DAY – 12

1. Dijikstrs algorithm

import heapq

def dijkstra(graph, start):

pq = [(0, start)]

distances = {vertex: float('infinity') for vertex in graph}

distances[start] = 0

visited = set()

while pq:

current\_distance, current\_vertex = heapq.heappop(pq)

if current\_vertex in visited:

continue

visited.add(current\_vertex)

for neighbor, weight in graph[current\_vertex].items():

distance = current\_distance + weight

if distance < distances[neighbor]:

distances[neighbor] = distance

heapq.heappush(pq, (distance, neighbor))

return distances

graph = {'A': {'B': 1, 'C': 4},

'B': {'A': 1, 'C': 2, 'D': 5},

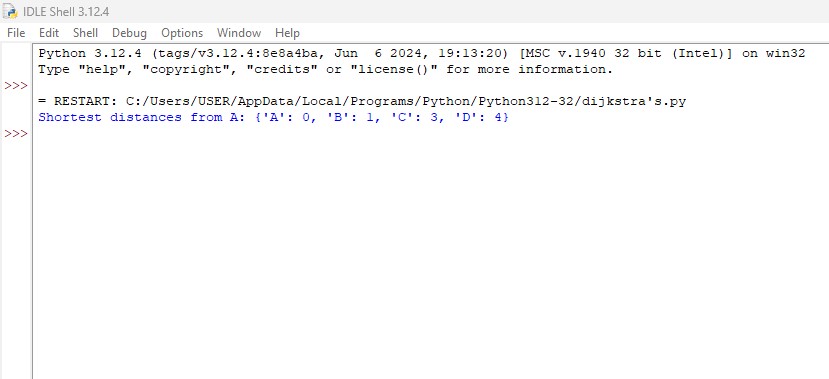
'C': {'A': 4, 'B': 2, 'D': 1},

'D': {'B': 5, 'C': 1}}

start\_vertex = 'A'

distances = dijkstra(graph, start\_vertex)

print("Shortest distances from {}: {}".format(start\_vertex, distances))



2. Huffman codes

from collections import Counter

class NodeTree(object):

def \_\_init\_\_(self, left=None, right=None):

self.left = left

self.right = right

def children(self):

return self.left, self.right

def \_\_str\_\_(self):

return self.left, self.right

def huffman\_code\_tree(node, binString=''):

'''

Function to find Huffman Code

'''

if type(node) is str:

return {node: binString}

(l, r) = node.children()

d = dict()

d.update(huffman\_code\_tree(l, binString + '0'))

d.update(huffman\_code\_tree(r, binString + '1'))

return d

def make\_tree(nodes):

'''

Function to make tree

:param nodes: Nodes

:return: Root of the tree

'''

while len(nodes) > 1:

(key1, c1) = nodes[-1]

(key2, c2) = nodes[-2]

nodes = nodes[:-2]

node = NodeTree(key1, key2)

nodes.append((node, c1 + c2))

nodes = sorted(nodes, key=lambda x: x[1], reverse=True)

return nodes[0][0]

if \_\_name\_\_ == '\_\_main\_\_':

string = 'BCAADDDCCACACAC'

freq = dict(Counter(string))

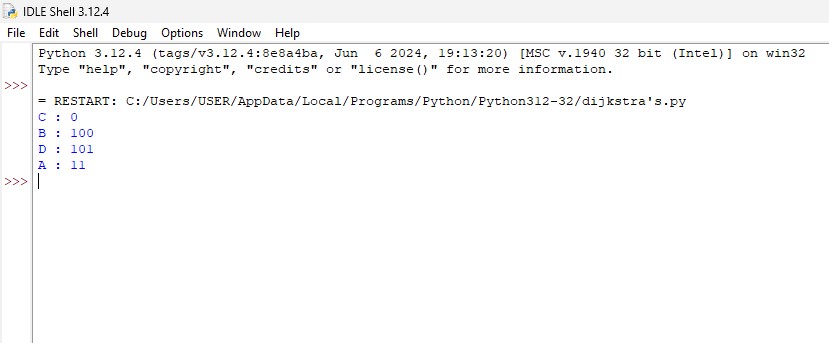
freq = sorted(freq.items(), key=lambda x: x[1], reverse=True)

node = make\_tree(freq)

encoding = huffman\_code\_tree(node)

for i in encoding:

print(f'{i} : {encoding[i]}')



