**GREEDY METHOD LAB PROGRAMS**

**1.COIN CHANGE PROBLEM**

def coin\_change(coins,amount):

coins.sort(reverse=True)

num\_coin=0

for coin in coins:

num\_coin +=amount//coin

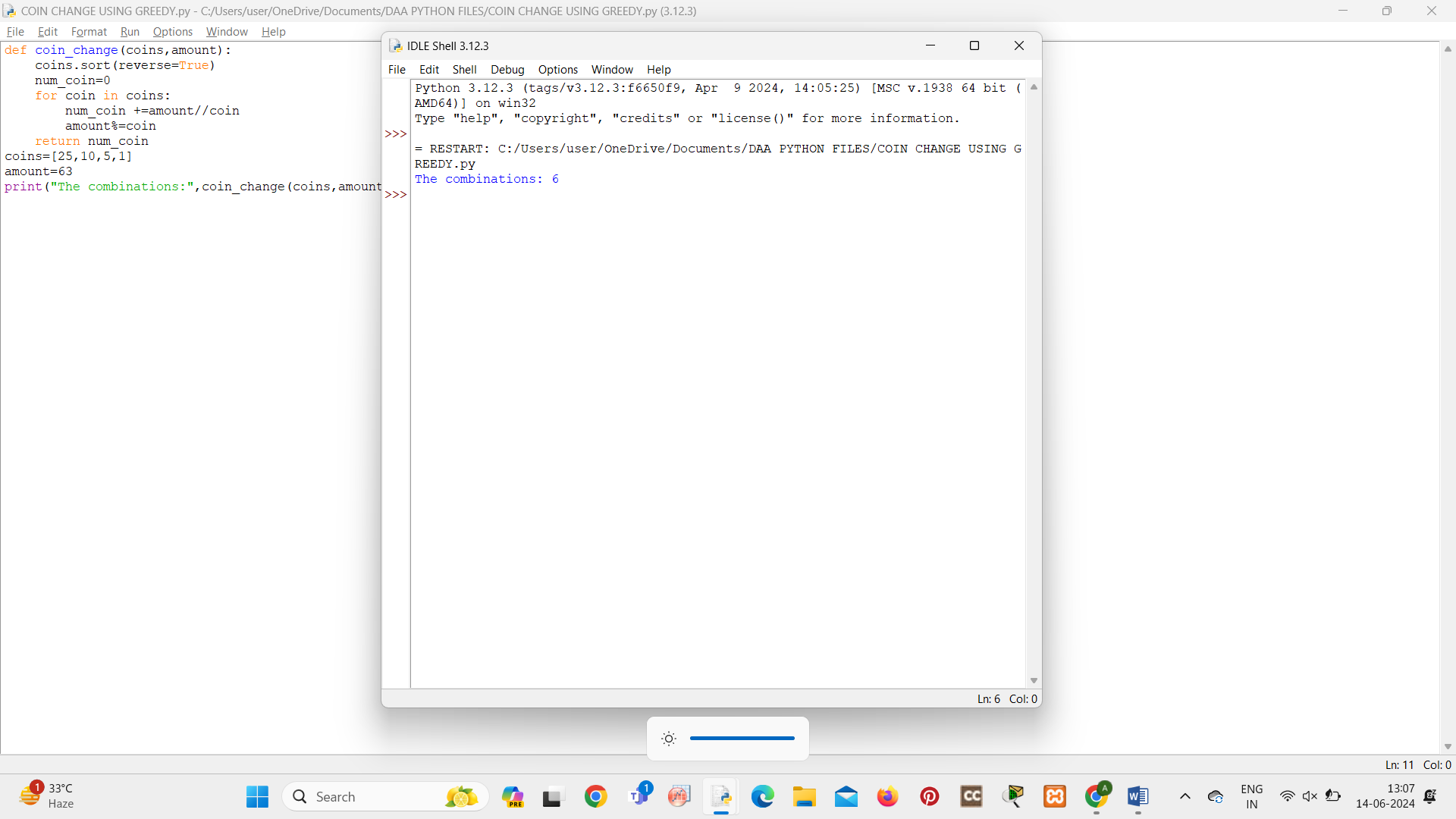
amount%=coin

return num\_coin

coins=[25,10,5,1]

amount=63

print("The combinations:",coin\_change(coins,amount))



**2.FRACTIONAL KNAPSACK**

import operator

def fractional\_knapsack(values, weights, capacity):

items = list(zip(values, weights))

items.sort(key=operator.itemgetter(0), reverse=True)

items.sort(key=operator.itemgetter(1), reverse=False)

total\_value = 0

knapsack = [0] \* len(values)

for value, weight in items:

if capacity == 0:

break

elif weight <= capacity:

knapsack[values.index(value)] = 1

total\_value += value

capacity -= weight

else:

fraction = capacity / weight

total\_value += value \* fraction

capacity = 0

return total\_value, knapsack

values = [60, 100, 120]

