Last name ______________________

First name ______________________

LARSON—MATH 356–SAGE WORKSHEET 11
Prufer Codes and Euler Circuits

Reminders

1. Homework #6 (h06) is due on Tuesday.
2. Check Blackboard to see if you have every grade you have submitted.
3. Read ahead in our textbook. Up next is Euler circuits (Sec. 4.1)

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 **s11**, 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 s11 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. Generate and display a `random_weighted_graph` with 5 vertices.

Recursive Functions & a Tree theorem

A **recursive** function is a function that calls itself. It must always have a *base case* so that the recursion eventually stops.

5. Now write a function `recursive_is_tree` to test whether a graph is a tree by seeing if it has a leaf, peeling it off, and repeating for the remaining graph. You'll need a base case. What will it be?

Prufer Codes

6. How can we find a *cut vertex* in a graph?

We will need the following two functions for our Prufer code construction algorithm.

7. Write a function `find_leafs(T)` to find all leafs in a tree T.

8. Write a function `find_cut_vertices(T)` to find all cut vertices in a tree T.

9. Write a function `find_prufer_vertex(T)` to find the label of the cut vertex incident to a leaf with smallest label in a tree T (assuming T's vertex labels go from 0 to $\nu-1$, for coding purposes, any list of linearly ordered labels will do).

The **Prufer code** of a labeled tree T with labels $t_1, t_1, \ldots, t_\nu$ (from 0 to $\nu - 1$, for coding purposes, any list of linearly ordered labels will do) is a list $s_1, \ldots, s_{\nu-2}$ where s_1 is the label of the cut vertex v_1 adjacent to the leaf w_1 with the smallest label in tree $T = T_1$; s_2 is is the label of the cut vertex v_2 adjacent to the leaf w_2 with the smallest label in the tree $T_2 = T_1 - w_1$ (formed by deleting leaf w_1 and its incident edge from T_1; etc.

In general s_i is is the label of the cut vertex v_i adjacent to the leaf w_i with the smallest label in the tree $T_i = T_{i-1} - w_{i-1}$ (formed by deleting leaf w_{i-1} and its incident edge from T_{i-1}.

10. Write a function `prufer_code(T)` that takes a labeled tree T (with labels from 0 to $\nu - 1$, for coding purposes, any list of linearly ordered labels will do) as input and outputs the Prufer Code of that tree.

Euler Circuits

An *Euler circuit* in a graph is a closed walk that contains every edge of the graph (so vertices may be repeated, every edge will be used exactly once, and we return to the starting vertex. (Since it is a circuit we can of course began at *any* vertex). If it does we say the graph is *Eulerian*. (Note too this is a graph *property*: either a graph is Eulerian or it is not).

Not every connected graph has an Euler circuit—the Bull graph for instance does not. How can we test if a graph has an Euler circuit? Or better, find an Euler circuit in the case that it does not?

A first observation is that if a graph has an Euler circuit every degree must be even (because the number of times our trail enters a vertex must equal the number of times it leaves that vertex). So that is a *necessary* condition. In fact, we will prove that this (together with being connected) is also a *sufficient* condition. That is, we'll prove: **A graph is Eulerian if and only of it is connected and every vertex has even degree**.

11. Write a function `is_eulerian(g)` that tests if an input graph g is Eulerian. This is either True or False, so our function should return a boolean value.

    We'll prove that we can find an Eulerian circuit in a connected graph whose vertices all have even degree. An algorithmic idea is to start at any vertex, greedily find a cycle (we can argue that it must return to that very same vertex), and then *extend* that cycle.

    How? How can we extend the cycle? If the cycle doesn't contain every graph edge, then some cycle vertex v must have adjacent edges. Delete the cycle and check the degrees of the cycle vertices. Find a new cycle, add it to the existing cycle and repeat.

    Here's how we'll break all this down (assuming our graph has an Eulerian circuit):

    (a) Write a function `find_cycle(g,v)` that takes a graph g and vertex v and greedily finds a cycle starting and ending with v.

    (b) Write a function `find_remaining_subgraph(g,C)` that takes the original graph g and the cycle C found so far, deletes the cycle edges, and returns the remaining subgraph.

    (c) Write a function `find_start_vertex(h,C)` that takes a (non-empty) graph h (subgraph of original graph g) and cycle C and returns a cycle vertex v that has positive degree in h (this will be the start vertex for extending C).

    (d) Write a function `extend_cycle(h,C,v)` that takes a (non-empty) graph h (subgraph of original graph g) and cycle C, a vertex v of C of positive degree, finds a cycle C' in h starting at v, and returns the cycle formed by gluing C and C' together.

    (e) Write a function `find_eulerian_cycle(g)` by putting these auxilliary functions ("ingredients") together.