Week 03

Seth Childers

1 3-1 Exercise 01

Problem:
a. Give an example of an algorithm that should not be considered an application of the brute-force approach.
b. Give an example of a problem that cannot be solved by a brute-force algorithm.

Code written in Python

```
########################################################################
# Exercises 3.1 - #01
# a. Give an example of an algorithm that should not be
# considered an application of the brute-force approach.
#
# A binary search is an example of an algorithm that should
# not be considered for a brute-force approach. This is
# because the binary search is used to find a single item
# that matches what you are looking for, where going through
# each element would be inefficient and unnecessary.
#
# b. Give an example of a problem that cannot be solved
# by a brute-force algorithm.
#
# A problem that cannot be solved by a brute-force algorithm
# is one that solves for the next best set of moves for a
# particular card or board game. This is not feasible because
# certain games have so many possible plays each move, and calculating
# the probability of each opponent's most likely moves means that
# the possibilities for a set of moves is incredibly exponential.
# This is possible for simpler games, but machine learning is necessary
# for the more complex games in order to cut down on computations.
########################################################################
```

2 3-1 Exercise 08

Problem: Sort the list E, X, A, M, P, L, E in alphabetical order by selection sort

Code written in Python

```
##########################################################################
# Exercises 3.1 - #08
#
# Sort the list E, X, A, M, P, L, E in alphabetical order by selection sort
##########################################################################
import sys

# Got help from https://www.geeksforgeeks.org/selection-sort/
def selectionSort(unsorted):
    for i in range(len(unsorted)):
        nextIndex = i
        for j in range(i+1, len(unsorted)):
            if unsorted[nextIndex] > unsorted[j]:
                nextIndex = j

        unsorted[i], unsorted[nextIndex] = unsorted[nextIndex], unsorted[i]

    print("Sorted array")
    for i in range(len(unsorted)):
        print("{}".format(unsorted[i] ))

unsorted = ['E', 'X', 'A', 'M', 'P', 'L', 'E']
selectionSort(unsorted)
```

Results

```
Sorted array
A
E
E
L
M
P
X
```

3 3.4 Exercise 06

Problem: Odd pie fight - There are n>= 3 people positioned on a field (Euclidean plane) so that each has a unique nearest neighbor. Each person has a cream pie. At a signal, everbody hurls his or her pie at the nearest neighbor. Assuming that n is odd and that nobody can miss his or her target, true or false: There always remains at least one person not hit by a pie.

Code written in Python

```
###############################################################################
# Exercises 3.4 - #06
#
# Odd pie fight - There are n>= 3 people positioned on a field (Euclidean
    plane)
# so that each has a unique nearest neighbor. Each person has a cream pie.
# At a signal, everbody hurls his or her pie at the nearest neighbor. Assuming
# that n is odd and that nobody can miss his or her target, true or false:
# There always remains at least one person not hit by a pie.
#
# True, no matter what there will always be at least one person not hit by a
# pie because the two people that have the smallest distance between them will
# hit each other, instead of hitting other people, which will throw off the
# possibility to hit everyone.
###############################################################################
```

4 3-5 EXERCISE 04

Problem:

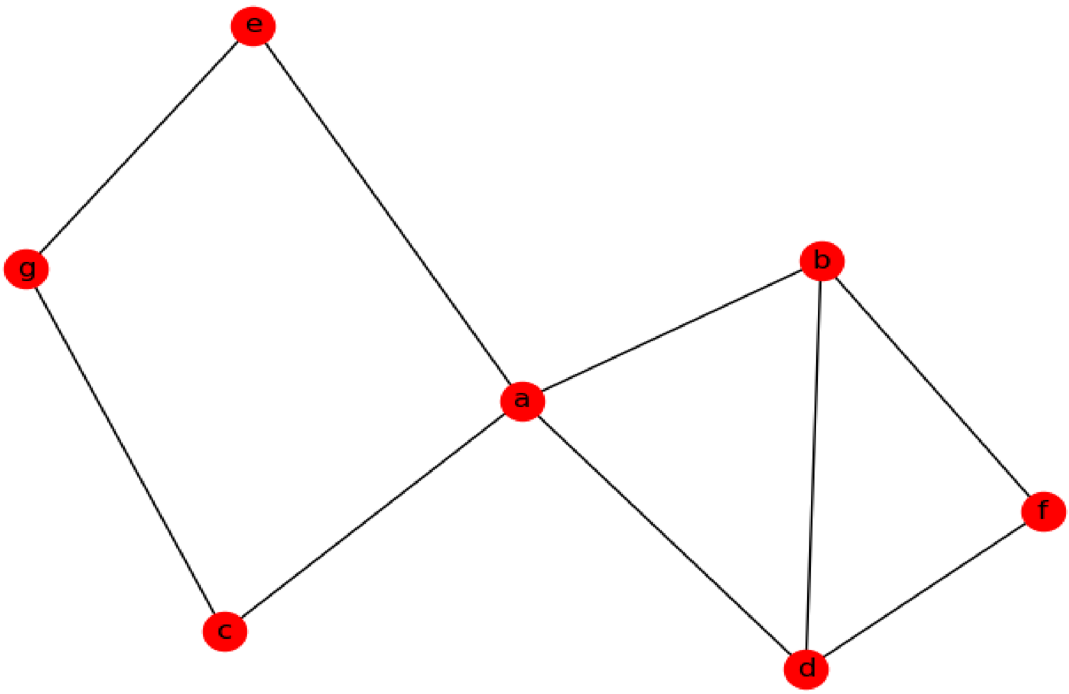
Code written in Python

```
###########################################################################
# Exercises 3.5 - #04
#
# Traverse the graph of Problem 1 by breadth-first search and construct
# the corresponding breadth-first search tree. Start the traversal at
# vertex 'a' and resolve ties by the vertex alphabetical order.
#
# Problem 1 graph:
# f --- b    c --- g
# \   / \   /     /
#   d --- a ---- e
###########################################################################
import matplotlib as plt
import networkx as nx

def main():
    # make the graph
    graph = createGraph()
    # print the graph info
    print(nx.info(graph))
    # draw and show the graph
    drawGraph(graph)
    # print out the breadth first search info
    print(list(breadthFirst(graph)))

def breadthFirst(graph):
    return nx.bfs_edges(graph, 'a')

def drawGraph(graph):
    # draw the graph
    nx.draw(graph, with_labels=True)
    # show the graph
    plt.pyplot.savefig('3-5_graph.png')

def createGraph():
    G = nx.Graph()
    G.add_nodes_from(['a', 'b', 'c', 'd', 'e', 'f', 'g'])
    G.add_edges_from([('a', 'c'), ('a', 'b'), ('a', 'e'), ('a', 'd')])
    G.add_edges_from([('d', 'f'), ('d', 'b'), ('f', 'b')])
    G.add_edges_from([('c', 'g'), ('a', 'e'), ('e', 'g')])

    return G
```

```python
if __name__ == "__main__":
    main()
```

