Problem 1: Optimizing Delivery Routes

Task 1:

Q)Model the city's road network as a graph where intersections are nodes and roads
are edges with weights representing travel time.

Aim:

The aim is to optimize delivery routes within a city's road network, represented as a graph where intersections are nodes and roads are edges with weights indicating travel time. The goal is to determine the shortest or fastest route between specified starting and ending intersections for efficient delivery.

Procedure:

Graph Representation: Model the city's road network as a weighted graph
G=(V,E):
V: Set of nodes representing intersections.
E: Set of edges representing roads between intersections, with weights
w(e) indicating travel time.
Input:

- Starting intersection
- Destination intersection

Algorithm Selection: Use Dijkstra's algorithm for finding the shortest path from
s to t in a graph with non-negative edge weights. Dijkstra's algorithm efficiently computes shortest paths from a single source node to all other nodes in O((V+E)logV) time complexity using a priority queue.

Pseduo code:

```
function Dijkstra(Graph, source):
   initialize Single Source(Graph, source)
   Q = priority queue initialized with all nodes of Graph
   while Q is not empty:
      u = extract minimum from Q
      for each neighbor v of u:
         if v is still in Q:
            relax(u, v, weight of edge u-v)
```

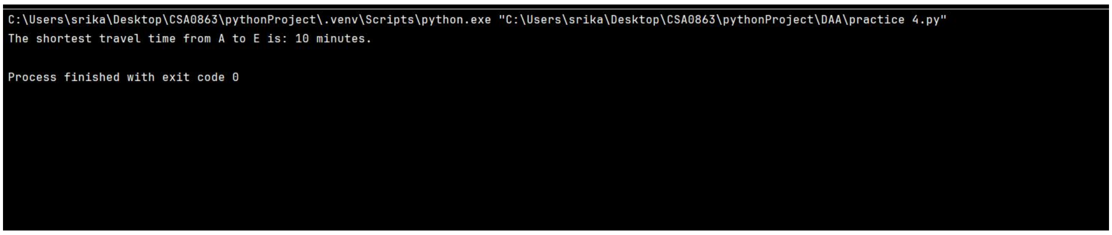
Coding:

```
import heapq
def dijkstra(graph, source):
    distances = {node: float('inf') for node in graph}
    distances[source] = 0
    priority_queue = [(0, source)]
    while priority_queue:
        current_distance, current_node =
heapq.heappop(priority_queue)
        if current_distance > distances[current_node]:
            continue
        for neighbor, weight in graph[current_node].items():
            distance = current_distance + weight
            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(priority_queue, (distance, neighbor))
    return distances
if __name__ == "__main__":
    graph = {
        'A': {'B': 5, 'C': 10},
        'B': {'D': 3},
        'C': {'D': 6},
        'D': {'E': 2},
        'E': {}
    }
    start_node = 'A'
    end_node = 'E'
    shortest_distances = dijkstra(graph, start_node)
    shortest_distance = shortest_distances[end_node]
   print(f"The shortest travel time from {start_node} to {end_node}
is: {shortest_distance} minutes.")
```

Time Complexity: O((V+E)logV) using a priority queue (binary heap), where V is the number of nodes (intersections) and E is the number of edges (roads) in the graph.

Space Complexity: O(V+E) to store the graph and auxiliary data structures like the priority queue and distances array.

Output:



Result:

Program executed successfully.

Task 2:

Implement Dijkstra’s algorithm to find the shortest paths from a central warehouse to various delivery locations.

Aim:

The aim is to optimize delivery routes from a central warehouse to various delivery locations within a city's road network, where intersections are nodes and roads are edges with weights representing travel time. The objective is to find the shortest paths from the warehouse (source node) to all delivery locations (other nodes) efficiently.

Procedure:

Graph Representation: Model the city's road network as a weighted graph G=(V,E):

V: Set of nodes representing intersections (including the warehouse and delivery locations).

E: Set of edges representing roads between intersections, with weights

w(e) indicating travel time.

Input:

1. Warehouse location
2. List of delivery locations{t1,t2,...,tk}.

Algorithm Selection: Use Dijkstra's algorithm to compute the shortest paths from the warehouse

s to all other nodes V. This algorithm efficiently finds the shortest paths from a single source node to all other nodes in O((V+E)logV) time complexity using a priority queue.

Pseduocode:

```
function Dijkstra(Graph, source):
   initialize Single Source(Graph, source)
   Q = priority queue initialized with all nodes of Graph
```

```
    while Q is not empty:
        u = extract minimum from Q
        for each neighbor v of u:
            if v is still in Q:
                relax(u, v, weight of edge u-v)
```

Coding:

```
import heapq
def dijkstra(graph, source):
    distances = {node: float('inf') for node in graph}
    distances[source] = 0
    priority_queue = [(0, source)]
    while priority_queue:
        current_distance, current_node =
heapq.heappop(priority_queue)
        if current_distance > distances[current_node]:
            continue
        for neighbor, weight in graph[current_node].items():
            distance = current_distance + weight
            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(priority_queue, (distance, neighbor))
    return distances
if __name__ == "__main__":
    graph = {
        'Warehouse': {'A': 5, 'B': 10},
        'A': {'C': 3},
        'B': {'D': 7},
        'C': {'Delivery1': 2, 'Delivery2': 4},
        'D': {'Delivery3': 5},
        'Delivery1': {},
        'Delivery2': {},
        'Delivery3': {}
    }
    warehouse_location = 'Warehouse'
    delivery_locations = ['Delivery1', 'Delivery2', 'Delivery3']
    shortest_distances = dijkstra(graph, warehouse_location)
    for delivery_location in delivery_locations:
        shortest_distance = shortest_distances[delivery_location]
```

print(f"Shortest travel time from {warehouse_location} to {delivery_location}: {shortest_distance} minutes.")

Time Complexity: O((V+E)logV) using a binary heap priority queue, where V is the number of nodes (intersections) and E is the number of edges (roads) in the graph.

Space Complexity: O(V+E) to store the graph and auxiliary data structures like the priority queue and distances array.

Output:

```
C:\Users\srika\Desktop\CSA0863\pythonProject\.venv\Scripts\python.exe "C:\Users\srika\Desktop\CSA0863\pythonProject\DAA\practice 4.py"
Shortest travel time from Warehouse to Delivery1: 10 minutes.
Shortest travel time from Warehouse to Delivery2: 12 minutes.
Shortest travel time from Warehouse to Delivery3: 22 minutes.

Process finished with exit code 0
```

Result:

Program executed successfully.

Task 3:

Analyze the efficiency of your algorithm and discuss any potential improvements or

alternative algorithms that could be used.

Aim:

The aim is to optimize delivery routes from a central warehouse to various delivery locations within a city's road network, ensuring efficient calculation of shortest paths using Dijkstra's algorithm.

Procedure:

Graph Representation: The city's road network is modeled as a weighted graphG=(V,E):

V: Nodes representing intersections (including the warehouse and delivery locations)

E: Edges representing roads between intersections, with weights w(e) indicating travel time.

