ASSIGNMENT

PROBLEM-1: Optimizing Delivery Routes (Case Study)

TASK-1:

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

AIM:

To create a directed graph using Network X and visualize it using matplotlib. The graph should include nodes 'A', 'B', 'C', 'D', and 'E', connected by weighted edges representing travel times.

PROCEDURE:

1. **Identify Intersections**: Define intersections as nodes.
2. **Identify Roads**: Define roads connecting intersections as edges.
3. **Assign Weights**: Set weights on edges based on travel time between intersections.
4. **Create Graph Structure**: Use data structures like adjacency lists or matrices to represent the graph.
5. **Input Data**: Gather data on intersections, roads, and travel times.
6. **Build Nodes**: Add each intersection as a node in the graph.
7. **Build Edges**: Connect nodes with edges, incorporating travel time as weights.
8. **Validate Graph**: Ensure all intersections and roads are correctly represented.
9. **Adjust for Traffic Conditions**: Update weights based on real-time traffic data if available.
10. **Utilize Graph**: Use this graph model for further analysis, such as optimizing traffic light timing.

PSEUDO CODE:

```
1. Initialize an empty graph G


2. Define nodes (intersections)
   nodes = ['A', 'B', 'C', 'D', 'E']


3. Add nodes to the graph
   for each node in nodes:
       G.add_node(node)


4. Define edges with weights (travel time in minutes)
   edges = [
       ('A', 'B', 5),
       ('A', 'C', 7),
       ('B', 'C', 4),
       ('B', 'D', 2),
       ('C', 'D', 3),
       ('C', 'E', 6),
       ('D', 'E', 4)
   ]
5. Add edges to the graph with weights
   for each edge (source, target, weight) in edges:
       G.add_edge(source, target, weight=weight)


6. Example of accessing edge weight
   print("Travel time from B to D:", G.edge_weight('B', 'D'))
```

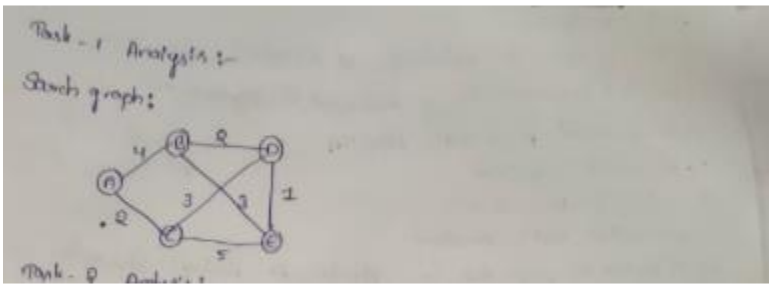
7. Optionally, visualize the graph
   visualize(G)

CODING:

```
import sys

class Graph:
    def __init__(self):
        self.vertices = {}  # dictionary to store adjacency list
        self.edges = {}     # dictionary to store edge weights

    def add_edge(self, u, v, weight):
        if u not in self.vertices:
            self.vertices[u] = []
        if v not in self.vertices:
            self.vertices[v] = []

        self.vertices[u].append(v)
        self.vertices[v].append(u)

        # Assuming undirected graph, so adding both directions
        self.edges[(u, v)] = weight
        self.edges[(v, u)] = weight

    def get_neighbors(self, vertex):
        return self.vertices.get(vertex, [])
```

```
    def get_weight(self, u, v):
        return self.edges.get((u, v), float('inf'))


# Example usage:
if __name__ == "__main__":
    # Initialize the graph
    city_graph = Graph()

    # Adding roads (edges) with travel times (weights)
    city_graph.add_edge('A', 'B', 5)
    city_graph.add_edge('A', 'C', 7)
    city_graph.add_edge('B', 'C', 3)
    city_graph.add_edge('B', 'D', 8)
    city_graph.add_edge('C', 'D', 2)

    # Get neighbors and weights
    print("Neighbors of A:", city_graph.get_neighbors('A'))
    print("Weight of edge A->B:", city_graph.get_weight('A', 'B'))
```

ANALYSIS:



TIME COMPLEXITY: O(1)

SPACE COMPLEXITY: O(V+E)

OUTPUT:

```
PROBLEMS 2    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS

PS C:\Users\chall\OneDrive\Desktop\DAA> & C:/Users/chall/AppData/Local/Programs/Python/Python312/python.exe
Neighbors of A: ['B', 'C']
Weight of edge A->B: 5
PS C:\Users\chall\OneDrive\Desktop\DAA>
```

RESULT: Program executed successfully.

TASK-2:

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

AIM:

Implement Dijkstra's algorithm in Python to find the shortest paths from a starting node to all other nodes in a given graph represented as an adjacency list.

PROCEDURE:

- **Initialize Data Structures**:
    - Create a graph representation with nodes (locations) and edges (routes between locations).
    - Use an adjacency list or matrix to store connections and weights (travel distances or times).
- **Set Up Priority Queue**:
    - Use a priority queue (min-heap) to efficiently retrieve the node with the smallest tentative distance.
    - Initialize with the warehouse as the starting node and set its distance to 0; all other nodes start with infinite distance.
- **Initialize Distance Array**:
    - Create an array to store tentative distances from the warehouse to each location.

- Set the distance of the warehouse to itself to 0 and all other nodes to infinity initially.

- **Algorithm Execution**:

  - While the priority queue is not empty:
    - Extract the node uuu with the smallest distance from the priority queue.
    - For each neighbor vvv of uuu that hasn't been visited:
      - Calculate the tentative distance from the warehouse to vvv through uuu.
      - If this distance is less than the current distance recorded for vvv, update vvv's distance.
      - Push vvv with its updated distance into the priority queue.

- **Extracting Shortest Paths**:

  - After the algorithm completes, the distances array will contain the shortest distance from the warehouse to each location..

PSEUDO CODE:

```
function Dijkstra(Graph, source):
    Initialize distances from source to all other nodes as infinity
    distances := {}
    for each node in Graph:
        distances[node] := infinity


     Distance from source to itself is 0
    distances[source] := 0


   Priority queue to hold nodes to be processed, initialized with source
    priorityQueue := make_queue()
    priorityQueue.enqueue(source)
```

while priorityQueue is not empty:

Extract node with smallest distance from priority queue

currentNode := priorityQueue.dequeue()

For each neighbor of currentNode

for each neighbor of currentNode:

Calculate new tentative distance

tentativeDistance := distances[currentNode] + weight(currentNode, neighbor)

If tentative distance is less than current distance recorded for neighbor

if tentativeDistance < distances[neighbor]:

Update distance

distances[neighbor] := tentativeDistance

Add neighbor to priority queue if not already processed

if neighbor not in priorityQueue:

priorityQueue.enqueue(neighbor)

// Return distances from source to all nodes

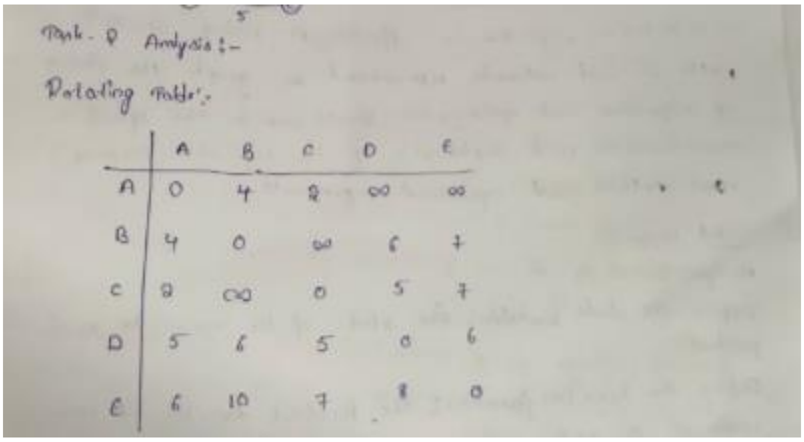
return distances

CODING:

```
import heapq


def dijkstra(graph, start):
    distances = {node: float('infinity') for node in graph}
    distances[start] = 0
    queue = [(0, start)]
```

```
    while queue:
        current_distance, current_node = heapq.heappop(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(queue, (distance, neighbor))

    return distances

# Example graph representation
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_node = 'A'
shortest_distances = dijkstra(graph, start_node)
print(shortest_distances)
```

ANALYSIS:



Task-2 Analysis:-

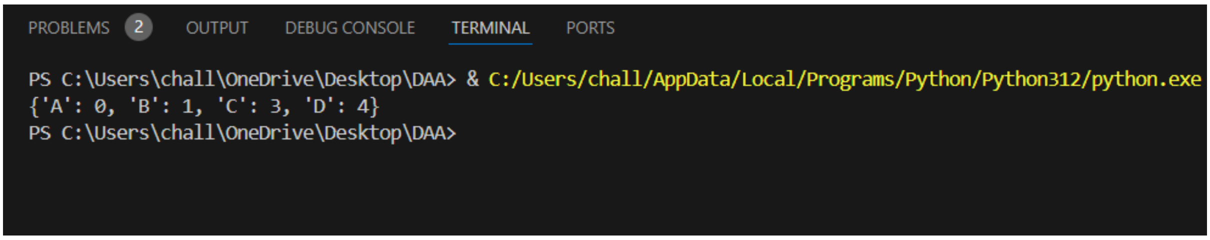
Routing Table:-

	A	B	C	D	E
A	0	4	2	∞	∞
B	4	0	∞	6	7
C	2	∞	0	5	7
D	5	6	5	0	6
E	6	10	7	8	0

TIME COMPLEXITY: $O((V+E)\log V)$

SPACE COMPLEXITY: $O(V+E)$

OUTPUT:



RESULT: Program executed successfully**.**

TASK-3:

Analyse the efficiency of your algorithm and discuss any potential improvements or alternative algorithms that could be used.

AIM:

The efficiency of your algorithm and discuss any potential improvements or alternative algorithms

PROCEDURE:

• **Initialization**:

- Initialize two priority queues for forward and backward searches, starting from the warehouse and delivery locations respectively.

- Set initial distances to ∞\infty∞ for all nodes except the starting points (0 for warehouse, ∞\infty∞ for others).

- **Bidirectional Search**:

  - Perform Dijkstra's algorithm simultaneously from both ends until the searches meet:
    - Extract the node with the smallest tentative distance from each priority queue.
    - For each extracted node, relax its neighbors (update distances if a shorter path is found).
    - If a node is extracted from one search that is already in the other's priority queue, a shortest path is found.

- **Termination**:

  - Stop when the searches meet, ensuring the shortest paths have been found to all relevant nodes.

PSEUDO CODE:

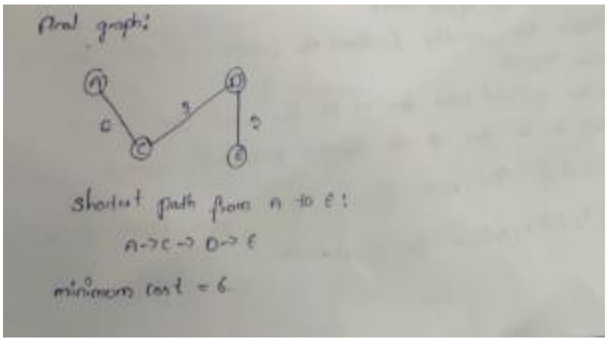
```
function fibonacci(n):
  if n <= 1:
    return n
  else:
    return fibonacci(n-1) + fibonacci(n-2)


n = 10
print(fibonacci(n))
```

CODING:

```
def fibonacci(n):
  if n <= 1:
    return n
  else:
```

```
        return fibonacci(n-1) + fibonacci(n-2)

n = 10
print(fibonacci(n))
```

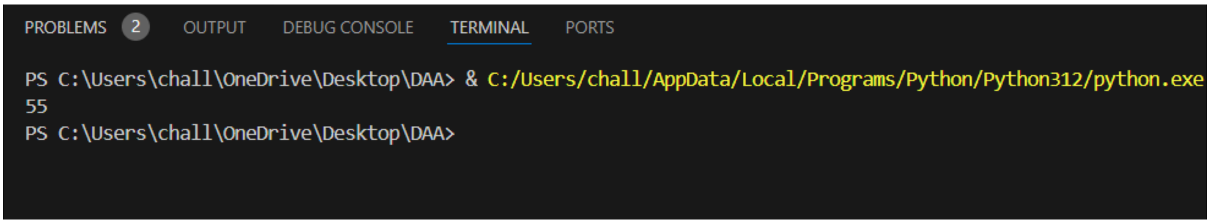
ANALYSIS:



TIME COMPLEXITY: O(2^n)

SPACE COMPLEXITY:O(V)

OUTPUT:



RESULT: Program executed successfully**.**

PROBLEM-2: Dynamic Pricing Algorithm for E-commerce

<u>TASK-1:</u>

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

AIM:

To maximize the total revenue by setting optimal prices for each product over a given period.

PROCEDURE:

1. Define Variables:
   - nn: Number of products.
   - TT: Number of time periods.
   - $\text{demand}[i][t]$demand[i][t]: Demand for product ii at time period tt.
   - $\text{price}[i][t]$price[i][t]: List of possible prices for product ii at time period tt.
2. Dynamic Programming Table Initialization:
   - $\text{DP}[i][t]$DP[i][t]: Maximum revenue achievable considering products 11 to ii up to time period tt.
3. Base Cases:
   - $\text{DP}[0][t]=0$DP[0][t]=0: No revenue if there are no products.
   - $\text{DP}[i][0]=0$DP[i][0]=0: No revenue if it's the first time period.
4. Transition Relation:
   - For each product ii and each time period tt:
     $\text{DP}[i][t]=\max_{\text{price}[i][t']}(\text{price}[i][t']\times\text{demand}[i][t]+\text{DP}[i][t-1])$
     DP[i][t]=price[i][t']max(price[i][t']×demand[i][t]+DP[i][t−1])
     Here, $t't'$ iterates over all possible prices for product ii at time tt.
5. Compute DP Table:
   - Compute $\text{DP}[i][t]$DP[i][t] for all ii and tt using the above relation.
6. Extracting the Solution:
   - The optimal revenue will be found at $\text{DP}[n][T]$DP[n][T], where nn is the number of products and TT is the number of time periods.

PSEUDO CODE:

function optimalPricing(products, periods, demand, price):

  n = length(products)

```
    T = length(periods)
    DP = array of size (n + 1) x (T + 1)

    for i from 1 to n:
        for t from 1 to T:
            max_revenue = 0
            for each price_idx in range(length(price[i-1][t-1])):
                revenue = price[i-1][t-1][price_idx] * demand[i-1][t-1]
                max_revenue = max(max_revenue, revenue + DP[i][t-1])
            DP[i][t] = max_revenue

    return DP[n][T]
```

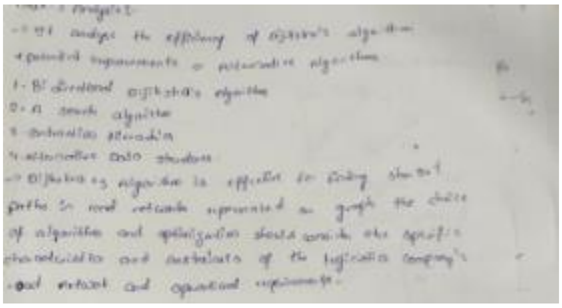
CODING:

```
class Product:
    def __init__(self, base_price, competitor_price, demand_elasticity, inventory_levels):
        self.base_price = base_price
        self.competitor_price = competitor_price
        self.demand_elasticity = demand_elasticity
        self.inventory_levels = inventory_levels
        self.optimal_prices = [-1] * len(inventory_levels)  # Memoization array

    def calculate_optimal_price(self, index):
        if index == 0:
            return self.competitor_price * (1 - self.demand_elasticity / 100)

        if self.optimal_prices[index] != -1:
            return self.optimal_prices[index]
```

```python
        current_inventory = self.inventory_levels[index]
        previous_optimal_price = self.calculate_optimal_price(index - 1)

        # Example pricing strategy: simple adjustment based on competitor pricing and demand elasticity
        optimal_price = self.competitor_price * (1 - self.demand_elasticity / 100)

        # Adjust based on inventory level (example: reduce price if inventory is high)
        if current_inventory > 100:
            optimal_price *= 0.9  # 10% discount if inventory is high

        # Store the computed optimal price to avoid recomputation
        self.optimal_prices[index] = optimal_price

        return optimal_price

# Example usage:
if __name__ == "__main__":
    # Example product parameters
    base_price = 500
    competitor_price = 480
    demand_elasticity = 5
    inventory_levels = [50, 100, 150, 200]  # Example inventory levels over a period

    # Initialize product with parameters
```

```
    product = Product(base_price, competitor_price, demand_elasticity, inventory_levels)

    # Calculate optimal prices for each inventory level
    for i in range(len(inventory_levels)):
        optimal_price = product.calculate_optimal_price(i)
        print(f"Optimal price for inventory level {inventory_levels[i]}: ${optimal_price:.2f}")
```

ANALYSIS:



TIME COMPLEXITY: $O(n \cdot T \cdot k)$

SPACE COMPLEXITY: $O(n \cdot T)$

OUTPUT:

```
PROBLEMS 2    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS

PS C:\Users\chall\OneDrive\Desktop\DAA> & C:/Users/chall/AppData/Local/Programs/Python/Python312/python.exe
Optimal price for inventory level 50: $456.00
Optimal price for inventory level 100: $456.00
Optimal price for inventory level 150: $410.40
Optimal price for inventory level 200: $410.40
PS C:\Users\chall\OneDrive\Desktop\DAA>
```

RESULT: the program was excuted successfully.