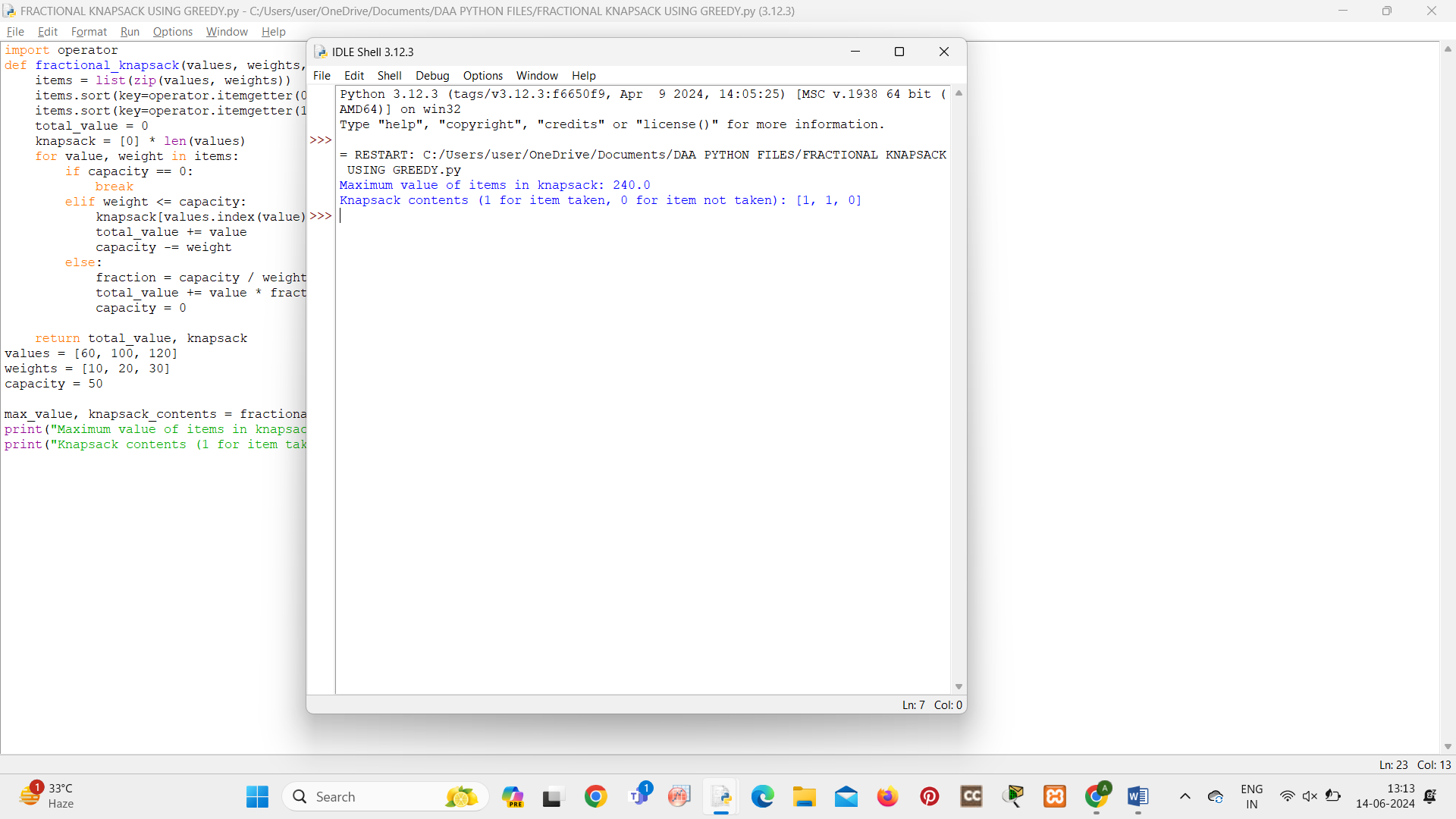
weights = [10, 20, 30]

capacity = 50

max\_value, knapsack\_contents = fractional\_knapsack(values, weights, capacity)

print("Maximum value of items in knapsack:", max\_value)

print("Knapsack contents (1 for item taken, 0 for item not taken):", knapsack\_contents)



**3.JOB SEQUENCE WITH DEADLINES**

def job\_sequencing\_with\_deadlines(jobs):

jobs.sort(key=lambda x: x[2], reverse=True)

max\_deadline = max(jobs, key=lambda x: x[1])[1]

sequence = [0] \* max\_deadline

total\_profit = 0

for job in jobs:

deadline = job[1]

while deadline > 0:

if sequence[deadline - 1] == 0:

sequence[deadline - 1] = job[0]

total\_profit += job[2]

break

else:

deadline -= 1

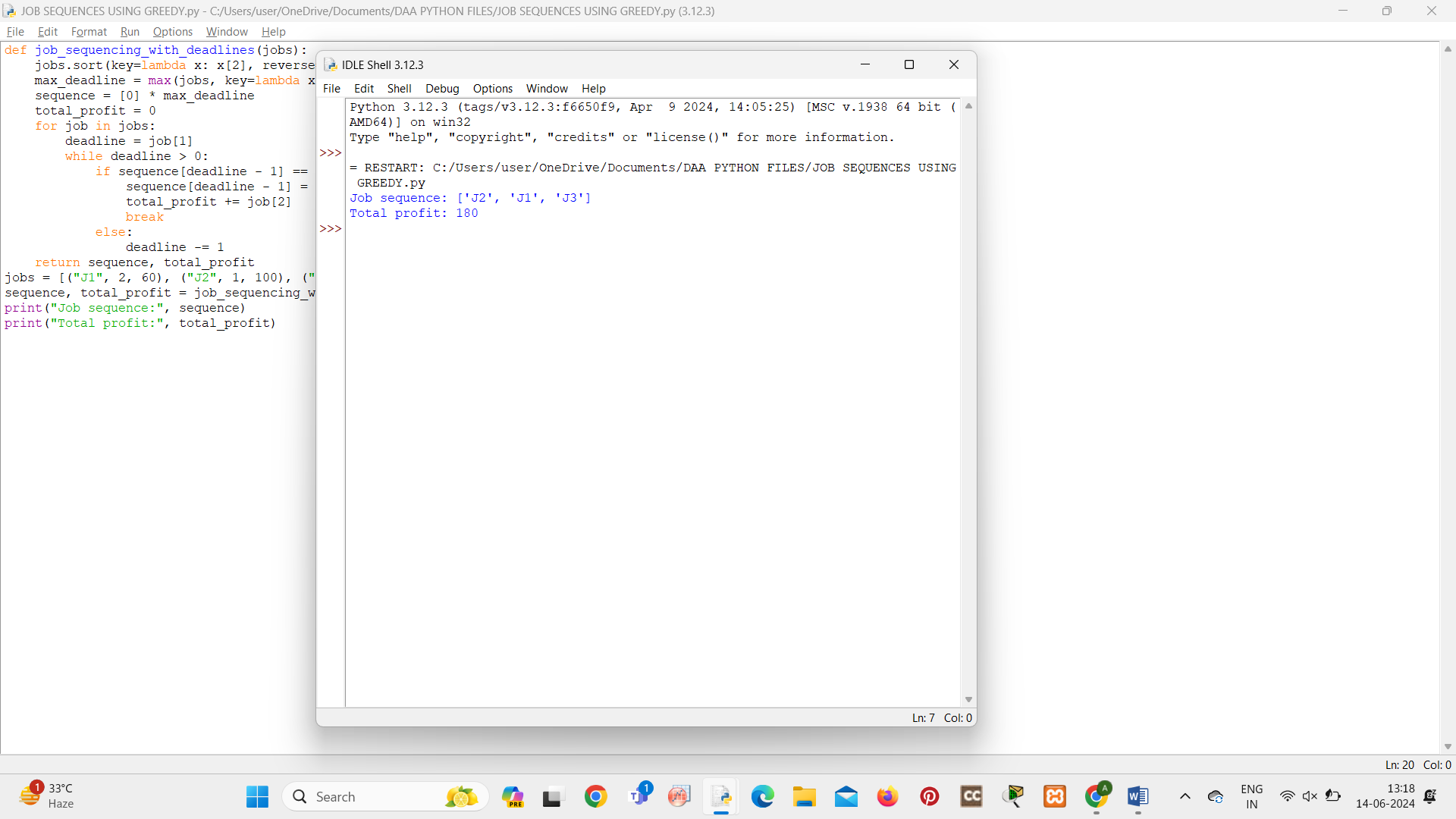
return sequence, total\_profit

jobs = [("J1", 2, 60), ("J2", 1, 100), ("J3", 3, 20), ("J4", 2, 40)]

sequence, total\_profit = job\_sequencing\_with\_deadlines(jobs)

print("Job sequence:", sequence)

print("Total profit:", total\_profit)



**4.SINGLE SOURCE SHORTEST PATH**

import heapq

def dijkstra(graph, source):

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

distances[source] = 0

priority\_queue = [(0, source)]

while priority\_queue:

current\_distance, current\_vertex = heapq.heappop(priority\_queue)

if current\_distance > distances[current\_vertex]:

continue

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

distance = current\_distance + weight

if distance < distances[neighbor]:

distances[neighbor] = distance

heapq.heappush(priority\_queue, (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}

