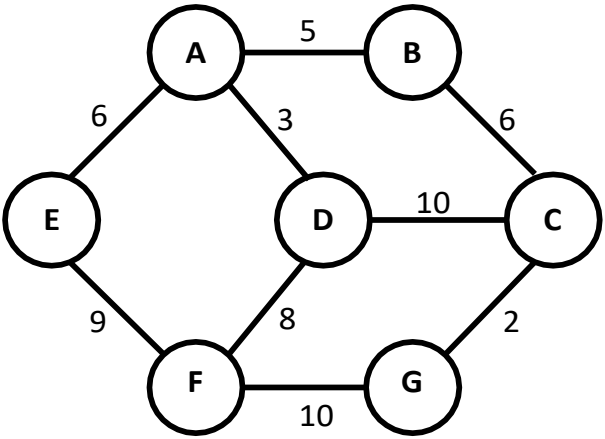


### Exercise # 1a: Minimum Spanning Tree

Write a function named “MSTPrim(G, SV)” and implement Prim’s algorithm to determine the minimum spanning tree in the following graph.

Input	<p>G: Adjacency list of the graph shown below</p>  <p>N: List of nodes in the graph shown above</p> <p>SV: A</p> <p>Note: (you can choose any vertex as a starting vertex)</p>
Output	<pre>[ ('A', 'D', 3), ('A', 'B', 5), ('B', 'C', 6),   ('C', 'G', 2), ('A', 'E', 6), ('D', 'F', 8) ]</pre>

First we will create Graph Adjacency List:

```
G = {  
    'A': [('B', 5), ('E', 6), ('D', 3)],  
    'B': [('A', 5), ('C', 6)],  
    'C': [('B', 6), ('D', 10), ('G', 2)],  
    'D': [('A', 3), ('C', 10), ('F', 8)],  
    'E': [('A', 6), ('F', 9)],  
    'F': [('D', 8), ('E', 9), ('G', 10)],  
    'G': [('C', 2), ('F', 10)]  
}
```

Now, we will create a function MSTPrim(G, SV) and implement Prim’s algorithm to determine the minimum spanning tree in the following graph.

Note: Please test your code for different source vertex.

*The approach shared is a little longer and involves lot of indexing, hence might be difficult to follow. We would advise you to follow the algorithm/ pseudo-code/ approach taught by your instructor. :)*

One of the approaches could be:

Step#1:

- ListOfNodes = list(G.keys())
- infinity = 9999
- ShortestDistance = []
- Now we will populate our ShortestDistance List:  
Loop over ListOfNodes:  
If the node is Starting Vertex, append (StartingVertex, StartingVertex, 0)  
Else, append [ "", node, infinity]

After first step:

ListOfNodes = [A, B, C, D, E, F, G]

ShortestDistance = [['A', 'A', 0], ['', 'B', 9999], ['', 'C', 9999], ['', 'D', 9999], ['', 'E', 9999], ['', 'F', 9999], ['', 'G', 9999]]

In ShortestDistance, For every member in the list:

ShortestDistance[i][0] is the Parent

ShortestDistance[i][1] is the Node

ShortestDistance[i][2] is the Shortest Weight (Weight from Parent to Node)

*It is better to create separate dictionaries for storing parent and shortest weight.*

Step#2:

- UNVISITED = ListOfNodes.copy()
- VISITED = []

After second step:

UNVISITED = [A, B, C, D, E, F, G]

VISITED = []

*If you are using Priority Queue, push (starting\_vertex, 0) in Priority Queue.*

- *In Priority Queue's enqueue() function, we would check if node already exists in queue, update its distance otherwise push it directly in the priority queue.*
- *In Priority Queue's dequeue() function, we would search and pop the node with the shortest distance.*

### Step#3

While our UNVISITED list is not empty:

#### Step#3a:

1. We will set lowest\_distance = infinity and Current\_Vertex = ""
2. Now we will loop over ShortestDistance List to find the node with shortest distance which is not in the VISITED list. In first iteration it will be Node A with distance ZERO.

We will loop over ShortestDistance:

If ShortestDistance[i][2] <= lowest\_distance and ShortestDistance[i][1] not in visited:

Set lowest\_distance = ShortestDistance[i][2]

Set Current\_Vertex = ShortestDistance[i][1]

*It is better to create a priority queue instead of finding the minimum vertex each time. Here, if you are using Priority Queue, you will dequeue() from the priority queue.*

#### Step#3b:

Now, we will loop over Neighbors of the Current\_Vertex; i.e. G[Current\_Vertex].

