Last name ______________________

First name ______________________

LARSON—MATH 356–SAGE WORKSHEET 07
Trees!

Reminders

1. Read ahead in our textbook. We're into Chp. 2 and trees!

Coding Algorithms

1. Log in to your Sage/CoCalc account.
   (a) Start the Chrome browser.
   (b) Go to `http://cocalc.com` and sign in.
   (c) You should see an existing Project for our class. Click on that.
   (d) Click "New", call it **s07**, then click "Sage Worksheet".
   (e) For each problem number, label it in the Sage cell where the work is. So for Problem 1, the first line of the cell should be `#Problem 1`.
   (f) When you are finished with the worksheet, click "make pdf", email me the pdf (at `clarson@vcu.edu`, with a header that says **Math 356 s07 worksheet attached**).

Saving and Re-using Code

We've coded several graphs now, and have added code for functions of graph invariants and auxiliary functions and stored them in "graphs.sage". I pushed my updated version to your Handouts folder. Either copy that file to your Home directory—or add the new stuff to your own "graphs.sage" file. We'll need those functions.

2. I've updated the copy of "graphs.sage" in your Handouts folder to include what we've added in class. *Copy* the current version from Handouts to your Home directory.

3. *Load* your copy of "graphs.sage". Run: `load('graphs.sage')`.

4. Run: `my_graphs` to see what graphs we have so far. (This is partly a test to make sure your file is loaded.)

More Dijkstra's Algorithm

We now have a working algorithm that takes a weighted (connected) graph and two vertices and finds the smallest total weight path between them. Let's convert this algorithm into one that finds a shortest length path between two vertices in an unweighted graph. One way we can do this is just to add weights 1 to each edge and use our existing code!

5. Write a function `add_unit_weights(g)` that takes an unweighted graph g and adds a weight of 1 to each edge.

6. Now write a function `shortest_path(g,v,w)` that finds the length of a shortest path from v to w in graph g. We can *re-use* our Dijkstra's algorithm code!

7. Run tests on some of the graphs we've already coded to make sure that its working.

Trees

How can we test if a graph is a tree? There are many ways. In class we'll prove that a graph is a tree if and only if it is connected and every edge is a cut-edge.

We've already written a test for if a graph is connected. So writing a function that tests if every edge of a graph is all we need to do. A *cut edge* in a graph is a graph such that if it is removed the number of components of the graph increases.

8. We'll need the concept of the number of *components* of a graph. How can we code that?

9. How can we test if an edge is a cut edge?

10. Now write a function `is_tree(g)` that returns True if a graph g is a tree and False otherwise.

11. Test it!