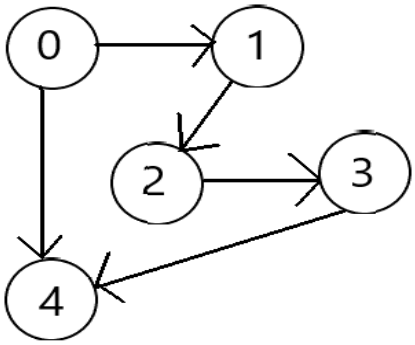
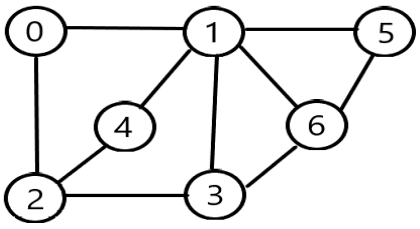
Problem 1:



Create the following graph. Implement the following functions.

1. **addEdge()** that takes two vertices as two parameters and creates an Edge between them.
2. **nonAdjacentVertices()** that takes a single vertex as parameter, and returns an array of vertices that are **NOT** adjacent to that vertex.
3. Write a function **addUnknownVertices()** that takes a vertex as a parameters, and creates an Edge with the vertices that are **NOT** adjacent to it.
4. Write a method **searchVertex()** which takes an array of Vertices as parameters. The first member of the array will be the starting vertex and the last member will be the Vertex you want to reach. For example **searchVertex(Vertex 0, Vertex 4, Vertex 3)** will output "Vertex cannot be reached" since you can reach from 0 to 4, but cannot from 4 to 3. Again, **searchVertex(Vertex 0, Vertex 1, Vertex 2, Node 3)** will output "Vertex can be reached" since you can reach from 0 to 1, 1 to 2, 2 to 3.

Problem 2:



Create the following graph. Implement the following functions.

1. **addEdge()** that takes two vertices as two parameters and creates an Edge between them.

2. **adjacentVertices()** that takes a single vertex as parameter, and returns an array of vertices adjacent to that vertex.

3. Write a function **showCommonVertices()** that takes two vertices as two parameters, and prints the common adjacent vertices between them.

4. Write a function **shortestPath(Edge start, Edge end)** to find the shortest path to reach from start Edge to end Edge. Your function will print the path it takes to reach the end Edge and also the total weight to reach the Edge. For example, **shortestPath(0,5)** will give the output 0, 1, 5 and also the total weight of the edges are 2. You should consider the weight between two Edges are 1 since this is an unweighted graph. Again, **shortestPath(0,6)** will output 0,1,6 and also the total weight is 2.

You can follow the given algorithm to find the shortest Path.

- Set all vertices distances = infinity except for the source vertex, set the source distance = 0.
- Push the source vertex in a min-priority queue in the form (distance, vertex), as the comparison in the min-priority queue will be according to vertices distances.
- Pop the vertex with the minimum distance from the priority queue (at first the popped vertex = source).
- Update the distances of the connected vertices to the popped vertex in case of "current vertex distance + edge weight < next vertex distance", then push the vertex with the new distance to the priority queue.
- If the popped vertex is visited before, just continue without using it.
- Apply the same algorithm again until the priority queue is empty.

The following algorithm is known as Djikstra's Algorithm.

IMPORTANT POINTS TO NOTICE:

1. For each problem you must create a Graph class as well as Node class (if required) and implement the functions as a method of the class.
2. You must figure out the return type of each method based on the problem description.
3. Implement the methods and write a small program that shows that each of your methods work for various cases.