Name=Saransh Mittal
Rollno=2000290130145

Day-24

Problem 1 : Strongly Connected Components – Kosaraju's Algorithm: G-54 Problem Statement: Given a Directed Graph with V vertices (Numbered from 0 to V-1) and E edges, Find the number of strongly connected components in the graph.

```
from collections import defaultdict


class Graph:
  def __init__(self, vertices):
    self.V = vertices
    self.graph = defaultdict(list)

  def add_edge(self, u, v):
    self.graph[u].append(v)

  def dfs(self, v, visited, stack):
    visited[v] = True
    for i in self.graph[v]:
      if not visited[i]:
        self.dfs(i, visited, stack)
    stack.append(v)

  def transpose(self):
    g = Graph(self.V)
    for i in self.graph:
      for j in self.graph[i]:
        g.add_edge(j, i)
    return g

  def count_scc(self):
    stack = []
    visited = [False] * self.V
    for i in range(self.V):
```

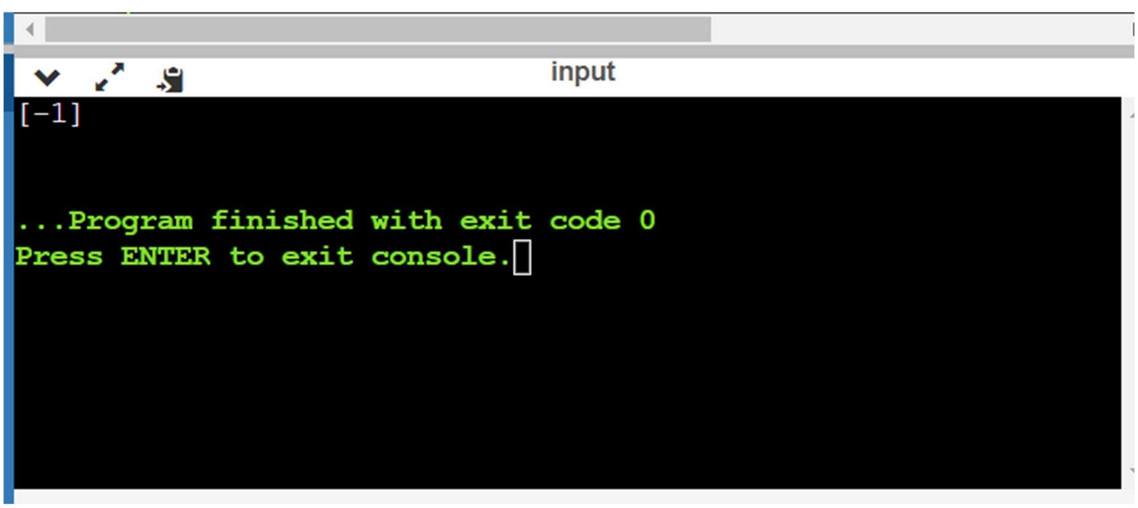
```
            if not visited[i]:
                self.dfs(i, visited, stack)

        transposed_graph = self.transpose()

        visited = [False] * self.V
        scc_count = 0

        while stack:
            v = stack.pop()
            if not visited[v]:
                transposed_graph.dfs(v, visited, [])
                scc_count += 1

        return scc_count


V = 5
graph = Graph(V)
graph.add_edge(0, 1)
graph.add_edge(1, 2)
graph.add_edge(2, 0)
graph.add_edge(1, 3)
graph.add_edge(3, 4)
num_scc = graph.count_scc()
print("Number of Strongly Connected Components:", num_scc)
```

```
input
Number of Strongly Connected Components: 3


...Program finished with exit code 0
Press ENTER to exit console.
```