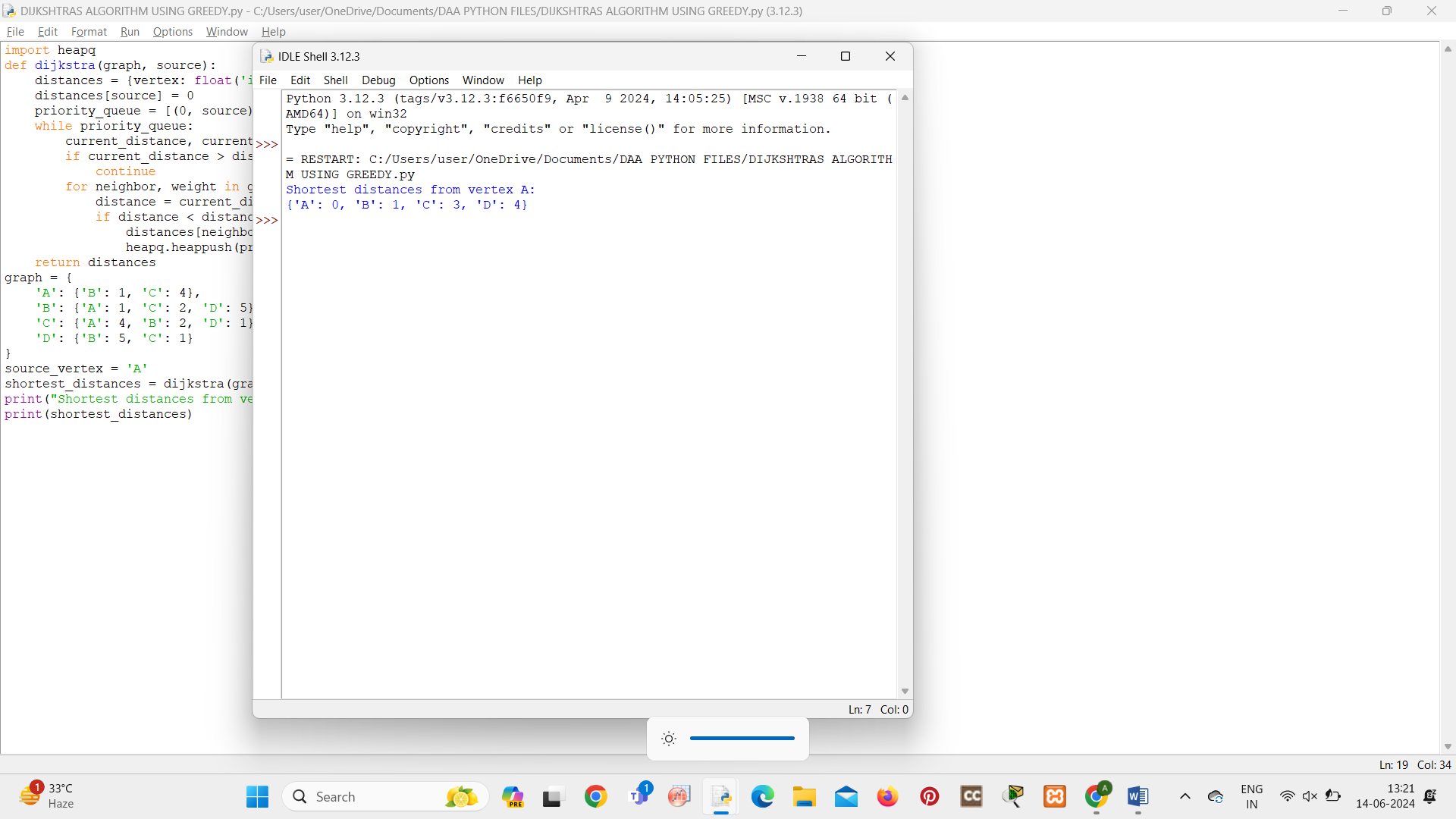
}

source\_vertex = 'A'

shortest\_distances = dijkstra(graph, source\_vertex)

print("Shortest distances from vertex", source\_vertex + ":")

print(shortest\_distances)



**5.HUFFMAN CODING AND TREE**

import heapq

from collections import Counter, namedtuple

Node = namedtuple('Node', ['char', 'freq', 'left', 'right'])

def \_\_lt\_\_(self, other):

return self.freq < other.freq

Node.\_\_lt\_\_ = \_\_lt\_\_

def build\_huffman\_tree(text):

char\_freq = Counter(text)

pq = [Node(char, freq, None, None) for char, freq in char\_freq.items()]

heapq.heapify(pq)

while len(pq) > 1:

left = heapq.heappop(pq)

right = heapq.heappop(pq)

merged\_freq = left.freq + right.freq

merged\_node = Node(None, merged\_freq, left, right)

heapq.heappush(pq, merged\_node)

return pq[0] if pq else None

def print\_huffman\_tree(node, level=0):

if node:

if node.left is None and node.right is None:

print(" " \* level + f"({node.char}, {node.freq})")

else:

print(" " \* level + f"({node.freq})")

print\_huffman\_tree(node.left, level + 1)

print\_huffman\_tree(node.right, level + 1)

text = "Huffman coding is a greedy algorithm"