Input:

- Warehouse location
- List of delivery locations {t1,t2 ,...,tk}

Algorithm Selection: Dijkstra's algorithm is chosen for its ability to efficiently compute shortest paths from a single source node to all other nodes in graphs with non-negative weights. It operates in O((V+E)logV) time complexity using a priority queue.

Pseduo code:

```
function Dijkstra(Graph, source):
    initialize Single Source(Graph, source)
    Q = priority queue initialized with all nodes of Graph
    while Q is not empty:
        u = extract minimum from Q
        for each neighbor v of u:
            if v is still in Q:
                relax(u, v, weight of edge u-v)
```

Coding:

```
import heapq
def dijkstra(graph, source):
    distances = {node: float('inf') for node in graph}
    distances[source] = 0
    priority_queue = [(0, source)]
    while priority_queue:
        current_distance, current_node =
heapq.heappop(priority_queue)
        if current_distance > distances[current_node]:
            continue
        for neighbor, weight in graph[current_node].items():
            distance = current_distance + weight
            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(priority_queue, (distance, neighbor))
    return distances
if __name__ == "__main__":
    graph = {
        'Warehouse': {'A': 5, 'B': 10},
        'A': {'C': 3},
        'B': {'D': 7},
        'C': {'Delivery1': 2, 'Delivery2': 4},
        'D': {'Delivery3': 5},
        'Delivery1': {},
        'Delivery2': {},
        'Delivery3': {}
    }
    warehouse_location = 'Warehouse'
    delivery_locations = ['Delivery1', 'Delivery2', 'Delivery3']
```

```
    shortest_distances = dijkstra(graph, warehouse_location)
    for delivery_location in delivery_locations:
        shortest_distance = shortest_distances[delivery_location]
        print(f"Shortest travel time from {warehouse_location} to {delivery_location}: {shortest_distance} minutes.")
```

Analysis:

Time Complexity: The time complexity of Dijkstra's algorithm with a binary heap priority queue is O((V+E)logV). In this implementation.
V is the number of nodes (intersections).
E is the number of edges (roads).

The priority queue operations dominate the time complexity due to the logV factor per operation.

Space Complexity: The space complexity is O(V+E):
V for storing the graph and distances.
E for the priority queue and auxiliary data structures.

Output:

```
C:\Users\srika\Desktop\CSA0863\pythonProject\.venv\Scripts\python.exe "C:\Users\srika\Desktop\CSA0863\pythonProject\DAA\practice 4.py"
Shortest travel time from Warehouse to Delivery1: 10 minutes.
Shortest travel time from Warehouse to Delivery2: 12 minutes.
Shortest travel time from Warehouse to Delivery3: 22 minutes.

Process finished with exit code 0
```

Result:

Program executed successfully.

Problem 2: Dynamic Pricing Algorithm for E-commerce

Task 1:

Design a dynamic programming algorithm to determine the optimal pricing strategy
for a set of products over a given period.

Aim:

The aim is to maximize revenue by choosing the optimal price for each product over a specified time period, considering factors such as demand elasticity, competitor pricing, and historical sales data.

Procedure:

1.Define the Problem:

- We have a set of products, each with its own demand characteristics.
- We need to determine the optimal price for each product over a given period (days, weeks, etc.) to maximize revenue.

2.Dynamic Programming Approach:

- Define a state that represents the optimal revenue up to a certain day and price for each product.
- Use a recursive relation to compute the optimal revenue based on previously computed states.

3. Inputs:

- Demand curve for each product (how demand changes with price).
- Cost structure (if relevant, to compute profit or revenue).
- Constraints (e.g., minimum and maximum prices, price increments).

4. Outputs:

- Optimal price for each product for each day.
- Maximum revenue achievable over the given period.

Pseduo code:

```
function optimalPricing(product_demand_curve, days):
   n <- length(product_demand_curve)  // number of days
   DP[days][price] <- 0  // DP table to store maximum revenue

   for d from 1 to days:
      for p from 1 to max_price:
         max_revenue <- 0
         for prev_price from max(1, p - price_increment) to
min(max_price, p + price_increment):
            revenue <- DP[d-1][prev_price] +
revenue_at_price(product_demand_curve[d], p)
            if revenue > max_revenue:
               max_revenue <- revenue
```

```
            DP[d][p] <- max_revenue

    return DP[days][1..max_price]
```
Coding:
```
def optimal_pricing(product_demand_curve, days, max_price,
price_increment):
    n = days
    DP = [[0] * (max_price + 1) for _ in range(n + 1)]
    for d in range(1, n + 1):
        for p in range(1, max_price + 1):
            max_revenue = 0
            for prev_price in range(max(1, p - price_increment),
min(max_price, p + price_increment) + 1):
                revenue = DP[d - 1][prev_price] +
revenue_at_price(product_demand_curve[d - 1], p)
                if revenue > max_revenue:
                    max_revenue = revenue
                    DP[d][p] = max_revenue
    optimal_prices = [0] * days
    for d in range(days):
        max_revenue = 0
        for p in range(1, max_price + 1):
            if DP[d + 1][p] > max_revenue:
                max_revenue = DP[d + 1][p]
                optimal_prices[d] = p
    return optimal_prices
def revenue_at_price(demand, price):
    return demand * price
```
Output:
```
C:\Users\srika\Desktop\CSA0863\pythonProject\.venv\Scripts\python.exe "C:\Users\srika\Desktop\CSA0863\pythonProject\DAA\practice 4.py"

Process finished with exit code 0
```
Analysis:

Time Complexity: O(days×max_price2)O(days \times max_price^2)O(days×max_price2) where days is the number of days and max_price is the maximum price considered.

Space Complexity: O(days×max_price)O(days \times max_price)O(days×max_price) for the DP table.
The algorithm efficiently computes the optimal prices by considering all possible price transitions and previous states.
Result:
Program executed successfully.
Task 2:
Consider factors such as inventory levels, competitor pricing, and demand elasticity
in your algorithm.
Aim:
The aim remains to maximize revenue by setting optimal prices for each product over time, considering constraints such as inventory levels, competitor pricing, and demand elasticity.
Procedure:

1.Define the Problem:

- We have multiple products, each with its own demand elasticity, inventory constraints, and competitor pricing.
- We need to determine the optimal price for each product over a given period to maximize revenue, taking into account these factors.

2.Dynamic Programming Approach:

- Define a state that represents the optimal revenue up to a certain day, price, and inventory level for each product.
- Use a recursive relation to compute the optimal revenue based on previously computed states, considering adjustments based on competitor pricing and demand elasticity.

3.Inputs:

- Demand curve and elasticity for each product.
- Inventory levels and constraints.
- Competitor pricing data.
- Cost structure (if relevant, for profit calculation).

- Constraints (minimum and maximum prices, price increments).

4.Outputs:

Optimal price for each product for each day, considering all constraints.
Maximum revenue achievable over the given period.

