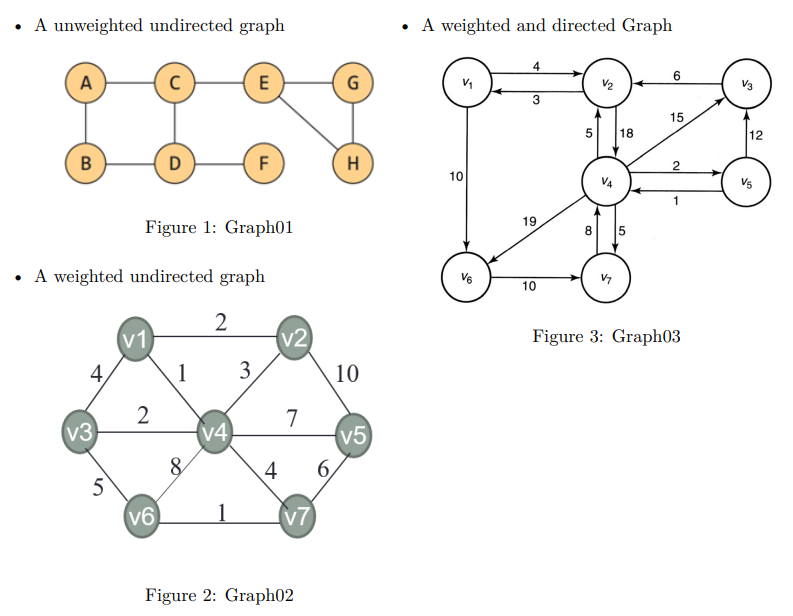
**Data Structures 2023-2**

**Lab 07: Graph Data Structures**

**ID: \_\_2018136121\_\_ Name: \_조원석\_\_**

**Task-1: Implement Graph Data Structure**

1. Write code for the graph data structures
2. Encode graphs given in Figures 1, 2, 3 and run the operations

****

**Code**

Graph.py

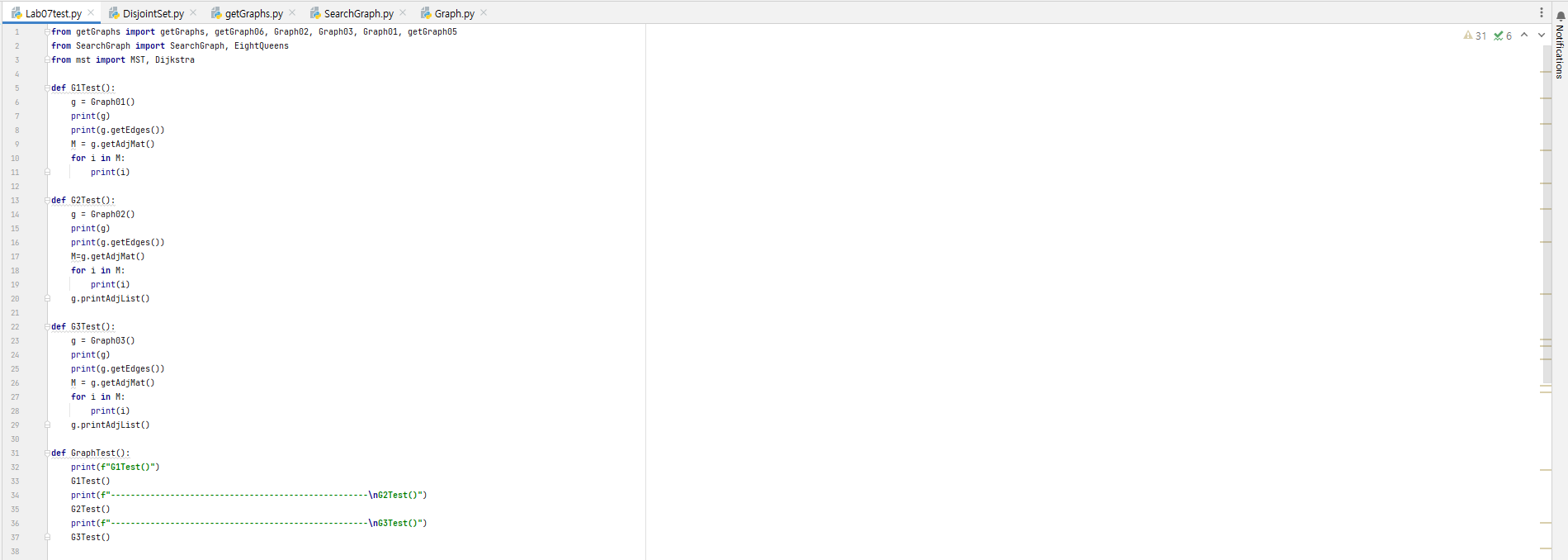
**from** collections **import** OrderedDict  
**from** DisjointSet **import** DisjointSet  
**class** Graph:  
 **def** \_\_init\_\_(self, directed = **False**, gdict=**None**):  
 **if** gdict **is None**:  
 gdict = {}  
 self.gdict = gdict  
 self.directed = directed  
 self.keyIndex = {}  
 **def** isDirected(self):  
 **return** self.directed  
  
 **def** getVertexList(self):  
 dict1 = OrderedDict(sorted(self.keyIndex.items()))  
 **return** list(dict1.keys())  
  
 **def** getOrder(self):  
 **return** len(self.gdict.keys())  
  
 **def** getEdgeList(self):  
 eList = []  
 **for** vtx **in** self.gdict:  
 **for** e **in** self.gdict[vtx]:  
 eList.append(e)  
 **return** eList  
 **def** getSize(self):  
 size = len(self.getEdgeList())  
 **if not** self.isDirected():  
 size //= 2  
 **return** size  
  
 **def** getDegree(self, vtx): **return** self.getOutDegree(vtx), self.getInDegree(vtx)  
  
 **def** getOutDegree(self,vtx): **return** len(self.gdict[vtx])  
  
 **def** printOutDegree(self):  
 **for** vtx **in** self.gdict:  
 print(**f"Out degree of vertex {**vtx**} = {**self.getOutDegree(vtx)**}"**)  
 **def** getInDegree(self, vtx):  
 **return** len(self.getInwardEdges(vtx))  
 **def** getInwardEdges(self, vtx):  
 eList = []  
 **for** e **in** self.getEdgeList():  
 **if** vtx == e.getV():  
 eList.append(e)  
 **return** eList  
  
 **def** printInDegree(self):  
 **for** vtx **in** self.gdict:  
 print(**f"Out degree of vertex {**vtx**} = {**self.getInDegree(vtx)**}"**)  
  
 **def** printAdjList(self):  
 aList = self.getAdjList()  
 **for** vtx **in** aList:  
 print(**f"{**vtx**} : {**aList[vtx]**}"**)  
 **def** getNeighborVertices(self, vtx):  
 elist = self.gdict.get(vtx)  
 vlist = []  
 **for** e **in** elist:  
 vlist.append(e.getV())  
 **return** vlist  
  
 **def** getNeighborEdges(self, vtx):  
 **return** self.gdict[vtx]  
  
 **def** getNeighbors(self, v):  
 nList = []  
 eList = self.gdict.get(v)  
 **for** e **in** eList:  
 nList.append(e.getV())  
 **return** nList  
  
 **def** getAdjList(self):  
 aList = {}  
 **for** vtx **in** self.gdict:  
 aList[vtx] = set(self.getNeighbors(vtx))  
 **return** aList  
  
 **def** getEdges(self):  
 eList = []  
 **for** vtx **in** self.gdict:  
 **for** e **in** self.gdict[vtx]:  
 eList.append(e)  
 **return** eList  
  
  
 **def** getAdjMat(self):  
 adjMat = [[0 **for** x **in** range(len(self.keyIndex))]**for** y **in** range(len(self.keyIndex))]  
 **for** e **in** self.getEdges():  
 adjMat[self.keyIndex[e.getU()] -1][self.keyIndex[e.getV()]-1] = e.getW()  
 **return** adjMat  
  
 **def** printAdjMat(self):  
 print(**"\n Adjacency Matrix:"**)  
 adjMatrix = self.getAdjMat()  
 **for** i **in** range(len(self.keyIndex)):  
 print()  
 **for** j **in** range(len(self.keyIndex)):  
 print(**"{0:>3d}"**.format(adjMatrix[i][j]), end=**""**)  
 print()  
  
 **def** getWeight(self):  
 w = 0  
 **for** e **in** self.getEdgeList():  
 w += e.getW()  
 **if not** self.isDirected():  
 w = w // 2  
 **return** w  
  
 **def** isCycle(self):  
 cycle =**False** ds = DisjointSet()  
 **for** vtx **in** self.getVertexList():  
 ds.makeSet(vtx)  
  
 **for** e **in** self.getEdgeList():  
 x = ds.find(e.getU())  
 y = ds.find(e.getV())  
 **if** x != y:  
 ds.Union(x, y)  
 **else**:  
 **return True  
 return** cycle  
 **def** \_\_repr\_\_(self):  
 gs =**""  
 for** vtx **in** self.gdict:  
 gs += **f"{**vtx**} : {**self.gdict[vtx]**}\n"  
 return** gs  
  
 **def** \_\_str\_\_(self):  
 gs =**""  
 for** vtx **in** self.gdict:  
 gs += **f"{**vtx**} : {**self.gdict[vtx]**}\n"  
 return** gs  
  
 **def** addVertex(self, vtx):  
 **if** vtx **in** self.gdict:  
 print(**"Vertex is already added.."**)  
 **else**:  
 self.gdict[vtx] = []  
 self.keyIndex[vtx] = len(self.keyIndex)+1  
  
