Last name ______________________

First name ______________________

LARSON—MATH 356—LAB WORKSHEET 03
Graphs!

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 **s03**, 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 356 s03 worksheet attached**).

   More Lists and Control

2. You can use *list comprehension* to apply a function to all the elements of a list. Evaluate `[abs(x) for x in [-1,2,-3]]`.

   It is often useful to manipulate and/or create tuples.

3. Here is a function that takes 2 numbers as inputs and returns a tuple (pair) with twice numbers.

```
def tuple_test(x,y):
    t=(2*x,2*y)
    return t
```

   Evaluate. Let `s=tuple_test(3,4)`. Evaluate.

4. Now write a function `pair_square(x,y)` that takes any numbers x and y and returns a tuple (pair) that is the squares of these numbers.

   A *for loop* is what we use when we want our code to run through every item x in a list.

5. Evaluate and test the following function. What do you think this function will do?

```
def for_loop_test():
    for i in [0..5]:
        print(i^2)
```

6. Modify your code to print the squares of the integers from 5 to 9. How did you change it?

7. Modify the code to print just the squares of 2, 5, 7 , 9, and 23. How did you change it?

8. The function `list_evens(n)` returns all the even integers from 0 to n. Evaluate and test the following code.

```
def list_evens(n):
    M=[]
    for x in [0..n]:
        if x%2==0:
            M.append(x)
    return M
```

9. Write a function list_primes(n) that **returns a list** of all the primes up to n. Use Sage's built-in `is_prime(n)` function. Test it.

A *while loop* runs a block of code while a condition is still satisfied.

10. Type in and evaluate the function `while_test()`. What do you think this function will do?

```
def while_test():
   i=0
   while i<5:
       print(i^2)
       i=i+1
```

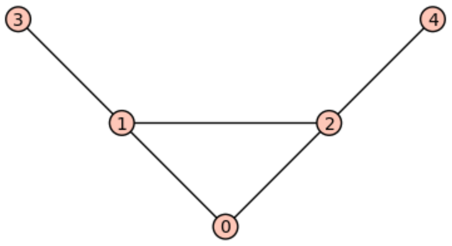
11. A common way to use a while loop is in a test where you don't know precisely when the test condition will be met. Here we will write a function that prints the first n primes. We will use a *counter* to keep track of how many we have so far. The first version works and the 2^{nd} runs forever. Why?

```
def print_first_n_primes2(n):
    count = 0
    current_number = 0
    while count < n:
        if is_prime(current_number) == True:
            print(current_number)
            count = count + 1
        current_number = current_number + 1
```

```
def print_first_n_primes(n):
    count = 0
    current_number = 0
    while count < n:
        if is_prime(current_number) == True:
            print(current_number)
            count = count + 1
    current_number = current_number + 1
```

Review: Making a Graph from Scratch.

12. Now use `Graph()`, and `add_edge()` to make the *bull*.



Start by letting `bull=Graph(5)` to get a graph with 5 vertices and no edges. Now add the edges that you see in the diagram of the bull using `bull.add_edge()`. Remember that the layout of the graph doesn't matter—only that it has the same edges. When you are done you can view it with `bull.show()`.

13. Use Sage to find an incidence matrix for this graph.

14. Use Sage to find an adjacency matrix for this graph.

15. Many many graphs are already pre-coded (built in) to Sage/Cocalc. How can we access the built-in Petersen Graph?

16. Use Sage to find an incidence matrix for this graph.

17. Use Sage to find an adjacency matrix for this graph.

Graph Methods in Sage

Recall that 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()`.

18. Find the order and size of the cube graph. Use `cube=graphs.CubeGraph()`

19. Find the order and size of the icosahedron graph. Use `icos=graphs.IcosahedralGraph()`

20. Find the order and size of the dodecahedron graph. Use `dode=graphs.DodecahedralGraph()`

21. Find the order and size of the tetrahedron graph. Use `tetra=graphs.TetrahedralGraph()`

22. Find the order and size of the octahedral graph. Use `octa=graphs.OctahedralGraph()`

Recursive Functions

A **recursive** function is a function that calls itself. It must always have a *base case* so that the recursion eventually stops.

23. Here is an example of a recursive definition of the *factorial* function. The base case here is the case where the input is 0 or 1.

```
def factorial(n):
   if n==0 or n==1:
       return 1
   else:
       return n*factorial(n-1)
```

Now try `factorial(0)`, `factorial(1)`, `factorial(2)`, `factorial(3)`, and `factorial(10)`.

24. The Fibonacci numbers are: 1,1,2,3,5,8,13,21..., where the next number is the sum of the previous two numbers. Write a recursive function function that computes the n^{th} Fibonacci number.