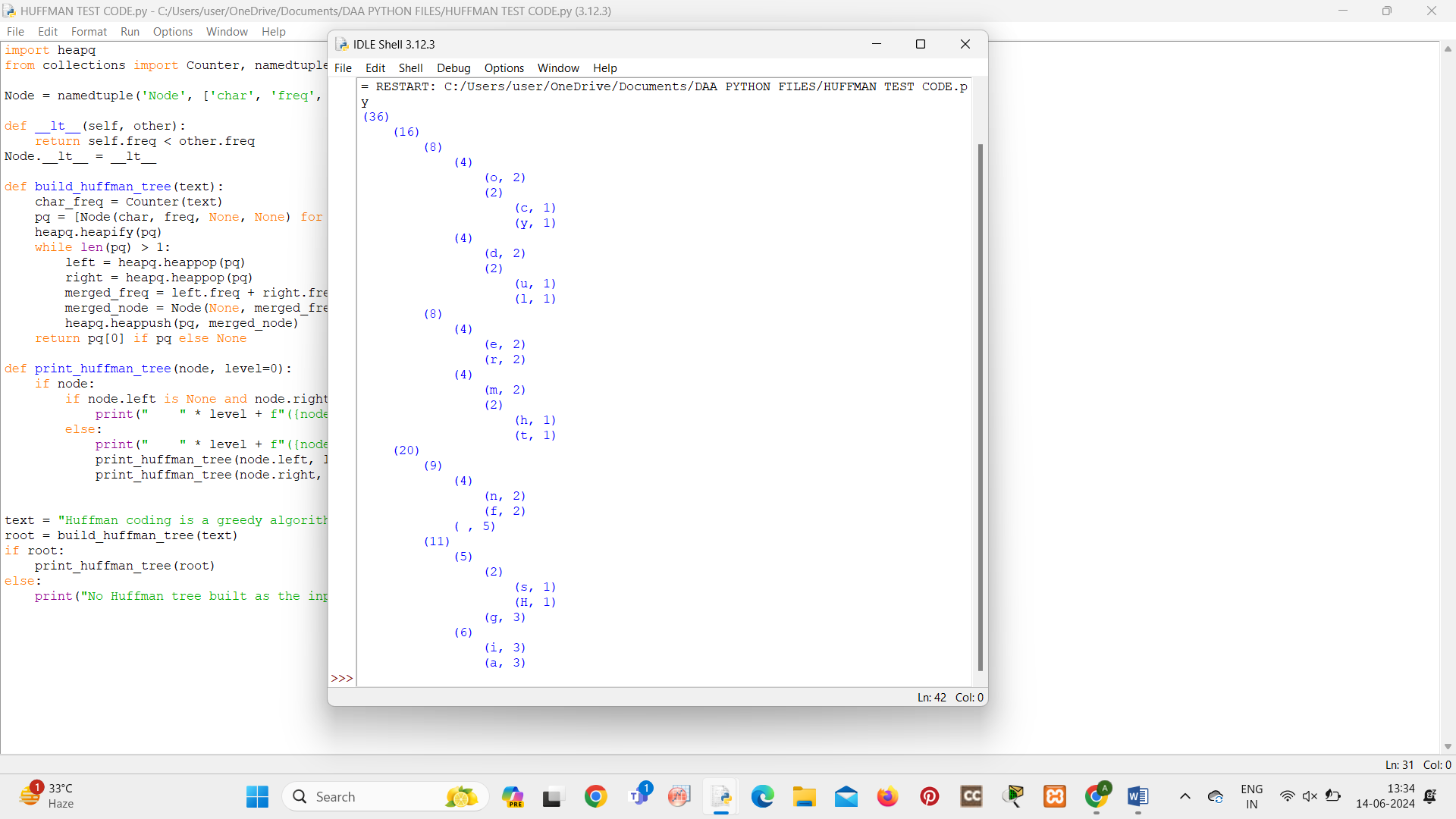
root = build\_huffman\_tree(text)

if root:

print\_huffman\_tree(root)

else:

print("No Huffman tree built as the input text is empty")



**6.CONTAINER LOADING**

def load\_containers(items, container\_capacity):

items.sort(reverse=True)

containers = []

current\_container = []

current\_capacity = container\_capacity

for item in items:

if item <= current\_capacity:

current\_container.append(item)

current\_capacity -= item

else:

containers.append(current\_container)

current\_container = [item]

current\_capacity = container\_capacity - item

if current\_container:

containers.append(current\_container)