 **def** addEdge(self, e):  
 **if** e.getU() **in** self.gdict:  
 self.gdict[e.getU()].append(e)  
 **if not** self.directed:  
 e2 = Edge(e.getV(), e.getU(), e.getW())  
 self.gdict[e2.getU()].append(e2)  
  
**class** Edge:  
 **def** \_\_init\_\_(self, u=**None**, v = **None**, w= **None**):  
 self.u=u  
 self.v=v  
 self.w=w  
 **def** getU(self):  
 **return** self.u  
  
 **def** getV(self):  
 **return** self.v  
  
 **def** getW(self):  
 **return** self.w  
  
 **def** setU(self, u):  
 self.u =u  
  
 **def** setV(self, v):  
 self.v =v  
  
 **def** setW(self, w):  
 self.w = w  
 **def** \_\_str\_\_(self):  
 **return f"( ({**self.u**}, {**self.v**}) -> {**self.w**} )"  
 def** \_\_repr\_\_(self):  
 **return f"( ({**self.u**}, {**self.v**}) -> {**self.w**} )"  
  
 def** \_\_hash\_\_(self):  
 **return** hash(self.w)  
  
 **def** \_\_eq\_\_(self, other):  
 **return** self.w == other.w  
  
 **def** \_\_ne\_\_(self, other):  
 **return** self.w != other.w  
  
 **def** \_\_lt\_\_(self, other):  
 **return** self.w < other.w  
  
 **def** \_\_le\_\_(self, other):  
 **return** self.w <= other.w  
  
 **def** \_\_gt\_\_(self, other):  
 **return** self.w > other.w  
  
 **def** \_\_ge\_\_(self, other):  
 **return** self.w >= other.w  
  
  
**class** Vertex:  
 **def** \_\_init\_\_(self, key = **None**):  
 self.key = key  
  
 **def** \_\_str\_\_(self):  
 **return** str(self.key)  
  
 **def** \_\_repr\_\_(self):  
 **return** str(self.key)  
  
 **def** setData(self, k):  
 self.key = k  
  
 **def** getData(self):  
 **return** self.key  
  
 **def** \_\_hash\_\_(self):  
 **return** hash(self.key)  
  
 **def** \_\_eq\_\_(self, other):  
 **return** self.key == other.key  
  
 **def** \_\_ne\_\_(self, other):  
 **return** self.key != other.key  
 **def** \_\_lt\_\_(self, other):  
 **return** self.key < other.key  
  
 **def** \_\_le\_\_(self, other):  
 **return** self.key <= other.key  
  
 **def** \_\_gt\_\_(self, other):  
 **return** self.key > other.key  
  
 **def** \_\_ge\_\_(self, other):  
 **return** self.key >= other.key

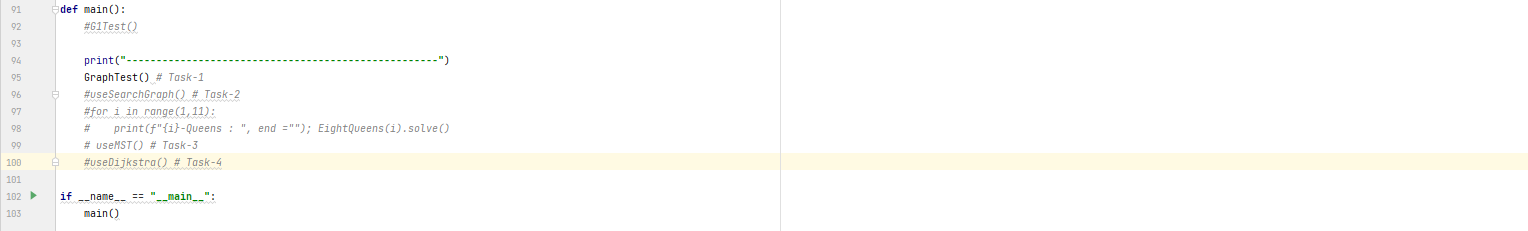
getGraphs.py

**from** Graph **import** \*  
**from** SearchGraph **import** \*  
  
  
**def** Graph01():  
 g = Graph(**False**)  
  
 v1 = Vertex(**"A"**)  
 v2 = Vertex(**"B"**)  
 v3 = Vertex(**"C"**)  
 v4 = Vertex(**"D"**)  
 v5 = Vertex(**"E"**)  
 v6 = Vertex(**"F"**)  
 v7 = Vertex(**"G"**)  
 v8 = Vertex(**"H"**)  
  
 e1 = Edge(v1, v2, 1)  
 e2 = Edge(v1, v3, 1)  
 e3 = Edge(v2, v4, 1)  
 e4 = Edge(v3, v4, 1)  
 e5 = Edge(v4, v6, 1)  
 e6 = Edge(v3, v5, 1)  
 e7 = Edge(v5, v7, 1)  
 e8 = Edge(v5, v8, 1)  
 e9 = Edge(v7, v8, 1)  
  
 g.addVertex(v1)  
 g.addVertex(v2)  
 g.addVertex(v3)  
 g.addVertex(v4)  
 g.addVertex(v5)  
 g.addVertex(v6)  
 g.addVertex(v7)  
 g.addVertex(v8)  
  
 g.addEdge(e1)  
 g.addEdge(e2)  
 g.addEdge(e3)  
 g.addEdge(e4)  
 g.addEdge(e5)  
 g.addEdge(e6)  
 g.addEdge(e7)  
 g.addEdge(e8)  
 g.addEdge(e9)  
 **return** g  
  
**def** Graph02():  
 sg = SearchGraph()  
 g = Graph(**False**)  
 v1 = Vertex(**"v1"**); v2 = Vertex(**"v2"**); v3 = Vertex(**"v3"**); v4 = Vertex(**"v4"**); v5 = Vertex(**"v5"**); v6 = Vertex(**"v6"**); v7 = Vertex(**"v7"**)*#; v8 = Vertex("v8")* g.addVertex(v1); g.addVertex(v2); g.addVertex(v3); g.addVertex(v4)  
 g.addVertex(v5); g.addVertex(v6); g.addVertex(v7)  
  
 sg.findCC(g.getAdjList())  
 e1 = Edge(v1, v2, 2); e2 = Edge(v1, v3, 4); e3 = Edge(v1, v4, 1)  
 e4 = Edge(v2, v4, 3); e5 = Edge(v2, v5, 10); e6 = Edge(v3, v4, 2)  
 e7 = Edge(v3, v6, 5); e8 = Edge(v4, v5, 7); e9 = Edge(v4, v6, 8)  
 e10 = Edge(v4, v7, 4); e11 = Edge(v5, v7, 6); e12 = Edge(v6, v7, 1)  
  
 g.addEdge(e1); g.addEdge(e2); g.addEdge(e3)  
 sg.findCC(g.getAdjList())  
  
 g.addEdge(e4); g.addEdge(e5); g.addEdge(e6)  
 sg.findCC(g.getAdjList())  
  
 g.addEdge(e7); g.addEdge(e8); g.addEdge(e9)  
 sg.findCC(g.getAdjList())  
  
 g.addEdge(e10); g.addEdge(e11); g.addEdge(e12)  
 sg.findCC(g.getAdjList())  
 **return** g  
  
  
**def** Graph03():  
 g = Graph(**True**)  
 v1=Vertex(**"v1"**)  
 v2=Vertex(**"v2"**)  
 v3=Vertex(**"v3"**)  
 v4=Vertex(**"v4"**)  
 v5=Vertex(**"v5"**)  
 v6=Vertex(**"v6"**)  
 v7=Vertex(**"v7"**)  
  
 g.addVertex(v1)  
 g.addVertex(v2)  
 g.addVertex(v3)  
 g.addVertex(v4)  
 g.addVertex(v5)  
 g.addVertex(v6)  
 g.addVertex(v7)  
  
 e1 = Edge(v1, v2, 4); e2 = Edge(v1, v6, 10); e3 = Edge(v2, v1, 3); e4 = Edge(v2, v4, 18)  
 e5 = Edge(v3, v2, 6); e6 = Edge(v4, v2, 5); e7 = Edge(v4, v5, 2); e8 = Edge(v4, v6, 19)  
 e9 = Edge(v4, v7, 5); e10 = Edge(v5, v4, 1); e11 = Edge(v6, v7, 10); e12 = Edge(v7, v4, 8)  
 e13 = Edge(v5, v3, 12); e14 = Edge(v4, v3, 15)  
 g.addEdge(e1)  
 g.addEdge(e2)  
 g.addEdge(e3)  
 g.addEdge(e4)  
 g.addEdge(e5)  
 g.addEdge(e6)  
 g.addEdge(e7)  
 g.addEdge(e8)  
 g.addEdge(e9)  
 g.addEdge(e10)  
 g.addEdge(e11)  
 g.addEdge(e12)  
 g.addEdge(e13)  
 g.addEdge(e14)  
  
