each.
3. Edit the `my_collect_legal_colorings` function to give an implementation of your design, and set `MODE='mine'` to enable it. Test and debug it as necessary, using only examples without any borders for now.
4. Adjust your algorithm to make use of your `is_a_legal_coloring` function. Test and debug as necessary.

Remember to commit your final work when you are done, and at any significant points along the way.

Extra Credit: Once you have completed the full-credit work (including testing and debugging), attempt any of the following extra credit problems:

1. Copy the `graph_coloring.py` file to `graph_coloring_many_colors.py`, and in that new file extend your existing "full credit" solution to also handle any number of colors. You are encouraged to use loops for the additions you make, while leaving the basic recursive structure in place ... if you know how to use Python's "for", that is probably the clearest and easiest way to make this extension (and you are encouraged to read up about it on Python.org if you don't know it and want an extra challenge). Alternatively, you can of course attempt to do this extension without using loops.
2. Copy the `graph_coloring.py` file to `graph_coloring_loopy.py`, and in that new file replace the recursive coloring algorithm with one that uses Python's build-in looping features such as "for" or "while", but not recursion (except of course for the the implicit recursive nature of "for" and "while"). Your solution should be correct and clearly commented, and also contain a comment discussing the relative merits you see of the recursive and loop-based approaches.