Problem – 2 : Print Shortest Path – Dijkstra's Algorithm

```
import heapq


def dijkstra_shortest_path(graph, n):
    # Initialize distance and visited arrays
    distance = [float('inf')] * (n + 1)
    distance[1] = 0
    visited = [False] * (n + 1)

    # Priority queue to keep track of nodes to visit
    priority_queue = [(0, 1)]

    while priority_queue:
        dist, node = heapq.heappop(priority_queue)

        if visited[node]:
            continue

        visited[node] = True

        for neighbor, weight in graph[node]:
            if distance[neighbor] > distance[node] + weight:
                distance[neighbor] = distance[node] + weight
                heapq.heappush(priority_queue, (distance[neighbor], neighbor))

    if distance[n] == float('inf'):
        return [-1]
    else:
        path = []
        current = n
        while current != 0:
```

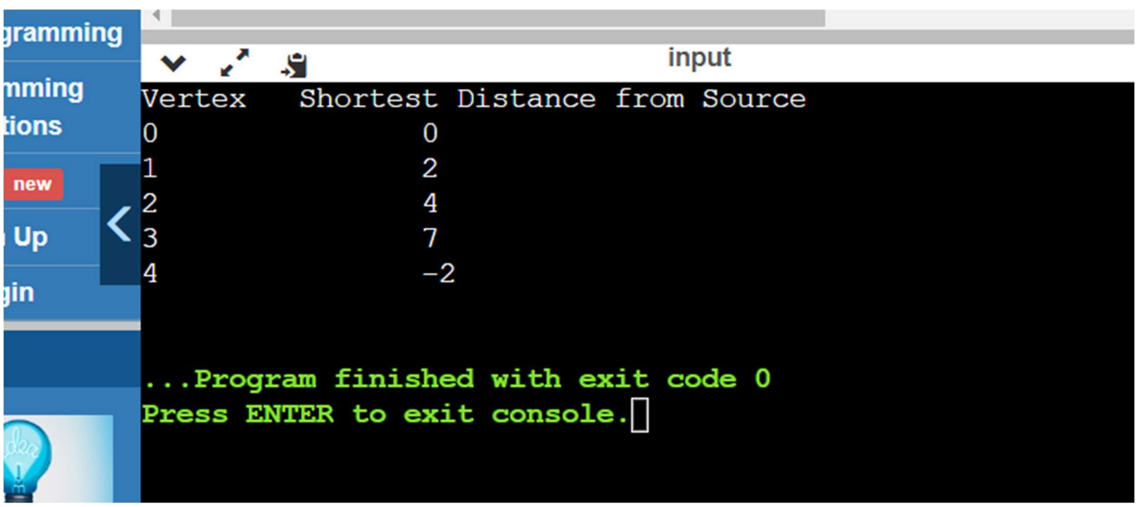
```
            path.append(current)
            for neighbor, weight in graph[current]:
                if distance[current] == distance[neighbor] + weight:
                    current = neighbor
                    break

        return path[::-1]

n = 4
m = 6



graph = {
    0: [(1, 2), (2, 4)],
    1: [(0, 2), (2, 1), (3, 7)],
    2: [(0, 4), (1, 1), (3, 3)],
    3: [(1, 7), (2, 3)]
}
shortest_path = dijkstra_shortest_path(graph, n)
print(shortest_path)
```



Problem – 3 Bellman Ford Algorithm

```
class Graph:
    def __init__(self, vertices):
        self.V = vertices
        self.graph = []

    def add_edge(self, u, v, w):
        self.graph.append([u, v, w])

    def print_solution(self, dist):
        print("Vertex \t Shortest Distance from Source")
        for i in range(self.V):
            print(f"{i}\t\t{dist[i]}")

    def bellman_ford(self, src):
        dist = [float("inf")] * self.V
        dist[src] = 0

        # Relax all edges V-1 times
        for _ in range(self.V - 1):
            for u, v, w in self.graph:
                if dist[u] != float("inf") and dist[u] + w < dist[v]:
                    dist[v] = dist[u] + w

        for u, v, w in self.graph:
            if dist[u] != float("inf") and dist[u] + w < dist[v]:
                print("Graph contains negative weight cycle")
                return

        self.print_solution(dist)
```