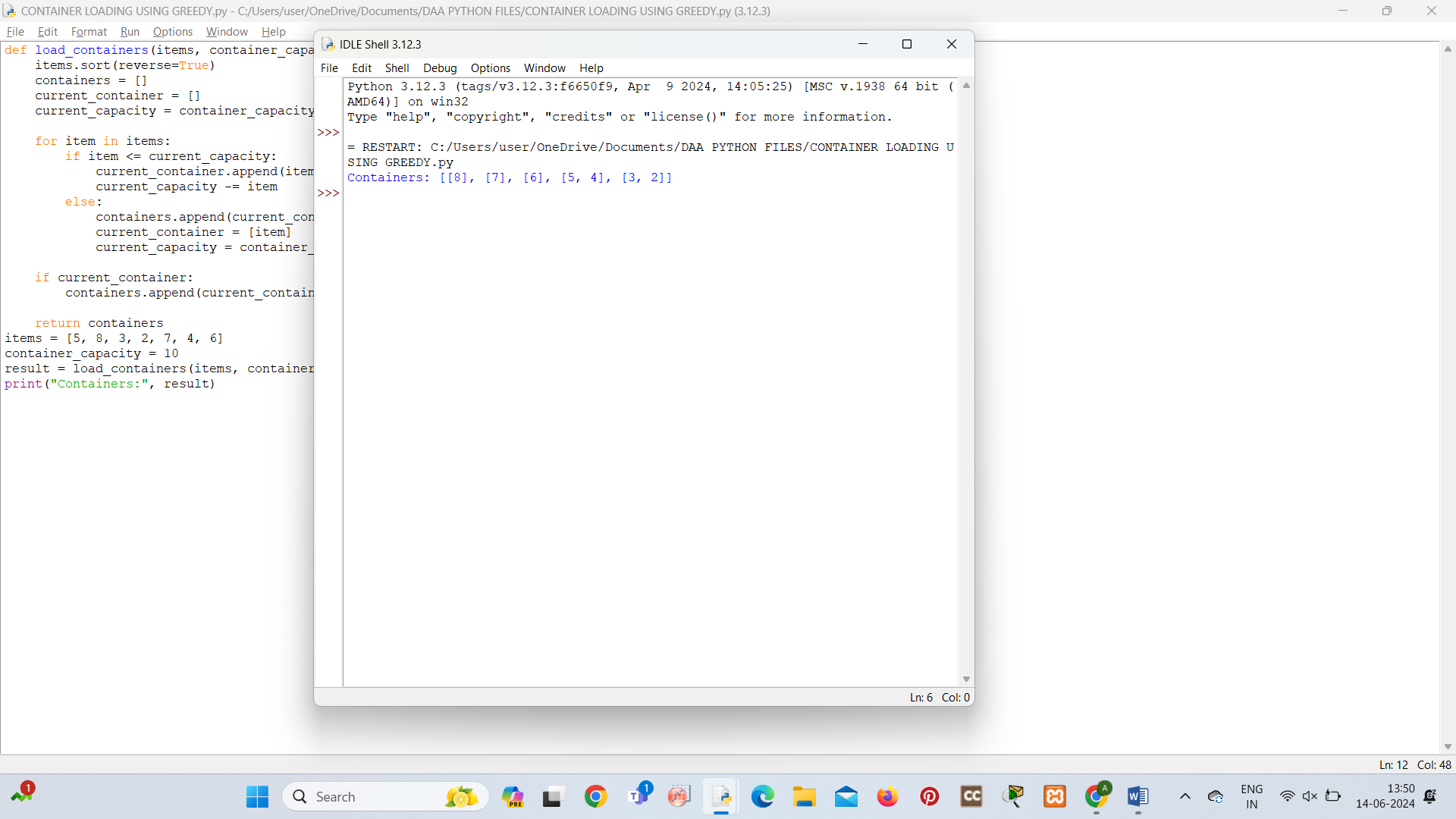
return containers

items = [5, 8, 3, 2, 7, 4, 6]

container\_capacity = 10

result = load\_containers(items, container\_capacity)

print("Containers:", result)



**7.KRUSKALS ALGORITHM FOR MINIMUM SPANNING TREE**

class DisjointSet:

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

self.parent = [i for i in range(n)]

self.rank = [0] \* n

def find(self, u):

if self.parent[u] != u:

self.parent[u] = self.find(self.parent[u])

return self.parent[u]

def union(self, u, v):

pu, pv = self.find(u), self.find(v)

if pu != pv:

if self.rank[pu] < self.rank[pv]:

self.parent[pu] = pv

elif self.rank[pu] > self.rank[pv]:

self.parent[pv] = pu

else:

self.parent[pv] = pu

self.rank[pu] += 1

def kruskals\_algorithm(graph):

edges = [(u, v, weight) for u, neighbors in enumerate(graph) for v, weight in neighbors]

edges.sort(key=lambda x: x[2])

n = len(graph)

mst = []

disjoint\_set = DisjointSet(n)

for u, v, weight in edges:

if disjoint\_set.find(u) != disjoint\_set.find(v):

disjoint\_set.union(u, v)

mst.append((u, v, weight))

