DAY – 11

1.Bellman Ford algorithm

class Graph:

def \_\_init\_\_(self, vertices):

self.V = vertices

self.edges = []

def add\_edge(self, u, v, w):

self.edges.append((u, v, w))

def bellman\_ford(self, src):

dist = [float('inf')] \* self.V

dist[src] = 0

for \_ in range(self.V - 1):

for u, v, w in self.edges:

if dist[u] != float('inf') and dist[u] + w < dist[v]:

dist[v] = dist[u] + w

for u, v, w in self.edges:

if dist[u] != float('inf') and dist[u] + w < dist[v]:

print("Graph contains negative weight cycle")

return

self.print\_solution(dist)

def print\_solution(self, dist):

print("Vertex Distance from Source")

for i in range(self.V):

print(f"{i}\t\t{dist[i]}")

g = Graph(5)

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

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

g.add\_edge(1, 2, 3)

g.add\_edge(1, 3, 2)

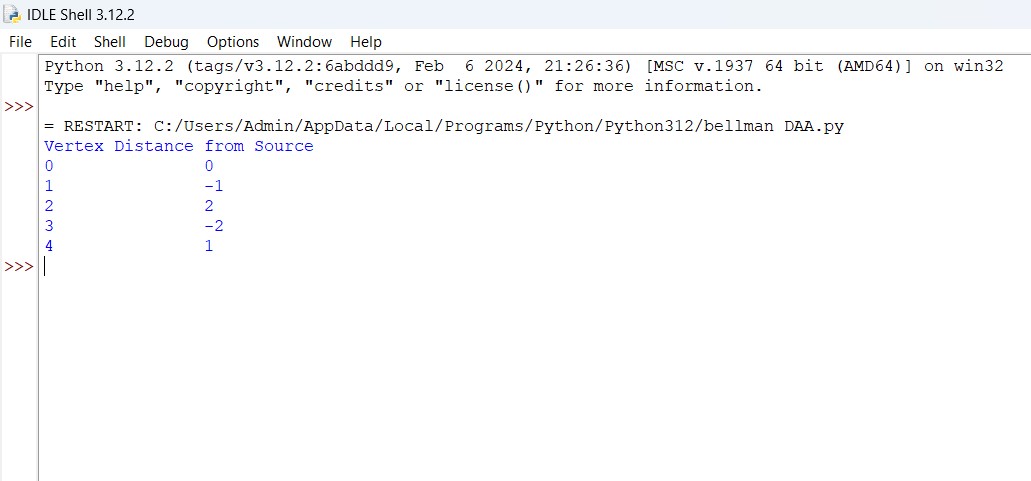
g.add\_edge(1, 4, 2)

g.add\_edge(3, 2, 5)

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

g.add\_edge(4, 3, -3)

g.bellman\_ford(0)



2. Warshalls algorithm

def warshall\_algorithm(graph):

V = len(graph)

dist = [[graph[i][j] for j in range(V)] for i in range(V)]

for k in range(V):

for i in range(V):

for j in range(V):

dist[i][j] = dist[i][j] or (dist[i][k] and dist[k][j])

print\_solution(dist)

def print\_solution(dist):

print("Following matrix shows the transitive closure of the given graph:")

for i in range(len(dist)):

for j in range(len(dist)):

print(dist[i][j], end=" ")

print()

graph = [[1, 1, 0, 0],

[0, 1, 1, 0],

[0, 0, 1, 1],

[0, 0, 0, 1]]

warshall\_algorithm(graph)



3. Coin change problem

def min\_coins(coins, amount):

dp = [float('inf')] \* (amount + 1)

dp[0] = 0

for coin in coins:

for x in range(coin, amount + 1):