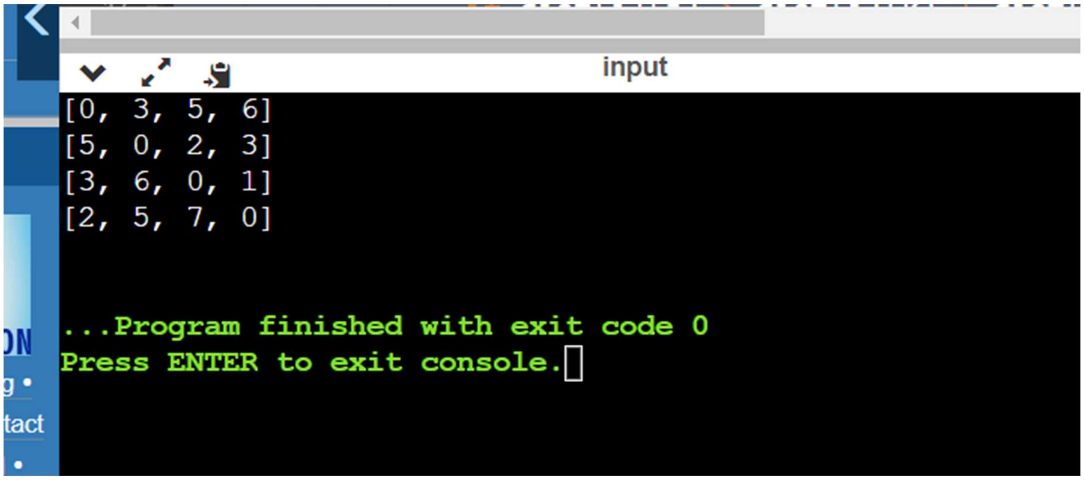
```
vertices = 5
graph = Graph(vertices)
graph.add_edge(0, 1, 6)
graph.add_edge(0, 3, 7)
graph.add_edge(1, 2, 5)
graph.add_edge(1, 3, 8)
graph.add_edge(1, 4, -4)
graph.add_edge(2, 1, -2)
graph.add_edge(3, 2, -3)
graph.add_edge(3, 4, 9)
graph.add_edge(4, 0, 2)
graph.add_edge(4, 2, 7)

source_vertex = 0
graph.bellman_ford(source_vertex)
```



Problem – 4 Floyd Warshall Algorithm

```
def floyd_warshall(graph):
    num_vertices = len(graph)
    dist = [[float('inf') for _ in range(num_vertices)] for _ in range(num_vertices)]

    for i in range(num_vertices):
        for j in range(num_vertices):
            if i == j:
                dist[i][j] = 0
            elif graph[i][j] is not None:
                dist[i][j] = graph[i][j]

    for k in range(num_vertices):
        for i in range(num_vertices):
            for j in range(num_vertices):
                # Check if the path through vertex k is shorter
                if dist[i][j] > dist[i][k] + dist[k][j]:
                    dist[i][j] = dist[i][k] + dist[k][j]

    return dist

graph = [
    [0, 3, None, 7],
    [8, 0, 2, None],
    [5, None, 0, 1],
    [2, None, None, 0]
]

result = floyd_warshall(graph)
```

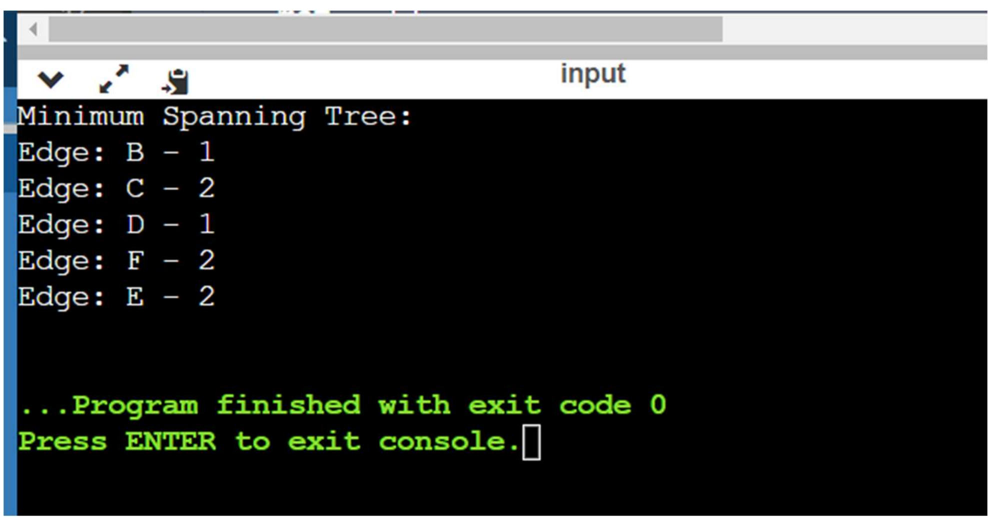
```
for row in result:
    print(row)
```



