Last name _____

First name _____

LARSON—MATH 756–SAGE WORKSHEET 07
Property-relations CONJECTURING

1. Login to your Sage/Cocalc account.
   (a) Start the Chrome browser.
   (b) Go to `http://cocalc.com`
   (c) Login. You created a new Project for our class. Click on that.
   (d) Click "New", then "Worksheets", then call it **s07**.
2. Run `load("independence.sage")` and then evaluate `pete.show()` and `independence_number(pete)` to see that the loaded functions are working.

Sufficient Condition Conjectures

The property-relations version of the CONJECTURING program takes as inputs a list of graphs, a list of graph properties, the index of the graph properties to investigate (produce conjectured bounds for), and whether the user wants upper or lower bounds.

3. Run `load("conjecturing.py")` to get started—this loads both versions of the conjecturing program.
4. First let's generate a sufficient condition conjecture for the property of a graph being König-Egervary. (we coded up an `is_KE` property—that's in the "independence.sage" file. We'll start with just two graphs and a few properties that are built-in or that we've coded in class.

```
objects = [pete,k_2_3]

properties = [Graph.is_bipartite, Graph.is_clique, is_KE,
is_independence_irreducible]

investigate = properties.index(is_KE)

propertyBasedConjecture(objects, properties, investigate, sufficient=True)
```

5. The conjecture you got is guaranteed to be true for every graph in `objects`. The two graphs are not equally "important' with respect to this conjecture. Why not?

6. The conjecture is of course true. So we need a non-bipartite KE graph in order to get an interesting conjecture. Let's add the *paw*. Run the following code. `show` the graph to make sure it's what you expect. Then add this graph to your "independence.sage" file for permanent future use.

```
paw = graphs.CycleGraph(3)
paw.add_vertex()
paw.add_edge(2,3)
```

7. Now update your list of `objects` (do this in a fresh Sage cell so you have a history of all your conjecturing runs (and of their evolution) and run the program again:

```
objects = [pete,k_2_3, paw]
```

8. Is the conjecture true? Can you find a counterexample by hand?

9. If you don't know the truth of a conjecture and you want to search for a counterexample, we can use `nauty_geng` to systematically generate all connected graphs with any small number of vertices. First we need to grab those conjectures. Let's re-do our last run and call the program's output "conjs":

```
objects = [pete,k_2_3, paw]

properties = [Graph.is_bipartite, Graph.is_clique, is_KE,
is_independence_irreducible]

investigate = properties.index(is_KE)

conjs=propertyBasedConjecture(objects,properties,investigate,sufficient=True)
for c in conjs:
    print c
```

10. Evaluate `conjs` to see what that is.

11. We can give the things in `conjs` names. Evaluate `conj0 = conjs[0]`. Then evaluate `conj0`.

12. `conj0` is a conjecture-object. In particular, it can act like a boolean function, with a graph as input, and output whether or not the conjecture is true for the input graph. Evaluate: `conj0(paw)`. Try it too for `pete` or any other graph.

13. This also means we can test this conjecture-function for arbitrary graphs. Maybe we can find a counterexample? Let's start by searching the connected graphs on 5 vertices:

```
for g in graphs.nauty_geng("5 -c"):
    if conj0(g) == False:
        print g.graph6_string()
        g.show()
        break
```

14. So there it is. Give it a name. Do you remember how to construct it from a g6 string? Then save it for future use. I called the graph `spaw` (for "super paw").

15. Add this graph to `objects` and repeat the last conjecturing run (in a new Sage cell of course!)

16. Hmmm... what happened?! Let's turn on the debugger to see if there is any useful information. Change the call to the conjecturing program:

```
conjs = propertyBasedConjecture(objects, properties, investigate,
sufficient=True, debug=True)
```

17. There are many ways to proceed here. Probably the best at this point is to add more properties to the `properties` input list. We need to code more properties or find more built-in ones. Here's a web page with some:
`doc.sagemath.org/html/en/reference/graphs/sage/graphs/generic_graph.html`
Scroll down until you see Graph Properties. Do you recognize any of these?

18. Let's add being eulerian, regular, planar, and chordal. Update the properties list (in a new Sage cell) and run the program again:

```
properties = [Graph.is_bipartite, Graph.is_clique, is_KE,
is_independence_irreducible, Graph.is_eulerian,
Graph.is_chordal, Graph.is_regular, Graph.is_planar]
```

19. Maybe the only graphs that are KE in the list of objects are also bipartite? What's the truth here?

20. OK, let's find more graphs that are not bipartite and that are KE to add.

```
for g in graphs.nauty_geng("5 -c"):
    if not g.is_bipartite() and is_KE(g):
        print g.graph6_string()
        g.show()
```

Lets add them all. Use the built-in `Graph` constructor. Call the graphs $g1$, $g2$ and $g3$. Add these to `objects` and run again.

```
objects = [pete,k_2_3, paw, spaw, g1, g2, g3]
```

21. What did you get? How can you proceed?

22. Just like with invariant-relations conjectures, we can add *theorems* We can keep the program from producing these by adding them as `theory`. The produced conjectures must be better for at least one input graph object than the existing `theory`—so the program can't reproduce these true conjectures. It doesn't really matter here—but will reduce clutter when there are lots of conjectures). We'll again use the `theory` paramaeter.

```
objects = [pete,k_2_3, paw, spaw, g1, g2, g3]

properties = [Graph.is_bipartite, Graph.is_clique, is_KE,
is_independence_irreducible, Graph.is_eulerian,
Graph.is_chordal, Graph.is_regular, Graph.is_planar]

investigate = properties.index(is_KE)

theorems = [Graph.is_bipartite]

conjs = propertyBasedConjecture(objects, properties, investigate,
theory = theorems, sufficient=True, debug=True)
for c in conjs:
    print c
```

Necessary Condition Conjectures

To generate necessary condition conjectures just change the "sufficient" parameter in the conjecturing program call to `False`. Make sure to remove the list of theorems. They aren't necessary conditions!

```
objects = [pete,k_2_3, paw, spaw]

properties = [Graph.is_bipartite, Graph.is_clique, is_KE,
is_independence_irreducible, Graph.is_eulerian,
Graph.is_chordal,Graph.is_regular,Graph.is_planar]

investigate = properties.index(is_KE)

conjs = propertyBasedConjecture(objects, properties, investigate,
sufficient=False, debug = True)
for c in conjs:
    print c
```

23. What did you get?

24. For each conjecture, either it is true or there is a counterexample. Can you prove or find a counterexample for either conjecture?

25. Find a counterexample to one of these conjectures, code it up, give it a name, add it to the list of `objects` and rerun the code block. Notice that the falsified conjecture is no longer produced.

26. CONJECTURE^{TM} and Play! See if you can generate a non-trivial conjecture.