Last name _____

First name _____

LARSON—MATH 756–SAGE WORKSHEET 02
Getting Started with Sage/Colcalc.

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 **s02**.

   Sage includes the `graphs` class which contains a number of *methods*. Some of these include constructors for making well-known graphs. "pete" is an arbitrarily chosen name in what follows.

2. Evaluate:

```
pete=graphs.PetersenGraph()
pete.show()
```

   There is a built-in (state-of-the-art, using Östergård's `cliquer` algorithm) maximum independent set method. The following is a modification so that we have a function which produces the cardinality of a maximum independent set.

3. To find a maximum independent set in the Petersen graph, evaluate `pete.independent_set()`. What do you get?

   To find out the options for this function, evaluate `pete.independent_set?`. One of the options shows you how to return the independence number α.

4. According to the help you read, we can use `pete.independent_set(value_only=True)`. Evaluate.

   If we wanted an independence number function we could use this. Evaluate the following code:

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

5. Now try `independence_number(pete)`. What did you get?

Graph Invariants in Sage

An *invariant* is a number associated with a graph (or a function which returns a number); it may be an integer or a real number, etc. The independence number of a graph is a graph invariant. The *order* of a graph is the number of vertices it has. The *size* of a graph is the number of edges it has. How many vertices and edges does the Petersen graph have? Evaluate `pete.order()` and `pete.size()`. These are built-in methods.

6. Find the order, size and independence number of the icosahedron graph. Use `icos=graphs.IcosahedralGraph()`.

7. Find the order, size and independence number of the dodecahedron graph. Use `dode=graphs.DodecahedralGraph()`.

8. Find the order, size and independence number of the tetrahedron graph. Use `tetra=graphs.TetrahedralGraph()`.

9. Find the order, size and independence number of the octahedral graph. Use `octa=graphs.OctahedralGraph()`.

10. Lovasz's ϑ is built-in. Evaluate: `pete.lovasz_theta()` to find that value.

Here's a list of various graph functions built-in to Sage. The invariants are scattered throughout.

11. Eigenvalues are built-in to Sage. To get the eigenvalues of the adjacency matrix of the Petersen graph, evaluate: `pete.adjacency_matrix().eigenvalues()`.

12. Eigenvalues aren't by definition graph invariants. But things like the largest eigenvalue are indeed graph invariants. So is the number of non-negative eigenvalues. (This is in fact an upper bound for the independence number of a graph). Write a function whose input is a graph and whose output is the number of non-negative eigenvalues.

13. The *Cvetkovic bound* is the minimum of the number of non-negative or non-positive eigenvalues. This too an upper bound for the independence number of a graph (which we'll prove soon enough). Write a function whose input is a graph and whose output is the Cvetkovic bound.

14. See how many invariants you can identify:

    `http://doc.sagemath.org/html/en/reference/graphs/sage/graphs/generic_graph.html`.

Graph Properties in Sage

A *property* is a boolean (or a function which returns a boolean): True or False.

15. Is the Petersen graph bipartite? Sage has a built-in test to check. Evaluate `pete.is_bipartite()`. What do you get?

16. Other built-in properties (so these are *methods*) include `is_hamiltonian` and `is_clique`. Use Sage to test if the Petersen graph is hamiltonian.

    Here's a list of *some* graph properties that are built-in in Sage:

    `http://doc.sagemath.org/html/en/reference/graphs/sage/graphs/generic_graph.html`.

Searching Graphs In Sage

Sage includes Brendon McKay's state-of-the-art graph generator `geng` and graph isomorphism checker `nauty`.

17. To generate and `show` all connected non-isomorphic graphs on 4 vertices, use:

```
for g in graphs.nauty_geng("4 -c"):
    g.show()
```

18. To generate and `show` all connected non-isomorphic graphs on 4 vertices where the independence number equals Lovasz's ϑ, use:

```
for g in graphs.nauty_geng("4 -c"):
    if g.lovasz_theta() == independence_number(g):
        g.show()
```

    The *graph-6* (or *g6*) string is another graph representation—and a great way for mathematicians to email graphs to each other. This is also an invention of McKay's.

19. To get the g6-string for the Petersen graph, evaluate: `pete.graph6_string()`.

20. We can also *construct* a graph from a g6-string. "`ShCHGD@?K?_@?@?C_GGG@??cG?G?GK_?C`" is a g6-string. Evaluate: `g = Graph("ShCHGD@?K?_@?@?C_GGG@??cG?G?GK_?C")`. Then `show` graph g. What graph is it?

