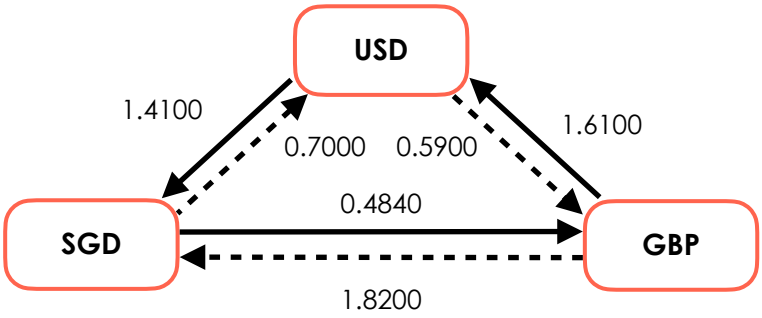
Advanced: PSET 4

Arbitrage

In this problem, you will be implementing a program to detect and extract negative cycles in a graph of exchange rates, a strategy known as arbitrage!

How does arbitrage work?

Imagine the following exchange rates between **USD**, **GBP** and **SGD**:



Each edge represents the conversion rate for one unit of currency to another. For instance, **1 SGD** will become **0.700 USD.**

In the graph above, you will notice that if we follow the following paths of conversion:

(USD –> SGD –> GBP –> SGD),

assuming we start off at 1 USD, we will come back with 1.0987 USD:

(1 USD –> 1.4100 SGD –> 0.68244GBP –> 1.0987 USD)

This is what arbitrage is: leveraging imbalances in markets to make profits.

How to detect arbitrage?

A. Mathematics behind arbitrage

If you wish to only learn about the steps to determine arbitrage, you may skip this section.

In arbitrage, we are multiplying currencies. For instance, for the above example where we followed the conversions: **USD –> SGD –> GBP –> SGD**, all we were doing was really following each edge weight:

```
Final Result (in USD) = 1 * 14100 * 0.4840 * 1.6100 = 1.0987
```

If we add a log to both sides:

```
log(1 * 14100 * 0.4840 * 1.6100) = log(1.0987)
```

The left hand side is essentially equivalent to:

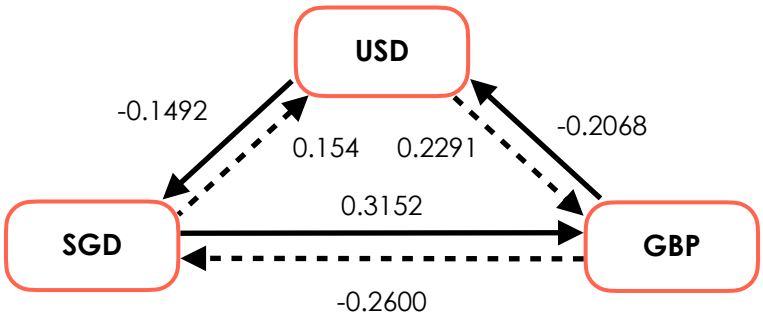
`log(1) + log(14100) + log(0.4840) + log(1.6100) = log(1.0987)`

It is important to now note that for $\log_{10}$**x** (base 10), for all values of **x greater than 1**, the value of **logx** will be **positive**, and for all values of **x less than 1**, the value of **logx** will be **negative**. Thus, essentially, when we find arbitrage and apply the log function to it, **result** should be positive.