 **return** g  
  
**def** getGraph06():  
 g = Graph(**True**)  
 v1=Vertex(**"v1"**)  
 v2=Vertex(**"v2"**)  
 v3=Vertex(**"v3"**)  
 v4=Vertex(**"v4"**)  
 v5=Vertex(**"v5"**)  
 v6=Vertex(**"v6"**)  
 v7=Vertex(**"v7"**)  
  
 g.addVertex(v1)  
 g.addVertex(v2)  
 g.addVertex(v3)  
 g.addVertex(v4)  
 g.addVertex(v5)  
 g.addVertex(v6)  
 g.addVertex(v7)  
  
 e1 = Edge(v1, v2, 4); e2 = Edge(v1, v6, 10); e3 = Edge(v2, v1, 3); e4 = Edge(v2, v4, 18)  
 e5 = Edge(v3, v2, 6); e6 = Edge(v4, v2, 5); e7 = Edge(v4, v5, 2); e8 = Edge(v4, v6, 19)  
 e9 = Edge(v4, v7, 5); e10 = Edge(v5, v4, 1); e11 = Edge(v6, v7, 10); e12 = Edge(v7, v4, 8)  
 e13 = Edge(v5, v3, 12); e14 = Edge(v4, v3, 15)  
  
 g.addEdge(e1)  
 g.addEdge(e2)  
 g.addEdge(e3)  
 g.addEdge(e4)  
 g.addEdge(e5)  
 g.addEdge(e6)  
 g.addEdge(e7)  
 g.addEdge(e8)  
 g.addEdge(e9)  
 g.addEdge(e10)  
 g.addEdge(e11)  
 g.addEdge(e12)  
 g.addEdge(e13)  
 g.addEdge(e14)  
  
 **return** g  
  
  
**def** getGraph05():  
 sg = SearchGraph()  
 g = Graph(**False**)  
 v1 = Vertex(**"v1"**); v2 = Vertex(**"v2"**); v3 = Vertex(**"v3"**); v4 = Vertex(**"v4"**); v5 = Vertex(**"v5"**); v6 = Vertex(**"v6"**); v7 = Vertex(**"v7"**)*#; v8 = Vertex("v8")* g.addVertex(v1); g.addVertex(v2); g.addVertex(v3); g.addVertex(v4)  
 g.addVertex(v5); g.addVertex(v6); g.addVertex(v7)  
  
 sg.findCC(g.getAdjList())  
  
 e1 = Edge(v1, v2, 2); e2 = Edge(v1, v3, 4); e3 = Edge(v1, v4, 1)  
 e4 = Edge(v2, v4, 3); e5 = Edge(v2, v5, 10); e6 = Edge(v3, v4, 2)  
 e7 = Edge(v3, v6, 5); e8 = Edge(v4, v5, 7); e9 = Edge(v4, v6, 8)  
 e10 = Edge(v4, v7, 4); e11 = Edge(v5, v7, 6); e12 = Edge(v6, v7, 1)  
  
 g.addEdge(e1); g.addEdge(e2); g.addEdge(e3)  
 sg.findCC(g.getAdjList())  
  
 g.addEdge(e4); g.addEdge(e5); g.addEdge(e6)  
 sg.findCC(g.getAdjList())  
  
 g.addEdge(e7); g.addEdge(e8); g.addEdge(e9)  
 sg.findCC(g.getAdjList())  
  
 g.addEdge(e10); g.addEdge(e11); g.addEdge(e12)  
 sg.findCC(g.getAdjList())  
 print(**"Do Topological Search"**)  
 sg.doTS(g.getAdjList())  
 **return** g  
  
**class** getGraphs:  
  
 **def** getG6(self):  
 sg = SearchGraph()  
 g=Graph(**False**)  
 v1 = Vertex(**"v1"**); v2 = Vertex(**"v2"**); v3 = Vertex(**"v3"**); v4 = Vertex(**"v4"**)  
 v5 = Vertex(**"v5"**); v6 = Vertex(**"v6"**); v7 = Vertex(**"v7"**); v8 = Vertex(**"v8"**)  
  
 g.addVertex(v1); g.addVertex(v2); g.addVertex(v3); g.addVertex(v4)  
 g.addVertex(v5); g.addVertex(v6); g.addVertex(v7)  
  
 sg.findCC(g.getAdjList())  
 e1 = Edge(v1, v2, 2); e2 = Edge(v1, v3, 4); e3 = Edge(v1, v4, 1)  
 e4 = Edge(v2, v4, 3); e5 = Edge(v2, v5, 10); e6 = Edge(v3, v4, 2)  
 e7 = Edge(v3, v6, 5); e8 = Edge(v4, v5, 7); e9 = Edge(v4, v6, 8)  
 e10 = Edge(v4, v7, 4); e11 = Edge(v5, v7, 6); e12 = Edge(v6, v7,1)  
  
 g.addEdge(e1); g.addEdge(e2); g.addEdge(e3)  
 sg.findCC(g.getAdjList())  
  
 g.addEdge(e4); g.addEdge(e5); g.addEdge(e6)  
 sg.findCC(g.getAdjList())  
  
 g.addEdge(e7); g.addEdge(e8); g.addEdge(e9)  
 sg.findCC(g.getAdjList())  
  
 g.addEdge(e10); g.addEdge(e11); g.addEdge(e12)  
 sg.findCC(g.getAdjList())  
 **return** g  
 **def** getG5(self):  
 g = Graph(**True**)  
 v1 = Vertex(**"A"**)  
 v2 = Vertex(**"B"**)  
 v3 = Vertex(**"C"**)  
 v4 = Vertex(**"D"**)  
 v5 = Vertex(**"E"**)  
  
 g.addVertex(v1)  
 g.addVertex(v2)  
 g.addVertex(v3)  
 g.addVertex(v4)  
 g.addVertex(v5)  
  
 e1 = Edge(v1, v2, 1)  
 e2 = Edge(v1, v3, 1)  
 e3 = Edge(v2, v4, 1)  
 e4 = Edge(v2, v5, 1)  
 e5 = Edge(v3, v4, 1)  
 e6 = Edge(v4, v5, 1)  
  
 g.addEdge(e1)  
 g.addEdge(e2)  
 g.addEdge(e3)  
 g.addEdge(e4)  
 g.addEdge(e5)  
 g.addEdge(e6)  
  
 **return** g  
  
 **def** getG1(self):  
 sg = SearchGraph()  
 g = Graph(**False**)  
  
  
 v1 = Vertex(**"A"**)  
 v2 = Vertex(**"B"**)  
 v3 = Vertex(**"C"**)  
 v4 = Vertex(**"D"**)  
 v5 = Vertex(**"E"**)  
 v6 = Vertex(**"F"**)  
 v7 = Vertex(**"G"**)  
 v8 = Vertex(**"H"**)  
  
 sg.findCC(g.getAdjList())  
 e1 = Edge(v1, v2, 1)  
 e2 = Edge(v1, v3, 1)  
 e3 = Edge(v2, v4, 1)  
 e4 = Edge(v3, v4, 1)  
 e5 = Edge(v4, v6, 1)  
  
 sg.findCC(g.getAdjList())  
 e6 = Edge(v3, v5, 1)  
 e7 = Edge(v5, v7, 1)  
 e8 = Edge(v5, v8, 1)  
 e9 = Edge(v7, v8, 1)  
  
 g.addVertex(v1)  
 g.addVertex(v2)  
 g.addVertex(v3)  
 g.addVertex(v4)  
 g.addVertex(v5)  
 g.addVertex(v6)  
 g.addVertex(v7)  
 g.addVertex(v8)  
  
 g.addEdge(e1)  
 g.addEdge(e2)  
 g.addEdge(e3)  
 g.addEdge(e4)  
 g.addEdge(e5)  
 g.addEdge(e6)  
 g.addEdge(e7)  
 g.addEdge(e8)  
 g.addEdge(e9)  
  
 **return** g  
  
 **def** getG2(self):  
 g=Graph(**False**)  
 v1 = Vertex(**"A"**)  
 v2 = Vertex(**"B"**)  
 v3 = Vertex(**"C"**)  
 v4 = Vertex(**"D"**)  
 v5 = Vertex(**"E"**)  
  
 g.addVertex(v1)  
 g.addVertex(v2)  
 g.addVertex(v3)  
 g.addVertex(v4)  
 g.addVertex(v5)  
  
 e1 = Edge(v1, v2, 13)  
 e2 = Edge(v1, v3, 10)  
 e3 = Edge(v3, v4, 27)  
 e4 = Edge(v2, v4, 25)  
 e5 = Edge(v4, v5, 34)  
 e6 = Edge(v2, v5, 18)  
  
 g.addEdge(e1)  
 g.addEdge(e2)  
 g.addEdge(e3)  
 g.addEdge(e4)  
 g.addEdge(e5)  
 g.addEdge(e6)  
  
