**Homework #9**

**Graphs**

total points: 160

0. If you have not already done so, it is very important to complete the official [feedback form](https://dickinson.campuslabs.com/courseeval/) for this course.

1. Consider the directed graph shown to the right:

a. (5 points) Draw the adjacency matrix representation for the graph. Set each entry in the matrix to either 1, to indicate the edge exists, or 0 to indicate the edge does not exist.

b. (5 points) Draw the adjacency list representation for the graph.

2. (10 points) Implement the getNeighbors method in the CS232DirectedAdjacencyMatrixGraph class. The No2Tests class contains tests that you can use to check your implementation of this functionality.

3. Implement the following methods in the CS232DirectedAdjacencyListGraph class:

a. (10 points) addEdge and getEdge. The No3aTests class contains tests that you can use to check your implementation of this functionality.

b. (10 points) removeEdge. The No3bTests class contains tests that you can use to check your implementation of this functionality. Note: No3bTests also repeats all of the No3aTests to ensure that the new functionality does not break any of the old functionality.

c. (10 points) getNeighbors. The No3cTests class contains tests that you can use to check your implementation of this functionality.

4. Implement the inDegree method in each of the following classes:

a. (10 points) CS232DirectedAdjacencyMatrixGraph. The No4aTests class contains tests that you can use to check your implementation of this functionality.

b. (10 points) CS232DirectedAdjacencyListGraph. The No4bTests class contains tests that you can use to check your implementation of this functionality.

c. (10 points) Give the asymptotic bounds for each of these implementations.

5. (20 points) Create a class named CS232UndirectedAdjacencyMatrixGraph and complete its implementation. Complete this class with as little code as possible – it should require very little code. Hint: Use inheritance, override the necessary methods, and use the superclass methods. Additional hint: Given an undirected graph $G$, how could you convert it into a directed graph $G'$ that is essentially equivalent, such that every vertex in $G'$ has the same set of neighbors as in $G$? The No5Tests class contains tests that you can use to check your implementation of this functionality.

6. Consider the Graph ADT implementation for undirected graphs in problem #5. This implementation trades off simple code in favor of wasting space.

 a. (5 points) Explain why there is wasted space in this implementation.

 b. (5 points) Explain how the wasted space could be eliminated.

 c. (5 points) Describe how this would complicate the code.

7. For the directed graph from question #1:

a. (5 points) List the vertices in the order in which they would be printed by a DFS starting at vertex #1 (assume adjacent vertices are considered in numeric order).

b. (5 points) List the vertices in the order in which they would be printed in a BFS starting at vertex #1 (assume adjacent vertices are considered in numeric order).

8. (15 points) Complete the BFS (breadth-first search) method in the CS232GraphAlgorithms class. In situations where BFS gives equal priority to some vertices, the vertex with the smallest identifier should be visited first. The No8Tests class contains tests that you can use to check your implementation of this functionality.

9. (10 points) Some applications in which a graph data structure may be used will typically yield low-density graphs (with few edges) while others may yield high-density graphs (with many edges). Discuss how the expected density of the graphs in a given application might affect whether you use an adjacency matrix or an adjacency list representation.

10. (10 points) Discuss how Java abstract classes are similar to and different from interfaces. When are abstract classes useful?