If the Neighbour not in visited:

We will check:

If Neighbour's Weight in ShortestDistance > Neighbour's Weight in Graph:

We will set Neighbour's Weight in ShortestDistance = Neighbour's Weight in Graph

We will set Neighbour's Parent in ShortestDistance = Current\_Vertex

*If you are using a priority queue, enqueue (Neighbour, Neighbour's Weight in ShortestDistance) in priority queue.*

#### Step#3c:

After we have loop over all the neighbors of the Current\_Vertex and performed desired operations, we will pop the Current\_Vertex from the UNVISITED list and push the Current\_Vertex in the VISITED list.

### Step#4:

After UNVISITED list is empty, our ShortestDistance List contains Minimum Spanning Tree.

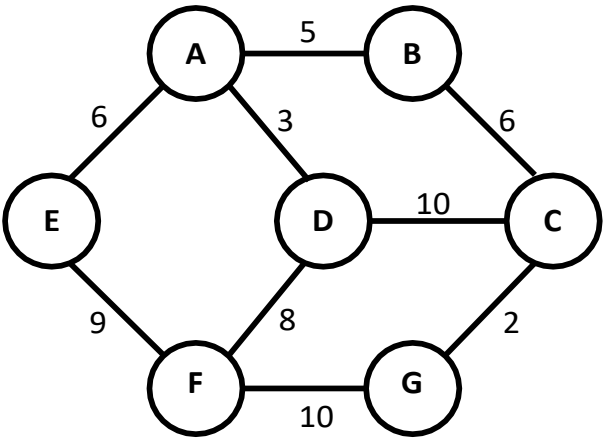
[[ 'A', 'A', 0], [ 'A', 'B', 5], [ 'B', 'C', 6], [ 'A', 'D', 3], [ 'A', 'E', 6], [ 'D', 'F', 8], [ 'C', 'G', 2]]

Remove [A, A, 0] from the ShortestDistance List and you are done! 😊

[[ 'A', 'B', 5], [ 'B', 'C', 6], [ 'A', 'D', 3], [ 'A', 'E', 6], [ 'D', 'F', 8], [ 'C', 'G', 2]]

### Exercise # 1c: Maximum Spanning Tree

Write a function named “MSTPrimMax(G, SV)” and implement Prim’s algorithm to determine the maximum spanning tree in the following graph.

Input	<p>G: Adjacency list of the graph shown below</p>  <p>SV: A</p> <p>Note: (you can choose any vertex as a starting vertex)</p>
Output	<pre>[ ('C', 'B', 6), ('D', 'C', 10), ('F', 'D', 8),   ('A', 'E', 6), ('E', 'F', 9), ('F', 'G', 10) ]</pre>

First we will create Graph Adjacency List:

```
G = {  
    'A': [('B', 5), ('E', 6), ('D', 3)],  
    'B': [('A', 5), ('C', 6)],  
    'C': [('B', 6), ('D', 10), ('G', 2)],  
    'D': [('A', 3), ('C', 10), ('F', 8)],  
    'E': [('A', 6), ('F', 9)],  
    'F': [('D', 8), ('E', 9), ('G', 10)],  
    'G': [('C', 2), ('F', 10)]  
}
```

Now, we will create a function MSTPrimMax(G, SV) and modify Prim’s algorithm to determine the maximum spanning tree in the following graph.

Note: Please test your code for different source vertex.

*The approach shared is a little longer and involves lot of indexing, hence might be difficult to follow. We would advise you to follow the algorithm/ pseudo-code/ approach taught by your instructor. :)*

One of the approaches could be:

Step#1:

- ListOfNodes = list(G.keys())
- infinity = -1
- ShortestDistance = []
- Now we will populate our ShortestDistance List:  
Loop over ListOfNodes:  
If the node is Starting Vertex, append (StartingVertex, StartingVertex, 0)  
Else, append [ "", node, infinity]

After first step:

ListOfNodes = [A, B, C, D, E, F, G]

ShortestDistance = [['A', 'A', 0], ['', 'B', -1], ['', 'C', -1], ['', 'D', -1], ['', 'E', -1], ['', 'F', -1], ['', 'G', -1]]

In ShortestDistance, For every member in the list:

ShortestDistance[i][0] is the Parent

ShortestDistance[i][1] is the Node

ShortestDistance[i][2] is the Shortest Weight (Weight from Parent to Node)

*It is better to create separate dictionaries for storing parent and shortest weight.*