21. We may want to search graphs and get output we can reuse. Do this by printing or storing g6 strings:

```
for g in graphs.nauty_geng("4 -c"):
    if g.lovasz_theta() == independence_number(g):
        print g.graph_6()
```

22. One way to make a graph is to start with a number of vertices and then for each pair of vertices n and m, flip a coin to decide whether to put an edge between those vertices. `random()` gives a random number between 0 and 1. Try this:

```
g=Graph(10)
for i in [0..9]:
    for j in [0..9]:
        if i<j and random() < 0.5:
            g.add_edge(i,j)
g.size()
g.show()
```

23. The study of *random graphs* is huge and important and was initiated in a 1959 paper of Erdős and Renyi. Sage has a built in function to do this: `graphs.RandomGNP(n,p)`, where n is the number of vertices you want, and p is the probability of an edge ($0 \leq p \leq 1$). To simulate a coin flip, use $p = 0.5$. Run the following code a few times.

```
g=graphs.RandomGNP(10,0.5)
g.size()
g.show()
```

Graph Algorithms in Sage

We will write an algorithm to find the independence number. We will write it as an ordinary function (rather than as a Graph method). The first thing we need to be able to do is to test whether the vertices S from a graph g are *independent*. This means there are no edges between the vertices in S. So we need to *search* through the edges of g.

24. Evaluate: `pete.edges(labels=False)` to get a list of the edges of the Petersen graph. Evaluate `E=pete.edges(labels=False)` to give this list the name E. Evaluate: `(0,1) in E` to test if $(0, 1)$ is the collection of edges E. Now check if $(0, 2)$ is an edge.

25. Now we'll test every pair i and j from a set of vertices S to check if S is independent. If S is independent then the test for (i, j) will be false for each possible pair.

```
def is_independent(g, S):
    E=g.edges(labels=False)
    for i in S:
        for j in S:
            if (i,j) in E:
                return False
    return True
```

26. Use `is_independent()` to test if the sets $[1, 2, 3]$ and $[1, 2]$ are independent in $p3$. Find the largest independent set S you can find in the Petersen graph. Test if it is independent.

The naive (and inefficient) way to find a largest independent set in a graph is to test every subset of vertices, check if it is independent, and then keep track of the largest one you've seen up to that point.

27. Let's try this by hand first for a smallish example. List all 8 subsets of the point set $\{0, 1, 2\}$ of $p3$.

28. Now, for each of those 8 subsets, check (by hand) if it is a independent set.

29. Of these 8 sets which is the largest independent set?

30. We'll use the subsets generator of a list to run through the subsets. Let `V=p3.vertices()`. Evaluate V to see what you have. lets see how it works. Let `L=subsets(V)`. Now try:

```
for S in L:
    print S
```

31. Now we'll write our first function to find a maximum independent set of a graph.

```
def maximum_independent_set(g):
    independent = []
    L=subsets(g.vertices())
    for S in L:
        if is_independent(g,S)==True:
            if len(S) > len(independent):
                independent = S
    return independent
```

The next big idea for finding a maximum independent set was due to Tarjan and Trojanowski in the 1970s: they noted that each vertex v of a graph is either in a maximum independent set *or* it is not. And, if v is in a maximum independent set then none of the points it is touching (that it is *adjacent* to is), called the *neighbors* of v, can be in that set.

So let's find the neighbors of a vertex v in a graph g.

```
def neighbors(g,v):
    N = []
    E = g.edges(labels=False)
    for w in g.vertices():
        if (v,w) in E or (w,v) in E:
            N.append(w)
    return N
```

Find the neighbors of vertex 0 in `pete`. Use `pete.show()` to check. Find the neighbors of vertex 9 in the Petersen graph.

So our problem of finding a maximum independent set in a graph can be *reduced* to the problem of finding a maximum independent set in two smaller subgraphs: (1) the graph formed by removing vertex v and (2) the graph formed by removing v and its neighbors. In this case, we assume that v is in the maximum independent set.

32. Now we need to form these graphs. Let g be a graph with vertex set V. Let S be any subset of V. Then you can find the graph formed by S together with all the edges that are between points of S in the original graph g with the command `g.subgraph(S)`. This is a new graph. We can give it a name, say h by `h=g.subgraph(S)`. Let `S=[2,3,5,7,8]`. Now try `h=g.subgraph(S)` and then `h.show()`.

33. Lets see the graphs that need to be formed when we apply the Tarjan-Trojanowski idea to vertex 0 of the Petersen graph. We'll need to form two sets $S1$ and $S2$ and the corresponding graphs. $S1$ is all the points of g except 0 and $S2$ is all the points of g except 0 and its neighbors.

    Try `S1=g.vertices()`, `S1.remove(0)`. Now evaluate $S1$ to see this set. Then try `h=g.subgraph(S1)` and then `h.show()`.

34. Now removing v and its neighbors will require more work:

```
S2=g.vertices()
S2.remove(0)
for w in neighbors(g,0):
    S2.remove(w)
```

    Evaluate $S2$ to see this set. Now try `h=g.subgraph(S2)` and `h.show()`.

35. To simplify things in the future, we should write a function to remove a vertex and its neighbors from a graph and produce a new graph with v and its neighbors removed.

```
def remove_vertex_and_neighbors(g,v):
    S2=g.vertices()
    S2.remove(v)
    for w in neighbors(g,v):
        S2.remove(w)
    return g.subgraph(S2)
```

    Try `remove_vertex_and_neighbors(g,0)`. How come it didn't do anything???

36. Remember what happens to V and the graph's points when we `pop()` a vertex off of the end of V. Evaluate: `g.vertices()`, then `V = g.vertices()`, then `v = V.pop()`, then `V`, and finally `g.vertices()`.