Assign 4

**SELECTION SORTING**

L=[3,4,5,6,1.7]

Print ('areay before sorting", 1)

min=0

for i in range (1.len (2)):

if L[min] > L[i] :

min = i

L[min], L[0]=L[0],L[min]

L[min], L[1] = L[1] , [min]

L [min], L[2] = L[2], L[min] L[min], L[3]=L[3], L [min]

Print("Sorted array", L)

**2.Prim’s Minimal spanning TREE Algorithm**

INF 9999999

V = 5

G = [[0, 9, 75, 0, 0),

[9, 0, 95, 19, 42],

[75, 95, 0, 51, 66],

[0, 19, 51, 0, 31],

[0, 42, 66, 31, 0]]

selected = [0, 0, 0, 0, 0]

no\_edge = 0

selected[0] = True

print("Edge: Weight\n")

while (no\_edge < V - 1):

minimum = INF

x = 0

y = 0

for i in range(V):

if selected[i]:

for j in range(V):

if ((not selected[j]) and G[i][j]):

if minimum > G[i][j]:

minimum = G[i][j]

x = i

y = j

print(str(x) + "-" + str(y) + ":" + str(6[x][y]))

selected[y] = True

no\_edge += 1

**3.Single Source Shortest Path Problem**

class Graph:\_\_init

def\_init\_(self,vertices):

self.V vertices

self.graph = []

def add\_edge(self, s,d, w):

self.graph.append([s, d, w])

def print\_solution (self, dist):

print("Vertex Distance from Source")

for i in range(self.V- 1):

print("{0}\t\t{1}".format(\*args: i, dist[i]))

def bellman\_ford(self, src):

dist = [float("Inf")) \* self.V

dist[src] = 0

for in range(self.V - 1):

for s, d, w in self.graph:

if dist[s] != float("Inf") and dist[s] + w < dist[d]:

dist[d] = dist[s] + w

for s, d, win self.graph:

if dist[s] != float("Inf") and dist[s] + w < dist[d]:

print("Graph contains negative weight cycle")

return

self.print\_solution (dist)

g = Graph(5)

g.add\_edge( 0, 1, 5)

g.add\_edge(0, 2, 4)

g.add\_edge( 1, 3, 3)

g.add\_edge(2, 1, 6)

g.add\_edge(3, 2, 2)

g.bellman\_ford(0)

**4.Job Scheduling Problem**

def sjn\_schedule (jobs):

jobs.sort(key=lambda x: x[1]) # Each job is (id, burst\_time)

schedule = []

current\_time=0

for job\_id, burst\_time in jobs:

schedule.append((job\_id, current\_time, current\_time + burst\_time))

current\_time += burst\_time

class Job:

def \_init\_(self, id, deadline, profit):

self.id = id

self.deadline = deadline

self.profit = profit

def job\_scheduling (jobs, n):

jobs.sort(key=lambda x: x.profit, reverse=True)

result = [None] \* n

slot = [False] \* n

total\_profit = 0

for job in jobs:

for j in range(min(n, job.deadline) 1, -1, -1):

if not slot[j]:

result[j] = job.id

slot[j] = True

total\_profit += job.profit

break

return total\_profit, result

jobs = [

job(‘j1’, ’2’ , ‘100’),

job(‘j2’, ’1’ , ‘19’),

job(‘j3’, ’2’ , ‘27’),

job(‘j4’, ’1’ , ‘25’),

job(‘j5’, ’3’ , ‘15’) ]

n = len(jobs)

max\_profit, scheduled\_jobs = job\_scheduling(jobs, n)

print(f"Maximum profit: {max\_profit}")

print(f"Scheduled jobs: {scheduled\_jobs}")

jobs = [("Job1", 6), ("Job2", 8), ("Job3", 7), ("Job4", 3)]

schedule = sjn\_schedule(jobs)

print("Scheduled Jobs:", schedule)

**5.Dijkstra’s Minimal Spanning Tree Algorithm**

Import sys

V=5

def min\_distance(dist, visited):

min\_val=sys.maxsize

min\_index= -1

for v in range (V):

if not visited [v] and dist[v] < min\_val:

min\_val dist[v]

min\_index=v

return min\_index

def dijkstra (graph,src):

dist=[sys.maxsize]\*V

dist[src]=0

visited=[False]\*V

for in range(V-1): -

u=min\_distance (dist, visited)

visited[u]=True

for v in range (V):

if graph[u] [v] and not visited[v] and dist[u]+ graph [u] [v] < dist[v]

dist[v]=dist[u] + graph [u] [v]

print("vertex\t distance from source")

for i in range (V):

print(1,"\t\t\t", dist[i])

graph=[[0,9,75,0,0],

[9,0,95,19,42],

[75,95,0,51,66],

[0,19,51,0,31],

[0,42,66,31,0]]

dijkstra (graph, src: 0)

**6. Kruskal’s Minimal Spanning Tree 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 find(self, parent, i):

if parent[i] == i:

return i

return self.find(parent, parent[i])

def apply\_union(self, parent, rank, x, y):

xroot = self.find(parent, x)

yroot = self.find(parent, y)

if rank[xroot] < rank[yroot]:

parent[xroot] = yroot

elif rank[xroot] > rank[yroot]:

parent[yroot] = xroot

else:

parent[yroot] = xroot

rank[xroot] += 1

def kruskal\_algo(self):

result = []

i, e = 0, 0

self.graph = sorted(self.graph, key=lambda item: item[2])

parent = []

rank = []

for node in range(self.V):

parent.append(node)

rank.append(0)

while e < self.V - 1:

u, v, w = self.graph[i]

i = i + 1

x = self.find(parent, u)

y = self.find(parent, v)

if x != y:

e = e + 1

result.append([u, v, w])

self.apply\_union(parent, rank, x, y)