1. Quiz: Eulerian Path

A Eulerian path starting at Javier will end on Marcie since they both have odd degrees in the friend network.

1. Quiz: Counting Eulerian Paths

Six different paths to cover all of the nodes and connections.

1. Quiz: Create Graph With Eulerian Tour

import itertools

def create\_tour(nodes):

node\_connections = itertools.combinations(nodes, 2)

connection\_list = list()

for item in node\_connections:

if item[1] == nodes[nodes.index(item[0]) + 1]:

connection\_list.append(item)

connection\_list.append(tuple([nodes[-1], nodes[0]]))

return connection\_list

1. Quiz: Representing A Graph

An advantage of representing a graph as a list of tuples is the conciseness of each entry being simply the two nodes that are connected. This may save on space as compared to an adjacency matrix. A disadvantage is that the nodes are ordered in each tuple, which may imply direction even in an undirected graph.

1. Quiz: Naive Multiplication Algorithm

naive(63, 12) is at some point x = 20, y = 12, z = ?

z = 43 \* 12 = 516

1. Quiz: Recursive Naïve

17 additions for rec\_naive(17, 6)

1. Quiz: Russian Multiplication Algorithm

We’re computing russian(63, 12). At some point during the execution, we have x = 7 and z = 84. What is y at this moment?

ab = xy + z

63 \* 12 = 7 \* y + 84

x = 63 = 111111, y = 12 = 1100; x = 7 = 111, y = 1100000 = 96

1. Quiz: Clique

12

1. Quiz: General Clique

def count(n):

return 2 + sum(range(n + 1)) if n > 0 else 2

def clique(n):

print("in a clique...")

for j in range(n):

for i in range(j):

print(i, "is friends with", j)

print(count(4)) # 12

print(count(5)) # 17

1. Quiz: Challenge Find Eulerian Tour

def create\_node\_list(graph):

"""

Creates a list of all the unique nodes found inside graph.

:param graph:

:return:

"""

node\_list = list()

for item in graph:

for node in item:

if node not in node\_list:

node\_list.append(node)

node\_list.sort()

return node\_list

def find\_eulerian\_tour(graph):

"""

Finds an Eulerian Tour within the given graph.

:param graph:

:return:

"""

# It is not perfect, as it should be able to find the tour no matter the starting point on a valid graph, but the

# idea was if for whatever reason a starting node did not work, it would try the next one in the graph and hopefully

# find a valid tour. The flaw comes with having an optimized for loop always taking the first traversable edge,

# rather than building a queue of possible paths and iterating through them. Still going to upload this version, but

# maybe tackle it properly in the future.

tour = list()

# Loops that will change the starting node in the tour.

for edge in graph:

for node in edge:

# Copy the graph and remove the edge in use.

temp\_graph = list(graph) # graph.copy() # changed for Udacity interpreter

# node is where we start!

tour.append(node)

# Add second node to tour.

if node == edge[0]:

location = edge[1]

tour.append(location)

else:

location = edge[0]

tour.append(location)

temp\_graph.remove(edge)

last\_size = len(temp\_graph) + 1 # last\_size ensures while loop does not iterate over the same nodes.

while 0 < len(temp\_graph) < last\_size:

last\_size = len(temp\_graph) # helped in test 2, first try with node 0 got stuck with last 4 edges.

for temp\_edge in temp\_graph:

if location in temp\_edge:

if location == temp\_edge[0]:

location = temp\_edge[1]

tour.append(location)

else:

location = temp\_edge[0]

tour.append(location)

temp\_graph.remove(temp\_edge)

if not temp\_graph:

return tour

else:

tour = list() # tour.clear() # changed for Udacity interpreter

return tour