3. Container loading

class Item:

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

self.name = name

self.weight = weight

def container\_loading(items, max\_weight):

items.sort(key=lambda item: item.weight, reverse=True)

total\_weight = 0

selected\_items = []

for item in items:

if total\_weight + item.weight <= max\_weight:

selected\_items.append(item)

total\_weight += item.weight

return selected\_items, total\_weight

if \_\_name\_\_ == "\_\_main\_\_":

items = [Item('item1', 10),

Item('item2', 40),

Item('item3', 20),

Item('item4', 30),

Item('item5', 50),

Item('item6', 5),]

max\_weight = 100

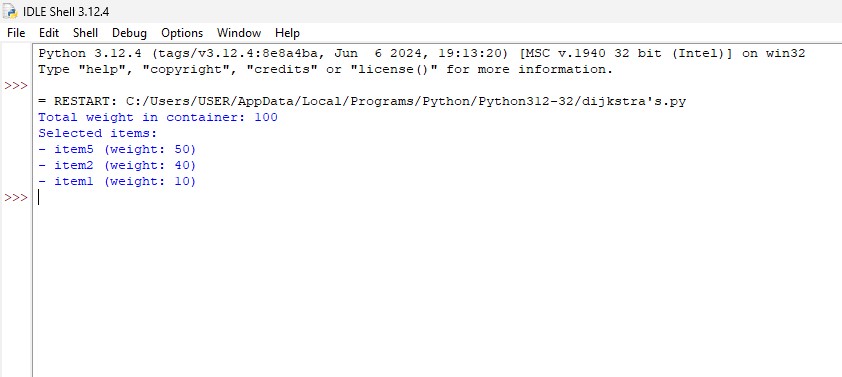
selected\_items, total\_weight = container\_loading(items, max\_weight)

print(f"Total weight in container: {total\_weight}")

print("Selected items:")

for item in selected\_items:

print(f"- {item.name} (weight: {item.weight})")



4. Minimum spanning tree

INF = 9999999

N = 5

G = [[0, 19, 5, 0, 0],

[19, 0, 5, 9, 2],

[5, 5, 0, 1, 6],

[0, 9, 1, 0, 1],

[0, 2, 6, 1, 0]]

selected\_node = [0, 0, 0, 0, 0]

no\_edge = 0

selected\_node[0] = True

print("Edge : Weight\n")

while (no\_edge < N - 1):

minimum = INF

a = 0

b = 0

for m in range(N):

if selected\_node[m]:

for n in range(N):

if ((not selected\_node[n]) and G[m][n]):

if minimum > G[m][n]:

minimum = G[m][n]

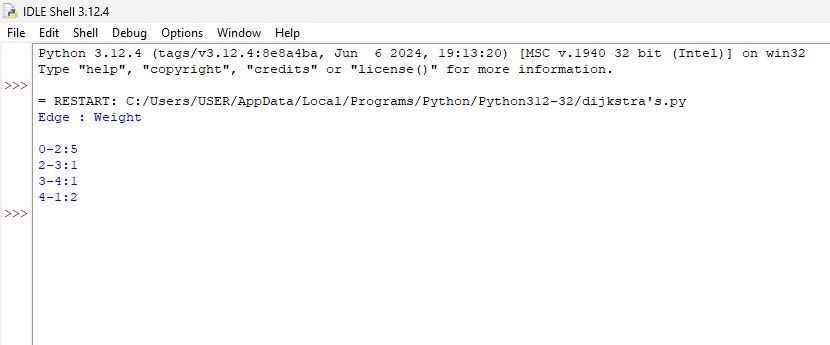
a = m

b = n

print(str(a) + "-" + str(b) + ":" + str(G[a][b]))

selected\_node[b] = True

no\_edge += 1



5. Kruskals algorithm

class Edge:

def \_\_init\_\_(self, src, dest, weight):

self.src = src

self.dest = dest

self.weight = weight

class Graph:

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

self.V = vertices

self.edges = []

def add\_edge(self, src, dest, weight):

self.edges.append(Edge(src, dest, weight))

def find(self, parent, i):

if parent[i] == i:

return i

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

def union(self, parent, rank, x, y):

root\_x = self.find(parent, x)

root\_y = self.find(parent, y)

if rank[root\_x] < rank[root\_y]:

parent[root\_x] = root\_y

elif rank[root\_x] > rank[root\_y]:

parent[root\_y] = root\_x

else:

parent[root\_y] = root\_x

rank[root\_x] += 1

def kruskal\_mst(self):

result = []

i = 0

e = 0

self.edges = sorted(self.edges, key=lambda edge: edge.weight)

parent = []

rank = []

for node in range(self.V):

parent.append(node)

rank.append(0)

while e < self.V - 1:

src, dest, weight = self.edges[i].src, self.edges[i].dest, self.edges[i].weight

i += 1

x = self.find(parent, src)

y = self.find(parent, dest)

if x != y:

e += 1

result.append((src, dest, weight))

self.union(parent, rank, x, y)

return result

if \_\_name\_\_ == "\_\_main\_\_":

g = Graph(4)

g.add\_edge(0, 1, 10)

g.add\_edge(0, 2, 6)

g.add\_edge(0, 3, 5)

g.add\_edge(1, 3, 15)

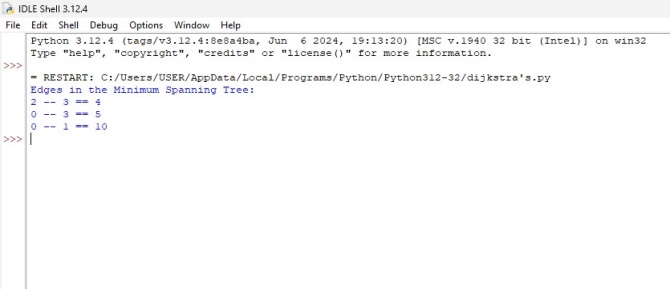
g.add\_edge(2, 3, 4)

mst = g.kruskal\_mst()

print("Edges in the Minimum Spanning Tree:")

for src, dest, weight in mst:

print(f"{src} -- {dest} == {weight}")



6. Boruvkas algorithm

class Graph:

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

self.V = vertices

self.edges = []

def add\_edge(self, src, dest, weight):

self.edges.append([src, dest, weight])

def find(self, parent, i):

if parent[i] == i:

return i

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

def union(self, parent, rank, x, y):

root\_x = self.find(parent, x)

root\_y = self.find(parent, y)

if rank[root\_x] < rank[root\_y]:

parent[root\_x] = root\_y

elif rank[root\_x] > rank[root\_y]:

parent[root\_y] = root\_x

else:

parent[root\_y] = root\_x

rank[root\_x] += 1

def boruvka\_mst(self):

parent = []

rank = []

cheapest = [-1] \* self.V

num\_trees = self.V

mst\_weight = 0

mst\_edges = []

for node in range(self.V):

parent.append(node)

rank.append(0)

while num\_trees > 1:

for i in range(len(self.edges)):

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

set1 = self.find(parent, u)

set2 = self.find(parent, v)

if set1 != set2:

if cheapest[set1] == -1 or cheapest[set1][2] > w:

cheapest[set1] = [u, v, w]

if cheapest[set2] == -1 or cheapest[set2][2] > w:

cheapest[set2] = [u, v, w]

for node in range(self.V):

if cheapest[node] != -1:

u, v, w = cheapest[node]

set1 = self.find(parent, u)

set2 = self.find(parent, v)

if set1 != set2:

self.union(parent, rank, set1, set2)

mst\_weight += w

mst\_edges.append((u, v, w))

num\_trees -= 1

cheapest = [-1] \* self.V

return mst\_edges, mst\_weight

if \_\_name\_\_ == "\_\_main\_\_":

g = Graph(4)

g.add\_edge(0, 1, 10)

g.add\_edge(0, 2, 6)

g.add\_edge(0, 3, 5)

g.add\_edge(1, 3, 15)

g.add\_edge(2, 3, 4)

mst\_edges, mst\_weight = g.boruvka\_mst()

print("Edges in the Minimum Spanning Tree:")

for u, v, weight in mst\_edges:

print(f"{u} -- {v} == {weight}")

print(f"Total weight of MST: {mst\_weight}")