return mst

graph = [

[(1, 4), (2, 3)],

[(0, 4), (2, 5), (3, 2)],

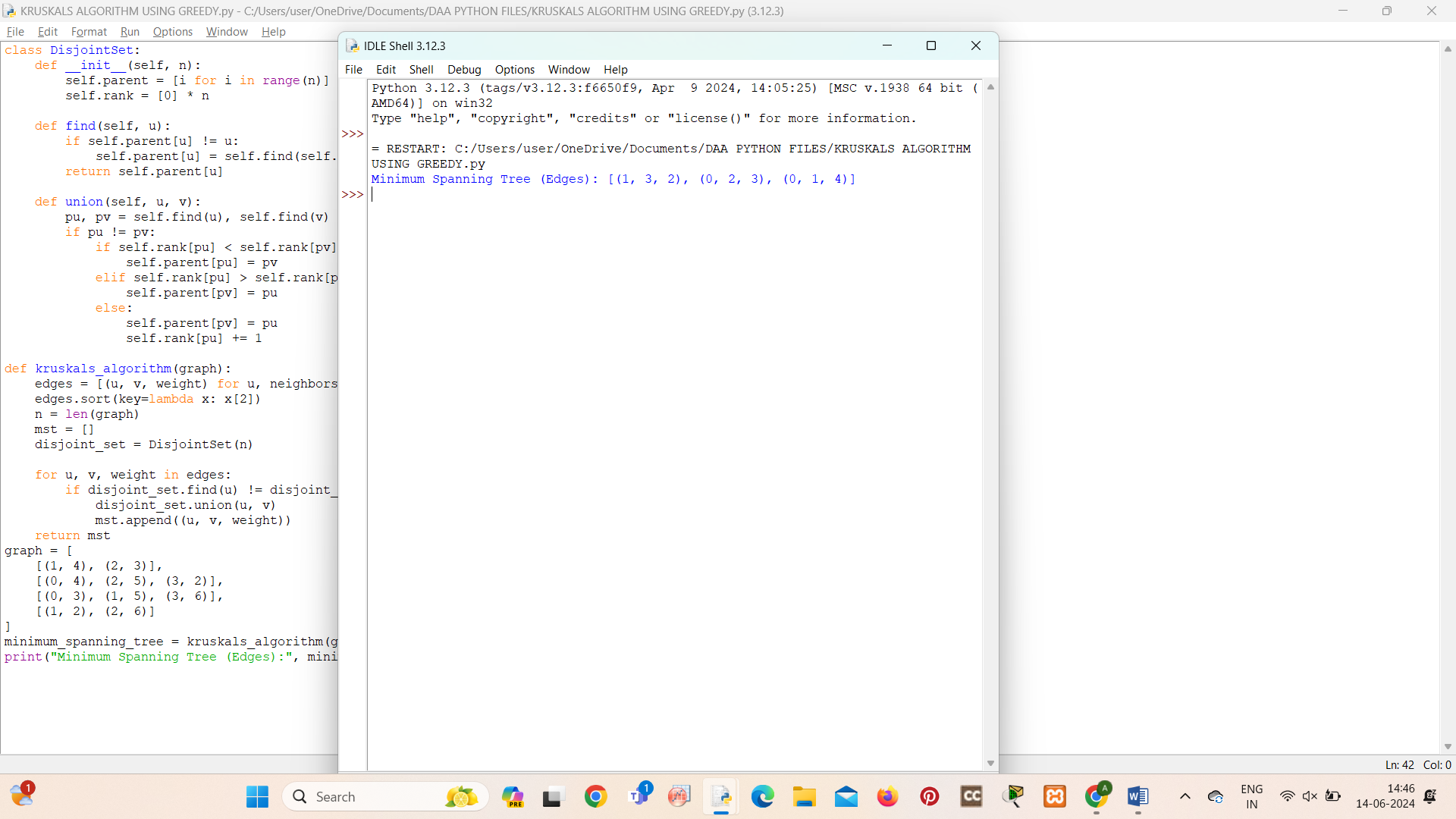
[(0, 3), (1, 5), (3, 6)],

[(1, 2), (2, 6)]

]

minimum\_spanning\_tree = kruskals\_algorithm(graph)

print("Minimum Spanning Tree (Edges):", minimum\_spanning\_tree)



**8.PRIMS ALGORITHM FOR MINIMUM SPANNING TREE**

import heapq

def prim\_algorithm(graph):

n = len(graph)

visited = [False] \* n

min\_heap = [(0, 0)]

mst = []

while min\_heap:

weight, u = heapq.heappop(min\_heap)

if not visited[u]:

visited[u] = True

mst.append((u, weight))

for v, w in graph[u]:

if not visited[v]:

heapq.heappush(min\_heap, (w, v))

