1. Consider the following directed graph (digraph) $G = (V, E)$:

![Graph Image]

a) What is the set of vertices? $V = \{v_0, v_1, v_2, v_3, v_4\}$

b) An edge can be represented by a tuple (from vertex, to vertex, weight). What is the set of edges?

$\begin{bmatrix}
v_0 & v_1 & v_2 & v_3 & v_4 \\
0 & 1 & \infty & 1 & 5 \\
9 & 0 & 3 & 3 & \infty \\
\infty & \infty & 0 & 4 & \infty \\
3 & \infty & 2 & 0 & 3 \\
\end{bmatrix}$

From vertex to vertex

$\{v_0, v_1, v_2, v_3, v_4\}$ or $\{v_0, v_1, v_2, v_3, v_4\}$

c) A path is a sequence of vertices that are connected by edges. In the graph $G$ above, list two different paths from $v_0$ to $v_3$.

$N_{o_1, v_3}$ or $N_{o_1, N_{v_1, v_3}}$

d) A cycle in a directed graph is a path that starts and ends at the same vertex. Find a cycle in the above graph.

$N_{v_1, v_0}$

2. Like most data structures, a graph can be represented using an array, or as a linked list of nodes. The array representation is a two-dimensional array (called an adjacency matrix) whose elements contain information about the edges and the vertices corresponding to the indices. (Python could use a list-of-lists)

a) Complete the following adjacency matrix for the above graph. (Here a missing edge is represented by $\infty$.)

(from vertex)

<table>
<thead>
<tr>
<th></th>
<th>$v_0$</th>
<th>$v_1$</th>
<th>$v_2$</th>
<th>$v_3$</th>
<th>$v_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v_0$</td>
<td>0</td>
<td>1</td>
<td>$\infty$</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>$v_1$</td>
<td>9</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td>$\infty$</td>
</tr>
<tr>
<td>$v_2$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>0</td>
<td>4</td>
<td>$\infty$</td>
</tr>
<tr>
<td>$v_3$</td>
<td>3</td>
<td>$\infty$</td>
<td>2</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>$v_4$</td>
<td>$\infty$</td>
<td>2</td>
<td>0</td>
<td>$\infty$</td>
<td>0</td>
</tr>
</tbody>
</table>

b) What is the big-oh to determine the edge-weight between any two vertices?

$O(1)$

d) The linked representation maintains a linked-list (or Python dictionary) of vertices with each vertex maintaining a linked list of other vertices that it connects to. Complete the adjacency list representation below:

Vertex List

- $v_0$
- $v_1$
- $v_2$
- $v_3$
- $v_4$

Edge Lists for each vertex containing connected-to and edge-weight information

- $v_0$: $v_1$ (1)
- $v_1$: $v_0$ (9), $v_3$ (3)
- $v_2$: $v_3$ (4)
- $v_3$: $v_2$ (2), $v_4$ (3)
- $v_4$: $v_3$ (3)

e) What is the big-oh to determine the edge-weight between any two vertices in an adjacency list?

$O(|V|)$

f) What is the big-oh amount of storage used to store the adjacency list?

$O(|V| + |E|)$
3. Below is the textbook’s edge, vertex, and graph implementations.

a) How does this graph implementation maintain its set of vertices?

b) How does this graph implementation maintain its set of edges?