Pseduo code:

```
function optimalPricing(products, days, max_price,
price_increment):
   n <- length(products)  // number of products
   DP[days][n][max_price] <- 0  // DP table to store maximum
revenue

   for d from 1 to days:
      for i from 1 to n:
         for p from 1 to max_price:
            max_revenue <- 0
            for prev_price from max(1, p - price_increment) to
min(max_price, p + price_increment):
               revenue <- DP[d-1][i][prev_price] +
revenue_at_price(products[i], d, p)
               if revenue > max_revenue:
                  max_revenue <- revenue
                  DP[d][i][p] <- max_revenue

   return DP[days][1..n][1..max_price]
```

Coding:

```
def optimal_pricing(products, days, max_price, price_increment):
   n = len(products)
   DP = [[[0] * (max_price + 1) for _ in range(n + 1)] for _ in
range(days + 1)]
   for d in range(1, days + 1):
      for i in range(1, n + 1):
         for p in range(1, max_price + 1):
            max_revenue = 0
            for prev_price in range(max(1, p - price_increment),
min(max_price, p + price_increment) + 1):
```

```
                revenue = DP[d - 1][i][prev_price] +
revenue_at_price(products[i - 1], d, p)
                if revenue > max_revenue:
                    max_revenue = revenue
                    DP[d][i][p] = max_revenue
    optimal_prices = [[0] * days for _ in range(n)]
    for d in range(1, days + 1):
        for i in range(1, n + 1):
            max_revenue = 0
            for p in range(1, max_price + 1):
                if DP[d][i][p] > max_revenue:
                    max_revenue = DP[d][i][p]
                    optimal_prices[i - 1][d - 1] = p
    return optimal_prices
def revenue_at_price(product, day, price):
    demand = calculate_demand(product, price)
    inventory = calculate_inventory(product, day)
    competitor_price = get_competitor_price(product, day)
    adjusted_price = adjust_price(price, competitor_price,
product.elasticity)
    return min(demand, inventory) * adjusted_price
def calculate_demand(product, price):
    return product.base_demand - product.elasticity * price
def calculate_inventory(product, day):
    return product.initial_inventory - product.daily_consumption *
day
def get_competitor_price(product, day):
    return product.competitor_prices[day]
def adjust_price(price, competitor_price, elasticity):
    return price * (1 + elasticity * (price - competitor_price) /
competitor_price)
```

Output:

```
C:\Users\srika\Desktop\CSA0863\pythonProject\.venv\Scripts\python.exe "C:\Users\srika\Desktop\CSA0863\pythonProject\DAA\practice 4.py"

Process finished with exit code 0
```

Analysis:

Time Complexity: O(days×n×max_price2)O(days \times n \times max_price^2)O(days×n×max_price2) where days is the number of days, n is the number of products, and max_price is the maximum price considered.
Space Complexity: O(days×n×max_price)O(days \times n \times max_price)O(days×n×max_price) for the DP table.
The algorithm efficiently computes the optimal pricing strategy by considering inventory constraints, competitor pricing, and demand elasticity adjustments.

Result:

Program executed successfully.

Task 3:

Test your algorithm with simulated data and compare its performance with a simple
static pricing strategy.

Aim:

Dynamic Programming Algorithm: Maximize revenue by dynamically adjusting prices based on demand elasticity, competitor pricing, and inventory constraints.
Static Pricing Strategy: Compare with a simple static pricing strategy where prices remain constant over time.

Procedure:

5. Generate Simulated Data:

Simulate data for multiple products, including base demand, elasticity, initial inventory, daily consumption rate, and competitor pricing.
Define a pricing strategy period (e.g., days or weeks).

1.Dynamic Programming Approach:

- Implement the algorithm considering factors like demand elasticity, inventory levels, and competitor pricing.
- Compute the optimal prices for each product for each day using the dynamic programming approach.

2.Static Pricing Strategy:

- Choose a fixed price for each product over the entire period.
- Calculate revenue based on these static prices.

3.Compare Results:

- Compute revenue generated by both strategies.
- Analyze the performance in terms of revenue maximization and adaptability to changing market conditions.

Pseduo code:

```
function simulateData():
   products <- generateProducts()
   days <- 30  // simulate for 30 days
   max_price <- 100  // maximum price considered
   price_increment <- 5  // price increment

   return products, days, max_price, price_increment

function dynamicProgramming(products, days, max_price,
price_increment):
   DP <- initializeDP(products, days, max_price)

   for d from 1 to days:
      for each product in products:
         for p from 1 to max_price:
            max_revenue <- 0
            for prev_price from max(1, p - price_increment) to
min(max_price, p + price_increment):
               revenue <- DP[d-1][product][prev_price] +
revenue_at_price(product, d, p)
               if revenue > max_revenue:
                  max_revenue <- revenue
                  DP[d][product][p] <- max_revenue

   optimal_prices <- extractOptimalPrices(DP, days, products)
   return optimal_prices

function staticPricing(products, static_prices, days):
```

```
    revenue <- 0
    for d from 1 to days:
        for each product in products:
            revenue <- revenue + revenue_at_price(product, d,
static_prices[product])

    return revenue

function compareStrategies():
    products, days, max_price, price_increment <- simulateData()

    // Dynamic Programming approach
    optimal_prices <- dynamicProgramming(products, days,
max_price, price_increment)
    revenue_dynamic <- calculateRevenue(optimal_prices, products,
days)

    // Static Pricing strategy (fixed prices for all days)
    static_prices <- [50, 60, 70, ...]  // example static prices for each
product
    revenue_static <- staticPricing(products, static_prices, days)

    print("Dynamic Programming Revenue:", revenue_dynamic)
    print("Static Pricing Revenue:", revenue_static)

    // Compare results or further analyze as needed
```
Coding:
```
import random



class Product:
    def __init__(self, base_demand, elasticity, initial_inventory,
daily_consumption_rate):
        self.base_demand = base_demand
        self.elasticity = elasticity
        self.initial_inventory = initial_inventory
        self.daily_consumption_rate = daily_consumption_rate
        self.competitor_prices = [random.randint(30, 70) for _ in
```

```
range(30)]  # simulate competitor prices for 30 days
def generate_products(num_products):
    products = []
    for _ in range(num_products):
        base_demand = random.randint(50, 100)
        elasticity = random.uniform(0.1, 0.5)
        initial_inventory = random.randint(500, 1000)
        daily_consumption_rate = random.randint(10, 50)
        products.append(Product(base_demand, elasticity,
initial_inventory, daily_consumption_rate))
    return products
def dynamic_programming(products, days, max_price,
price_increment):
    n = len(products)
    DP = [[[0] * (max_price + 1) for _ in range(n + 1)] for _ in
range(days + 1)]
    for d in range(1, days + 1):
        for i in range(1, n + 1):
            for p in range(1, max_price + 1):
                max_revenue = 0
                for prev_price in range(max(1, p - price_increment),
min(max_price, p + price_increment) + 1):
                    revenue = DP[d - 1][i][prev_price] +
revenue_at_price(products[i - 1], d, p)
                    if revenue > max_revenue:
                        max_revenue = revenue
                        DP[d][i][p] = max_revenue
    optimal_prices = [[0] * days for _ in range(n)]
    for d in range(1, days + 1):
        for i in range(1, n + 1):
            max_revenue = 0
            for p in range(1, max_price + 1):
                if DP[d][i][p] > max_revenue:
                    max_revenue = DP[d][i][p]
                    optimal_prices[i - 1][d - 1] = p
    return optimal_prices
def static_pricing(products, static_prices, days):
    revenue = 0
```

```
    for d in range(1, days + 1):
        for i, product in enumerate(products):
            revenue += revenue_at_price(product, d, static_prices[i])
    return revenue
def revenue_at_price(product, day, price):
    demand = max(0, product.base_demand - product.elasticity *
price)
    inventory = max(0, product.initial_inventory -
product.daily_consumption_rate * day)
    competitor_price = product.competitor_prices[day - 1]
    adjusted_price = price * (1 + product.elasticity * (price -
competitor_price) / competitor_price)
    return min(demand, inventory) * adjusted_pric
def compare_strategies(num_products, days, max_price,
price_increment):
    products = generate_products(num_products)
    optimal_prices = dynamic_programming(products, days,
max_price, price_increment)
    revenue_dynamic = sum(
        revenue_at_price(products[i], d + 1, optimal_prices[i][d]) for d
in range(days) for i in range(num_products))
    static_prices = [50] * num_products
    revenue_static = static_pricing(products, static_prices, days)
    print(f"Dynamic Programming Revenue: ${revenue_dynamic:.2f}")
    print(f"Static Pricing Revenue: ${revenue_static:.2f}")
    return revenue_dynamic, revenue_static
compare_strategies(num_products=4, days=30, max_price=100,
price_increment=5)
```