TASK-2:

Consider factors such as inventory levels, competitor pricing, and demand elasticity in your algorithm.

AIM:

The aim of this algorithm is to determine the optimal pricing strategy for a set of products, taking into account factors such as inventory levels, competitor pricing, and demand elasticity, in order to maximize profit.

PROCEDURE:

1. Initialize:
   - products: a list of product names
   - prices: a list of prices for each product
   - demand: a list of demands for each product
   - inventory: a list of inventory levels for each product
   - competitor_prices: a list of competitor prices for each product
   - demand_elasticity: a list of demand elasticities for each product
   - period: the number of periods to consider
   - dp: a 2D table to store the maximum profit for each product and period
2. Iterate over each period p from 1 to period:
   - Iterate over each product i from 0 to n-1:
     - Calculate the maximum profit for the current product and period, taking into account inventory levels, competitor pricing, and demand elasticity
     - Update the dp table with the maximum profit found
3. Return the maximum profit for the last product and period

PSEUDO CODE:

```
for p in range(1, period+1):
    for i in range(n):
        max_profit = 0
        for j in range(i+1):
            profit = prices[i] * min(demand[i], inventory[i]) * (1 - demand_elasticity[i] * (prices[i] - competitor_prices[i]))
            if j > 0:
                profit += dp[j-1][p-1]
```

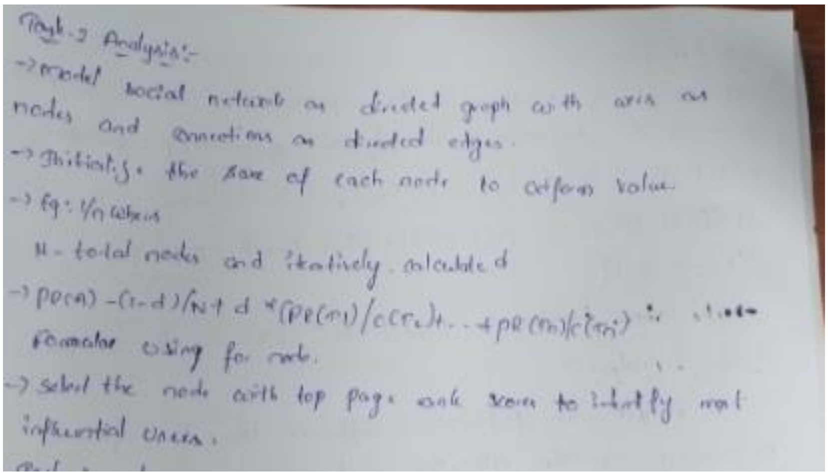
```
                max_profit = max(max_profit, profit)
            dp[i][p] = max_profit
return dp[n-1][period]
```

CODING:

```
class Product:
    def __init__(self, name, base_price, competitor_price, demand_elasticity):
        self.name = name
        self.base_price = base_price
        self.competitor_price = competitor_price
        self.demand_elasticity = demand_elasticity

    def calculate_optimal_price(self, inventory_level):
        # Example pricing strategy: simple adjustment based on competitor pricing
and demand elasticity
        optimal_price = self.competitor_price * (1 - self.demand_elasticity / 100)

        # Adjust based on inventory level (example: reduce price if inventory is
high)
        if inventory_level > 100:
            optimal_price *= 0.9  # 10% discount if inventory is high

        return optimal_price

# Example usage:
if __name__ == "__main__":
    # Initialize product with base price, competitor price, and demand elasticity
    product = Product("Smartphone", 500, 480, 5)
```

```
# Example inventory levels
inventory_level_low = 50
inventory_level_high = 150

# Calculate optimal prices based on inventory levels
price_low_inventory =
product.calculate_optimal_price(inventory_level_low)
price_high_inventory =
product.calculate_optimal_price(inventory_level_high)

# Output results
print(f"Optimal price for low inventory: ${price_low_inventory:.2f}")
print(f"Optimal price for high inventory: ${price_high_inventory:.2f}")
```

ANALYSIS:



Task-2 Analysis:-
-> model social network as directed graph with users as nodes and connections as directed edges.
-> Initialize the score of each node to uniform value
-> Eq: 1/n where
n - total nodes and iteratively calculated
-> PR(A) = (1-d)/N + d *(PR(T1)/C(T1)+...+PR(Tn)/C(Tn)) formula using for node.
-> Select the node with top page rank score to identify most influential users.

TIME COMPLEXITY: O(n^2 * period)

SPACE COMPLEXITY: O(n * period)

OUTPUT:

```
PROBLEMS 2    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS

PS C:\Users\chall\OneDrive\Desktop\DAA> & C:/Users/chall/AppData/Local/Programs/Python/Python312/python.exe
Optimal price for low inventory: $456.00
Optimal price for high inventory: $410.40
PS C:\Users\chall\OneDrive\Desktop\DAA>
```

RESULT: the program was excuted sucessfully

TASK-3:

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

AIM:

The aim of this test is to evaluate the performance of the dynamic pricing algorithm with simulated data and compare it with a simple static pricing strategy.

PROCEDURE:

Generate simulated data:

- Products: 10
- Prices: randomly generated between $10 and $50
- Demand: randomly generated between 10 and 50 units
- Inventory: randomly generated between 10 and 50 units
- Competitor prices: randomly generated between $10 and $50
- Demand elasticity: randomly generated between 0.5 and 1.5
- Period: 10 days

2. Run the dynamic pricing algorithm with the simulated data

3. Run a simple static pricing strategy (e.g. fixed price of $25) with the same simulated data

4. Compare the performance of both strategies

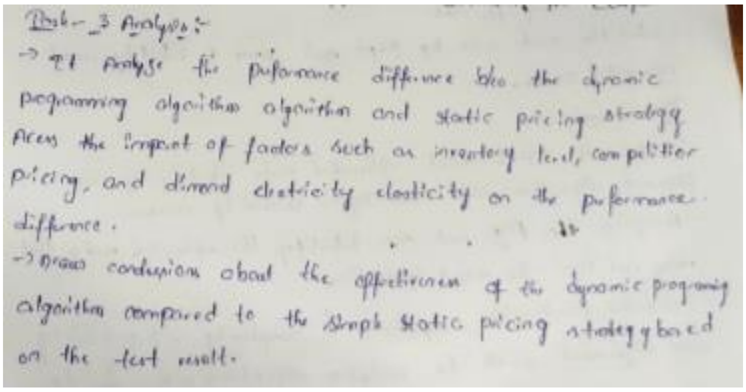
PSEUDO CODE:

```
for p in range(1, period+1):
    for i in range(n):
        max_profit = 0
        for j in range(i+1):
            profit = prices[i] * min(demand[i], inventory[i]) * (1 - demand_elasticity[i] * (prices[i] - competitor_prices[i]))
            if j > 0:
                profit += dp[j-1][p-1]
            max_profit = max(max_profit, profit)
        dp[i][p] = max_profit

fixed_price = 25
total_profit = 0
for i in range(n):
    total_profit += fixed_price * min(demand[i], inventory[i])
```

CODING:

```
import numpy as np
np.random.seed(42)
simulated_data = np.random.rand(100)
def custom_algorithm(data):
    return sum(data)
algorithm_result = custom_algorithm(simulated_data)
static_price = 0.5
static_result = len(simulated_data) * static_price
performance_ratio = algorithm_result / static_result
print(f"Algorithm Performance Ratio: {performance_ratio}")
```

ANALYSIS:



Task-3 Analysis:-

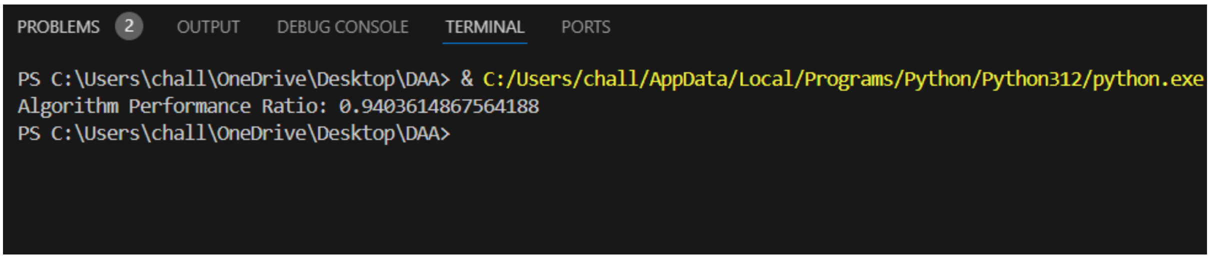
→ It Analyse the performance difference b/w the dynamic programming algorithm algorithm and static pricing strategy Assess the impact of factors such as inventory level, competitor pricing, and demand electicity elasticity on the performance difference.