return mst

graph = [

[(1, 4), (2, 3)],

[(0, 4), (2, 5), (3, 2)],

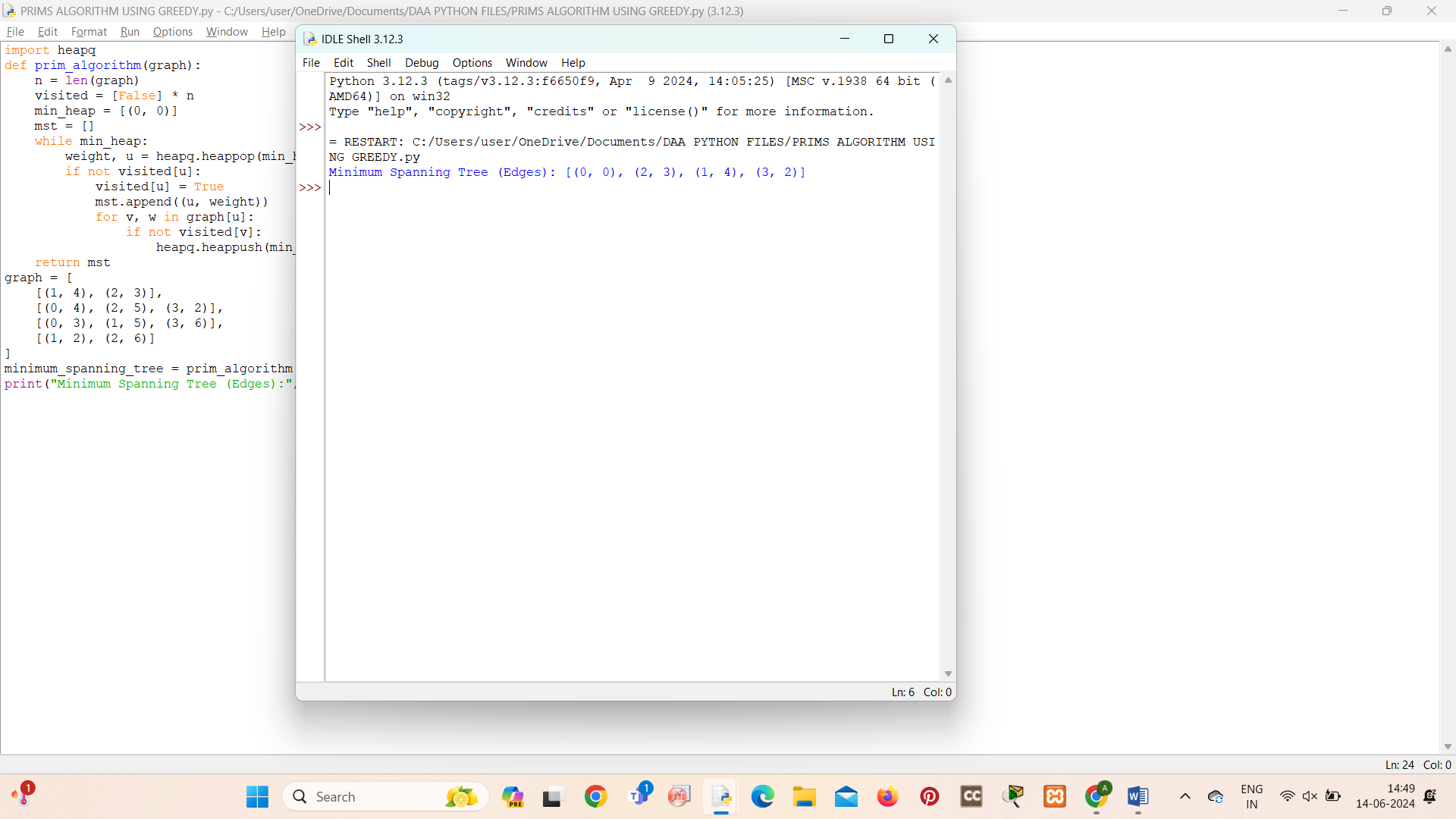
[(0, 3), (1, 5), (3, 6)],

[(1, 2), (2, 6)]

]

minimum\_spanning\_tree = prim\_algorithm(graph)

print("Minimum Spanning Tree (Edges):", minimum\_spanning\_tree)



**9.BORUVKAS ALGORITHM**

from collections import defaultdict

class Graph:

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

self.V = vertices

self.graph = defaultdict(list)

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

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

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

def boruvka\_mst(self):

parent = []

cheapest = []

min\_cost = 0

subset = []

for node in range(self.V):

parent.append(node)

cheapest.append(-1)

subset.append(-1)

num\_trees = self.V

while num\_trees > 1:

for i in range(self.V):

cheapest[i] = -1

for u in range(self.V):

for v, w in self.graph[u]:

set1 = self.find(subset, u)

set2 = self.find(subset, v)

if set1 != set2:

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

cheapest[set1] = (v, w)

for node in range(self.V):

if cheapest[node] != -1:

u = node

v, w = cheapest[node]

set1 = self.find(subset, u)

set2 = self.find(subset, v)

if set1 != set2:

min\_cost += w

self.union(subset, set1, set2)

print(f"Edge {u} - {v} weight: {w}")

num\_trees -= 1

print(f"Minimum cost: {min\_cost}")

def find(self, subset, i):

if subset[i] == -1:

return i

else:

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

def union(self, subset, x, y):

xroot = self.find(subset, x)

yroot = self.find(subset, y)

subset[xroot] = yroot

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)

g.boruvka\_mst()