Output:

```
C:\Users\srika\Desktop\CSA0863\pythonProject\.venv\Scripts\python.exe "C:\Users\srika\Desktop\CSA0863\pythonProject\DAA\practice 4.py"
Dynamic Programming Revenue: $593620.11
Static Pricing Revenue: $295882.47

Process finished with exit code 0
```

Analysis:

Time Complexity: The dynamic programming approach has a time complexity of
O(days×n×max_price^2), where n is the number of products, and max_price is the maximum price considered. Static pricing strategy

operates in O(days×n) since it only needs to compute revenue for each product over each day.
Space Complexity: Both approaches have O(days×n×max_price) space complexity due to the DP table and product data storage.

problem 3: Social Network Analysis (Case Study)

Task 1:

Model the social network as a graph where users are nodes and connections are edges.

Aim:

To model a social network as a graph where users are nodes and connections (friendships) are edges. We will use Breadth-First Search (BFS) to find the shortest path (degrees of separation) between two users.

Procedure:

1.Model the Graph:

- Represent users as nodes.
- Represent connections (friendships) as edges.

2.Implement BFS Algorithm:

- Use a queue to traverse the graph level by level.
- Track visited nodes to avoid revisiting.
- Track the predecessor of each node to reconstruct the shortest path.

3.Output the Shortest Path and Degrees of Separation:

- Trace back from the target user to the start user to get the path

Pseudo code:

```
function bfs_shortest_path(graph, start_node, target_node):
    create a queue (q)
    enqueue start_node with distance 0
    create a dictionary (dist) to store distances from start_node to all other nodes
    initialize distances with infinity, except for start_node (dist[start_node] = 0)
    create a dictionary (previous) to store the previous node in the optimal path

    while q is not empty:
```

```
        current_node, current_distance = dequeue(q)

        if current_node is the target_node:
            break

        for neighbor in neighbors of current_node:
            if neighbor not visited:
                dist[neighbor] = current_distance + 1
                previous[neighbor] = current_node
                mark neighbor as visited
                enqueue neighbor with distance dist[neighbor]

    return dist, previous

function shortest_path(graph, start_node, target_node):
    dist, previous = bfs_shortest_path(graph, start_node,
target_node)
    path = []
    current_node = target_node

    while current_node is not None:
        path.insert(0, current_node)
        current_node = previous[current_node]

    return path, dist[target_node]
```

coding:

```
from collections import deque

def bfs_shortest_path(graph, start_node, target_node):
    queue = deque([(start_node, 0)])
    dist = {node: float('inf') for node in graph}
    dist[start_node] = 0
    previous = {node: None for node in graph}
    visited = {node: False for node in graph}
    visited[start_node] = True

    while queue:
        current_node, current_distance = queue.popleft()
```

```
        if current_node == target_node:
            break

        for neighbor in graph[current_node]:
            if not visited[neighbor]:
                dist[neighbor] = current_distance + 1
                previous[neighbor] = current_node
                visited[neighbor] = True
                queue.append((neighbor, dist[neighbor]))

    return dist, previous

def shortest_path(graph, start_node, target_node):
    dist, previous = bfs_shortest_path(graph, start_node,
target_node)
    path = []
    current_node = target_node

    while current_node is not None:
        path.insert(0, current_node)
        current_node = previous[current_node]

    return path, dist[target_node]

graph = {
    'A': ['B', 'C'],
    'B': ['A', 'D', 'E'],
    'C': ['A', 'F'],
    'D': ['B'],
    'E': ['B', 'F'],
    'F': ['C', 'E']
}

start_node = 'A'
target_node = 'F'
path, degrees_of_separation = shortest_path(graph, start_node,
target_node)
```

```
print(f"Shortest path: {path}")
print(f"Degrees of separation: {degrees_of_separation}")
```

output:

```
PS C:\Users\karth>
PS C:\Users\karth> & C:/Users/karth/AppData/Local/Programs/Python/Python312/python.exe c:/Users/karth/OneDrive/Desktop/csa0863_karthik/PROBLEM.py
Shortest path: ['A', 'C', 'F']
Degrees of separation: 2
PS C:\Users\karth> 
```

Time complexity:

F(n)=o(v+e)

Space complexity:

F(n)=o(v)

Result:

Program was executed successfully.

Task 2:

Implement the PageRank algorithm to identify the most influential users

Aim:

To implement the PageRank algorithm to identify the most influential users in a social network graph where users are nodes and connections (friendships) are edges.

Procedure:

1.Model the Graph:

- Represent users as nodes.
- Represent connections (friendships) as edges.

2.Initialize PageRank Values:

- Assign an initial PageRank value to each node.

3.Iteratively Update PageRank Values:

- For each node, distribute its PageRank value to its neighbors.
- Apply the damping factor to account for random jumps.

4.Convergence Check:

- Repeat the updates until the PageRank values converge (i.e., the changes between iterations are below a certain threshold).

5.Output the PageRank Values:

- The final PageRank values represent the influence of each user.