Problem 5 Prim's Algorithm – Minimum Spanning Tree

```
import heapq
class Graph:
    def __init__(self):
        self.graph = {}

    def add_edge(self, u, v, weight):
        if u not in self.graph:
            self.graph[u] = []
        if v not in self.graph:
            self.graph[v] = []
        self.graph[u].append((v, weight))
        self.graph[v].append((u, weight))

    def prim_mst(self):
        start_vertex = next(iter(self.graph))
        mst = []
        visited = set()
        priority_queue = [(0, start_vertex)]

        while priority_queue:
            weight, current_vertex = heapq.heappop(priority_queue)
            if current_vertex in visited:
                continue

            visited.add(current_vertex)
            if weight != 0:
                mst.append((current_vertex, weight))

            for neighbor, edge_weight in self.graph[current_vertex]:
```

```
            if neighbor not in visited:
                heapq.heappush(priority_queue, (edge_weight, neighbor))

        return mst

g = Graph()
g.add_edge('A', 'B', 1)
g.add_edge('A', 'C', 4)
g.add_edge('B', 'C', 2)
g.add_edge('B', 'D', 5)
g.add_edge('C', 'D', 1)
g.add_edge('D', 'E', 3)
g.add_edge('E', 'F', 2)
g.add_edge('F', 'D', 2)

mst = g.prim_mst()
print("Minimum Spanning Tree:")
for vertex, weight in mst:
    print(f"Edge: {vertex} - {weight}")
```



Problem – 6 : Kruskal's Algorithm – Minimum Spanning Tree

```
class UnionFind:
    def __init__(self, n):
        self.parent = list(range(n))
        self.rank = [0] * n

    def find(self, x):
        if self.parent[x] != x:
            self.parent[x] = self.find(self.parent[x])  # Path compression
        return self.parent[x]

    def union(self, x, y):
        root_x, root_y = self.find(x), self.find(y)

        if root_x == root_y:
            return False
        if self.rank[root_x] < self.rank[root_y]:
            self.parent[root_x] = root_y
        elif self.rank[root_x] > self.rank[root_y]:
            self.parent[root_y] = root_x
        else:
            self.parent[root_y] = root_x
            self.rank[root_x] += 1
        return True

def kruskal(graph):
    n = len(graph)
    edges = []
    for i in range(n):
        for j in range(i + 1, n):
            if graph[i][j] != 0:
```

```
                edges.append((i, j, graph[i][j]))

    edges.sort(key=lambda x: x[2])
    uf = UnionFind(n)
    mst = []

    for edge in edges:
        u, v, weight = edge
        if uf.union(u, v):
            mst.append(edge)

    return mst

graph = [
    [0, 2, 0, 6, 0],
    [2, 0, 3, 8, 5],
    [0, 3, 0, 0, 7],
    [6, 8, 0, 0, 9],
    [0, 5, 7, 9, 0],
]

minimum_spanning_tree = kruskal(graph)
print("Minimum Spanning Tree:")
for edge in minimum_spanning_tree:
    print(f"{edge[0]} -- {edge[1]}: weight {edge[2]}")
```

```
Minimum Spanning Tree:
0 -- 1: weight 2
1 -- 2: weight 3
1 -- 4: weight 5
0 -- 3: weight 6


...Program finished with exit code 0
Press ENTER to exit console.
```