#Selection sort

def heuristic(arr):

sorted\_arr = sorted(arr)

mismatch\_count = sum(1 for a, b in zip(arr, sorted\_arr) if a != b)

return mismatch\_count

# Compares the current array with its sorted version. Counts how many elements are not in their correct position. Returns this count (called **mismatch count**).

def selectionSort(arr):

n = len(arr)# Store the number of elements.

print(f"Initial array: {arr}")

print(f"Heuristic: {heuristic(arr)}")

for i in range(n):

min\_index = i#Start from position i and assume it's the minimum.

for j in range(i + 1, n):

if arr[j] < arr[min\_index]:

min\_index = j #from i+1 to end of array, find the smallest value.

if i != min\_index:

arr[i], arr[min\_index] = arr[min\_index], arr[i]

#Swap current position i with found min\_index if they differ.

print(f"After step {i + 1}: {arr}, Heuristic: {heuristic(arr)}")

return arr

print("Name:selection sort")

print("The heuristic value shows how many numbers are misplaced")

arr = list(map(int, input("Enter the array elements separated by space(EG:5 10 18 2 53 22): ").split()))

sorted\_arr = selectionSort(arr)

print(f"Sorted array: {sorted\_arr}")

#minimum spanning tree(krushkals )

class Graph:

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

self.V = vertices

self.graph = []

self.heuristics = {}

# initialize Number of vertices.Empty list for edges

def addEdge(self, u, v, w, h=0):

# Add edge to the graph and heuristic values to the dictionary

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

self.heuristics[(u, v)] = h

self.heuristics[(v, u)] = h

def find(self, parent, i):

# Find function with path compression

if parent[i] != i:

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

return parent[i]

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

# Union function to connect two components

if rank[x] < rank[y]:

parent[x] = y

elif rank[x] > rank[y]:

parent[y] = x

else:

parent[y] = x

rank[x] += 1

#Merges two trees based on rank

def KruskalMST(self):

result = [] # To store the resulting MST

i = 0 # Initial edge index

e = 0 # Initial count of edges in the MST

# Sort edges based on the weight and heuristic

self.graph.sort(key=lambda item: item[2] + self.heuristics.get((item[0], item[1]), 0))

# Initialize disjoint sets

parent = list(range(self.V))

rank = [0] \* self.V

while e < self.V - 1 and i < len(self.graph):

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

i += 1

x = self.find(parent, u)

y = self.find(parent, v)

if x != y: # If u and v are not in the same set, add edge to MST

e += 1

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

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

# Print the result

minimumCost = sum([weight for \_, \_, weight in result])

print("\nEdges in the constructed MST:")

for u, v, weight in result:

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

print("Minimum Spanning Tree Cost:", minimumCost)

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

print("Name: minimum spanning tree-krushkals ")

print("Kruskal's Algorithm is a greedy algorithm used to find the Minimum Spanning Tree (MST) .we sort all edges here pick smallest edge and check if it doesn'St form cycle ")

print("Heuristic is used for helping to choose the next edge. Edge with low weight+ low heuristic value is chosen")

# Input the number of vertices

vertices = int(input("Enter the number of vertices: "))

g = Graph(vertices)

# Input the number of edges

edges = int(input("Enter the number of edges: "))

print("Enter the edges in the format: u v weight heuristic")

# Input edges and heuristics

for \_ in range(edges):

u, v, w, h = map(int, input().split()) # Read edge: u, v, weight, heuristic

g.addEdge(u, v, w, h)

# Run Kruskal's Algorithm to find the Minimum Spanning Tree (MST)

g.KruskalMST()

#sample input

#Enter the number of vertices: 5

#Enter the number of edges\*n-1): 4

#Enter the edges in the format: u v weight heuristic

#0 1 10 2

# 2 4 6 1

# 0 3 5 0

#1 3 15 1