Due: November 1, 4:00pm
(note: Second part of the lab will start in November 1 lab period, so while we won’t collect this
assignment until later, you should have the work completed).

This is a pair programming assignment. Select a partner and work with them throughout
the entirety of Lab 5.

We will be using Python for this lab. To make things easy to manage and submit, you can
develop most of your code in Jupyter notebooks. Your final submission may ask you to
turn some of the code into a package, but it should be convenient for you to do all
development for this and several other parts of the lab in Jupyter (or Google Colab).

In this lab, we will work on building the code for the Minimum Spanning Tree algorithms. Our
starting point is Kruskal’s algorithm. Kruskal’s algorithm needs to be supported by an abstract
data type (ADT) that represents a collection of disjoint sets. There are three operations we need
to implement:

- **MakeSet(v):** (where \( v \) is a label of sorts - in your code this will be a node label) - add a
  new set consisting of a single element \( v \) (\{\( v \)\}). The label of the new set will be \( v \).
- **FindSet(v):** given a label \( v \) find the set in the disjoint set collection that it belongs
  to. This function returns a label of the set \( v \) belongs to.
- **Union(v,u):** given two set labels, \( v \), and \( u \), merge these two sets into a single set. Set
  (and return) the label for the new set.

In addition to this set of operations, we need to implement one more operation:

- **Initialize:** creates an empty data structure that contains no sets. (note: MakeSet() adds
  a new set to this data structure).

One possible way to implement this abstract data type in Python is through the use of
dictionaries.

- **Initialize()** creates an empty dictionary
- **MakeSet(v)** adds a new key-value pair \( v:v \) to the dictionary
- **FindSet(v)** returns the current value for the key (node) \( v \)
- **Union(v,u)** finds all labels "x" such that their value (i.e., their set designation) is
  equal to \( v \), and replaces them with "u"

Note, that this is NOT the most efficient implementation of the disjoint sets data structure (while
lookup (FindSet) is \( O(1) \), union is \( O(|V|) \) in the worst case). However, for our current purposes,
this approach will do.
Task 1. Implement the Abstract Data Type DisjointSets using the approach discussed above. Specifically,

- implement the class constructor/ Initialize() method
- implement the MakeSet() method
- implement the FindSet() method
- implement the Union() method

Task 2. In preparation for implementing Kruskal's algorithm (as well as any other graph algorithm), we need to develop a data structure to store graphs. Additionally, we need a way to generate some graphs for testing purposes.

The simplest representation of a graph is an Adjacency Matrix. Our work in this lab is with undirected, weighted graphs, which means that:

- your Adjacency Matrix must be symmetric (and have zeros on the main diagonal)
- the value in the intersection of row j and column i (for i != j) is the edge weight for the edge (i,j). A value of 0 means there is no edge between i and j

You can use NumPy arrays to store Adjacency Matrices.

While some of our future testing might be done on specific graphs provided to you, a lot of debugging and testing along the way can be done on randomly generated graphs.

Write a function getRandomGraph() that takes as input three parameters:

- n - number of nodes in the graph
- m - number of edges in the graph
- maxWeight - maximum weight of an edge in the graph

The function shall return back an adjacency matrix (e.g., a NumPy array) representing a randomly generated graph with specified number of nodes and number of edges, and with randomly assigned weights ranging from 0 to maxWeight. If the combination of n and m is invalid (e.g., n=2 and m = 100), your function can return a matrix of zeros of size n x n.

Note: you can assume that node labels in such graphs are row/column ids, i.e., node labels of a graph with n vertices are "0", "1", ..., "n-1" (or "1",... "n" - whichever you find more natural).
**Task 3.** To test your disjoint set ADT, write a function $\text{Graph2DisjointSets}(A)$ that takes as input an adjacency matrix of a weighted undirected graph (the output of your $\text{getRandomGraph()}$ function) and returns back the disjoint set structure in which each node from the graph represented by $A$ is placed in a separate set. (That is, this function should return the result of Initializing a disjoint set ADT and then calling a MakeSet on each vertex of the graph represented by $A$).

**Task 4.** Write some test code that generates a random graph with 10 nodes, creates a disjoint set representing the graph, conducts six (6) merge operations (pick which ones you want), and then returns the disjoint set label (i.e., calls $\text{FindSet()}$) for each node in the graph. The results of this code must all be printed out explicitly.

**Submission.** Place all this code into a python notebook named $\text{Lab5-MST.ipynb}$. At the end of Lab 5 you will submit this notebook as one of your deliverables, but as you will add code to it, there is no need to submit anything just yet.