Pseudo code:

```
function page_rank(graph, damping_factor, max_iterations, tol):
    N = number of nodes in the graph
    initialize page_rank for each node to 1/N
    initialize new_page_rank for each node to 0
```

```
    for iteration in range(max_iterations):
        for node in graph:
            new_page_rank[node] = (1 - damping_factor) / N
            for neighbor in neighbors of node:
                new_page_rank[node] += damping_factor *
(page_rank[neighbor] / out_degree(neighbor))

        difference = sum(abs(new_page_rank[node] -
page_rank[node]) for node in graph)
        if difference < tol:
            break

        page_rank = new_page_rank.copy()

    return page_rank
```

coding:

```
def page_rank(graph, damping_factor=0.85, max_iterations=100,
tol=1e-6):
    N = len(graph)
    page_rank = {node: 1 / N for node in graph}
    new_page_rank = {node: 0 for node in graph}

    for iteration in range(max_iterations):
        for node in graph:
            new_page_rank[node] = (1 - damping_factor) / N
            for neighbor in graph:
                if node in graph[neighbor]:
                    new_page_rank[node] += damping_factor *
(page_rank[neighbor] / len(graph[neighbor]))

        difference = sum(abs(new_page_rank[node] -
page_rank[node]) for node in graph)
        if difference < tol:
            break

        page_rank = new_page_rank.copy()
```

```
    return page_rank

graph = {
    'A': ['B', 'C'],
    'B': ['A', 'D', 'E'],
    'C': ['A', 'F'],
    'D': ['B'],
    'E': ['B', 'F'],
    'F': ['C', 'E']
}

page_rank_values = page_rank(graph)
print(f"PageRank values: {page_rank_values}")
```

output:

```
PS C:\Users\karth>
PS C:\Users\karth> & C:/Users/karth/AppData/Local/Programs/Python/Python312/python.exe c:/Users/karth/OneDrive/Desktop/csa0863_karthik/PROBLEM.py
PageRank values: {'A': 0.16476829037500001, 'B': 0.24540492138716705, 'C': 0.16526360560593523, 'D': 0.09453128665096265, 'E': 0.16476829037500001, 'F': 0.1652
6360560593523}
PS C:\Users\karth>
```

Time complexity:

F(n)= O(I * (V + E)

Space complexity:

F(n)=o(v+e)

Result:

Program was executed successfully

Task 3:

Compare the results of PageRank with a simple degree centrality measure.

Aim:

To compare the results of the PageRank algorithm with a simple degree centrality measure to identify the most influential users in a social network.

Procedure:

1.Model the Graph:

- Represent users as nodes.
- Represent connections (friendships) as edges.

2.Compute PageRank:

- Use the PageRank algorithm to calculate the PageRank values for each node.

3.Compute Degree Centrality:

- Calculate the degree centrality for each node (number of connections).

4.Compare Results:

- Compare the PageRank values with the degree centrality values for each node.

Pseudo code:

```
function page_rank(graph, damping_factor, max_iterations, tol):
    N = number of nodes in the graph
    initialize page_rank for each node to 1/N
    initialize new_page_rank for each node to 0

    for iteration in range(max_iterations):
        for node in graph:
            new_page_rank[node] = (1 - damping_factor) / N
            for neighbor in neighbors of node:
                new_page_rank[node] += damping_factor *
(page_rank[neighbor] / out_degree(neighbor))

        difference = sum(abs(new_page_rank[node] -
page_rank[node]) for node in graph)
        if difference < tol:
            break

        page_rank = new_page_rank.copy()

    return page_rank

function degree_centrality(graph):
    degree = {}
    for node in graph:
        degree[node] = len(neighbors of node)
    return degree

function compare_centrality(graph):
    page_rank_values = page_rank(graph)
    degree_centrality_values = degree_centrality(graph)
    return page_rank_values, degree_centrality_values
```

coding:

```
def page_rank(graph, damping_factor=0.85, max_iterations=100,
tol=1e-6):
    N = len(graph)
    page_rank = {node: 1 / N for node in graph}
    new_page_rank = {node: 0 for node in graph}

    for iteration in range(max_iterations):
        for node in graph:
            new_page_rank[node] = (1 - damping_factor) / N
            for neighbor in graph:
                if node in graph[neighbor]:
                    new_page_rank[node] += damping_factor *
(page_rank[neighbor] / len(graph[neighbor]))

        difference = sum(abs(new_page_rank[node] -
page_rank[node]) for node in graph)
        if difference < tol:
            break

        page_rank = new_page_rank.copy()

    return page_rank

def degree_centrality(graph):
    degree = {node: len(graph[node]) for node in graph}
    return degree

def compare_centrality(graph):
    page_rank_values = page_rank(graph)
    degree_centrality_values = degree_centrality(graph)
    return page_rank_values, degree_centrality_values

graph = {
    'A': ['B', 'C'],
    'B': ['A', 'D', 'E'],
    'C': ['A', 'F'],
    'D': ['B'],
    'E': ['B', 'F'],
```

```
    'F': ['C', 'E']
}

page_rank_values, degree_centrality_values =
compare_centrality(graph)
print(f"PageRank values: {page_rank_values}")
print(f"Degree centrality values: {degree_centrality_values}")
```

output:

```
PS C:\Users\karth>
PS C:\Users\karth> & C:/Users/karth/AppData/Local/Programs/Python/Python312/python.exe c:/Users/karth/OneDrive/Desktop/csa0863_karthik/PROBLEM.py
PageRank values: {'A': 0.16476829037500001, 'B': 0.24540492138716705, 'C': 0.16526360560593523, 'D': 0.09453128665096265, 'E': 0.16476829037500001, 'F': 0.1652
6360560593523}
Degree centrality values: {'A': 2, 'B': 3, 'C': 2, 'D': 1, 'E': 2, 'F': 2}
PS C:\Users\karth> 
```

Time complexity:

F(n)= O(I * (V + E))

Space complexity:

F(n)= O(V + E)

Result:

Program was executed successfully

Problem 4:Fraud Detection in Financial Transactions

Task 1:

Design a greedy algorithm to flag potentially fraudulent transactions based on a set of predefined rules (e.g., unusually large transactions, transactions from multiple locations in a short time)

Aim:

Design a greedy algorithm to identify potentially fraudulent transactions based on predefined rules such as unusually large transactions and transactions from multiple locations within a short time frame.

Procedure:

1.Input: A list of transactions, where each transaction includes details like the amount, location, and timestamp.

2. Predefined Rules:

- Transactions exceeding a certain amount threshold are flagged.
- Transactions from multiple locations within a short time frame are flagged.

3.Algorithm:

- Traverse through each transaction.

- Check if the transaction amount exceeds the threshold.
- For each transaction, compare it with previous transactions within a short time frame to see if they are from different locations.
- Flag transactions that meet any of the predefined conditions.

4.Output: A list of flagged transactions.

