**1 - MERGE SORT**

**def merge\_sort(orders):**

**if len(orders) <= 1:**

**return orders**

**mid = len(orders) // 2**

**left = merge\_sort(orders[:mid])**

**right = merge\_sort(orders[mid:])**

**return merge(left, right)**

**def merge(left, right):**

**sorted\_orders = []**

**i = j = 0**

**while i < len(left) and j < len(right):**

**if left[i]["timestamp"] <= right[j]["timestamp"]:**

**sorted\_orders.append(left[i])**

**i += 1**

**else:**

**sorted\_orders.append(right[j])**

**j += 1**

**sorted\_orders.extend(left[i:])**

**sorted\_orders.extend(right[j:])**

**return sorted\_orders**

**# Example usage**

**orders = [**

**{"id": 101, "timestamp": "2025-09-22T18:45:00"},**

**{"id": 102, "timestamp": "2025-09-22T18:30:00"},**

**{"id": 103, "timestamp": "2025-09-22T19:00:00"},**

**]**

**sorted\_orders = merge\_sort(orders)**

**for order in sorted\_orders:**

**print(order)**

**2 – QUICK SORT**

def quicksort(movies, key):

if len(movies) <= 1:

return movies

pivot = movies[len(movies) // 2][key]

left = [x for x in movies if x[key] < pivot]

middle = [x for x in movies if x[key] == pivot]

right = [x for x in movies if x[key] > pivot]

return quicksort(left, key) + middle + quicksort(right, key)

# Example usage

movies = [

{"title": "Movie A", "imdb": 8.2, "year": 2020, "popularity": 15000},

{"title": "Movie B", "imdb": 7.5, "year": 2022, "popularity": 20000},

{"title": "Movie C", "imdb": 9.0, "year": 2018, "popularity": 18000},

]

# Sort by IMDb rating

sorted\_by\_imdb = quicksort(movies, key="imdb")

for movie in sorted\_by\_imdb:

print(movie)

3 – FRACTINAL KNAPSACK

def fractional\_knapsack(items, capacity):

# Sort items by value-to-weight ratio in descending order

items.sort(key=lambda x: x['value'] / x['weight'], reverse=True)

total\_value = 0.0

selected\_items = []

for item in items:

if capacity == 0:

break

if item['divisible']:

# Take as much as possible

take\_weight = min(item['weight'], capacity)

total\_value += take\_weight \* (item['value'] / item['weight'])

selected\_items.append({

'name': item['name'],

'taken\_weight': take\_weight,

'taken\_value': take\_weight \* (item['value'] / item['weight'])

})

capacity -= take\_weight

else:

# Take whole item only if it fits

if item['weight'] <= capacity:

total\_value += item['value']

selected\_items.append({

'name': item['name'],

'taken\_weight': item['weight'],

'taken\_value': item['value']

})

capacity -= item['weight']

return total\_value, selected\_items

# Example items

relief\_items = [

{"name": "Food Packets", "weight": 10, "value": 60, "divisible": True},

{"name": "Medical Kit", "weight": 20, "value": 100, "divisible": False},

{"name": "Water Bottles", "weight": 30, "value": 120, "divisible": True},

{"name": "Blankets", "weight": 15, "value": 45, "divisible": False}

]

# Boat capacity

max\_capacity = 50

# Run algorithm

max\_value, selected = fractional\_knapsack(relief\_items, max\_capacity)

# Output result

print(f"Maximum Utility Value: {max\_value}")

print("Items Selected:")

for item in selected:

print(f" - {item['name']}: {item['taken\_weight']}kg, Value: {item['taken\_value']}")

4 – **DIJKSTRA ALGORITHM**

import heapq

def dijkstra(graph, source):

# Initialize distances and priority queue

distances = {node: float('inf') for node in graph}

distances[source] = 0

prev\_nodes = {node: None for node in graph}

pq = [(0, source)]

while pq:

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

# Skip if we already found a better path

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

continue

for neighbor, weight in graph[current\_node]:

distance = current\_distance + weight

if distance < distances[neighbor]:

distances[neighbor] = distance

prev\_nodes[neighbor] = current\_node

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

return distances, prev\_nodes

def reconstruct\_path(prev\_nodes, destination):

path = []

while destination:

path.append(destination)

destination = prev\_nodes[destination]

return path[::-1]

# Each node maps to a list of (neighbor, travel\_time)

city\_graph = {

'A': [('B', 5), ('C', 10)],

'B': [('D', 3)],

'C': [('D', 1)],

'D': [('E', 2)],

'E': []

}

source = 'A'

hospitals = ['D', 'E']

# Run Dijkstra

distances, prev\_nodes = dijkstra(city\_graph, source)

# Find nearest hospital

nearest = min(hospitals, key=lambda h: distances[h])

path = reconstruct\_path(prev\_nodes, nearest)

print(f"Nearest hospital: {nearest}")

print(f"Shortest path: {' -> '.join(path)}")

print(f"Travel time: {distances[nearest]} minutes")

def update\_traffic(graph, road, new\_weight):

for i, (neighbor, weight) in enumerate(graph[road[0]]):

if neighbor == road[1]:

graph[road[0]][i] = (neighbor, new\_weight)

break