c) What is the big-oh to determine the edge-weight between any two vertices $O(1)$

```python
""" File: graph.py """
from vertex import Vertex

class Graph:
    def __init__(self):
        self.vertList = {}
        self.numVertices = 0

def addVertex(self, key):
    self.numVertices = self.numVertices + 1
    newVertex = Vertex(key)
    self.vertList[key] = newVertex
    return newVertex

def getVertex(self, n):
    if n in self.vertList:
        return self.vertList[n]
    else:
        return None

def __contains__(self, n):
    return n in self.vertList

def addEdge(self, f, t, cost=0):
    if f not in self.vertList:
        nv = self.addVertex(f)
    if t not in self.vertList:
        nv = self.addVertex(t)
    self.vertList[f].addNeighbor \ (self.vertList[t], cost)

def getVertices(self):
    return self.vertList.keys()

def __iter__(self):
    return iter(self.vertList.values())

""" File: vertex.py """

class Vertex:
    def __init__(self, key, color='white',
                 dist=0, pred=None):
        self.id = key
        self.connectedTo = {}
        self.color = color
        self.predecessor = pred
        self.distance = dist
        self.discovery = 0
        self.finish = 0

    def addNeighbor(self, nbr, weight=0):
        self.connectedTo[nbr] = weight

    def __str__(self):
        return str(self.id) + ' connectedTo: ' + str([x.id for x in self.connectedTo])

    def getConnections(self):
        return self.connectedTo.keys()

    def getId(self):
        return self.id

    def getWeight(self, nbr):
        return self.connectedTo[nbr]

    def getColor(self):
        return self.color

    def setColor(self, newColor):
        self.color = newColor

    def getPred(self):
        return self.predecessor

    def setPred(self, newPred):
        self.predecessor = newPred

    def getDistance(self):
        return self.distance

    def setDistance(self, newDistance):
        self.distance = newDistance

    def getDiscovery(self):
        return self.discovery

    def setDiscovery(self, newDiscovery):
        self.discovery = newDiscovery

    def getFinish(self):
        return self.finish

    def setFinish(self, newFinish):
        self.finish = newFinish

    def getNumVertices(self):
        return self.numVertices

    def getNeighbor(self, key):
        return self.connectedTo[key]
```
4. Graphs can be used to solve many problems by modeling the problem as a graph and using "known" graph algorithm(s). For example, consider the word-ladder puzzle where you transform one word into another by changing one letter at a time, e.g., transform FOOL into SAGE by FOOL → FOIL → FAIL → FALL → PALL → PALE → SALE → SAGE.

We can use a graph algorithm to solve this problem by constructing a graph such that
- a word represents a vertex
- an edge represents connecting words that differ by one letter
- a word ladder transformation from one word to another represents a path

4. For the words listed below, draw the graph of question 3

[Graph diagram]

a) List a different transformation from FOOL to SAGE
   FOOL → POOL → POLL → POLE → POLE → PAGE → SAGE

b) If we wanted to find the shortest transformation from FOOL to SAGE, what does that represent in the graph?

   Shortest path

C) There are two general approaches for traversing a graph from some starting vertex s:
- Breadth First Search (BFS) where you find all vertices a distance 1 (directly connected) from s, before finding all vertices a distance 2 from s, etc.
- Depth First Search (DFS) where you explore as deeply into the graph as possible. If you reach a "dead end," we backtrack to the deepest vertex that allows us to try a different path.

Which of these traversals would be helpful for finding the shortest solution to the word-ladder puzzle?

BFS
1. There are two general approaches for traversing a graph from some starting vertex \( s \):

- Depth First Search (DFS) where you explore as deeply into the graph as possible. If you reach a “dead end,” we backtrack to the deepest vertex that allows us to try a different path.
- Breadth First Search (BFS) where you find all vertices a distance 1 (directly connected) from \( s \), before finding all vertices a distance 2 from \( s \), etc.

What data structure would be helpful in each type of search? Why?

a) Breadth First Search (BFS): 
   - Use a queue: \( \text{FIFO queue} \)
   - Example:
     - \( \text{ enqueue } q \)
     - \( \text{ while } q \text{ is not empty} \)
     - Get a vertex \( \text{ current vertex } \) \( \text{ dequeue } \) \( q \)
     - For each \( n \) in \( \text{ get connections} \)
       - If \( n \) is \( \text{ color } \) \( w \)
         - \( n \).setDistance \( \text{ current vertex } \) \( \text{ getDistance} \)
         - \( n \).setPred \( \text{ current vertex } \)
         - \( n \).setColor \( \text{ gray} \)
         - \( q \).enqueue \( n \)

b) Depth First Search (DFS):
   - Recursion with run-time stack

2. Assuming a graph \( G \) containing the word-ladder graph from lecture 25, on the diagram trace the BFS algorithm by showing the value of each vertex’s color, predecessor, and distance attributes.