 **return** g

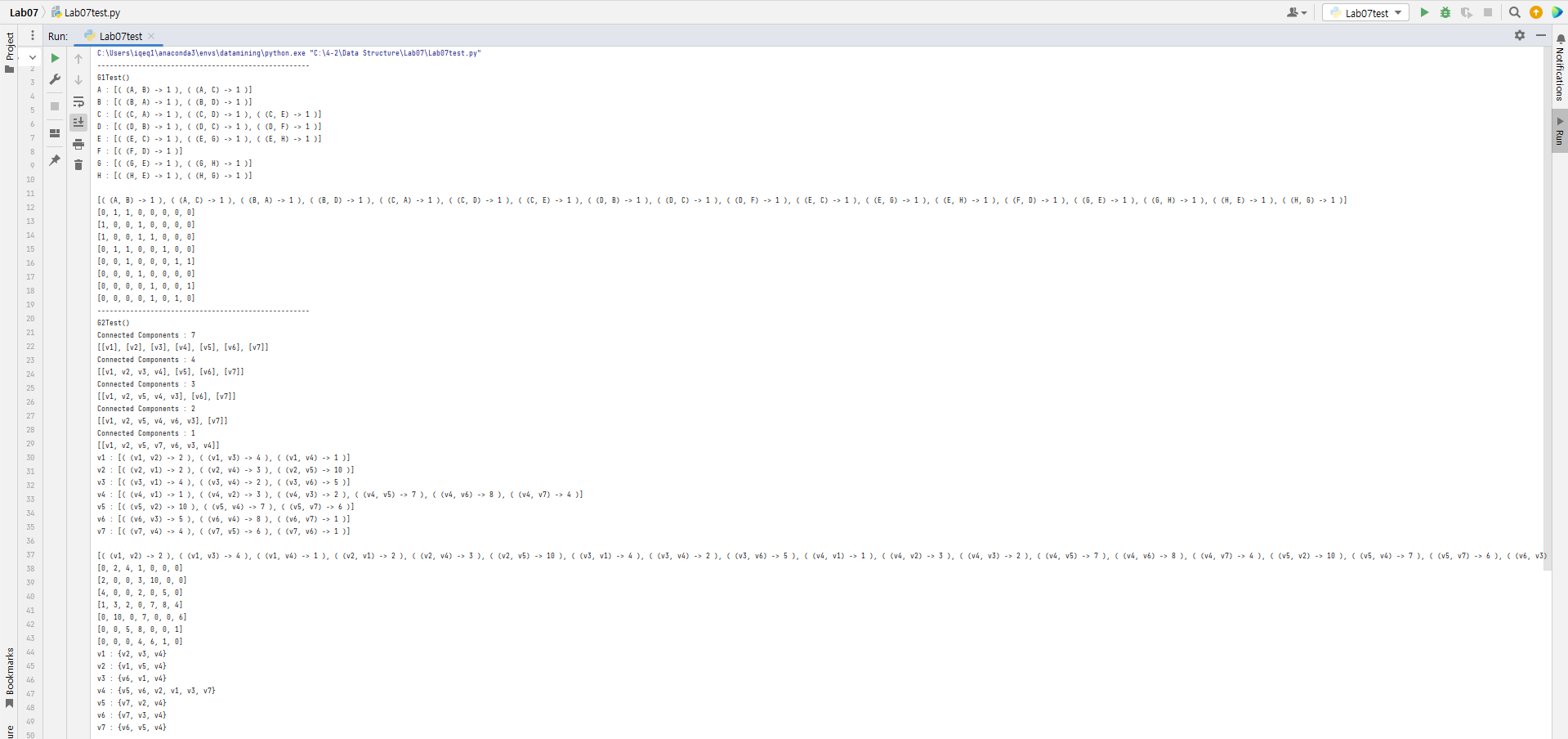
**Results/Output**

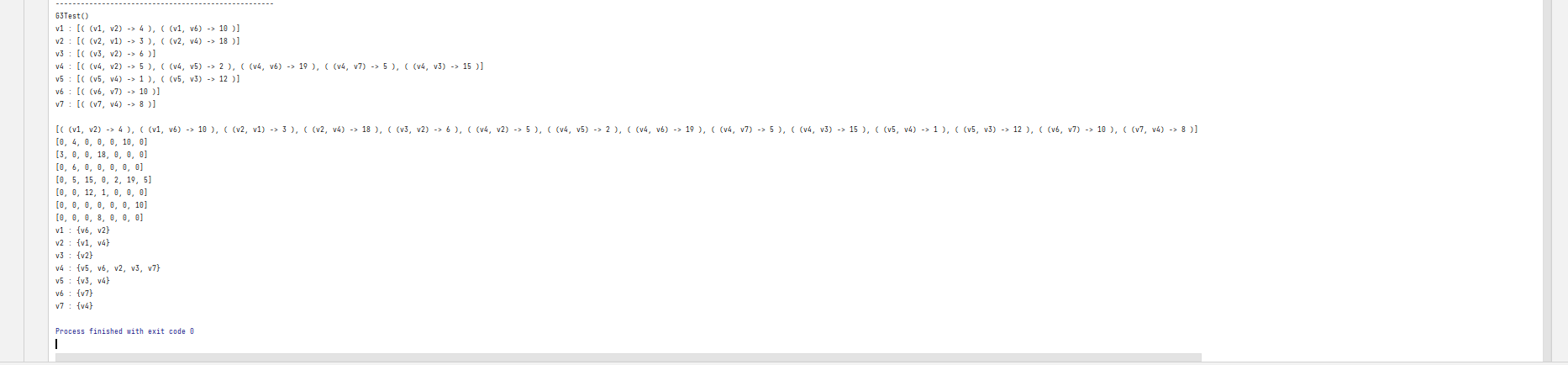
**Lab07Test.py**

****

****

**Result**



****

**Task2:** **DFS and BFS:**

1. Write code for the DFS and BFS algorithms and test then on graphs given in Figures 1, 2, 3
2. Write code for the following applications of the DFS
   1. Connected components
   2. Topological Sort
   3. N-Queens Problem

**Code**

1. Write code for the DFS and BFS algorithms and test then on graphs given in Figures 1, 2, 3

SearchGraph.py

**from** queue **import** Queue, LifoQueue  
**from** Graph **import** Graph  
**from** collections **import** defaultdict  
  
**class** SearchGraph:  
 **def** bfs(self, adjList, start):  
 visited = set()  
 print(**"BFS starting with Vertex "**, start)  
 visited.add(start)  
 Q = Queue()  
 Q.put(start)  
 **while not** Q.empty():  
 v = Q.get()  
 print(v, end=**'->'**)  
 nbr = adjList[v]  
 **for** u **in** nbr:  
 **if** u **not in** visited:  
 visited.add(u)  
 Q.put(u)  
  
 **def** dfs(self, adjList, start, visited=set()):  
 *#if visited is None:  
 # visited = set()* **if** start **not in** visited:  
 visited.add(start)  
 print(start, end=**' '**)  
 **for** next **in** adjList[start] - visited:  
 self.dfs(adjList, next, visited)  
  
 **def** dfs(self, adjList, start):  
 visited=set()  
 print(**"DFS : Starting with Vertex "**, start)  
 visited.add(start)  
 S = LifoQueue()  
 S.put(start)  
 **while not** S.empty():  
 v = S.get()  
 print(v, end=**"->"**)  
 neighber = adjList[v]  
 **for** u **in** neighber:  
 **if** u **not in** visited:  
 visited.add(u)  
 S.put(u)

1. Write code for the following applications of the DFS  
    ***#For task-2* Topological Sort** **def** doTS(self, adjList):  
    visited = defaultdict()  
    **for** vtx **in** adjList:  
    visited[vtx] = **False** result = []  
    **for** v **in** visited:  
    self.dfsTS(v, adjList, visited, result)  
    print(result)  
     
    **def** dfsTS(self, v, adjList, visited, result):  
    **if not** visited[v]:  
    visited[v] = **True  
    for** neighber **in** adjList[v]:  
    self.dfsTS(neighber, adjList, visited, result)  
    result.insert(0, v)  
    ***# for task-2* Connected components****def** findCC(self, adjList):  
    visited = set()  
    colonList = []  
     
    **for** v **in** adjList:  
    **if** v **not in** visited:  
    colon = self.dfsCC(adjList, [], v ,visited)  
    colonList.append(colon)  
    print(**f"Connected Components : {**len(colonList)**}"**)  
    print(colonList)  
     
    **def** dfsCC(self, adjList, colon, v, vistted):  
    **if** v **not in** vistted:  
    vistted.add(v)  
    colon.append(v)  
    n = adjList[v]  
    **for** vtx **in** n:  
    **if** vtx **not in** vistted:  
    self.dfsCC( adjList, colon, vtx, vistted)  
     
    **return** colon  
   ***# For task-2* N-Queens Problem****class** EightQueens:  
    **def** \_\_init\_\_(self, NQ):  
    self.NQ = NQ  
    self.solutions = 0  
    self.NN = 0  
     
    **def** solve(self):  
    board = [-1] \* self.NQ  
    self.dfsPQ(board, 0)  
    print(**"Found "**, self.solutions, **"solutions."**)  
    print(**"Nodes "**, self.NN)  
     
     
    **def** dfsPQ(self, board, row):  
    **if** row == self.NQ: *# if possible solution found  
    #print(board)* self.solutions += 1  
    **else**:  
    **for** col **in** range(self.NQ):  
    **if not** self.isAttack(board, row, col):  
    self.NN += 1  
    board[row] = col  
    self.dfsPQ(board, row+1)  
     