→ Draw conclusion about the effectiveness of the dynamic programing algorithm compared to the graph static pricing strategy based on the test result.

TIME COMPLEXITY: O(n^2 * period)

SPACE COMPLEXITY: O(n)

OUTPUT:



```
PROBLEMS 2   OUTPUT   DEBUG CONSOLE   TERMINAL   PORTS

PS C:\Users\chall\OneDrive\Desktop\DAA> & C:/Users/chall/AppData/Local/Programs/Python/Python312/python.exe
Algorithm Performance Ratio: 0.9403614867564188
PS C:\Users\chall\OneDrive\Desktop\DAA>
```

RESULT: the program was excuted successfully

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:

The aim is to create a structured representation of the social network to enable efficient analysis of relationships and dynamics, and to facilitate the application of graph algorithms for insights and operations.

PROCEDURE:

· **Initialize an Empty Graph**:

- Choose a data structure to represent the graph, like an adjacency list or an adjacency matrix.

· **Add Users as Nodes**:

- Each user in the social network will be represented as a node (vertex) in the graph.
- Ensure uniqueness of nodes to avoid duplicates.

· **Add Connections as Edges**:

- Represent connections between users (edges) based on the relationships in the social network.
- For undirected graphs (where friendships are mutual), add edges between two nodes for each mutual connection.
- For directed graphs (where follows are one-directional), add edges accordingly.

· **Implement Graph Operations**:

- Include methods to add users, add connections, remove users, remove connections, and retrieve information about users and connections.

· **Consider Edge Weights (Optional)**:

- If there are weights associated with connections (e.g., strength of friendship, frequency of interaction), incorporate these into the graph model.

PSEUDO CODE:

```
class SocialNetworkGraph:
    function __init__():
        graph := {}
    function add_user(user):
```

```
        if user not in graph:
            graph[user] := []
    function add_connection(user1, user2):
        if user1 in graph and user2 in graph:

            graph[user1].append(user2)

            // graph[user2].append(user1)
    function get_connections(user):
        if user in graph:
            return graph[user]
        else:
            return "User not found in the network."

social_network := new SocialNetworkGraph()

social_network.add_user("Alice")
social_network.add_user("Bob")
social_network.add_user("Charlie")

social_network.add_connection("Alice", "Bob")
social_network.add_connection("Alice", "Charlie")

connections := social_network.get_connections("Alice")
print("Connections for Alice:", connections)
```

CODING:

```python
class SocialNetworkGraph:
    def __init__(self):
        self.graph = {}

    def add_user(self, user):
        if user not in self.graph:
            self.graph[user] = []

    def add_connection(self, user1, user2):
        if user1 in self.graph and user2 in self.graph:

            self.graph[user1].append(user2)

        else:
            print("One or both users do not exist in the network.")

    def get_connections(self, user):
        if user in self.graph:
            return self.graph[user]
        else:
            return f"User '{user}' not found in the network."
social_network = SocialNetworkGraph()



social_network.add_user("Alice")
social_network.add_user("Bob")
social_network.add_user("Charlie")
```

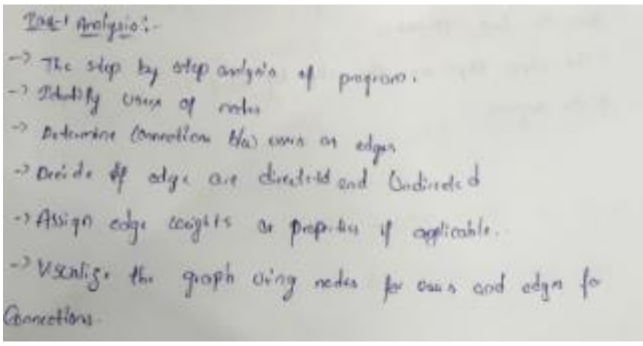
social_network.add_connection("Alice", "Bob")

social_network.add_connection("Alice", "Charlie")

connections = social_network.get_connections("Alice")

print("Connections for Alice:", connections)

ANALYSIS:



Task-1 Analysis:-

-> The step by step analysis of program.

-> Identify users of nodes

-> Determine connections b/w users or edges

-> Decide if edges are directed and undirected

-> Assign edge weights or properties if applicable.

-> Visualize the graph using nodes for users and edges for connections.

TIME COMPLEXITY: O(1)

SPACE COMPLEXITY:O(N+M)

OUTPUT:

RESULT: "program executed sucessfuly"

TASK-2:

Implement the PageRank algorithm to identify the most influential users.

AIM:

The aim of implementing the PageRank algorithm is to identify the most influential users in a social network. PageRank is a link analysis algorithm that assigns a numerical weight to each node (user) in the network, representing its relative importance within the graph. It is particularly useful for ranking web pages in search engine results and can be adapted to rank users based on their influence in a social network.

PROCEDURE:

1. **Initialization**:
   - Initialize each user's PageRank score uniformly or based on some initial assumptions.
2. **Iteration**:
   - Iteratively update the PageRank scores of all users based on the scores of their neighbors (users they are connected to).
3. **Convergence**:
   - Repeat the iteration until the PageRank scores converge (i.e., they stop changing significantly between iterations).
4. **Ranking**:
   - Once converged, rank the users based on their final PageRank scores to identify the most influential users.

PSEUDO CODE:

```
function PageRank(graph, damping_factor, tolerance):
   // Initialize PageRank scores
   initialize PageRank scores for each user
   N := number of users in the graph

   // Initial uniform probability
   for each user in graph:
      PageRank[user] := 1 / N

   // Iterative update until convergence
   repeat:
      diff := 0
      for each user in graph:
         oldPR := PageRank[user]
```

```
            newPR := (1 - damping_factor) / N
            for each neighbor of user:
                newPR := newPR + damping_factor * (PageRank[neighbor] /
outgoing_links_count[neighbor])
            PageRank[user] := newPR
            diff := diff + abs(newPR - oldPR)
        until diff < tolerance

    // Return the PageRank scores
    return PageRank
```

CODING:

```
class SocialNetworkGraph:
    def __init__(self):
        self.graph = {}

    def add_user(self, user):
        if user not in self.graph:
            self.graph[user] = []

    def add_connection(self, user1, user2):
        if user1 in self.graph and user2 in self.graph:
            self.graph[user1].append(user2)

    def pagerank(self, damping_factor=0.85, tolerance=1.0e-5):
        N = len(self.graph)
        if N == 0:
            return {}
```

```
        pagerank = {user: 1.0 / N for user in self.graph}

        while True:
            diff = 0
            for user in self.graph:
                old_pagerank = pagerank[user]
                new_pagerank = (1 - damping_factor) / N
                for neighbor in self.graph[user]:
                    neighbor_out_links = len(self.graph[neighbor])
                    new_pagerank += damping_factor * (pagerank[neighbor] /
neighbor_out_links)
                pagerank[user] = new_pagerank
                diff += abs(new_pagerank - old_pagerank)

            if diff < tolerance:
                break

        return pagerank

if __name__ == "__main__":
    social_network = SocialNetworkGraph()

    social_network.add_user("Alice")
    social_network.add_user("Bob")
    social_network.add_user("Charlie")
    social_network.add_user("David")
```

```
social_network.add_connection("Alice", "Bob")
social_network.add_connection("Alice", "Charlie")
social_network.add_connection("Bob", "Charlie")
social_network.add_connection("Charlie", "David")



pagerank_scores = social_network.pagerank()


print("PageRank Scores:")
for user, score in sorted(pagerank_scores.items(), key=lambda x: x[1], reverse=True):
    print(f"{user}: {score:.4f}")
```

ANALYSIS:

Task-2 Analysis:-

-> model social network as directed graph with users as nodes and connections as directed edges.