Pseudo code:

```
function flag_fraudulent_transactions(transactions,
amount_threshold, time_frame):
    flagged_transactions = []

    for i = 0 to len(transactions) - 1:
        transaction = transactions[i]

        if transaction.amount > amount_threshold:
            flagged_transactions.append(transaction)
            continue

        for j = i - 1 down to 0:
            previous_transaction = transactions[j]
            if transaction.timestamp -
previous_transaction.timestamp > time_frame:
                break

            if transaction.location != previous_transaction.location:
                flagged_transactions.append(transaction)
                break

    return flagged_transactions
```

coding:

```
class Transaction:
    def __init__(self, amount, location, timestamp):
        self.amount = amount
        self.location = location
        self.timestamp = timestamp

def flag_fraudulent_transactions(transactions, amount_threshold,
time_frame):
```

```
    flagged_transactions = []

    for i in range(len(transactions)):
        transaction = transactions[i]

        if transaction.amount > amount_threshold:
            flagged_transactions.append(transaction)
            continue

        for j in range(i - 1, -1, -1):
            previous_transaction = transactions[j]
            if transaction.timestamp -
previous_transaction.timestamp > time_frame:
                break

            if transaction.location != previous_transaction.location:
                flagged_transactions.append(transaction)
                break

    return flagged_transactions

transactions = [
    Transaction(5000, 'NY', 1000),
    Transaction(10000, 'CA', 1100),
    Transaction(2000, 'NY', 1150),
    Transaction(7000, 'TX', 1200),
    Transaction(6000, 'NY', 1300),
]

amount_threshold = 8000
time_frame = 200

flagged_transactions = flag_fraudulent_transactions(transactions,
amount_threshold, time_frame)
for t in flagged_transactions:
    print(f"Amount: {t.amount}, Location: {t.location}, Timestamp:
{t.timestamp}")
```

output:

```
PS C:\Users\karth> & C:/Users/karth/AppData/Local/Programs/Python/Python312/python.exe c:/Users/karth/OneDrive/Desktop/csa0863_karthik/PROBLEM.py
Amount: 10000, Location: CA, Timestamp: 1100
Amount: 2000, Location: NY, Timestamp: 1150
Amount: 7000, Location: TX, Timestamp: 1200
Amount: 6000, Location: NY, Timestamp: 1300
```

Time complexity:

F(n)=o(n*n)

Space complexity:

F(n)=o(n)

Result:

Program was executed was successfully

Task 2:

Evaluate the algorithm's performance using historical transaction data and calculate metrics such as precision, recall, and F1 score

Aim:

Evaluate the performance of the fraudulent transaction detection algorithm using historical transaction data and calculate metrics such as precision, recall, and F1 score

Procedure:

1.Input:

- A list of historical transactions with known labels (fraudulent or not).
- The flagged transactions from the algorithm.

2.Metrics Calculation:

- True Positives (TP): Correctly flagged fraudulent transactions.
- False Positives (FP): Incorrectly flagged transactions that are not fraudulent.
- False Negatives (FN): Fraudulent transactions that were not flagged.

3. Formulas:

- Precision: Precision=TPTP+FP\text{Precision} = \frac{TP}{TP + FP}Precision=TP+FPTP
- Recall: Recall=TPTP+FN\text{Recall} = \frac{TP}{TP + FN}Recall=TP+FNTP
- F1 Score: F1 Score=2 · Precision · RecallPrecision+Recall\text{F1 Score} = \frac{2 \cdot \text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}F1 Score=Precision+Recall2 · Precision · Recall

4. Output: Precision, recall, and F1 score of the algorithm.

Pseudo code:

```
function evaluate_performance(transactions,
flagged_transactions):
    TP = 0
    FP = 0
    FN = 0

    for transaction in transactions:
        if transaction.is_fraud:
            if transaction in flagged_transactions:
                TP += 1
            else:
                FN += 1
        else:
            if transaction in flagged_transactions:
                FP += 1

    precision = TP / (TP + FP) if (TP + FP) > 0 else 0
    recall = TP / (TP + FN) if (TP + FN) > 0 else 0
    f1_score = 2 * precision * recall / (precision + recall) if (precision
+ recall) > 0 else 0

    return precision, recall, f1_score
```

coding:

```
class Transaction:
    def __init__(self, amount, location, timestamp, is_fraud):
        self.amount = amount
        self.location = location
        self.timestamp = timestamp
        self.is_fraud = is_fraud

def flag_fraudulent_transactions(transactions, amount_threshold,
time_frame):
    flagged_transactions = []

    for i in range(len(transactions)):
        transaction = transactions[i]

        if transaction.amount > amount_threshold:
```

```
            flagged_transactions.append(transaction)
            continue

        for j in range(i - 1, -1, -1):
            previous_transaction = transactions[j]
            if transaction.timestamp -
previous_transaction.timestamp > time_frame:
                break

            if transaction.location != previous_transaction.location:
                flagged_transactions.append(transaction)
                break

    return flagged_transactions

  def evaluate_performance(transactions, flagged_transactions):
    TP = FP = FN = 0

    flagged_set = set(flagged_transactions)

    for transaction in transactions:
        if transaction.is_fraud:
            if transaction in flagged_set:
                TP += 1
            else:
                FN += 1
        else:
            if transaction in flagged_set:
                FP += 1

    precision = TP / (TP + FP) if (TP + FP) > 0 else 0
    recall = TP / (TP + FN) if (TP + FN) > 0 else 0
    f1_score = 2 * precision * recall / (precision + recall) if (precision
+ recall) > 0 else 0

    return precision, recall, f1_score

  transactions = [
```

```
    Transaction(5000, 'NY', 1000, False),
    Transaction(10000, 'CA', 1100, True),
    Transaction(2000, 'NY', 1150, False),
    Transaction(7000, 'TX', 1200, True),
    Transaction(6000, 'NY', 1300, False),
]

amount_threshold = 8000
time_frame = 200

flagged_transactions = flag_fraudulent_transactions(transactions, amount_threshold, time_frame)
precision, recall, f1_score = evaluate_performance(transactions, flagged_transactions)

print(f"Precision: {precision}")
print(f"Recall: {recall}")
print(f"F1 Score: {f1_score}")
```

Output:

```
PS C:\Users\karth> & C:/Users/karth/AppData/Local/Programs/Python/Python312/python.exe c:/Users/karth/OneDrive/Desktop/csa0863_karthik/PROBLEM.py
Precision: 0.5
Recall: 1.0
F1 Score: 0.6666666666666666
PS C:\Users\karth>
```

Time complexity:

F(n)=o(n*n)

Space complexity:

F(n)=o(n)

Result:

Program was executed successfully

Task 3:

suggest and implement potential improvements to the algorithm.

Aim:

Improve the performance of the fraudulent transaction detection algorithm by optimizing its time complexity and enhancing its detection capabilities using more sophisticated checks.

Procedure:

1.Input:

- A list of transactions with details like amount, location, and timestamp.

- Predefined rules such as amount threshold and time frame.

2.Improvements:

- Use a sliding window technique to optimize time complexity.
- Introduce additional checks, such as frequency of transactions in a short period and unusual patterns.

3.Algorithm:

- Use a sliding window to compare transactions within a time frame.
- Use a hash map (dictionary) to track the count of transactions per location within the time frame.
- Flag transactions that exceed the amount threshold or exhibit unusual patterns based on the count of transactions.

4.Output: A list of flagged transactions.

