Due on Apr. 22, 2020
65 points total

General directions: We will exclusively use Java for our programming assignment, and allow only the use of modules in the Java standard library, such as java.util or java.math.

For this coding assignment, please download the five skeleton files ("Problem1.java", "Problem2.java", "Problem3.java", "Problem4.java", and "Pair.java") from the course website and fill in the empty methods. Make sure to keep these files in the default package in your workspace. You should not rename these files or modify the method headers but you may add helper methods as you see fit. When you are done and ready to submit, upload "Problem1.java", "Problem2.java", "Problem3.java", and "Problem4.java" to Gradescope (you do not need to upload "Pair.java"). Do not upload any other files or put any of the four files in a folder. Submissions to Gradescope will be autograded so that you can see your score. You may submit to the autograder any number of times, up until the due date.

Finally, this is an individual assignment. You are **not** allowed to discuss your work with anybody. **Failure to adhere to these guidelines will be promptly reported to the relevant authority without exception.**

Note that one additional problem may be added at a later date to include material not yet covered in lecture.

Problem 1 (15 points)
Recall that any logical formula can be expressed in disjunctive normal form (DNF). For this problem you will implement the method `convertToDNF` which generates a DNF expression corresponding to a given truth table. The specifications of this method are described in more depth in "Problem1.java", available on the course website.

Problem 2 (15 points)
Recall that if R is a relation on a set A, then there are some properties of R that we often consider: reflexivity, transitivity, symmetry, and asymmetry. Implement methods to test whether given relations satisfy each of these properties.

The specifications of the methods are described in more depth in "Problem2.java", available on the course website.

Problem 3 (15 points)
Recall that we defined a graph as an ordered pair (V, E) where V is a non-empty finite set and E is a set of two-element subsets of V. Thus, one way to represent a graph is by explicitly listing its vertices and edges. However, in practice, the two most common representations are the *adjacency list* and *adjacency matrix* representations, which we define below.

Let $G = (V, E)$ be a graph with n vertices. For this problem, we assume $V = \{0, 1, \ldots, n-1\}$. Recall that for all $u \in V$, we say vertex v is a *neighbor* of u if $(u, v) \in E$. The *adjacency list* representation of G is a list L of lists of vertices such that $L[u]$ contains the neighbors of vertex u.

The *adjacency matrix* representation of un undirected graph G is an $n \times n$ symmetric matrix M where $M[u, v]$ is the weight of the u-v edge if v is a neighbor of u, and 0 otherwise. In an unweighted graph, $M[u, v]$ is 1 if v is a neighbor of u and 0 otherwise. (Recall that `M` is *symmetric* means $M[u, v] = M[v, u]$ for every pair of vertices u and v.)

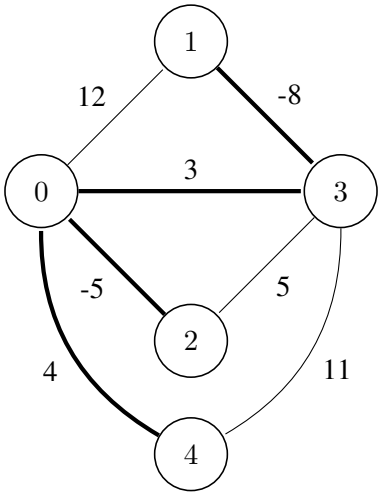
For this problem, you will implement two methods: `getAdjacencyList` and `getAdjacencyMatrix`, each of which takes a vertex set and edge set and returns an alternate representation of the graph. The specifications of these methods are described in more depth in "Problem3.java", available on the course website.

Problem 4 (20 points)
Recall the "cycle property" regarding minimum spanning trees (MSTs): the heaviest edge of any cycle in the graph is not in any MST. Thus, one way to find an MST is the following: while there exists a cycle C in the graph, remove the heaviest edge of C from the graph. Repeat this process until the graph is acyclic, and return this final graph.

For this problem, you will implement the algorithm above as method `findMST`, which takes an adjacency matrix for a weighted graph and returns the cost of the MST. See the example graph G and its corresponding matrix `M` below.

To implement `findMST`, you are to use the given function `findCycle` which returns `null`

if there is no cycle contained in the graph G, and otherwise returns a cycle C represented by a list of vertices (integers between 0 and $n-1$, inclusive) such that every pair of consecutive vertices are adjacent in G, as well as the first and last vertices in C. For example, for the matrix `M` below, `findCycle` might return `[0,3,1]`, `[2,3,4,0]`, or any other such cycle in G represented by M.



Above, the 5-vertex graph G represented by the matrix `M` below. The thick edges denote the (unique) MST in G, which has total cost -6.

$$\texttt{M} = \begin{bmatrix} None & 12 & -5 & 3 & 4 \\ 12 & None & None & -8 & None \\ -5 & None & None & 5 & None \\ 3 & -8 & 5 & None & 11 \\ 4 & None & None & 11 & None \end{bmatrix}$$

The specifications and usage of these methods are described in more depth in "Problem4.java", available on the course website.