We can make use of this property to treat arbitrage as a negative cycle by multiplying all values with a negative sign:

`−log(1) − log(14100) − log(0.4840) − log(1.6100) = −log(1.0987)`

Now, our final result will only ever be negative if our conversion has a final value greater than 1 (arbitrage). Let's take a look at what our graph looks like now:



As you can see, following the same conversion (USD –> SGD –> GBP –> SGD):

`− 0.1492 − 0.2068 + 0.3152 = − 0.0408` (Negative Cycle)

A. Steps to perform arbitrage

Let's now look at how arbitrage is computed using the concept of graphs. Using the above graph as an example, when we convert it to an adjacency list, it will look like the following:

```
{
    "USD": [("USD", "SGD", 1.4100), ("USD", "GBP", 0.5900)],
    "SGD": [("SGD", "USD", 0.7000), ("SGD", "GBP", 0.4840)],
    "GBP": [("GBP", "USD", 1.6100), ("GBP", "SGD", 1.8200)],
}
```

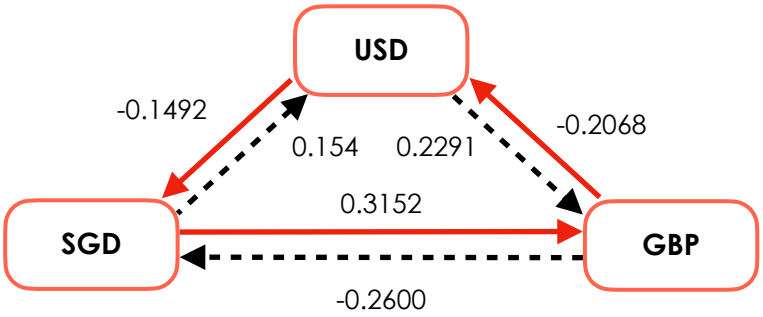
Note: In this problem set, we represent edges as a tuple with the first element as the source vertex, second element as the destination vertex and third element as the weight of that edge.

Step 1: Convert graph to negative log graph

All we need to do is apply the negative log (base 10) function to every edge weight and we will receive the new graph:

```
{
    'USD': [('USD', 'SGD', -0.14921), ('USD', 'GBP', 0.22914)],
    'SGD': [('SGD', 'USD', 0.154901), ('SGD', 'GBP', 0.31515)],
    'GBP': [('GBP', 'USD', -0.20682), ('GBP', 'SGD', -0.26007)]
}
```

Step 2: Perform Bellman-Ford algorithm to detect and retrieve negative cycle



The next step is to perform the Bellman-Ford algorithm, and compute the **shortest path tree**. The shortest path tree is essentially the edges contained in the **edgeTo** list.

Hint: The shortest path tree is considered after relaxing E edges V times!

Step 3: Search the shortest path tree for negative cycle

The last step is to employ a cycle search algorithm (recall Lesson 3) to search for a cycle in the shortest path tree. Due to the nature of the Bellman-Ford algorithm, if there is a cycle in the **shortest path tree**, it should be a **negative cycle**. Then, just return all edges involved in this cycle and that is your arbitrage opportunity!

Hint: To find a cycle in a graph / shortest path tree, keep track of which vertices are on the current recursion stack, and if you visit one of these vertices in your DFS traversal, then a cycle has been found.

A. Starting Code

`exchangeRates`

- The test cases for this problem have been provided to you in a file called exchangeRates.py

main

```
def main():
    try:
        n = int(argv[1]) - 1
    except:
        print("No CLA inputted, defaulting to exchangeRate 1")
        n = 0

    print(arbitrage(exchangeRates[n]))
```

- In main, we provide the option for you to input a the test case as a command line argument. For instance, running "**python arbitrage 1**" will use the first test case in the exchangeRates list
- If no command line argument is inputted, the first test case will be used
- The arbitrage function (which you will implement) will then be run and the result printed on the test case selected

arbitrage

```
def arbitrage(exchangeRates):

    # CONVERT EXCHANGE RATES TO LOG GRAPH
    logGraph = convertToLogGraph(exchangeRates)

    # RETRIEVES SHORTEST PATH TREE / NONE THROUGH BELLMAN FORD ALGO
    shortestPathTree = bellmanFord(logGraph)

    # RETRIEVES EDGES INVOLVED IN CYCLE IF THERE IS ONE
    return searchForCycle(shortestPathTree)
```

