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 c23, 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 255 c23 worksheet attached).

2. **Problem** (Ramanujan) 2, 9, 16, etc. can be written (uniquely) as the sum of 2 cubes $(1^3 + 1^3, 1^3 + 2^3, 2^3 + 2^3, \text{etc.})$. Find the smallest integer which can be written as the sum of 2 cubes in 2 different ways.

A **graph** is a mathematical object consisting of dots and lines (also called vertices and edges). A **tree** is a graph that contains no cycles.

The **order** of a graph is the number of vertices it has. The **size** of a graph is the number of edges it has.

The **stability number** of a graph is the largest number of vertices in the graph that have no edges between them. So if `g=graphs.PetersenGraph()`, its stability number is 4.

**Stable Set Algorithms**

3. Here is a first function to find a maximum stable set of a graph.

```python
def naive_maximum_stable_set(g):
    stable = []
    L=Subsets(g.vertices())
    for S in L:
        if is_stable(g,S)==True:
            if len(S) > len(stable):
                stable = S
    return stable
```
The next big idea for finding a maximum stable set was due to Tarjan and Trojanowski in the 1970s: they noted that each vertex \( v \) of a graph is either in a maximum stable set or it is not. And, if \( v \) is in a maximum stable set then none of the vertices it is touching (that it is adjacent to is), called the neighbors of \( v \), can be in that set.

4. Find the neighbors of vertex 9 in the Petersen graph.

5. So our problem of finding a maximum stable set in a graph can be reduced to the problem of finding a maximum stable 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 stable set.

6. To simplify things in the future, we wrote 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)
```

7. Type in the following (recursive!) code. Do you see how it implements the Tarjan-Trojanowski idea? Notice how this code builds up from pieces coded earlier. We’re writing sophisticated code here! There are a couple of subtle issues that I will tell you more about in class.

```python
def tt_maximum_stable_set(g, StableSet):
    V = g.vertices()
    if V == []:
        return StableSet
    v = V.pop()
    S1 = V
    S2 = remove_vertex_and_neighbors(g, v)
    g1 = g.subgraph(S1)
    g2 = g.subgraph(S2)
    Max1 = tt_maximum_stable_set(g1, StableSet)
    Max2 = tt_maximum_stable_set(g2, StableSet+[v])
    if len(Max1) > len(Max2):
        return Max1
    else:
        return Max2
```

Try `tt_maximum_stable_set(g, [])`. Now test this function with a variety of other graphs where you know the answer. Does it work?
Our Own Class

The Graph class is sprawling and complicated. Now we’ll design our own class to get a feeling for the main ideas. We’ll design a general class of Dungeons and Dragons character, sample concrete character objects, methods that can be accessed by any character objects, and functions that can be used on the characters.

8. The following code defines a class called `Character`, together with a single method which constructs new Characters. What we have in mind is a thing (think of it as a person) that has a *name*.

```python
class Character:
    def __init__(self, name):
        self.name = name

Now we can create characters. Try `c1=Character("John")` to create a character `c1` with the name “John”. We can create as many as we want. Try `c2=Character("Jenn")`.

9. Our characters can’t do anything yet. So let’s add a method so they can introduce themselves.

```python
class Character:
    def __init__(self, name):
        self.name = name
    def hello(self):
        print "Hello world! I am {}.".format(self.name)
```

Evaluate. We must create new characters in order to use the newly defined abilities. Try `c3=Character("Bilbo")`. Then try `c3.hello()`.

10. Now let’s add attributes to our Characters: intelligence, health, strength, and fortitude. We will randomly initialize these as integers from 1 to 10. We would also like to be able to get a status report on these values. So we will add a `status()` method.

```python
class Character:
    def __init__(self, name):
        self.name = name
        self.intelligence=randint(1,10)
        self.health=randint(1,10)
        self.strength=randint(1,10)
        self.fortitude=randint(1,10)
    def hello(self):
        print "Hello world! I am {}.".format(self.name)
    def status(self):
        print "My intelligence is {}".format(self.intelligence)
        print "My health is {}".format(self.health)
        print "My strength is {}".format(self.strength)
        print "My fortitude is {}".format(self.fortitude)
```

Evaluate. Let `c4=Character("Frodo")`. Try `c4.status()`.
11. Things may happen to our characters. Gandalf may drink a potion that effects his intelligence. Let’s add a method so we can change a character’s initial intelligence. We must be careful never to leave the range of 1 to 10.

class Character:
    def __init__(self, name):
        self.name = name
        self.intelligence = randint(1,10)
        self.health = randint(1,10)
        self.strength = randint(1,10)
        self.fortitude = randint(1,10)

def hello(self):
    print "Hello world! I am {}.".format(self.name)

def status(self):
    print "My intelligence is {}".format(self.intelligence)
    print "My health is {}".format(self.health)
    print "My strength is {}".format(self.strength)
    print "My fortitude is {}".format(self.fortitude)

def change_intelligence(self, amount):
    new = self.intelligence + amount
    if new < 1:
        self.intelligence = 1
    elif new > 10:
        self.intelligence = 10
    else:
        self.intelligence = new

Evaluate. Let c5=Character("Gandalf").
12. Now define the following function.

```python
def drink_potion(character):
    if random() < 0.5:
        character.change_intelligence(3)
        print "I feel smarter!"
    else:
        character.change_intelligence(-3)
        print "Uh oh!"
```

Try `c5.status()`, then `drink_potion(c5)`, then `c5.status()` again.

13. Perhaps we should award our characters “points” in certain situations? We can add a points value when we initialize the character. And also add it to our status reports. And there should be a way to change the number of points. So let’s add a `change_points()` method to the `Character` class.

```python
def change_points(self, amount):
    self.points = self.points + amount
```

Evaluate. Let `c6=Character("LittleJohn")`. Then try `c6.status()`.

14. Our characters may have to fight trolls. Define the following function.

```python
def fight_troll(character):
    if character.health > 5 and character.strength > 5:
        character.change_points(5)
        print "I have defeated the troll!"
    elif character.health < 4 or character.strength < 4:
        character.change_points(-5)
        print "You have defeated me this time!"
    else:
        print "Run away!"
```

15. Oh oh. LittleJohn has encountered a troll. Let’s see what happens. Evaluate `fight_troll(c6)`. Then check his status with `c6.status()`.

You see these classes, objects and methods can get very interesting!