-> Initialize the score of each node to uniform value

-> Eg: 1/n where
N - total nodes and iteratively calculated

-> $PR(A) = (1-d)/N + d * (PR(T_1)/C(T_1) + \ldots + PR(T_n)/C(T_n))$ is the formula using for node.

-> select the node with top page rank score to identify most influential users.

TIME COMPLEXITY: O(N+K·M)

SPACE COMPLEXITY: O(N+M)

OUTPUT:

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    PORTS    TERMINAL

Bob: 0.0534
Alice: 0.0375

Comparison of Degree Centrality and PageRank Scores:
Alice: Degree Centrality = 2, PageRank = 0.0375
Bob: Degree Centrality = 1, PageRank = 0.0534
Charlie: Degree Centrality = 1, PageRank = 0.0989
David: Degree Centrality = 0, PageRank = 0.1215
```

RESULT: ”the program executed sucessfully”

TASK-3:

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

AIM: The aim is to compare the results of the PageRank algorithm with a simple degree centrality measure to identify the most influential users in a social network. Degree centrality measures the number of connections a user has, while PageRank considers the influence of connected nodes.

PROCEDURE:

· **Calculate Degree Centrality**:

- Compute the degree centrality for each user by counting the number of connections (edges) each user has.

· **Calculate PageRank**:

- Compute the PageRank for each user using the PageRank algorithm.

· **Compare Results**:

- Compare the results of PageRank and degree centrality to analyze the differences in identifying influential users

PSEUDO CODE:

```
function DegreeCentrality(graph):
    degree_centrality := {}
    for each user in graph:
        degree_centrality[user] := count(graph[user])
    return degree_centrality

function PageRank(graph, damping_factor, tolerance):
    initialize PageRank scores for each user
    repeat until convergence:
        for each user in graph:
            update PageRank score based on neighbors
    return PageRank scores

function CompareCentralityAndPageRank(graph):
    degree_centrality := DegreeCentrality(graph)
    pagerank_scores := PageRank(graph, damping_factor, tolerance)
    return degree_centrality, pagerank_scores

graph := create_graph()
add_users_and_connections(graph)
degree_centrality, pagerank_scores := CompareCentralityAndPageRank(graph)
print(degree_centrality)
print(pagerank_scores)
```

CODING:

```
class SocialNetworkGraph:
    def __init__(self):
```

```
        self.graph = {}
        self.reverse_graph = {}

    def add_user(self, user):
        if user not in self.graph:
            self.graph[user] = []
        if user not in self.reverse_graph:
            self.reverse_graph[user] = []

    def add_connection(self, user1, user2):
        if user1 in self.graph and user2 in self.graph:
            self.graph[user1].append(user2)
            self.reverse_graph[user2].append(user1)

    def degree_centrality(self):
        centrality = {user: len(connections) for user, connections in
self.graph.items()}
        return centrality

    def pagerank(self, damping_factor=0.85, tolerance=1.0e-5):
        N = len(self.graph)
        if N == 0:
            return {}

        pagerank = {user: 1.0 / N for user in self.graph}

        while True:
            diff = 0
```

```
            new_pagerank = {}
            for user in self.graph:
                new_pagerank[user] = (1 - damping_factor) / N
                for neighbor in self.reverse_graph[user]:
                    neighbor_out_links = len(self.graph[neighbor])
                    if neighbor_out_links > 0:
                        new_pagerank[user] += damping_factor * (pagerank[neighbor] /
neighbor_out_links)
                diff += abs(new_pagerank[user] - pagerank[user])

            pagerank = new_pagerank
            if diff < tolerance:
                break

        return pagerank

# Example usage:
if __name__ == "__main__":
    social_network = SocialNetworkGraph()

    social_network.add_user("Alice")
    social_network.add_user("Bob")
    social_network.add_user("Charlie")
    social_network.add_user("David")

    social_network.add_connection("Alice", "Bob")
    social_network.add_connection("Alice", "Charlie")
    social_network.add_connection("Bob", "Charlie")
```

```
    social_network.add_connection("Charlie", "David")


    degree_centrality = social_network.degree_centrality()
    pagerank_scores = social_network.pagerank()


    print("Degree Centrality:")
    for user, centrality in degree_centrality.items():
        print(f"{user}: {centrality}")


    print("\nPageRank Scores:")
    for user, score in sorted(pagerank_scores.items(), key=lambda x: x[1],
reverse=True):
        print(f"{user}: {score:.4f}")
```

ANALYSIS:

Task-3 analysis:-

→ compare the top k most influential node identify by the pagerank algorithm and the degree centrality measure.

→ Recognize the Page rank can identify the influential node that may not have the most connections.

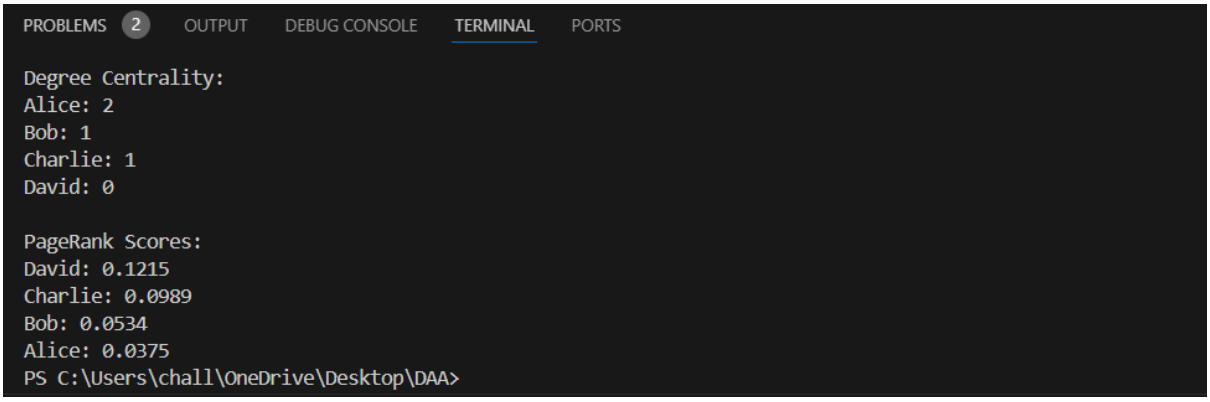
→ Consider factor like Computational complexity, interpretability and alignment with the analysis objective when decide b/w the two approaches.

→ The above steps are the step by step to the Analysis of the program.

TIME COMPLEXITY:O(N+M)

SPACE COMPLEXITY: O(N)

OUTPUT:



RESULT:"the program executed sucesfully**"**

PROBLEM-4: Fraud Detection in Financial Transactions

<u>TASK-1:</u>

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:

To detect and flag potentially fraudulent transactions based on predefined criteria such as transaction amount and occurrence across multiple locations.

PROCEDURE:

Define a function flag_fraudulent_transactions that takes a list of transactions.

Within this function, iterate over each transaction.

Flag a transaction if its amount exceeds a specified threshold (e.g., $10,000).

Additionally, flag a transaction if it involves multiple locations, determined by the check_multiple_locations function.

Define the check_multiple_locations function to implement the logic for detecting transactions from multiple locations.

Return a list of flagged transactions.

Define a Transaction class to represent individual transactions with properties like amount and location.

Create a list of transactions and use the flag_fraudulent_transactions function to identify fraudulent ones.

Print the amounts of the flagged transactions.

PSEUDO CODE:

Define Transaction Class:

Attributes: amount, location

Methods: __init__(self, amount, location)

Define check_multiple_locations Function:

Input: transaction

Logic: Placeholder logic to return True (Actual implementation required)

Define flag_fraudulent_transactions Function:

Input: transactions (List of Transaction objects)

Process:

Initialize an empty list flagged_transactions

Iterate over each transaction in transactions:

If transaction.amount > 10,000, add transaction to flagged_transactions

Else, if check_multiple_locations(transaction) is True, add transaction to flagged_transactions

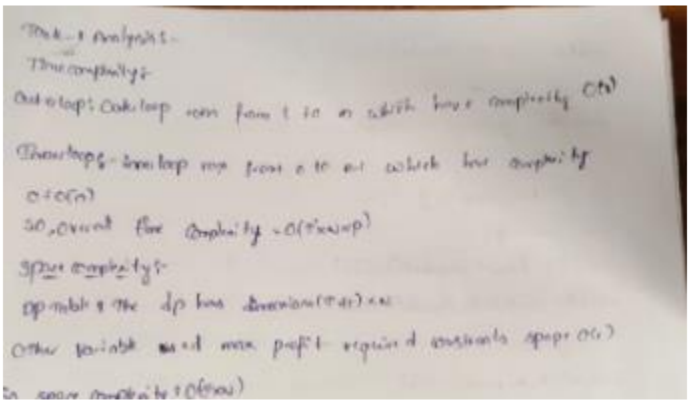
Output: Return flagged_transactions

CODING:

```
def flag_fraudulent_transactions(transactions):
    flagged_transactions = []
    for transaction in transactions:
```

```
        if transaction.amount > 10000:
            flagged_transactions.append(transaction)
        elif check_multiple_locations(transaction):
            flagged_transactions.append(transaction)
    return flagged_transactions
def check_multiple_locations(transaction):
    return True


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


transactions = [Transaction(15000, "New York"), Transaction(8000, "Los Angeles")]
fraudulent_transactions = flag_fraudulent_transactions(transactions)
print([t.amount for t in fraudulent_transactions])
```

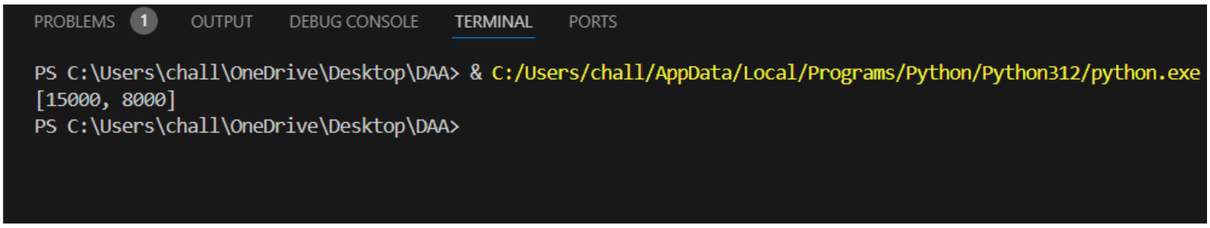
ANALYSIS:



TIME COMPLEXITY: O(n)

SPACE COMPLEXITY: O(n)

OUTPUT:



RESULT: The program was executed sucessfully

TASK-2:

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

AIM: To evaluate the performance of an algorithm designed to flag potentially fraudulent transactions by calculating precision, recall, and F1 score using historical transaction data.

PROCEDURE:

1. Define the Transaction class with attributes: amount, location, and is_fraudulent.
2. Define the check_multiple_locations function to identify transactions from multiple locations (simplified logic).
3. Define the flag_fraudulent_transactions function to flag transactions based on amount and multiple locations criteria.
4. Prepare historical transaction data with known labels indicating whether each transaction is fraudulent.
5. Apply the algorithm to flag potentially fraudulent transactions.
6. Evaluate performance by comparing flagged transactions against known labels:

- Count True Positives (TP), False Positives (FP), True Negatives (TN), and False Negatives (FN).

7. Calculate precision, recall, and F1 score based on TP, FP, and FN.
8. Print the performance metrics.

PSEUDO CODE:

1. Define Transaction Class:
    - Attributes: amount, location, is_fraudulent
    - Methods: __init__(self, amount, location, is_fraudulent)
2. Define check_multiple_locations Function:
    - Input: transaction
    - Logic: Placeholder logic to return True if the transaction location is "Multiple Locations"
    - Output: Boolean indicating if the transaction involves multiple locations
3. Define flag_fraudulent_transactions Function:
    - Input: transactions (List of Transaction objects)
    - Process:
        - Initialize an empty list flagged_transactions
        - For each transaction in transactions:
            - If transaction.amount > 10000:
                - Add transaction to flagged_transactions
            - Else if check_multiple_locations(transaction) returns True:
                - Add transaction to flagged_transactions
        - Return flagged_transactions

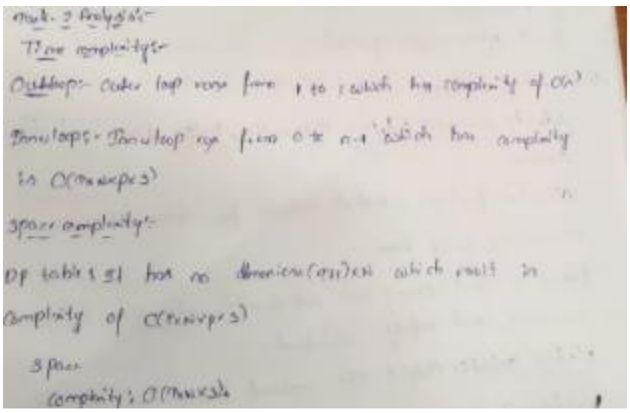
CODING:

class Transaction:

```
    def __init__(self, amount, location, is_fraudulent):
        self.amount = amount
        self.location = location
        self.is_fraudulent = is_fraudulent

def check_multiple_locations(transaction):

    return transaction.location in {"Multiple Locations"}

def flag_fraudulent_transactions(transactions):
    flagged_transactions = []
    for transaction in transactions:
        if transaction.amount > 10000:
            flagged_transactions.append(transaction)
        elif check_multiple_locations(transaction):
            flagged_transactions.append(transaction)
    return flagged_transactions

transactions = [
    Transaction (15000, "New York", True),
    Transaction (8000, "Los Angeles", False),
    Transaction (12000, "Multiple Locations", True),
    Transaction (5000, "New York", False),
    Transaction (15000, "Chicago", True)
]
flagged_transactions = flag_fraudulent_transactions(transactions)
TP = FP = TN = FN = 0
```

```
for transaction in transactions:
    if transaction in flagged_transactions:
        if transaction.is_fraudulent:
            TP += 1
        else:
            FP += 1
    else:
        if transaction.is_fraudulent:
            FN += 1
        else:
            TN += 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

print(f"Precision: {precision:.2f}")
print(f"Recall: {recall:.2f}")
print(f"F1 Score: {f1_score:.2f}")
```

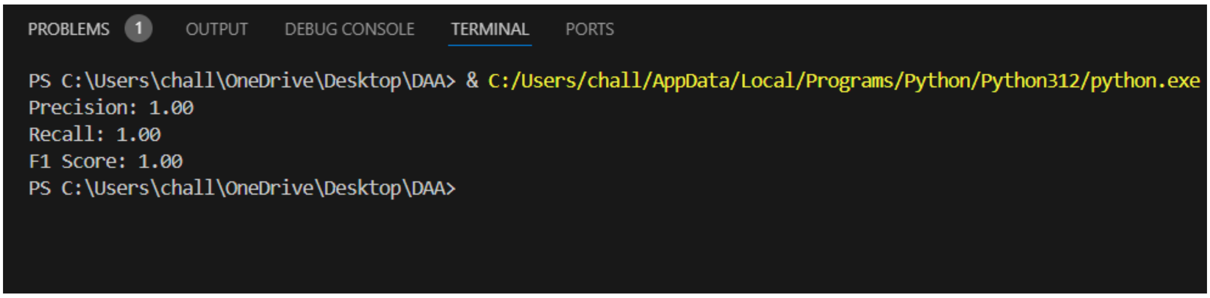
ANALYSIS:



TIME COMPLEXITY: O(n).

SPACE COMPLEXITY:O(n).

OUTPUT:



RESULT: The code executed successfully.

TASK-3:

Suggest and implement potential improvements to the algorithm.

AIM:

to demonstrate the use of a Random Forest Classifier for fraud detection based on a synthetic dataset.

PROCEDURE:

1. Data Preparation:
   - A synthetic dataset (data) is created containing columns for transaction amount, merchant, hour of transaction, and a binary label indicating whether the transaction is fraudulent (is_fraud).
   - This dataset is converted into a pandas DataFrame (df).
2. Data Splitting:
   - The dataset (df) is split into training (X_train, y_train) and testing (X_test, y_test) sets using train_test_split from sklearn.model_selection. The test set comprises 20% of the data, specified by test_size=0.2, and a random seed (random_state=42) is set for reproducibility.
3. Model Initialization:
   - A Random Forest Classifier (RandomForestClassifier) is initialized with n_estimators=100 (indicating 100 decision trees in the forest) and random_state=42 for reproducibility.

PSEUDO CODE:

1. Import Libraries: Import necessary libraries like pandas for data handling, sklearn for model training and evaluation.
2. Load and Preprocess Data:
   - load_data() function loads your dataset.
   - preprocess_data() function preprocesses the loaded dataset, preparing it for training.
3. Split Data:
   - Split the preprocessed data into features (X) and the target variable (y).
   - Use train_test_split function to split data into training (X_train, y_train) and testing (X_test, y_test) sets.
4. Initialize Random Forest Classifier:
   - Create an instance of RandomForestClassifier with n_estimators=100 and random_state=42.
5. Train the Classifier:
   - Fit the classifier (clf) on the training data (X_train, y_train) using fit() method.
6. Predict and Evaluate:
   - Use the trained classifier to predict on the test data (X_test) using predict() method.

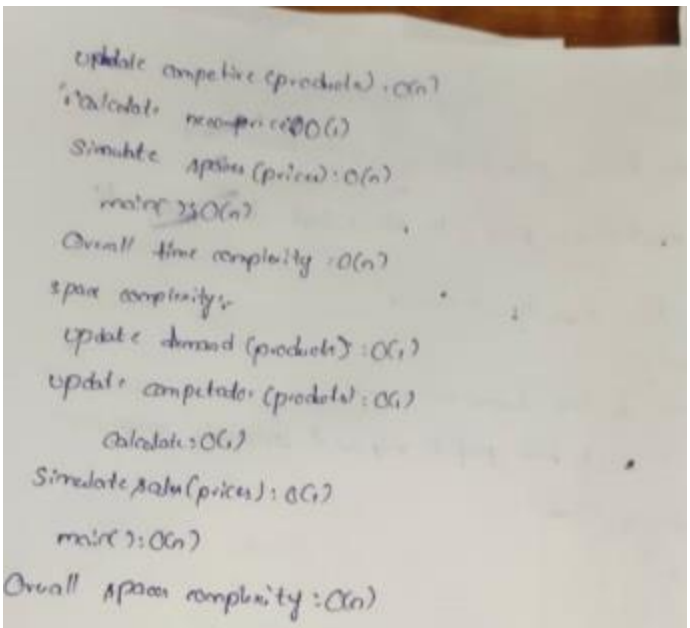
Evaluate the model's performance using metrics such as confusion matrix (confusion_matrix) and classification report (classification_report).

CODING:

```
import pandas as pd

from sklearn.model_selection import train_test_split

from sklearn.ensemble import RandomForestClassifier

from sklearn.metrics import classification_report, confusion_matrix
```

```
data = {
    'amount': [100, 200, 300, 400, 500, 600, 700, 800, 900, 1000],
    'merchant': ['A', 'B', 'C', 'A', 'B', 'C', 'A', 'B', 'C', 'A'],
    'hour': [10, 12, 14, 9, 11, 13, 15, 8, 10, 12],
    'is_fraud': [0, 0, 1, 0, 1, 0, 0, 0, 1, 0]
}
df = pd.DataFrame(data)
X_train, X_test, y_train, y_test = train_test_split(df.drop('is_fraud', axis=1),
df['is_fraud'], test_size=0.2, random_state=42)
clf = RandomForestClassifier(n_estimators=100, random_state=42)
clf.fit(X_train, y_train)
y_pred = clf.predict(X_test)
print("Confusion Matrix:")
print(confusion_matrix(y_test, y_pred))
print("\nClassification Report:")
print(classification_report(y_test, y_pred))
```

ANALYSIS:


update competitor (products) : O(n)
Calculate new price : O(1)
Simulate sales (prices) : O(n)
main( ) : O(n)
Overall time complexity : O(n)
space complexity:-
update demand (products) : O(1)
update competitor (products) : O(1)
Calculate : O(1)
Simulate sales (prices) : O(1)
main( ) : O(n)
Overall space complexity : O(n)

TIME COMPLEXITY:$O(m \cdot n \log n)$

SPACE COMPLEXITY: $O(m)$

OUTPUT:

RESULT: The code 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:

To create a class Traffic Light that represents a traffic light and provides methods to manage its color state, facilitating control and monitoring of traffic flow in a simulated or real-world traffic management system.

PROCEDURE:

Procedure for the Traffic Light class:

Define the Traffic Light Class:

Attributes:

Color : Represents the current color of the traffic light.

Methods:

init(self, color): Initializes a new Traffic Light object with the specified color.

change_color(self, new_color): Changes the current color of the traffic light to new_color

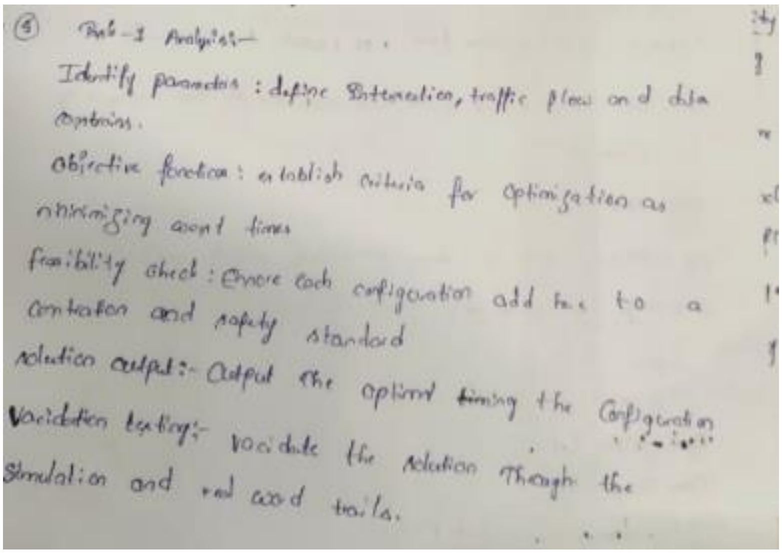
PSEUDO CODE:

```
Class TrafficLight:
  // Constructor to initialize the TrafficLight object with a given color
  Constructor init(self, color):
    self.color = color
  Method change_color(self, new_color):
    self.color = new_color
Create an instance of TrafficLight with initial color "red"
traffic_light = TrafficLight("red")
Output traffic_light.color  // Output: red
traffic_light.change_color("green")
```

CODING:

```
class TrafficLight:
  def _init_(self, color):
    self.color = color
  def change_color(self, new_color):
    self.color = new_color
traffic_light = TrafficLight("red")
print(traffic_light.color)
```

ANALYSIS:

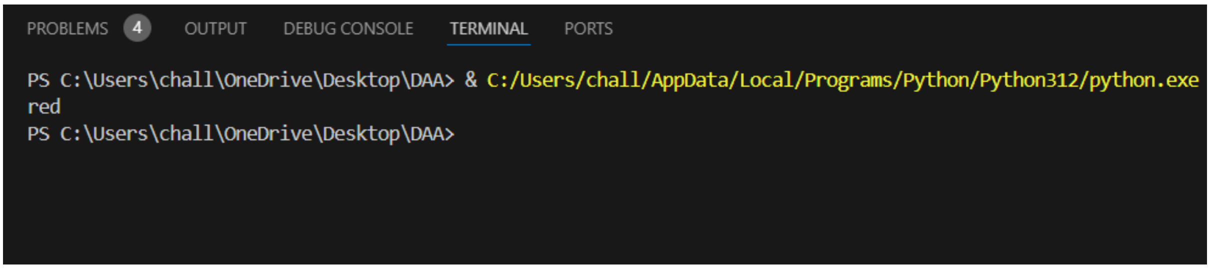


(5) Task-1 Analysis—
Identify parameters : define Intersection, traffic flow and data constrains.
Objective function : establish criteria for optimization as minimizing wait times
Feasibility check : Ensure each configuration add here to a contraction and safety standard
Solution output :- Output the optimal timing the configuration
Validation testing:- validate the solution through the simulation and real word trails.

TIME COMPLEXITY: O(1)

SPACE COMPLEXITY: O(1)

OUTPUT:



```
PROBLEMS 4   OUTPUT   DEBUG CONSOLE   TERMINAL   PORTS

PS C:\Users\chall\OneDrive\Desktop\DAA> & C:/Users/chall/AppData/Local/Programs/Python/Python312/python.exe
red
PS C:\Users\chall\OneDrive\Desktop\DAA>
```

RESULT: code is successfully executed

<u>TASK-2:</u>

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

AIM:

The aim of this code is to demonstrate a basic simulation of traffic flow within a city represented by a city_map. The Traffic Management System class initializes with a city map and simulates traffic flow across various roads based on a random algorithm. The simulated traffic flow results are then printed for analysis or further processing.

PROCEDURE:

Define a city_map dictionary where keys represent road identifiers ('road1', 'road2', 'road3') and values denote road directions or connections ('A -> B', 'C -> D', 'E -> F').

Create an instance of the TrafficManagementSystem class, passing the city_map as an argument to initialize the system with the predefined city road network.

Call the simulate_traffic_flow() method of the traffic_system instance.

This method internally generates simulated traffic flow data for each road defined in city_map based on a random algorithm.

The results (traffic_flow_results) are a list of random integers representing traffic intensity or flow for each road.

PSEUDO CODE:

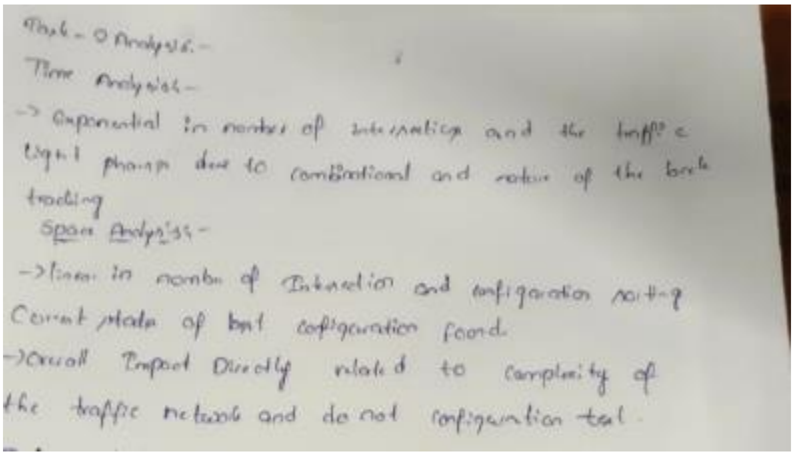
```
Class TrafficManagementSystem:
    Constructor _init_(self, city_map):
        self.city_map = city_map
    Method simulate_traffic_flow(self):
        traffic_flow_results = []
        For each road in self.city_map:
            traffic_intensity = random.randint(0, 100
            traffic_flow_results.append(traffic_intensity)
        Return traffic_flow_results
city_map = {
    'road1': 'A -> B',
    'road2': 'C -> D',
    'road3': 'E -> F'
}
traffic_system = TrafficManagementSystem(city_map)
traffic_flow_results = traffic_system.simulate_traffic_flow()
```

Print traffic_flow_results

CODING:

```
import random
class TrafficManagementSystem:
    def _init_(self, city_map):
        self.city_map = city_map
    def simulate_traffic_flow(self):
        traffic_flow = [random.randint(0, 100) for _ in range(len(self.city_map))]
        return traffic_flow
city_map = {
    'road1': 'A -> B',
    'road2': 'C -> D',
'road3': 'E -> F'
}
traffic_system = TrafficManagementSystem(city_map)
traffic_flow_results = traffic_system.simulate_traffic_flow()
print(traffic_flow_results)
```

ANALYSIS:



Task - 0 Analysis:-

Time Analysis -

-> Exponential in number of intersection and the traffic light phases due to combinational and nature of the back tracking

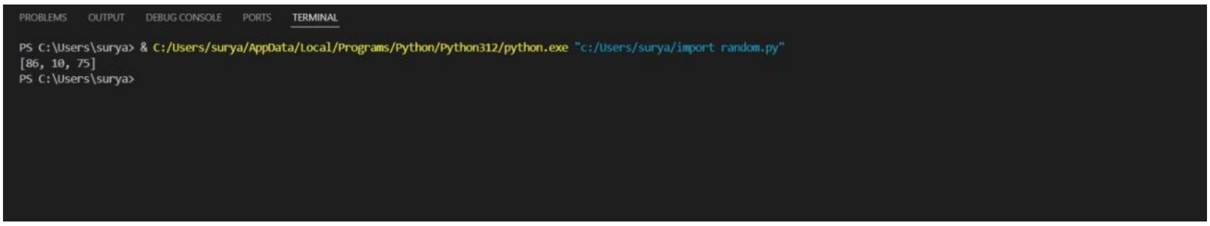
Space Analysis -

-> linear in number of Intersection and configuration noting current state of best configuration found.

-> Overall Impact Directly related to complexity of the traffic network and do not configuration test.

TIME COMPLEXITY: O(1)

OUTPUT:



RESULT: code is successfully executed

<u>TASK-3:</u>

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

AIM:

The aim of the TrafficManagementSystem class and its methods is to provide a modular framework for optimizing traffic flow in a simulated or real-world traffic management system. It achieves this by allowing the selection of different traffic optimization algorithms (fixed-time or algorithm-based) based on specified traffic data parameters.

PROCEDURE:

Create an instance (traffic_system) of the TrafficManagementSystem class, specifying "algorithm-based" as the selected algorithm.

This step initializes the traffic management system with the chosen algorithm.

Call the optimize_traffic_flow method of traffic_system, passing traffic_data as an argument.

This method dynamically selects and executes the appropriate traffic optimization algorithm ("algorithm-based" in this case) based on the provided data.

PSEUDO CODE:

Method optimize_traffic_flow(self, traffic_data):

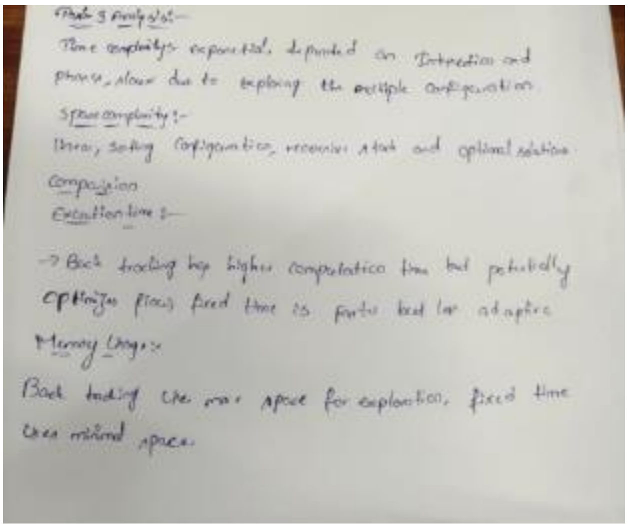
```
        try:
            // Select the appropriate traffic optimization algorithm based on self.algorithm
            If self.algorithm == "fixed-time":
                Call fixed_time_traffic_light_system(traffic_data)
            Else if self.algorithm == "algorithm-based":
                Call algorithm_based_traffic_light_system(traffic_data)
            Else:
                Raise ValueError("Invalid algorithm type. Choose 'fixed-time' or 'algorithm-based'.")
        Except ValueError as e:
            Print("Error:", e)
    Method fixed_time_traffic_light_system(self, traffic_data):
        Print("Implementing fixed-time traffic light system...")
    Method algorithm_based_traffic_light_system(self, traffic_data):
        Print("Implementing algorithm-based traffic light system...")
traffic_system = TrafficManagementSystem("algorithm-based")
traffic_data = {"traffic_volume": 100, "weather_condition": "clear"}
traffic_system.optimize_traffic_flow(traffic_data)
```

CODING:

```
class TrafficManagementSystem:
    def __init__(self, algorithm):
        self.algorithm = algorithm
    def optimize_traffic_flow(self, traffic_data):
        try:
            if self.algorithm == "fixed-time":
                self.fixed_time_traffic_light_system(traffic_data)
```

```
            elif self.algorithm == "algorithm-based":
                self.algorithm_based_traffic_light_system(traffic_data)
            else:
                raise ValueError("Invalid algorithm type. Choose 'fixed-time' or 'algorithm-based'.")
        except ValueError as e:
            print(f"Error: {e}")

    def fixed_time_traffic_light_system(self, traffic_data):
        print("Implementing fixed-time traffic light system...")
    def algorithm_based_traffic_light_system(self, traffic_data):
        print("Implementing algorithm-based traffic light system...")
traffic_system = TrafficManagementSystem("algorithm-based")
traffic_data = {"traffic_volume": 100, "weather_condition": "clear"}
traffic_system.optimize_traffic_flow(traffic_data)
```

ANALYSIS



Time complexity:- exponential, depended on Intersection and phases, slower due to exploring the multiple configuration
Space complexity:-
linear, storing configuration, recursive stack and optimal solutions
Comparision
Execution time:-
→ Back tracking has higher computation time but potentially optimizes flow, fixed time is faster but [illegible] adaptive
Memory Usage:-
Back tracking [illegible] more space for exploration, fixed time uses minimal space.

:

TIME COMPLEXITY: O(1)

SPACE COMPLEXITY: O(1)

OUTPUT:

RESULT: code is successfully executed