Last name ______________________

First name ______________________

LARSON—MATH 750–SAGE WORKSHEET 07
Graph Posets

1. Create a Sage/Cocalc account.

   (a) Start the Chrome browser.

   (b) Go to `http://cocalc.com`

   (c) Login. You should see an existing Project for our class. Click on that.

   (d) Click “New”, then “Worksheets”, then call it **s07**.

   The goal of today’s lab is a first investigation of “Taylor Series” for graphs.

2. Let’s reload. Here are definitions of three common graphs and a function that returns the independence number of the input graph.Code these and test them.

```
pete = graphs.PetersenGraph()
p3 = graphs.PathGraph(3)
c5 = graphs.CycleGraph(5)

def independence_number(g):
    return g.independent_set(value_only = True)
```

3. For graph posets we’ll want a ground set consisting of graphs and the relation (probably) to be the subset relation. Code the following function and test it on the graphs we have so far.

```
def is_subgraph(g1,g2):
    v1 = g1.vertices() #list
    v2 = g2.vertices()
    e1 = g1.edges(labels=False)
    e2 = g2.edges(labels=False)
    if not Set(v1).issubset(Set(v2)):
        return False
    if not Set(e1).issubset(Set(e2)):
        return False
    else:
        return True
```

4. The “right” way to make graph posets is probably to use graphs themselves as the ground set. This is intuitively simpler, allows us to imitate anything we do for general (non-graph) posets, and actually seems to be fast. The secret is that the input graphs must be *immutable* (non-changeable, and thus *hashable*). This first idea is to consider all the connected sub-graphs with up to m edges. Our hope is that computing NP-hard invariants for these might be enough to give us a reasonable approximation.

   The following code considers all subsets of the edge set of a graph with no more than m edges, makes a graph out of them (necessarily a subgraph of the parent graph) and checks if it is connected. If so, it adds (*appends*) it to a list of graphs the function will use as the ground set of a poset and outputs the poset. The poset includes the parent graph—we'll need that in calls to compute approximations.

```
def make_m_edge_poset(g, m):
    E = g.edges(labels=False)
    Gs = [g.copy(immutable = True)]
    for i in [1..m]:
        for S in Subsets(E,i):
            h = Graph(list(S))
            if h.is_connected():
                Gs.append(h.copy(immutable = True))
    return Poset((Gs, is_subgraph))
```

5. Run this then test it with calls like `P=make_m_edge_poset(c5,2))`. Run `P`. You will learn if your poset was created properly and how many elements it has. You can see what is in this poset with calls like this:

```
for g in P:
    g.show()
```

6. Now experiment with these functions for other (small) graph. You'll have to define the graph g and corresponding poset yourself, imitating what we've done.
7. All of these functions, together with new "posetic_G" and "mobius_F" functions (corresponding to the F and G from the Mobius Inversion theorem) can be found in "posets2.sage" in the handouts folder.
8. Here is a version of the mobius_F function that does not ever calculate the summand corresponding to some entered "top"—which we will think of as the entered graph. We don't want, for instance, in our approximating calculations to compute the independence number of that!

```
def truncated_mobius_F(P, F, x, top):
    sum = 0
    for y in P:
        if y != top and P.is_lequal(y,x):
            sum = sum + P.moebius_function(y,x)*posetic_G(P, F, y)
    return sum

def taylor(F, immutable_x, m):
    P = make_m_edge_poset(immutable_x, m)
    return truncated_mobius_F(P, F, immutable_x, immutable_x)
```

9. Try these functions with calls like:
`taylor(independence_number, c5.copy(immutable=True), 1)`
—for an approximation of the independence number of $c5$ using only 1-edge subgraphs. Then try it for 2,2, and 4. *Why* are the results so terrible? Is this a bug?