- In arbitrage, we implement the sequence of helper functions, and your job will be to implement these helper functions
- As explained above:
  - We first convert our exchangeRates into a new graph after applying the negative log function to the weights (**convertToLogGraph**)
  - Next, compute the shortest path tree, if there is a negative cycle, using the Bellman-Ford algorithm (**bellmanFord**)
  - Lastly, we retrieve the cycle from the shortest path tree (**searchForCycle**)

Note:

- The test cases are not representative of real life exchange rates and have been hardcoded for you purely for assignment purposes
- For all the test cases, you may assume that there is **at maximum one negative cycle among all the exchange rates** (one arbitrage opportunity)

B. Your Task

Implement the following functions:

1. `convertToLogGraph`

- This function takes in a **graph of exchange rates**, and returns an adjacency list with the same edges, but with **every weight having the negative $\log_{10}$ function applied**

For instance, passing in the following exchange rates into the function:

```
{
    "USD": [("USD", "SGD", 1.4100), ("USD", "GBP", 0.5900)],
    "SGD": [("SGD", "USD", 0.7000), ("SGD", "GBP", 0.4840)],
    "GBP": [("GBP", "USD", 1.6100), ("GBP", "SGD", 1.8200)],
}
```

will **return** the following:

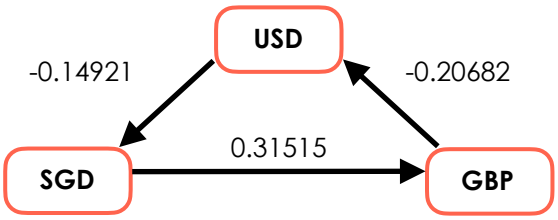
```
{
    'USD': [('USD', 'SGD', -0.14921), ('USD', 'GBP', 0.22914)],
    'SGD': [('SGD', 'USD', 0.154901), ('SGD', 'GBP', 0.31515)],
    'GBP': [('GBP', 'USD', -0.20682), ('GBP', 'SGD', -0.26007)]
}
```

2. `bellmanFord`

- This function will take in the logGraph (output of **`convertToLogGraph`** function), and returns the shortest path tree
- The shortest path tree is essentially an adjacent list made up of the edges involved in the **edgeTo** list.

For instance, passing the above graph into the bellmanFord function, the output will look like so:

```
{
  'GBP': [('GBP', 'USD', -0.20682)],
  'USD': [('USD', 'SGD', -0.14921)],
  'SGD': [('SGD', 'GBP', 0.31515)]
}
```



As you can see, many edges have been removed through the Bellman-Ford algorithm, producing the above "**shortest path tree**".

3. `searchForCycle`

- This function takes in the shortest path tree (output of bellmanFord), and returns a list containing edges (excluding the weights) involved in the **negative cycle**.
- For instance, passing in the above **shortest path tree** into this function will return the following: **`[('SGD', 'GBP'), ('GBP', 'USD'), ('USD', 'SGD')]`**

`Note:` The order in which the edges are added to the list is not important.

C. Sample Output

```
python arbitrage.py 1
[('SGD', 'GBP'), ('GBP', 'USD'), ('USD', 'SGD')]

python arbitrage.py 2
[('EUR', 'JPY'), ('JPY', 'USD'), ('USD', 'EUR')]

python arbitrage.py 3
[('SGD', 'GBP'), ('GBP', 'JPY'), ('JPY', 'SGD')]

python arbitrage.py 4
[]

python arbitrage.py 5
[]

python arbitrage.py 6
[]

python arbitrage.py 7
[('USD', 'GBP'), ('GBP', 'SGD'), ('SGD', 'USD')]

python arbitrage.py 8
[('AUD', 'USD'), ('USD', 'JPY'), ('JPY', 'MYR'), ('MYR', 'AUD')]
```

D. Submission

To test your code, run the following command:

```
python utils/test.py
```

To submit your code, run the following command:

```
python utils/submit.py
```

A report.txt should be generated for you to view your results

---