     
    **def** isAttack(self, board, row, col):  
    **for** i **in** range(row):  
    **if** board[i] == col **or** \  
    board[i] - i == col - row **or** \  
    board[i] + i == col + row:  
    **return True  
     
    return False**

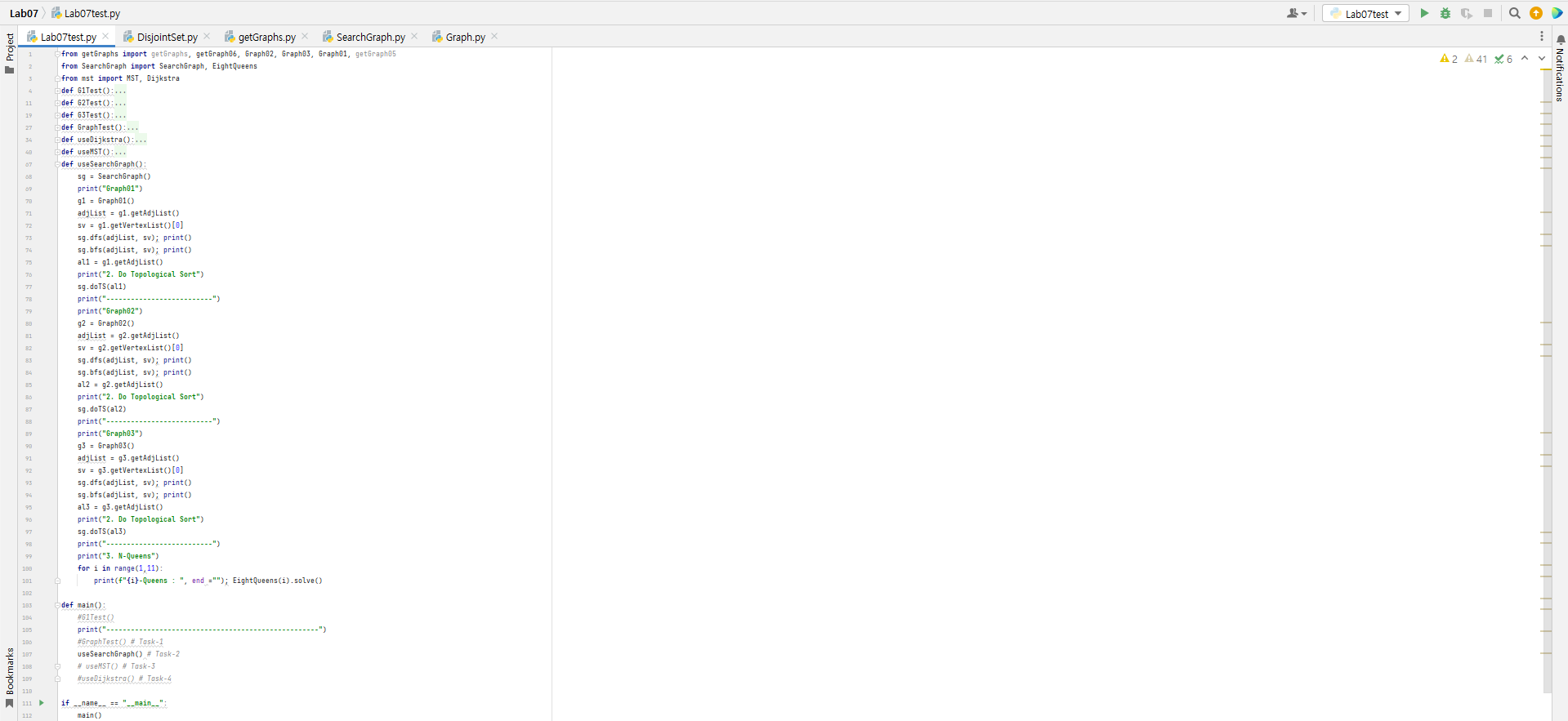
**from** Graph **import** \*  
**from** SearchGraph **import** \*  
  
  
**def** Graph01():  
 sg = SearchGraph()  
 g = Graph(**False**)  
 v1 = Vertex(**"A"**)  
 v2 = Vertex(**"B"**)  
 v3 = Vertex(**"C"**)  
 v4 = Vertex(**"D"**)  
 v5 = Vertex(**"E"**)  
 v6 = Vertex(**"F"**)  
 v7 = Vertex(**"G"**)  
 v8 = Vertex(**"H"**)  
  
 e1 = Edge(v1, v2, 1)  
 e2 = Edge(v1, v3, 1)  
 e3 = Edge(v2, v4, 1)  
 e4 = Edge(v3, v4, 1)  
 e5 = Edge(v4, v6, 1)  
 e6 = Edge(v3, v5, 1)  
 e7 = Edge(v5, v7, 1)  
 e8 = Edge(v5, v8, 1)  
 e9 = Edge(v7, v8, 1)  
  
 g.addVertex(v1)  
 g.addVertex(v2)  
 g.addVertex(v3)  
 g.addVertex(v4)  
 g.addVertex(v5)  
 g.addVertex(v6)  
 g.addVertex(v7)  
 g.addVertex(v8)  
 print(**"1. Connected Components"**)  
 sg.findCC(g.getAdjList())  
 g.addEdge(e1)  
 g.addEdge(e2)  
 g.addEdge(e3)  
 sg.findCC(g.getAdjList())  
 g.addEdge(e4)  
 g.addEdge(e5)  
 g.addEdge(e6)  
 sg.findCC(g.getAdjList())  
 g.addEdge(e7)  
 g.addEdge(e8)  
 g.addEdge(e9)  
 sg.findCC(g.getAdjList())  
 **return** g  
  
**def** Graph02():  
 sg = SearchGraph()  
 g = Graph(**False**)  
 v1 = Vertex(**"v1"**); v2 = Vertex(**"v2"**); v3 = Vertex(**"v3"**); v4 = Vertex(**"v4"**); v5 = Vertex(**"v5"**); v6 = Vertex(**"v6"**); v7 = Vertex(**"v7"**)  
  
 g.addVertex(v1); g.addVertex(v2); g.addVertex(v3); g.addVertex(v4)  
 g.addVertex(v5); g.addVertex(v6); g.addVertex(v7)  
  
 sg.findCC(g.getAdjList())  
 e1 = Edge(v1, v2, 2); e2 = Edge(v1, v3, 4); e3 = Edge(v1, v4, 1)  
 e4 = Edge(v2, v4, 3); e5 = Edge(v2, v5, 10); e6 = Edge(v3, v4, 2)  
 e7 = Edge(v3, v6, 5); e8 = Edge(v4, v5, 7); e9 = Edge(v4, v6, 8)  
 e10 = Edge(v4, v7, 4); e11 = Edge(v5, v7, 6); e12 = Edge(v6, v7, 1)  
  
 print(**"1. Connected Components"**)  
 g.addEdge(e1); g.addEdge(e2); g.addEdge(e3)  
 sg.findCC(g.getAdjList())  
  
 g.addEdge(e4); g.addEdge(e5); g.addEdge(e6)  
 sg.findCC(g.getAdjList())  
  
 g.addEdge(e7); g.addEdge(e8); g.addEdge(e9)  
 sg.findCC(g.getAdjList())  
  
 g.addEdge(e10); g.addEdge(e11); g.addEdge(e12)  
 sg.findCC(g.getAdjList())  
 **return** g  
  
  
**def** Graph03():  
 sg = SearchGraph()  
 g = Graph(**True**)  
 v1=Vertex(**"v1"**)  
 v2=Vertex(**"v2"**)  
 v3=Vertex(**"v3"**)  
 v4=Vertex(**"v4"**)  
 v5=Vertex(**"v5"**)  
 v6=Vertex(**"v6"**)  
 v7=Vertex(**"v7"**)  
  
 g.addVertex(v1)  
 g.addVertex(v2)  
 g.addVertex(v3)  
 g.addVertex(v4)  
 g.addVertex(v5)  
 g.addVertex(v6)  
 g.addVertex(v7)  
  
 e1 = Edge(v1, v2, 4); e2 = Edge(v1, v6, 10); e3 = Edge(v2, v1, 3); e4 = Edge(v2, v4, 18)  
 e5 = Edge(v3, v2, 6); e6 = Edge(v4, v2, 5); e7 = Edge(v4, v5, 2); e8 = Edge(v4, v6, 19)  
 e9 = Edge(v4, v7, 5); e10 = Edge(v5, v4, 1); e11 = Edge(v6, v7, 10); e12 = Edge(v7, v4, 8)  
 e13 = Edge(v5, v3, 12); e14 = Edge(v4, v3, 15)  
 print(**"1. Connected Components"**)  
 sg.findCC(g.getAdjList())  
 g.addEdge(e1)  
 g.addEdge(e2)  
 g.addEdge(e3)  
 g.addEdge(e4)  
 g.addEdge(e5)  
 sg.findCC(g.getAdjList())  
 g.addEdge(e6)  
 g.addEdge(e7)  
 g.addEdge(e8)  
 g.addEdge(e9)  
 g.addEdge(e10)  
 sg.findCC(g.getAdjList())  
 g.addEdge(e11)  
 g.addEdge(e12)  
 g.addEdge(e13)  
 g.addEdge(e14)  
 sg.findCC(g.getAdjList())  
 **return** g