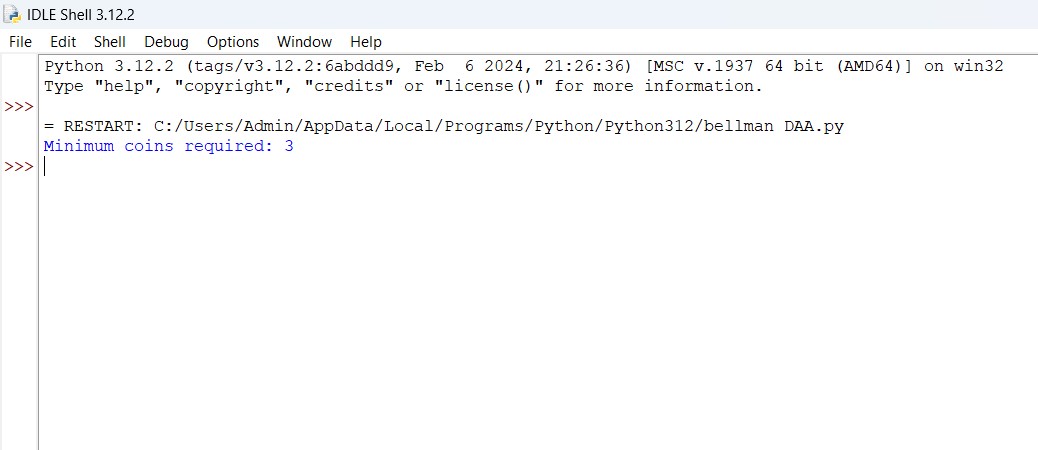
dp[x] = min(dp[x], dp[x - coin] + 1)

return dp[amount] if dp[amount] != float('inf') else -1

coins = [1, 2, 5]

amount = 11

print(f"Minimum coins required: {min\_coins(coins, amount)}")



4. Knapsack problem using greedy class Item:

def \_\_init\_\_(self, value, weight):

self.value = value

self.weight = weight

def knapsack\_greedy(items, capacity):

items = sorted(items, key=lambda item: item.value/item.weight, reverse=True)

total\_value = 0

for item in items:

if capacity >= item.weight:

capacity -= item.weight

total\_value += item.value

else:

fraction = capacity / item.weight

total\_value += item.value \* fraction

break

return total\_value

items = [Item(60, 10),

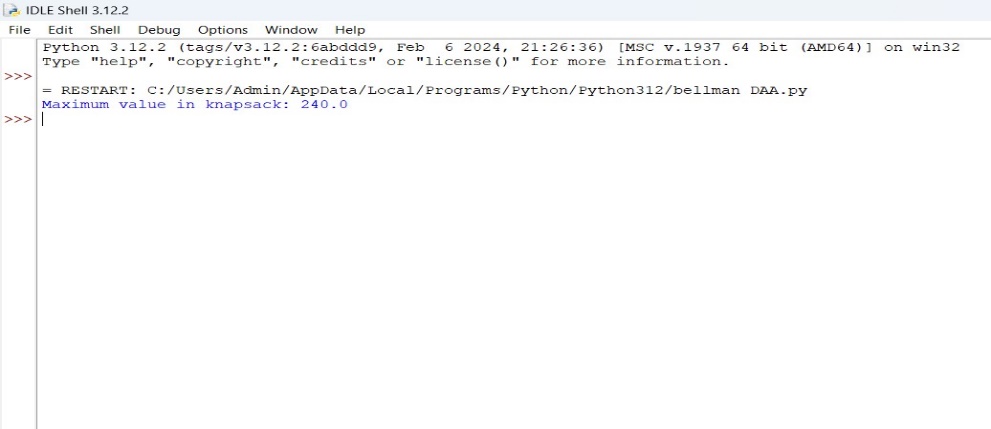
Item(100, 20),

Item(120, 30)]

capacity = 50

max\_value = knapsack\_greedy(items, capacity)

print(f"Maximum value in knapsack: {max\_value}")



5. Job sequence with deadlines

class Job:

def \_\_init\_\_(self, job\_id, deadline, profit):

self.job\_id = job\_id

self.deadline = deadline

self.profit = profit

def job\_scheduling(jobs):

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

max\_deadline = max(job.deadline for job in jobs)

result = [None] \* max\_deadline

slot = [False] \* max\_deadline

for job in jobs:

for j in range(min(max\_deadline - 1, job.deadline - 1), -1, -1):

if not slot[j]:

slot[j] = True

result[j] = job

break

print("Job sequence to maximize profit:")

for i in range(max\_deadline):

if result[i]:

print(f"Job ID: {result[i].job\_id}, Deadline: {result[i].deadline}, Profit: {result[i].profit}")

jobs = [Job('a', 2, 100),

Job('b', 1, 19),

Job('c', 2, 27),

Job('d', 1, 25),

Job('e', 3, 15)]

job\_scheduling(jobs)