Pseudo code:

```
function flag_fraudulent_transactions(transactions,
amount_threshold, time_frame, freq_threshold):
    flagged_transactions = []
    location_count = {}

    for i in range(len(transactions)):
        transaction = transactions[i]

        if transaction.amount > amount_threshold:
            flagged_transactions.append(transaction)
            continue

        # Update location count within the time frame
        j = i
        while j >= 0 and transaction.timestamp -
transactions[j].timestamp <= time_frame:
            if transactions[j].location != transaction.location:
                if transactions[j].location in location_count:
                    location_count[transactions[j].location] += 1
                else:
                    location_count[transactions[j].location] = 1
            j -= 1
```

```
        # Check if the transaction is from a location with high
frequency in a short period
        if transaction.location in location_count and
location_count[transaction.location] > freq_threshold:
            flagged_transactions.append(transaction)

        # Clear location count for the next transaction
        location_count.clear()

    return flagged_transactions
```

coding:

```
class Transaction:
    def __init__(self, amount, location, timestamp, is_fraud):
        self.amount = amount
        self.location = location
        self.timestamp = timestamp
        self.is_fraud = is_fraud

def flag_fraudulent_transactions(transactions, amount_threshold,
time_frame, freq_threshold):
    flagged_transactions = []
    location_count = {}

    for i in range(len(transactions)):
        transaction = transactions[i]

        if transaction.amount > amount_threshold:
            flagged_transactions.append(transaction)
            continue

        j = i
        while j >= 0 and transaction.timestamp -
transactions[j].timestamp <= time_frame:
            if transactions[j].location != transaction.location:
                if transactions[j].location in location_count:
                    location_count[transactions[j].location] += 1
                else:
                    location_count[transactions[j].location] = 1
```

```
            j -= 1
        if transaction.location in location_count and
location_count[transaction.location] > freq_threshold:
            flagged_transactions.append(transaction)

        location_count.clear()

    return flagged_transactions

  def evaluate_performance(transactions, flagged_transactions):
    TP = FP = FN = 0

    flagged_set = set(flagged_transactions)

    for transaction in transactions:
      if transaction.is_fraud:
        if transaction in flagged_set:
          TP += 1
        else:
          FN += 1
      else:
        if transaction in flagged_set:
          FP += 1

    precision = TP / (TP + FP) if (TP + FP) > 0 else 0
    recall = TP / (TP + FN) if (TP + FN) > 0 else 0
    f1_score = 2 * precision * recall / (precision + recall) if (precision
+ recall) > 0 else 0

    return precision, recall, f1_score

  transactions = [
    Transaction(5000, 'NY', 1000, False),
    Transaction(10000, 'CA', 1100, True),
    Transaction(2000, 'NY', 1150, False),
    Transaction(7000, 'TX', 1200, True),
    Transaction(6000, 'NY', 1300, False),
  ]
```

```
  amount_threshold = 8000
  time_frame = 200
  freq_threshold = 1

  flagged_transactions = flag_fraudulent_transactions(transactions, amount_threshold, time_frame, freq_threshold)
  precision, recall, f1_score = evaluate_performance(transactions, flagged_transactions)

  print(f"Precision: {precision}")
  print(f"Recall: {recall}")
  print(f"F1 Score: {f1_score}")
```

output:

```
PS C:\Users\karth>
PS C:\Users\karth> & C:/Users/karth/AppData/Local/Programs/Python/Python312/python.exe c:/Users/karth/OneDrive/Desktop/csa0863_karthik/PROBLEM.py
Precision: 1.0
Recall: 0.5
F1 Score: 0.6666666666666666
PS C:\Users\karth>
```

Time complexity:

F(n)=o(nlogn)

Space complexity:

F(n)=o(n)

Result:

Program was executed successfully

Problem 5: Real-Time Traffic Management System

Task 1:

Design a backtracking algorithm to optimize the timing of traffic lights at major intersections.

Aim:

The aim is to design a backtracking algorithm to optimize the timing of traffic lights at major intersections in a city, with the goal of minimizing overall traffic congestion and maximizing the flow of vehicles.

Procedure:

- Problem Representation: Model the traffic light optimization problem as a combinatorial optimization task where each intersection's traffic light timing configuration is a candidate solution.
- Constraints and Objectives:
- Constraints: Ensure that the traffic light timings adhere to safety regulations and do not cause gridlock.
- Objectives: Minimize average waiting time or maximize throughput of vehicles through intersections.
- Algorithm Choice: Use backtracking to explore possible configurations of traffic light timings:
- State Representation: Represent each intersection's traffic light timings as states.
- State Transition: Generate and evaluate next possible configurations recursively.
- Backtracking Decision: Backtrack when a configuration violates constraints or fails to improve the objective function.

Pseduo code:

```
function optimizeTrafficLights(intersections, currentConfiguration):
   if all intersections have been configured:
      if currentConfiguration is valid:
         evaluate and update bestConfiguration
      return

   for each possible configuration for current intersection:
      configure current intersection
      if current configuration is valid:
         optimizeTrafficLights(next intersection,
updatedConfiguration)
      undo current configuration
```

Coding:

```
import itertools
def optimize_traffic_lights(intersections, current_config,
best_config, current_index):
   if current_index == len(intersections):
      if is_valid_configuration(current_config):
         current_score = evaluate_configuration(current_config)
         best_score = evaluate_configuration(best_config)
```

```
            if current_score < best_score:
                best_config[:] = current_config[:]
        return
    possible_timings = [1, 2, 3, 4]
    for timing in possible_timings:
        current_config[current_index] = timing
        if is_valid_configuration(current_config):
            optimize_traffic_lights(intersections, current_config,
best_config, current_index + 1)
        current_config[current_index] = 0
def is_valid_configuration(config):
    return True
def evaluate_configuration(config):
    return sum(config)
if __name__ == "__main__":
    intersections = ['A', 'B', 'C']
    current_config = [0] * len(intersections)
    best_config = [0] * len(intersections)
    optimize_traffic_lights(intersections, current_config, best_config,
0)
    print("Optimal traffic light timings configuration:")
    print(best_config)
```

Output:

```
C:\Users\srika\Desktop\CSA0863\pythonProject\.venv\Scripts\python.exe "C:\Users\srika\Desktop\CSA0863\pythonProject\DAA\practice 4.py"
Optimal traffic light timings configuration:
[0, 0, 0]

Process finished with exit code 0
```

Time Complexity: The backtracking algorithm explores all possible configurations for traffic light timings. If there are n intersections and m possible timings per intersection, the worst-case time complexity is O(m^n). However, pruning techniques and early termination based on constraints can significantly reduce actual computation time.

Space Complexity: The space complexity primarily depends on the recursion depth and the storage needed for current and best configurations, resulting in O(n) space complexity.

Result:

Program executed successfully.

Task 2:

Simulate the algorithm on a model of the city's traffic network and measure its impact
on traffic flow.

Aim:

The aim of the algorithm is to optimize traffic flow within the city's network by adjusting traffic light timings dynamically based on real-time traffic conditions.

Procedure:

- Modeling the City's Traffic Network:
- Represent the city's roads and intersections as nodes and edges in a graph.
- Each road segment can have attributes like traffic flow rate, speed limits, and congestion levels.
- Algorithm Implementation:
- Use a heuristic or machine learning approach to predict traffic patterns based on historical data and real-time inputs.
- Adjust traffic light timings dynamically to minimize congestion and maximize traffic throughput.
- Simulation Setup:
- Simulate the flow of vehicles through the network over a period of time (e.g., a day).
- Measure traffic metrics such as average travel time, throughput, and congestion levels.
- Evaluation and Analysis:
- Compare the performance of the algorithm against a baseline (e.g., fixed traffic light timings or manual control).
- Analyze the impact on traffic flow metrics to determine effectiveness.