**getGraphs.py**

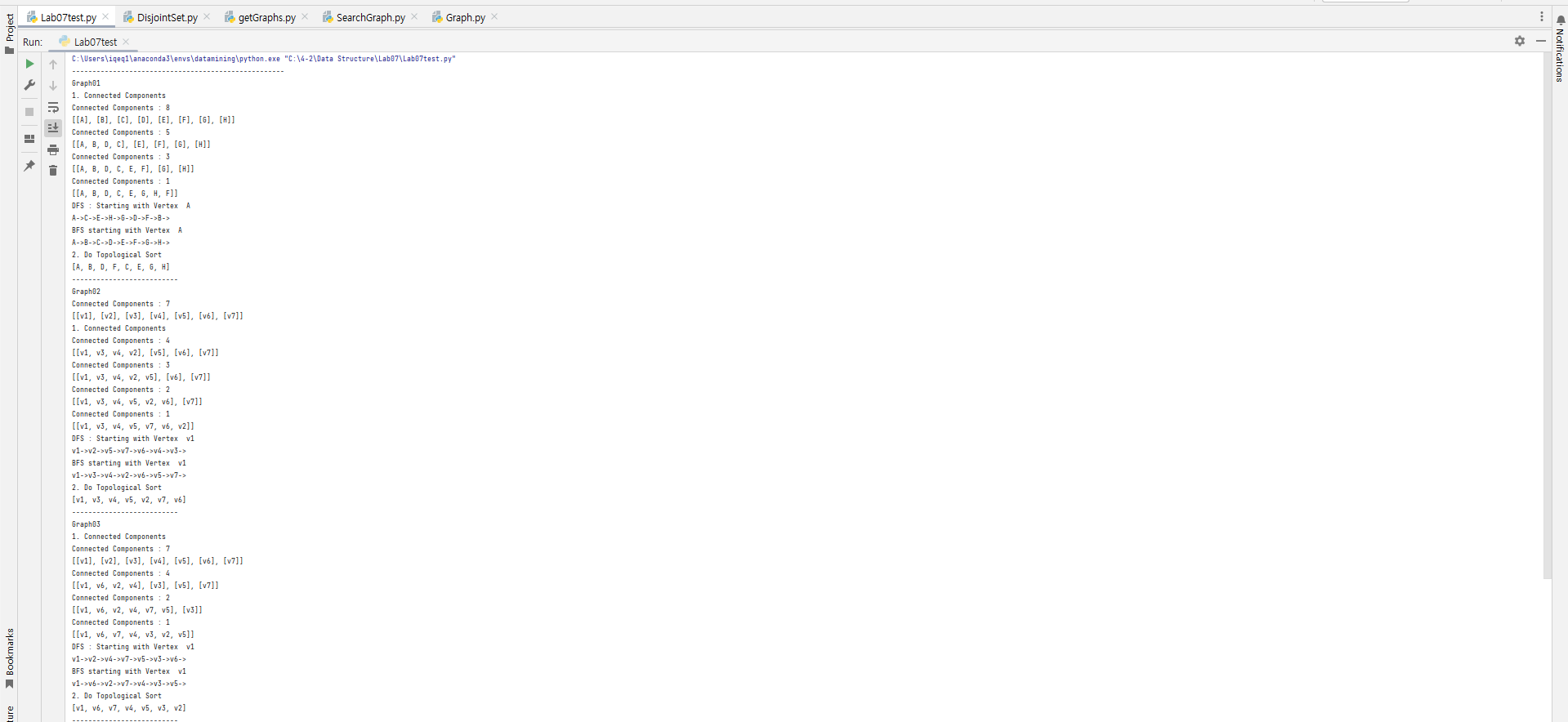
**from** Graph **import** \*  
**from** SearchGraph **import** \*  
  
  
**def** Graph01():  
 sg = SearchGraph()  
 g = Graph(**False**)  
 v1 = Vertex(**"A"**)  
 v2 = Vertex(**"B"**)  
 v3 = Vertex(**"C"**)  
 v4 = Vertex(**"D"**)  
 v5 = Vertex(**"E"**)  
 v6 = Vertex(**"F"**)  
 v7 = Vertex(**"G"**)  
 v8 = Vertex(**"H"**)  
  
 e1 = Edge(v1, v2, 1)  
 e2 = Edge(v1, v3, 1)  
 e3 = Edge(v2, v4, 1)  
 e4 = Edge(v3, v4, 1)  
 e5 = Edge(v4, v6, 1)  
 e6 = Edge(v3, v5, 1)  
 e7 = Edge(v5, v7, 1)  
 e8 = Edge(v5, v8, 1)  
 e9 = Edge(v7, v8, 1)  
  
 g.addVertex(v1)  
 g.addVertex(v2)  
 g.addVertex(v3)  
 g.addVertex(v4)  
 g.addVertex(v5)  
 g.addVertex(v6)  
 g.addVertex(v7)  
 g.addVertex(v8)  
 print(**"1. Connected Components"**)  
 sg.findCC(g.getAdjList())  
 g.addEdge(e1)  
 g.addEdge(e2)  
 g.addEdge(e3)  
 sg.findCC(g.getAdjList())  
 g.addEdge(e4)  
 g.addEdge(e5)  
 g.addEdge(e6)  
 sg.findCC(g.getAdjList())  
 g.addEdge(e7)  
 g.addEdge(e8)  
 g.addEdge(e9)  
 sg.findCC(g.getAdjList())  
 **return** g  
  
**def** Graph02():  
 sg = SearchGraph()  
 g = Graph(**False**)  
 v1 = Vertex(**"v1"**); v2 = Vertex(**"v2"**); v3 = Vertex(**"v3"**); v4 = Vertex(**"v4"**); v5 = Vertex(**"v5"**); v6 = Vertex(**"v6"**); v7 = Vertex(**"v7"**)  
  
 g.addVertex(v1); g.addVertex(v2); g.addVertex(v3); g.addVertex(v4)  
 g.addVertex(v5); g.addVertex(v6); g.addVertex(v7)  
  
 sg.findCC(g.getAdjList())  
 e1 = Edge(v1, v2, 2); e2 = Edge(v1, v3, 4); e3 = Edge(v1, v4, 1)  
 e4 = Edge(v2, v4, 3); e5 = Edge(v2, v5, 10); e6 = Edge(v3, v4, 2)  
 e7 = Edge(v3, v6, 5); e8 = Edge(v4, v5, 7); e9 = Edge(v4, v6, 8)  
 e10 = Edge(v4, v7, 4); e11 = Edge(v5, v7, 6); e12 = Edge(v6, v7, 1)  
  
 print(**"1. Connected Components"**)  
 g.addEdge(e1); g.addEdge(e2); g.addEdge(e3)  
 sg.findCC(g.getAdjList())  
  
 g.addEdge(e4); g.addEdge(e5); g.addEdge(e6)  
 sg.findCC(g.getAdjList())  
  
 g.addEdge(e7); g.addEdge(e8); g.addEdge(e9)  
 sg.findCC(g.getAdjList())  
  
 g.addEdge(e10); g.addEdge(e11); g.addEdge(e12)  
 sg.findCC(g.getAdjList())  
 **return** g  
  
  
**def** Graph03():  
 sg = SearchGraph()  
 g = Graph(**True**)  
 v1=Vertex(**"v1"**)  
 v2=Vertex(**"v2"**)  
 v3=Vertex(**"v3"**)  
 v4=Vertex(**"v4"**)  
 v5=Vertex(**"v5"**)  
 v6=Vertex(**"v6"**)  
 v7=Vertex(**"v7"**)  
  
 g.addVertex(v1)  
 g.addVertex(v2)  
 g.addVertex(v3)  
 g.addVertex(v4)  
 g.addVertex(v5)  
 g.addVertex(v6)  
 g.addVertex(v7)  
  
 e1 = Edge(v1, v2, 4); e2 = Edge(v1, v6, 10); e3 = Edge(v2, v1, 3); e4 = Edge(v2, v4, 18)  
 e5 = Edge(v3, v2, 6); e6 = Edge(v4, v2, 5); e7 = Edge(v4, v5, 2); e8 = Edge(v4, v6, 19)  
 e9 = Edge(v4, v7, 5); e10 = Edge(v5, v4, 1); e11 = Edge(v6, v7, 10); e12 = Edge(v7, v4, 8)  
 e13 = Edge(v5, v3, 12); e14 = Edge(v4, v3, 15)  
 print(**"1. Connected Components"**)  
 sg.findCC(g.getAdjList())  
 g.addEdge(e1)  
 g.addEdge(e2)  
 g.addEdge(e3)  
 g.addEdge(e4)  
 g.addEdge(e5)  
 sg.findCC(g.getAdjList())  
 g.addEdge(e6)  
 g.addEdge(e7)  
 g.addEdge(e8)  
 g.addEdge(e9)  
 g.addEdge(e10)  
 sg.findCC(g.getAdjList())  
 g.addEdge(e11)  
 g.addEdge(e12)  
 g.addEdge(e13)  
 g.addEdge(e14)  
 sg.findCC(g.getAdjList())  
 **return** g

**Results/Output**

**Lab07Test.py**

****

**Result**

****

****

**Task-3:**  **Minimum Spanning Trees:** Write code for Prims Algorithm and Kruskal Algorithm to get minimum spanning trees. Test your code using a graph in Figure 2.

**Code**

DisjointSet.py

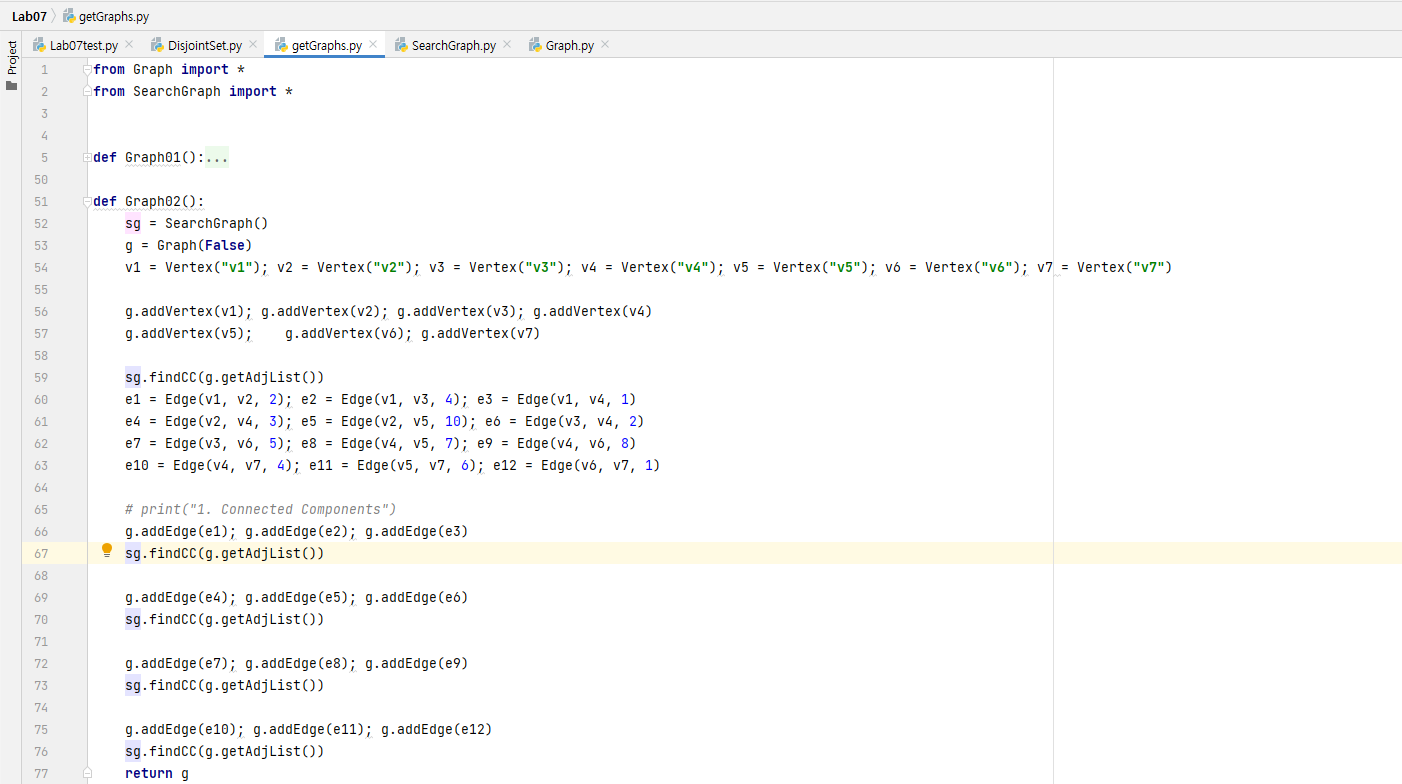
*# disjointSet***class** DisjointSet:  
 **def** \_\_init\_\_(self):  
 self.parent = dict()  
 self.rank = dict()  
  
