```
print("ROHAN WAYAL")

# Representing a college campus graph using adjacency matrix
# Nodes: Departments/Institutes
# Edges: Distance between departments

# Departments:
# 0: Computer Science (CS)
# 1: Information Technology (IT)
# 2: Electronics (EC)
# 3: Mechanical (ME)
# 4: Civil (CE)
# 5: Electrical (EE)

# Adjacency matrix with distances (0 means no direct edge)
graph = [
    [0, 2, 0, 6, 0, 0],
    [2, 0, 3, 8, 5, 0],
    [0, 3, 0, 0, 7, 0],
    [6, 8, 0, 0, 9, 0],
    [0, 5, 7, 9, 0, 1],
    [0, 0, 0, 0, 1, 0]
]

# Number of vertices
V = len(graph)

# --- Kruskal's Algorithm Implementation ---
class DisjointSet:
    def __init__(self, n):
        self.parent = list(range(n))
```

```
        self.rank = [0]*n

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

    def union(self, u, v):
        root_u = self.find(u)
        root_v = self.find(v)

        if root_u == root_v:
            return False

        if self.rank[root_u] < self.rank[root_v]:
            self.parent[root_u] = root_v
        elif self.rank[root_u] > self.rank[root_v]:
            self.parent[root_v] = root_u
        else:
            self.parent[root_v] = root_u
            self.rank[root_u] += 1
        return True

def kruskal(graph):
    edges = []
    for i in range(V):
        for j in range(i+1, V):
            if graph[i][j] != 0:
                edges.append((graph[i][j], i, j))
    edges.sort(key=lambda x: x[0])
```

```
    ds = DisjointSet(V)
    mst = []
    total_weight = 0

    for weight, u, v in edges:
        if ds.union(u, v):
            mst.append((u, v, weight))
            total_weight += weight

    return mst, total_weight



# --- Prim's Algorithm Implementation ---
import heapq

def prim(graph):
    visited = [False] * V
    min_heap = [(0, 0)]  # (weight, vertex)
    mst = []
    total_weight = 0

    while min_heap:
        weight, u = heapq.heappop(min_heap)
        if visited[u]:
            continue
        visited[u] = True
        total_weight += weight
        # If weight is not zero, means not starting node, add edge
        if weight != 0:
            mst.append(u)
        for v in range(V):
```

```
            if graph[u][v] != 0 and not visited[v]:
                heapq.heappush(min_heap, (graph[u][v], v))

    # The above just tracks visited nodes and total weight, to print edges,
    # we need a slight modification to track parent nodes
    # Let's implement with parent array

def prim_with_edges(graph):
    visited = [False] * V
    parent = [-1] * V
    key = [float('inf')] * V
    key[0] = 0

    for _ in range(V):
        # Pick min key vertex from unvisited vertices
        min_key = float('inf')
        u = -1
        for i in range(V):
            if not visited[i] and key[i] < min_key:
                min_key = key[i]
                u = i

        visited[u] = True
        # Update keys and parents of adjacent vertices
        for v in range(V):
            if graph[u][v] != 0 and not visited[v] and graph[u][v] < key[v]:
                key[v] = graph[u][v]
                parent[v] = u

    mst = []
    total_weight = 0
```

```
    for i in range(1, V):
        mst.append((parent[i], i, graph[parent[i]][i]))
        total_weight += graph[parent[i]][i]

    return mst, total_weight


# --- Main Program ---

departments = ["CS", "IT", "EC", "ME", "CE", "EE"]

mst_kruskal, weight_kruskal = kruskal(graph)
print("\nMinimum Spanning Tree using Kruskal's Algorithm:")
for u, v, w in mst_kruskal:
    print(f"{departments[u]} -- {departments[v]} : {w}")
print("Total weight (distance):", weight_kruskal)

mst_prim, weight_prim = prim_with_edges(graph)
print("\nMinimum Spanning Tree using Prim's Algorithm:")
for u, v, w in mst_prim:
    print(f"{departments[u]} -- {departments[v]} : {w}")
print("Total weight (distance):", weight_prim)
```