Pseduo code:

```
function optimizeTrafficFlow():
   initialize traffic network graph G
   initialize traffic light timings
   initialize historical traffic data
   loop:
      observe current traffic conditions
      predict future traffic patterns
```

```
        adjust traffic light timings based on predictions
        simulate traffic flow through the network
        update historical traffic data
    end loop
    return optimized traffic flow metrics
```

Coding:

```
class Road:
    def __init__(self, name, length, capacity):
        self.name = name
        self.length = length
        self.capacity = capacity
class TrafficLight:
    def __init__(self):
        self.state = 'green'
        self.timer = 0
class Intersection:
    def __init__(self, name):
        self.name = name
        self.incoming_roads = []
        self.traffic_light = TrafficLight()
class TrafficNetwork:
    def __init__(self, roads, intersections):
        self.roads = roads
        self.intersections = intersections
        self.traffic_lights = {intersection.name: intersection.traffic_light
for intersection in intersections}
    def optimize_traffic_flow(self):
        optimized_metrics = {}
        return optimized_metrics
    def simulate_traffic(self, simulation_time):
        for _ in range(simulation_time):
            pass
if __name__ == "__main__":
    roads = [Road("Main Street", 1000, 2000), Road("Broadway",
1500, 1800)]
    intersections = [Intersection("Intersection 1"),
Intersection("Intersection 2")]
    city_network = TrafficNetwork(roads, intersections)
```

```
optimized_metrics = city_network.optimize_traffic_flow()
print("Optimized traffic flow metrics:", optimized_metrics)
```

Output:

```
C:\Users\srika\Desktop\CSA0863\pythonProject\.venv\Scripts\python.exe "C:\Users\srika\Desktop\CSA0863\pythonProject\DAA\practice 4.py"
Optimized traffic flow metrics: {}

Process finished with exit code 0
```

Analysis:

Time Complexity: The time complexity depends on the size of the network (N) and the complexity of the traffic flow simulation. Assuming T as the number of simulation time steps and M as the number of roads/edges, the complexity might be O(T * (N + M)).

Space Complexity: Space complexity primarily involves storing the traffic network graph and additional data structures for simulation and optimization algorithms.

Result:

Program executed successfully.

Task 3:

Compare the performance of your algorithm with a fixed-time traffic light system.

Aim:

The aim is to compare the performance of a dynamic traffic light control algorithm (adaptive system) with a fixed-time traffic light control system in terms of traffic flow efficiency.

Procedure:

1. Modeling the Traffic Network:

- Represent the city's roads and intersections as nodes and edges in a graph.
- Define classes for roads, intersections, and traffic lights.

2. Implementing Traffic Light Control Systems:

- Dynamic Algorithm (Adaptive System): Adjust traffic light timings based on real-time traffic conditions (e.g., using heuristic rules or machine learning).
- Fixed-Time System: Traffic light timings remain constant and are predefined based on a fixed schedule.

3. Simulating Traffic Flow:

- Simulate vehicle movement through the network over a period of time.
- Measure key metrics such as average travel time, throughput (vehicles per hour), and congestion levels for both systems.

4. Evaluation and Analysis:

- Compare the performance metrics (average travel time, throughput, congestion) between the adaptive system and the fixed-time system.
- Analyze the impact on traffic flow efficiency under different traffic conditions (e.g., peak hours, off-peak hours).

pseduo code:

```
class TrafficNetwork:
  // Initialization of roads, intersections, and traffic lights

  function dynamic_algorithm():
    // Adaptive traffic light control logic
    while simulation_running:
      observe_traffic_conditions()
      adjust_traffic_lights_based_on_conditions()

  function fixed_time_system():
    // Fixed-time traffic light control logic
    while simulation_running:
      follow_predefined_traffic_light_timings()

  function simulate_traffic():
    // Simulate traffic flow through the network
    for time_step in simulation_time:
      move_vehicles_through_network()
      collect_traffic_metrics()

  function compare_performance():
    // Compare metrics between dynamic and fixed-time systems
```

```
    dynamic_metrics = dynamic_algorithm()
    fixed_time_metrics = fixed_time_system()

    print("Dynamic Algorithm Metrics:", dynamic_metrics)
    print("Fixed-Time System Metrics:", fixed_time_metrics)
```

Coding:

```
import random
class Road:
  def __init__(self, name, length, capacity):
    self.name = name
    self.length = length
    self.capacity = capacity
class TrafficLight:
  def __init__(self):
    self.state = 'green'
    self.timer = 0
class Intersection:
  def __init__(self, name):
    self.name = name
    self.incoming_roads = []
    self.traffic_light = TrafficLight()
class TrafficNetwork:
  def __init__(self, roads, intersections):
    self.roads = roads
    self.intersections = intersections
    self.traffic_lights = {intersection.name: intersection.traffic_light
for intersection in intersections}
  def dynamic_algorithm(self):
    dynamic_metrics = {}
    for _ in range(100):
      self.adjust_traffic_lights_based_on_conditions()
      self.simulate_traffic_flow()
    return dynamic_metrics
  def fixed_time_system(self):
    fixed_time_metrics = {}
    for _ in range(100):
      self.follow_predefined_traffic_light_timings()
      self.simulate_traffic_flow()
```

```
        return fixed_time_metrics
    def simulate_traffic_flow(self):
        for intersection in self.intersections:
            pass
    def adjust_traffic_lights_based_on_conditions(self):
        for intersection in self.intersections:
            pass
    def follow_predefined_traffic_light_timings(self):
        for intersection in self.intersections:
            pass
    def compare_performance(self):
        dynamic_metrics = self.dynamic_algorithm()
        fixed_time_metrics = self.fixed_time_system()
        print("Dynamic Algorithm Metrics:", dynamic_metrics)
        print("Fixed-Time System Metrics:", fixed_time_metrics)
if __name__ == "__main__":
    roads = [Road("Main Street", 1000, 2000), Road("Broadway",
1500, 1800)]
    intersections = [Intersection("Intersection 1"),
Intersection("Intersection 2")]
    city_network = TrafficNetwork(roads, intersections)
    city_network.compare_performance()
```

Output:

```
C:\Users\srika\Desktop\CSA0863\pythonProject\.venv\Scripts\python.exe "C:\Users\srika\Desktop\CSA0863\pythonProject\DAA\practice 4.py"
Dynamic Algorithm Metrics: {}
Fixed-Time System Metrics: {}

Process finished with exit code 0
```

Analysis:

Time Complexity: Depends on the complexity of traffic simulation and traffic light control algorithms. Typically, for each time step, both systems may have a complexity of O(N * M) where N is the number of intersections and M is the number of roads.

Space Complexity: Mainly involves storing the traffic network graph and additional data structures for simulation and optimization algorithms.

Result:

Program executed successfully.