Results

```
Type: Graph
Number of nodes: 7
Number of edges: 9
Average degree: 2.5714
[('a', 'c'), ('a', 'b'), ('a', 'e'), ('a', 'd'), ('c', 'g'), ('b', 'f')]
```



5 3-5 EXERCISE 08

Problem: A graph is said to be bipartite if all its vertices can be partitioned into two disjoint subsets X and Y so that every edge connects a vertex in X with a vertex in Y. (One can also say that a graph is bipartite if its vertices can be colored in two colors so that every edge has its vertices colored in different colors; such graphs are also called 2-colorable.)
a. Design a DFS-based algorithm for checking whether a graph is bipartite.
b. Design a BFS-based algorithm for checking whether a graph is bipartite.

Code written in Python

```
#########################################################################
# Exercises 3.5 - #08
#
# A graph is said to be bipartite if all its vertices can be partitioned
# into two disjoint subsets X and Y so that every edge connects a vertex
# in X with a vertex in Y. (One can also say that a graph is bipartite if
# its vertices can be colored in two colors so that every edge has its
# vertices colored in different colors; such graphs are also called
# 2-colorable.)
# For example, graph (i) is bipartite while graph (ii) is not.
#
#        (i)                           (ii)
#  x1 --- y1 --- x3                  a --- b
#  |      |      |                   |  /  |
#  y2 --- x2 --- y3                  c --- d
#
# a. Design a DFS-based algorithm for checking whether a graph is bipartite.
#   - While there is a next element to search AND the next element is not
#     the same color as the current element AND you haven't found what you're
#     looking for, go to the next element.
# b. Design a BFS-based algorithm for checking whether a graph is bipartite.
#   - While there are child nodes to search AND all child nodes are not the
#     same color as the current node AND you haven't found what you're looking
#     for, go to the next element.
#########################################################################
```

6 BARNEY 2.6

Problem:

Code written in JavaScript

```
/**
 * Exercise 2.6 - Find the Door
 *
 * You are facing a wall that stretches initely in both directions.
 * There is a door in the wall, but you know neither how far away nor in
 * which direction. You can see the door only when you are right next to it.
 * Design and write code for an algorithm that enables you to reach the door
 * by walking at most O(n) steps where n is the (unknown to you) number of
 * steps between your initial position and the door. (Hint: walk alternately
 * right and left going each time exponentially farther from your initial
 * position.)
 *
 * @param {number} stepsToTake
 * @param {number} doorLocation
 */
function findDoor(stepsToTake, doorLocation) {
    let found = false;
    let stepsTaken = 0;
    let currentLocation = 0;
    while (!found) {
        stepsToTake++;
        currentLocation += stepsToTake;
        stepsTaken += stepsToTake;
        if (doorLocation < stepsToTake && doorLocation > 0) found = true;
        currentLocation -= stepsToTake * 2;
        stepsTaken += stepsToTake * 2;
        if (doorLocation > stepsToTake && doorLocation < 0) found = true;
        currentLocation += stepsToTake;
        stepsTaken += stepsToTake;
    }
    return stepsTaken;
}

const doorLocation = Math.floor(Math.random() * 100);
console.log(‘Number of back and forths to find the door at spot
    ${doorLocation}: ${findDoor(1, doorLocation)}‘);
```

Results

```
Number of back and forths to find the door at spot 13: 416
```