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 **s04**.

**Review**
As we go you can (should) cut and paste any needed relevant definitions from work-sheets s01, s02 and s03 into this Sage Worksheet.

**Graph Algorithms in Sage**
The first non-trivial 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.

2. So let’s find the neighbors of a vertex \( v \) in a graph \( g \). This is a built-in method. For a given graph \( g \), use \( g.neighbors(v) \).
   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.

3. 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 \( h=pete.subgraph(S) \) and then \( h.show() \).
4. 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 $S_1$ and $S_2$ and the corresponding graphs. $S_1$ is all the points of $g$ except 0 and $S_2$ is all the points of $g$ except 0 and its neighbors.

Try $S_1=\text{pete.vertices()}, S_1.remove(0)$. Now evaluate $S_1$ to see this set. Then try $h=\text{pete.subgraph}(S_1)$ and then $h.show()$.

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

$S_2=g.vertices()$
$S_2.remove(0)$
for $w$ in $g.neighbors(0)$:
    $S_2.remove(w)$

Evaluate $S_2$ to see this set. Now try $h=\text{pete.subgraph}(S_2)$ and $h.show()$.

6. 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.

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

Try $\text{remove_vertex_and_neighbors}(g, 0)$. How come it didn’t do anything???

7. Remember what happens to $V$ and the graph’s points when we $\text{pop()}$ a vertex off of the end of $V$. Evaluate: $g.vertices()$, then $V = g.vertices()$, then $v = V.pop()$, then $V$, and finally $g.vertices()$.

8. Now we are ready to write our new maximum independent set function. We will need two new vertex sets $S_1$ and $S_2$. Note that this function is recursive.
def tt_maximum_independent_set_aux(g, IndependentSet):
    V = g.vertices()
    if V == []:
        return IndependentSet
    v = V.pop()
    S1 = V
    S2 = remove_vertex_and_neighbors(g,v)
    g1 = g.subgraph(S1)
    g2 = g.subgraph(S2)
    Max1 = tt_maximum_independent_set_aux(g1, IndependentSet)
    Max2 = tt_maximum_independent_set_aux(g2, IndependentSet+[v])
    if len(Max1) > len(Max2):
        return Max1
    else:
        return Max2

Try tt_maximum_independent_set_aux(pete, []). The function is initialized with an empty set which will grow into a maximum independent set.

9. The last function takes 2 inputs. A cleaner function would take the graph itself as the only input. We can use the function we just wrote as an auxiliary function.

def tt_maximum_independent_set(g):
    return def tt_maximum_independent_set_aux(g, [])\

Now try tt_maximum_independent_set(pete), and test this function with a variety of other graphs where you know the answer. Does it work?

**Integer and Linear Programming in Sage**

The relaxation of the independent set IP gives an optimum value that is necessarily an upper bound for the vertex packing number. Here’s how we can write this in Sage in a general way.

def fractional_independence_number(g):
    p = MixedIntegerLinearProgram(maximization=True)
    x = p.new_variable(nonnegative=True)
    p.set_objective(sum(x[v] for v in g.vertices()))
    for v in g.vertices():
        p.add_constraint(x[v], max=1)
    for (u,v) in g.edge_iterator(labels=False):
        p.add_constraint(x[u] + x[v], max=1)
    return round(2*p.solve())/2.0 #hack to solve imprecision in the LP solver
fractional_independence_number(g) returns the fractional independence number $\alpha_f = \alpha_f(g)$.

**Critical Independent Sets in Sage**

We will muster our theorems to code algorithms to find:

(a) A *maximum critical independent set* (or MCIS) $I$,
(b) its neighbors $N(I)$, and
(c) the (unique) half-set $X^c$ (where $X = I \cup N(I)$),

as given in the Independence Decomposition Theorem.

The main idea for finding a MCIS is that if there is a vertex $v$ that can be set to 1 while still yielding the optimal value $\alpha_f$ then that vertex can be included in a MCIS. So we can store that vertex, remove it and its neighbors, and repeat.

```python
def find_critical_vertex(g):
    for v in g.vertices():
        g_prime = remove_vertex_and_neighbors(g,v)
        alpha_f = fractional_independence_number #just a hack
        if alpha_f(g) == alpha_f(g_prime) + 1:
            return v
```

10. Let $p3=$graphs.PathGraph(3). Find a critical independent set by hand. Then try: find_critical_vertex(p3).

Notice the code above doesn’t have a return statement in the case that the whole for loop runs but never finds a critical vertex.