Step#2:

- UNVISITED = ListOfNodes.copy()
- VISITED = []

After second step:

UNVISITED = [A, B, C, D, E, F, G]

VISITED = []

*If you are using Priority Queue, push (starting\_vertex, 0) in Priority Queue.*

- *In Priority Queue's enqueue() function, we would check if node already exists in queue, update its distance otherwise push it directly in the priority queue.*
- *In Priority Queue's dequeue() function, we would search and pop the node with the largest distance.*

### Step#3

While our UNVISITED list is not empty:

#### Step#3a:

1. We will set `lowest_distance = infinity` and `Current_Vertex = ""`
2. Now we will loop over ShortestDistance List to find the node with shortest distance which is not in the VISITED list. In first iteration it will be Node A with distance ZERO.

We will loop over ShortestDistance:

If `ShortestDistance[i][2] >= lowest_distance` and `ShortestDistance[i][1]` not in visited:

Set `lowest_distance = ShortestDistance[i][2]`

Set `Current_Vertex = ShortestDistance[i][1]`

*It is better to create a priority queue instead of finding the maximum vertex each time. Here, if you are using Priority Queue, you will `deQueue()` from the priority queue.*

#### Step#3b:

Now, we will loop over Neighbors of the `Current_Vertex`; i.e. `G[Current_Vertex]`.

If the Neighbour not in visited:

We will check:

If Neighbour's Weight in ShortestDistance < Neighbour's Weight in Graph:

We will set Neighbour's Weight in ShortestDistance = Neighbour's Weight in Graph

We will set Neighbour's Parent in ShortestDistance = `Current_Vertex`

*If you are using a priority queue, `enqueue` (Neighbour, Neighbour's Weight in ShortestDistance) in priority queue.*

#### Step#3c:

After we have loop over all the neighbors of the `Current_Vertex` and performed desired operations, we will pop the `Current_Vertex` from the UNVISITED list and push the `Current_Vertex` in the VISITED list.

#### Step#4:

After UNVISITED list is empty, our ShortestDistance List contains Minimum Spanning Tree.

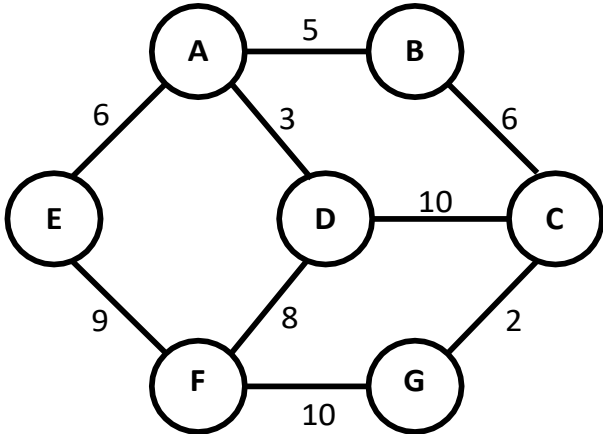
`[['A', 'A', 0], ['C', 'B', 6], ['D', 'C', 10], ['F', 'D', 8], ['A', 'E', 6], ['E', 'F', 9], ['F', 'G', 10]]`

Remove `[A, A, 0]` from the ShortestDistance List and you are done! 😊

`[ ['C', 'B', 6], ['D', 'C', 10], ['F', 'D', 8], ['A', 'E', 6], ['E', 'F', 9], ['F', 'G', 10]]`

## Exercise # 2: Minimum Spanning Tree

Write a function named MSTKruskal(G) and implement Kruskal's algorithm to determine the minimum spanning tree in the following graph.

Example 1 :	
Input	<p>G: Adjacency list of the graph shown below</p> 
Output	<pre>[('C', 'G', 2), ('A', 'D', 3), ('A', 'B', 5), ('A', 'E', 6), ('B', 'C', 6), ('D', 'F', 8)]</pre>

First we will create Graph Adjacency List:

```
G = {  
    'A': [('B', 5), ('E', 6), ('D', 3)],  
    'B': [('A', 5), ('C', 6)],  
    'C': [('B', 6), ('D', 10), ('G', 2)],  
    'D': [('A', 3), ('C', 10), ('F', 8)],  
    'E': [('A', 6), ('F', 9)],  
    'F': [('D', 8), ('E', 9), ('G', 10)],  
    'G': [('C', 2), ('F', 10)]  
}
```

Now, we will create a function MSTKruskal (G) and implement Kruskal's algorithm to determine the minimum spanning tree in the following graph.