 **def** makeSet(self, v):  
 self.parent[v] = v  
 self.rank[v] = 0  
  
 **def** find(self, v):  
 *# !root* **if** self.parent[v] != v:  
 self.parent[v] = self.find(self.parent[v])  
 **return** self.parent[v]  
  
 **def** Union(self, v1, v2):  
 root1 = self.find(v1)  
 root2 = self.find(v2)  
  
 **if** root1 != root2:  
 **if** self.rank[root1] > self.rank[root2]:  
 self.parent[root2] = root1  
  
 **else**:  
 self.parent[root2] = root2  
 **if** self.rank[root1] == self.rank[root2]:  
 self.rank[root2] =+ 1

**MST Class**

**from** queue **import** PriorityQueue  
**from** Graph **import** Graph  
**from** DisjointSet **import** DisjointSet **as** DS

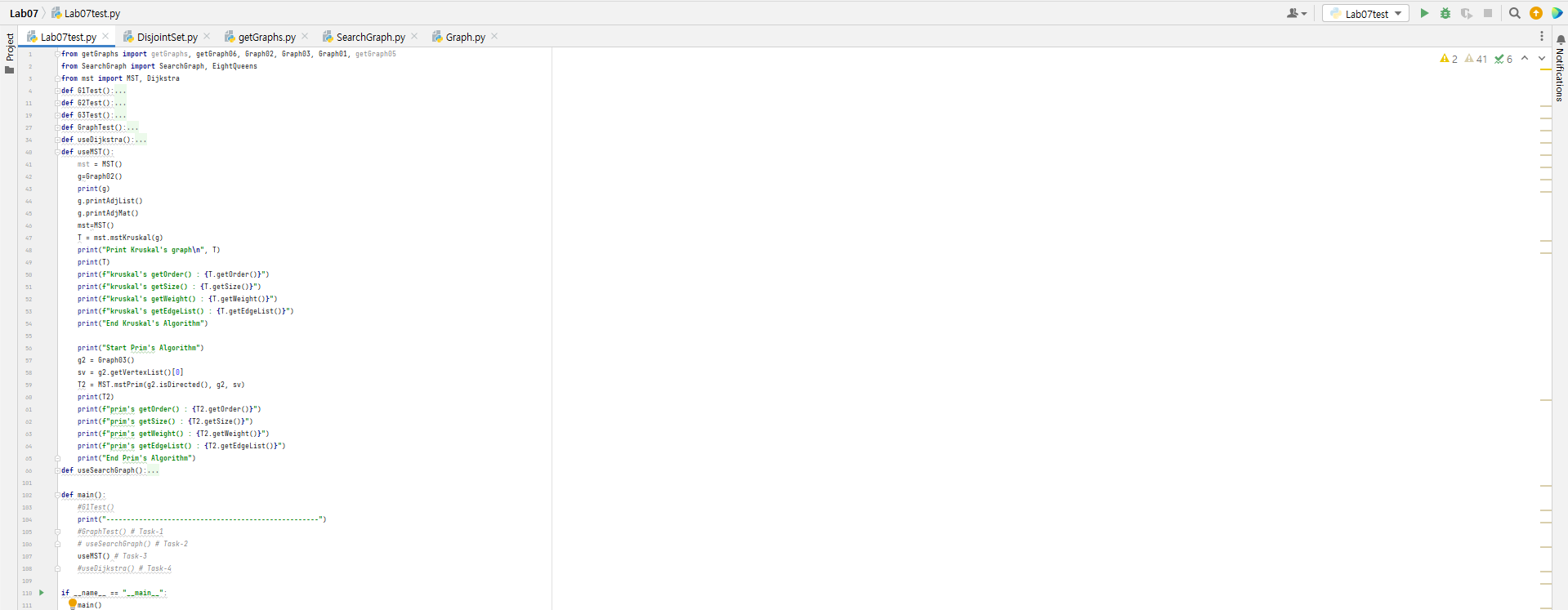
**class** MST:  
 **def** mstKruskal(self, g):  
 T = Graph(g.isDirected())  
 **for** v **in** g.getVertexList():  
 T.addVertex(v)  
 ds = DS()  
 **for** v **in** g.getVertexList():  
 ds.makeSet(v)  
 pq = PriorityQueue()  
 **for** e **in** g.getEdges():  
 pq.put(e)  
  
 numberOfEdges = 0  
 **while not** pq.empty():  
 e = pq.get()  
 p1 = ds.find(e.getU())  
 p2 = ds.find(e.getV())  
 **if** p1 != p2:  
 T.addEdge(e)  
 ds.Union(p1, p2)  
 numberOfEdges += 1  
 **return** T  
 **def** mstPrim(self, G, src):  
 T = Graph(G.isDirected())  
 eList = []  
  
 PQ = PriorityQueue()  
 **for** e **in** G.getNeighborEdges(src):  
 PQ.put(e)  
 T.addVertex(src)  
  
 **while** T.getOrder() != G.getOrder():  
 minE = PQ.get()  
 vtxV = minE.getV()  
 **if** vtxV **in** T.getVertexList():  
 **continue** T.addVertex(vtxV)  
 eList.append(minE)  
 **for** e **in** G.getNeighborEdges(vtxV):  
 PQ.put(e)  
 T.addEdge(e)  
  
