What’s it all about?

Given a graph representing currencies and the exchange rates between them, we want to find a cycle of currencies so that we could trade currencies around the cycle, ending up with more money than we had at the start.
Do this how???

We can do this by creating a new graph with edge weights that are a function of the exchange rates, then using the Bellman-Ford single source shortest path algorithm to detect a negative cycle. This negative cycle will correspond to a profitable trading cycle in the original graph.
Bellman-Ford

BELLMAN-FORD(G, w, s)
1 INITIALIZE-SINGLE-SOURCE(G, s)
2 for i = 1 to |G.V| - 1
3 for each edge (u,v) ∈ G.E
4 RELAX(u,v,w)
5 for each edge (u,v) ∈ G.E
6 if v.d > u.d + w(u,v)
7 return FALSE
8 return TRUE
Algorithm Modifications

Graph weights have to be converted to logs and made negative.

The output needs to be the predecessor nodes for the cycle.
Digraph object (partial)

class Digraph:
    def __init__(self):
        self.nodes = {}
        self.edges = defaultdict(dict)

    def add_node(self, identity, attributes=None):
        self.nodes[identity] = normalize_attributes(attributes)

    def add_edge(self, identity1, identity2, attributes=None):
        attributes = normalize_attributes(attributes)
        self.ensure_node(identity1)
        self.ensure_node(identity2)
        self.edges[identity1][identity2] = attributes

    def get_nodes(self):
        return self.nodes.keys()

    def get_edges(self):
        return [(a, b) for a in self.edges for b in self.edges[a]]

    def attributes_of(self, identity, identity2=None):
        if identity2 == None:
            return self.nodes[identity]
        return self.edges[identity][identity2]
Starter Code
You must use the supplied directed graph object, Digraph.
Study the code and answer these questions:

1. What is the *attribute* dictionary?
2. What can it be used to store?
What does this code do?

```
dag = digraph()
...
for v in dag.neighbors(node):
    adj_attr = dag.attributes_of(v)
    if adj_attr['color'] == 0:
        dag.attributes_of(v).update({‘parent’: node, ‘edge’: t})
    else:
        if adj_attr[‘parent’] == None:
            dag.attributes_of(v).update({‘parent’: node})
        if adj_attr[‘color’] == 1:
            dag.attributes_of(v).update({‘edge’: ‘b’})
```

Partial DFS code...
And how can you use it?

Based on what you figured out about attributes, how can you use them specifically in the arbitrage problem?

10 points – use Task-1 worksheet

To get credit for this, you need to make sure one of us checks your work by the end of class today.
Suggested Helpers

Study the code in the currency_arbitrage.py file.

“Fill in” the function

```python
find_profitable_trading_cycle(filename)
```

using appropriate helper functions. This means you also have to “fill in” the helper functions.

60 points – use Task-2 worksheet

To get credit for this, you need to make sure one of us checks your work by the end of class today.
Complexity Analysis

What is the theoretical complexity of this problem?
• How does your “implementation” compare?
• How could you improve its complexity?
• How will you show the complexity visually (i.e. graphing it – what are the independent and dependent variables, their units, and on what axes are they shown)?

10 points – use Task-3 worksheet

To get credit for this, you need to make sure one of us checks your work by the end of class today.
Grading

This in-class work is worth 80 points, which will be the value of this assignment.

You can get up to 20 points extra credit if you do the following:

1. Implement what we’ve worked on in class in Python3.
2. Make multiple commits of your work to GitHub.
3. Submit a zip file to Canvas that contains:
   • currency_arbitrage.py
   • any additional files it imports
   • a .pdf file that discusses complexity, including a graph of your implementation with graph sizes of 5, 20, 30, 50, 100, 120, 140, 160, 180, 200, 220, 240, and 260 (use the supplied generate_arbitrage_input.py for testing and timing)

If you choose to do this, the hard deadline is Saturday, Dec 9 at 11:59pm
Sample running times

```python
>>> find_profitable_trading_cycle('exchange_rates_5-0.txt')
Time: 0.0010583880000001322 seconds
>>> find_profitable_trading_cycle('exchange_rates_100-0.txt')
Time: 1.08585365500000002 seconds
Product of exchange rates: 1.02843175739944
['c62', 'cc', 'ce']
>>> find_profitable_trading_cycle('exchange_rates_200-0.txt')
Time: 8.620370018 seconds
Product of exchange rates: 1.055204996024095
['cc5', 'cc3', 'cf', 'cc4']
>>> find_profitable_trading_cycle('exchange_rates_260-0.txt')
Time: 19.129474692000002 seconds
Product of exchange rates: 1.0525973751607889
['cf', 'cff', 'cdc', 'cf3', 'cfd', 'cfe', 'cf1']
>>>```
Image Credits

arbitrage.png: http://www.quantumforquants.org/quantum-computing/qa-arbitrage/