One of the approaches could be:

Step#1:

- ListOfNodes = list(G.keys())
- DisJointSet = []
- Loop over the ListOfNodes to initialize DisJointSet:  
Append [ListOfNodes[i]] in DisJointSet.

After first step:

ListOfNodes = [A, B, C, D, E, F, G]

DisJointSet = [['A'], ['B'], ['C'], ['D'], ['E'], ['F'], ['G']]

Step#2:

- Extract Unique List of Edges (Without Repeated Edges) from the Graph and store it in a variable EL.
- Sort the Edges List (using any sorting algorithm) and store it in a variable SE.
- We will initialize MST = [] to store the Minimum Spanning Tree produced by the Graph.

After second step:

EL = [('A', 'B', 5), ('A', 'D', 3), ('A', 'E', 6), ('B', 'C', 6), ('C', 'D', 10), ('C', 'G', 2), ('D', 'F', 8), ('E', 'F', 9), ('F', 'G', 10)]

SE = [('C', 'G', 2), ('A', 'D', 3), ('A', 'B', 5), ('A', 'E', 6), ('B', 'C', 6), ('D', 'F', 8), ('E', 'F', 9), ('C', 'D', 10), ('F', 'G', 10)]

Step#3:

We will loop over SE (Sorted Edges List).

- U = SE[i][0]
- V = SE[i][1]
- We will find the index of U in DisJointSet and store it in IdxOfU.
- We will find the index of V in DisJointSet and store it in IdxOfV.
- If IdxOfU != IdxOfV, it means both are not in the same DisJointSet.  
If IdxOfU != IdxOfV:  
We will update the DisJointSet by calling function mergeAndDelete(IdxOfU, IdxOfV, DisJointSet).  
Append the edge in MST.

Step#3:

Return MST



Note: While MST contains your Minimum Spanning Tree, DisjointSet contains the list of connected components in the graph.

Implementing function: mergeAndDelete(IdxOfU, IdxOfV, DisJointSet)

1. We will create a list in which we will store the union of DisjointSet[idxOfu] & DisjointSet[idxOfv].

lst = DisjointSet[idxOfu] + DisjointSet[idxOfv]

2. u = DisjointSet[idxOfu]  
v = DisjointSet[idxOfv]

3. We will remove u and v from DisjointSet.

4. Append lst in DisjointSet.

5. Return DisjointSet.

### **Exercise # 3: Minimum Spanning Tree**

There are eight small islands in a lake, and the state wants to build seven bridges to connect them so that each island can be reached from any other one via one or more bridges. The cost of constructing a bridge is proportional to its length. The distances between pairs of islands are given in the following table.

Use the graph algorithms that you have implemented so far to find which bridges to build to minimize the total construction cost.

	1	2	3	4	5	6	7	8
1	-	240	210	340	280	200	345	120
2	-	-	265	175	215	180	185	155
3	-	-	-	260	115	350	435	195
4	-	-	-	-	160	330	295	230
5	-	-	-	-	-	360	400	170
6	-	-	-	-	-	-	175	205
7	-	-	-	-	-	-	-	305
8	-	-	-	-	-	-	-	-

Input	Graph: Adjacency list of a graph shown in the table above.
Output	Minimum spanning tree covering all of the above nodes.

**You need to first convert the given Adjacency Matrix to Adjacency List and then perform Minimum Spanning Tree's Algorithm. Both Prim & Kruskal Algorithm will result in the same minimum spanning Tree.**

a) Kruskal's algorithm

Kruskal's algorithm works by starting with each vertex in a separate cluster, and merging two clusters at each step. In this case, the clusters are merged in this order:

```
3 to 5 (115)
1 to 8 (120)
2 to 8 (155)
4 to 5 (160)
5 to 8 (170)
6 to 7 (175) [Note that 2 and 4 are already in the same cluster]
2 to 6 (180)
```

b) the Prim-Jarník algorithm

The Prim-Jarník algorithm works by adding a new vertex to the tree at each step. The order of adding will depend on the starting vertex, although the resulting tree should be the same. For example, if we choose vertex 8 as the starting vertex, the edges are added in this order:

```
8 to 1 (120)
8 to 2 (155)
8 to 5 (170)
5 to 3 (115)
5 to 4 (160)
2 to 6 (180)
6 to 7 (175)
```

For both (a) and (b), we end up with this graph:

```
1 -- 8 -- 2 -- 6 -- 7
    |
    3 -- 5 -- 4
```