Computer Science 112 - Fundamentals of Programming II
Lab Project 10
Due on Github 11:59pm Monday 6 December

Part I: Directed Acyclic Graphs

Create file **dag.py** containing a class **`DAG`**. Here's the test I used to get started, without output shown in blue italic.

```
print('Part I: DAG basics ------------')
dag = DAG(('a', 'b', 'c'), (('a', 'b'), ('a', 'c')))
print(len(dag))
```
3
```
print(dag)
```
{'a': ['b', 'c'], 'b': [], 'c': []}

As the output suggests, I found it easiest to back my DAG with an ordinary Python dictionary.

Part II: BFS and DFS

Using the figures on Slide 33:5 as a guide, add **`BFS`** and **`DFS`** methods to your **`DAG`** class. In my test suite I just labeled the vertices according to the BFS order:

```
print('\nPart II: search ------------')
dag = DAG((1, 2, 3, 4, 5, 6, 7, 8), ((1,2), (1,3), (1,4), (2,5), (2,6), (4,7), (6,8)))
print(dag)
```
{1: [2, 3, 4], 2: [5, 6], 3: [], 4: [7], 5: [], 6: [8], 7: [], 8: []}
```
print('BFS: ' + str(dag.bfs()))
```
BFS: [1, 2, 3, 4, 5, 6, 7, 8]
```
print('DFS: ' + str(dag.dfs()))
```
DFS: [1, 2, 5, 6, 8, 3, 4, 7]

Here again I suggest a keep-it-simple approach: instead of importing your Queue class or using the fancy ArrayStack and LinkedQueue structures shown in the slides, follow the approach in the Tree Search lecture and use a built-in Python list for both BFS and DFS, the only difference being where you put the neighbors (successors).

Part III: Topological sort

In this exercise we'll implement the Topological Sort algorithm from slide 33:13. To make the exercise relevant, we'll use the Computer Science major course graph in slide 32:16: 'CSCI111', 'MATH121', etc. To keep things simple I just used MATH121 as the math requirement, and pretended that both CSCI312 and CSCI313 were required for the major.

Here's what my output looked like for this part:

```
Part III: Topological sort -----------
['CSCI111', 'CSCI112', 'CSCI210', 'CSCI209', 'MATH121', 'CSCI312', 'CSCI211', 'CSCI313']
```

For the crucial feature of this algorithm – marking the nodes – I found it easier to maintain a sets of unvisited courses, rather than coming up some kind of complicated node structure to represent courses. Likewise, I found it easier to copy/paste/modify the code from the little **`dfs()`** function in slide 33:13, rather than trying to use my existing **`dfs()`** method for topo sort. This kind of trick – taking an algorithm that's easy to run on paper ("talking the talk") and making it work on the computer ("walking the walk") in a slightly different way – is a vital but non-intuitive skill that you will get better at as you continue your career in computer science. As usual, our university's motto *Omnia autem probate* (test everything!) will help you: if you're adding more than one or two lines of code without testing your intermediate results, you're setting yourself up for frustration.

Part IV: Undirected Graphs

Create a new file **ug.py** containing a class **`UG`**. Then use your **`UG`** class to instantiate the undirected graph in slide 33:15. Although this looks like a lot of work, a little inspection will show that there's a redundancy in the distance matrix shown in the slide: the distance between A and B is the same as the distance between B and A, etc. You can exploit this redundancy to simplify the code for building the graph. Here's my **`UG`** constructor call to get you started:

```
ug = UG(('A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I',  'J'),
        ( ('A','D', 462),
          ('A','F', 451),
          ... # several lines here!
          ('G','I', 389) ))
```

Once you've written your constructor, the next step as usual is to write your **`__str__`** method, which should return a string representation of the distance matrix shown in the slide, including column and row labels. My **`__str__`** method worked by first creating an *N*x*N* array of zeros with a doubly-nested list comprehension, then filling in the distances, then returning a formatted version of the matrix. *Hint*: producing this matrix will be easier if you go back to your constructor and have each dictionary entry be another dictionary; e.g., the entry for **`A`** ends up being **`{'D': 462, 'F': 451, 'H': 370}`**. Self-connections are always zero, and unspecified connections are set to **`math.inf`**. (which gets reported as -). Just make sure that your constructor works in a general way, as shown in the code above. For full credit, have the **`__str__`** method format its returned string in a nice readable way, just like the figure in the slides!

(*Continued next page ...*)

Part V: Floyd's Algorithm

To finish up, add a method **`floyd()`** that uses Floyd's Algorithm to produce a matrix of shortest distances like the one at the top of slide 33:16. To do this, I started by factoring the distance-matrix-building code from my **`__str__`** method into a helper method. Next I implemented Floyd's algorithm as shown in slide 33:18. Finally, I re-used the string-formatting code from my **`__str__`** method to report the output of the **`floyd()`** method in a nice, readable way.