11. Try: find_critical_vertex(pete).
Nothing! Secretly, under the hood, a function _always returns_ something: if its not a user-specified output then its (the special type) `None`. We'll actually _use_ that fact in the next function.

```python
def MCIS_aux(g, mcis):
    val = find_critical_vertex(g)
    if val == None:
        return mcis
    else:
        v = val
        mcis = mcis + [v]
        g_prime = remove_vertex_and_neighbors(g, v)
        return MCIS_aux(g_prime, mcis)
```

This is an _auxiliary_ function that will do all of the work—just like with the TT algorithm—it has a list of vertices as a parameter—that’s a way to transfer data to sub-calls of the function as we recurse. At the end we’ll _wrap_ it with a clearer function that just uses the input graph as a single parameter.

What the function does is test for a critical vertex and add it to the set we are maintaining of critical vertices. Then we pass it and the graph formed by removing it and its neighbors back to the function. When we can’t find any more critical vertices we return the set we’ve been building.

12. Evaluate: `MCIS_aux(p3, [])` to find a maximum critical independent set of `p3`. This initializes the list to build-on as the empty list.

OK, here is our clean Maximum Critical Independent Set (MCIS) function.

```python
def MCIS(g):
    return MCIS_aux(g, [])
```

13. Evaluate: `MCIS(p3)` to find a MCIS of `p3`. Let: `c6=graphs.CycleGraph(6)` and then find `MCIS(c6)`. Now find a MCIS of the Petersen graph.

We can now easily find the Independence Decomposition™: find a MCIS, find it’s neighbors, then find what’s left (the “half-set” `Xc`). All the “work” here is in finding the MCIS. So this function looks clean and simple. We’ll output a triple (3-tuple) of these three sets.

```python
def IDT(g):
    I = MCIS(g)
    N = []
    for v in I:
        for w in g.neighbors(v):
            if w not in N:
                N.append(w)

    Xc = []
    for v in g.vertices():
        if v not in I and v not in N:
            Xc.append(v)

    return I, N, Xc
```

14. So to find the Independence Decomposition for `p3`, evaluate: `IDT(p3)` and `IDT(c6)`.

Now find the Independence Decomposition of the Petersen graph.
Part of the Independence Decomposition Theorem says that the sum of the independence numbers of the subgraphs induced on the $X$ and $X^c$ sets is the independence number of the parent graph. We can code a function that checks this. Note here, because the IDT function outputs a triple, we need to access those 3 pieces by their indices.

```python
def check_IDT(g):
    decomposition = IDT(g)
    I = decomposition[0]
    N = decomposition[1]
    X = I + N
    Xc = decomposition[2]
    return independence_number(g) == independence_number(g.subgraph(X)) + independence_number(g.subgraph(Xc))
```

15. Now evaluate: `check_IDT(p3)` and `check_IDT(pete)`. 

16. Let’s do more substantial testing. Let’s see how it runs on a selection of random graphs. We print g6 strings here—in case there is an error, we can graph the error-causing graph and use it for testing and development.

```python
for i in [1..20]:
    g = graphs.RandomGNP(20,0.5)
    print check_IDT(g), g.graph6_string()
```

A corollary of the Independence Decomposition Theorem says that the sum of the matching numbers of the subgraphs induced on the $X$ and $X^c$ sets is the matching number of the parent graph. We can test that too.

```python
def check_matching_decomposition(g):
    decomposition = IDT(g)
    I = decomposition[0]
    N = decomposition[1]
    X = I + N
    Xc = decomposition[2]
    return matching_number(g) == matching_number(g.subgraph(X)) + matching_number(g.subgraph(Xc))
```

17. Check whether the Matching Decomposition corollary holds for `p3`, `c6`, and `pete`. 

18. Again we can do more substantial testing by generating and testing random graphs.

```python
for i in [1..20]:
    g = graphs.RandomGNP(20,0.5)
    print check_matching_decomposition(g), g.graph6_string()
```

19. What would you run to check that the IDT holds for all connected graphs on 7 vertices? 

20. What would you run to check that the Matching Decomposition corollary holds for all connected graphs on 7 vertices?