 **for** e **in** eList:  
 print(e)  
  
 **return** T

**getGraphs.py**

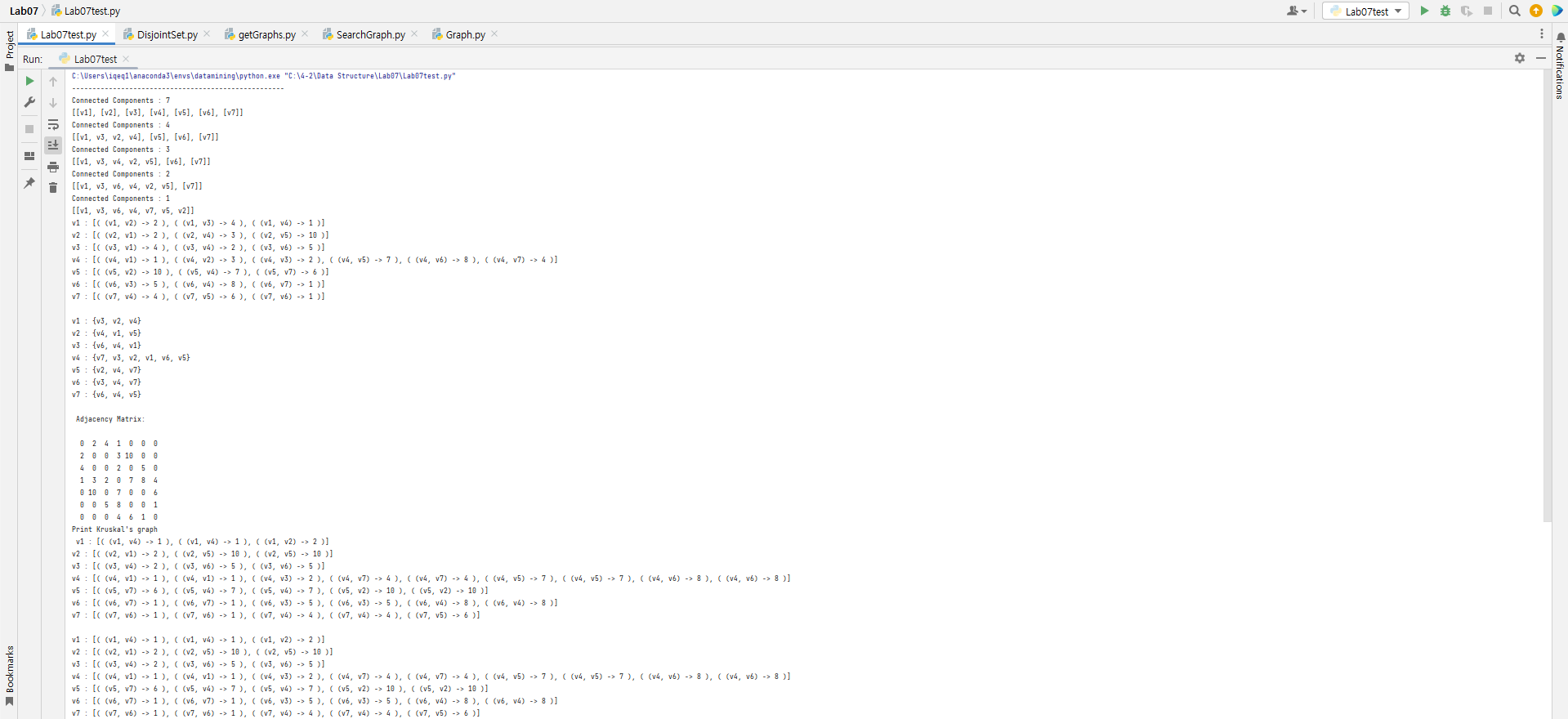
****

**Results/Output**

**Lab07Test.py**

****

Prim’s algorithm output

****

**Kruskal’s algorithm output**

****

**kruskal's getEdgeList() : [( (v1, v4) -> 1 ), ( (v1, v4) -> 1 ), ( (v1, v2) -> 2 ), ( (v2, v1) -> 2 ), ( (v2, v5) -> 10 ), ( (v2, v5) -> 10 ), ( (v3, v4) -> 2 ), ( (v3, v6) -> 5 ), ( (v3, v6) -> 5 ), ( (v4, v1) -> 1 ), ( (v4, v1) -> 1 ), ( (v4, v3) -> 2 ), ( (v4, v7) -> 4 ), ( (v4, v7) -> 4 ), ( (v4, v5) -> 7 ), ( (v4, v5) -> 7 ), ( (v4, v6) -> 8 ), ( (v4, v6) -> 8 ), ( (v5, v7) -> 6 ), ( (v5, v4) -> 7 ), ( (v5, v4) -> 7 ), ( (v5, v2) -> 10 ), ( (v5, v2) -> 10 ), ( (v6, v7) -> 1 ), ( (v6, v7) -> 1 ), ( (v6, v3) -> 5 ), ( (v6, v3) -> 5 ), ( (v6, v4) -> 8 ), ( (v6, v4) -> 8 ), ( (v7, v6) -> 1 ), ( (v7, v6) -> 1 ), ( (v7, v4) -> 4 ), ( (v7, v4) -> 4 ), ( (v7, v5) -> 6 )]**

**Task-4:**  **Single Source Shortest Paths:** Write code for the Dijkstra Algorithm to get single source shortest paths. Test your code using a graph in Figure 3.

**Code**

DisjointSet.py

*# disjointSet***class** DisjointSet:  
 **def** \_\_init\_\_(self):  
 self.parent = dict()  
 self.rank = dict()  
  
 **def** makeSet(self, v):  
 self.parent[v] = v  
 self.rank[v] = 0  
  
 **def** find(self, v):  
 *# !root* **if** self.parent[v] != v:  
 self.parent[v] = self.find(self.parent[v])  
 **return** self.parent[v]  
  
 **def** Union(self, v1, v2):  
 root1 = self.find(v1)  
 root2 = self.find(v2)  
  
 **if** root1 != root2:  
 **if** self.rank[root1] > self.rank[root2]:  
 self.parent[root2] = root1  
  
 **else**:  
 self.parent[root2] = root2  
 **if** self.rank[root1] == self.rank[root2]:  
 self.rank[root2] =+ 1

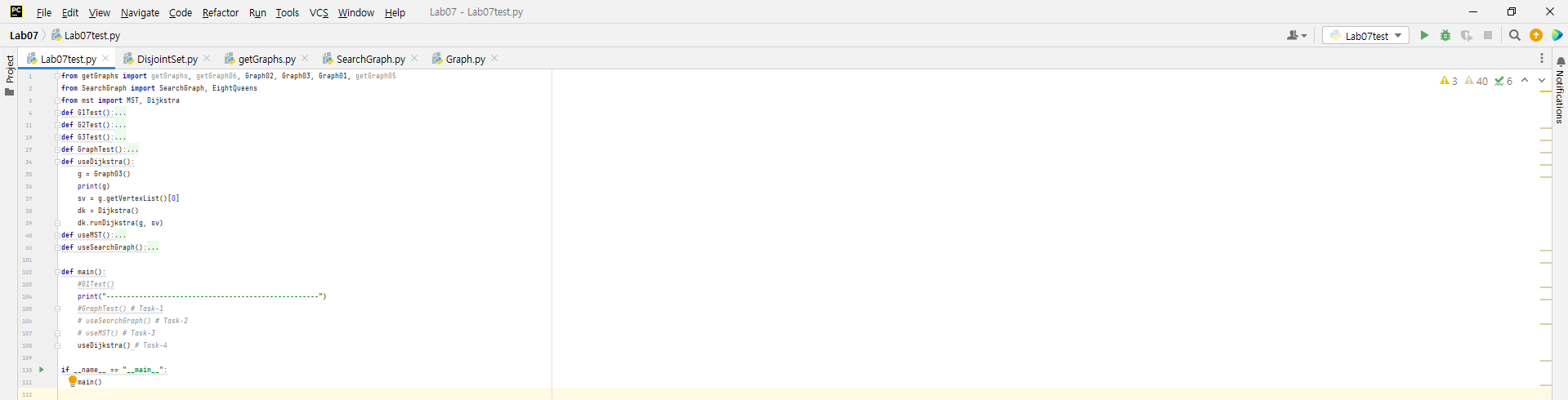
**Dijstra Class**

**from** queue **import** PriorityQueue  
**from** Graph **import** Graph  
**from** DisjointSet **import** DisjointSet **as** DS

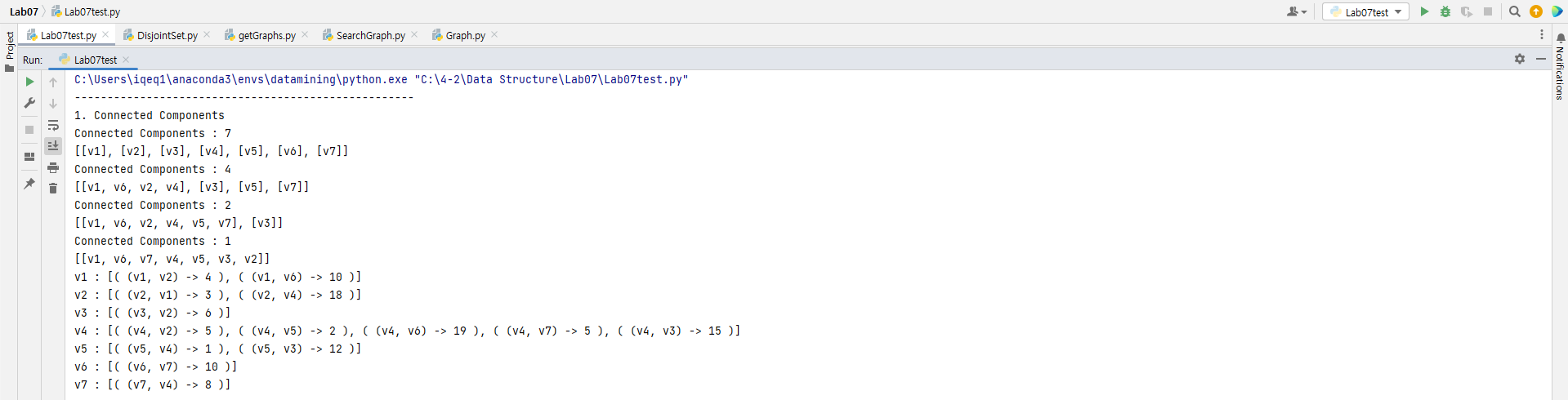
**class** Dijkstra:  
 **def** runDijkstra(self, G, src):  
 Known = {}  
 Dv = {}  
 Pv = {}  
 **for** vtx **in** G.getVertexList():  
 Known[vtx] = **False** Dv[vtx] = float(**"inf"**)  
 Pv[vtx] = **None** Known[src] = **True** Dv[src] = 0.0  
 PQ = PriorityQueue()  
 PQ.put((0, src))  
 EdgeDistance = 0.0  
 newDistance = 0.0  
  
 **while not** PQ.empty():  
 emin = PQ.get()[1]  
 **for** e **in** G.getNeighborEdges(emin):  
 EdgeDistance = e.getW()  
 newDistance =Dv[e.getU()]+ EdgeDistance  
 **if not** Known[e.getV()] **and** Dv[e.getV()] > newDistance:  
 Dv[e.getV()] = newDistance  
 Pv[e.getV()] = newDistance  
 PQ.put((newDistance, e.getV()))  
 Known[e.getU()] = **True  
 def** printConfigration(self, Known, Dv, Pv):  
 print(**"Configration"**)  
 **for** vtx **in** Known:  
 print(**f"{**vtx**}, {**Known[vtx]**}, {**Dv[vtx]**}, {**Pv[vtx]**}"**)

**Results/Output**

Lab07Test



Dijkstra Output

****

**Conclusion**

Conclude the Lab. Write your views about it, i.e. what have you learned from this lab? It was helpful or difficult etc

**Thank you. I can learn Graph data structures, DFS, BFS algorithms, MST such as “prim” and “Kruskal” algorithms , Dijkstra algorithms. These are good assignments for me.**