1) Scenario: You are working for a logistics company that wants to optimize its delivery routes to minimize the fuel consumption and delivery time. The company operates in a city with a complex road network.

TASKS:

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

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

3) Analyze the efficiency of your algorithm and discuss any potential improvements & alternative algorithm that could be used.

DELIVERABLES:-

* Graph model of the city's road network
* Pseudocode and implementation of Dijikstra's algorithm.
* Analyze of the algorithm's efficiency and potential improvements.

REASONING:-

Explain why Dijikstra's algorithm is suitable for this problem. Discuss any assumptions made & how different road conditions could affect your solution.

Ans: GRAPH MODEL OF THE CITY'S ROAD NETWORK:-

Graph representation:

* NODES:- Represent intersections & key locations
* EDGES:- Represent the edges by the travel time on each road.

Assumptions:

* All the edges weights are non-Negative
* The road network is connected, the path b/w the 2 Nodes
* Traffic conditions, road closures and other dynamic factors are not directly considered in the initial model.

ALGORITHM:

1) Mark the source node with a current distance of 0 and the rest with infinity.

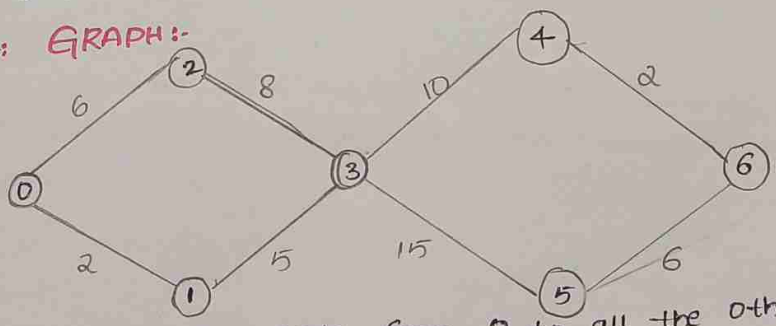
2) Set the Non-visited node with the smallest current distance as the current node.

3) For each neighbour, N of the current Node adds the current distance of adjacent node with the weight 0→1. If it is smaller than the current Node of distance, Set it as the N.

4) Mark the current node 1 as visited.

5) Go to 2nd step if there are any nodes unvisited.

~~CODE~~: GRAPH:-



Let's start the Node from 0 to all the other Nodes.

As, we can see the shortest path from.

0 - 1 = 2

0 - 2 = 6

0 - 3 = 7 ⇒ 5+2.

0 - 4 = 0-1-3-4 ⇒ 2+5+10 = 17,

0 - 6 = 0-2-3-4-6 ⇒ 6+8+10+2 = 26,

Step1:- Start from 0 to visited & check for adjacent nodes.

Unvisited Nodes = {0,1,2,3,4,5,6}

Distance: 0: 0
1: ∞
2: ∞
3: ∞
4: ∞
5: ∞
6: ∞

0 as The unmark.

Step2:- Mark Node1 as visited and add the Distance

Unvisited Nodes = {0, 1, 2, 3, 4, 5, 6}

Node 0 to 1 = 2.

Distance: 0: 0
1: 2
2: ∞
3: ∞
4: ∞
5: ∞
6: ∞

Step3:- Mark Node 3 as visited after considering the Optimal path & add the distance.

Unvisited Nodes = {0, 1, 2, 3, 4, 5, 6}

Distance: 0: 0
1: 2
2: 6
3: 7
4: ∞
5: ∞
6: ∞

Node 0-1-2 ⇒ 2+5 = 7.

Step4:- Again we have 2 choices for adjacent Nodes.

Unvisited Nodes = {0, 1, 2, 3, 4, 5, 6}

Distance: 0: 0
1: 2
2: 6
3: 7
4: 17
5: ∞
6: ∞

Node 0-1-3-4 = 2+5+10 = 17.

Steps:- Again, Move forward and check for the adjacent Node 6.

Unvisited Node = {0, 1, 2, 3, 4, 5, 6}

Distance:

0 : 0
1 : 2
2 : 6
3 : 7
4 : 17
5 : 22
6 : 19 ✓

Node 0-1-3-4-6 ⇒ 2+5+10+2 = 19.

The Shortest path from all the Nodes is 0-1 i.e, 2.

PYTON IMPLEMENTATION:-

```
import heapq
    def dijikstras (graph, start):
        priority_queue = [(0, start)]
    distances = {node: float ('infinity') for node in graph}
    distances [start] = 0
    while priority_queue:
        current_distance, current_node = heapq.heappop (priority_queue)
    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
```

* ANALYSIS OF ALGORITHM EFFICIENCY & POTENTIAL IMPROVEMENTS:-

Efficiency Analysis:-

* TIME COMPLEXITY: The algorithm runs in $O((|V|+|E|)\log|V|)$ time, where $|V|$ is the number of nodes and $|E|$ is the number of edges. This efficiency is due to the priority queue operations.

SPACE COMPLEXITY:- The space complexity is $O(|V|)$ for storing the distances and the priority queue.

2) Scenario: An e-commerce company wants to implement a dynamic pricing algorithm to adjust the prices of the products in the real-time based on demand and competitor prices.

TASKS:

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

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

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

DELIVERABLES:

* Pseudocode and implementation of the dynamic pricing algorithm.
* Simulation results comparing dynamic and static pricing strategies.
* Analysis of the benefits and drawbacks of dynamic pricing.

REASONING:

Justify the use of the dynamic programming to this problem. Explain how you incorporated different factors into your algorithm and discuss any challenges faced during implementation.

Ans: PROBLEM STATEMENT: An e-commerce company wants to adjust the prices of its product in real-time based on factors such as demand, competitor prices. The goal is to maximize over the given period.

DYNAMIC PROGRAMMING JUSTIFICATION:

Dynamic programming is appropriate here because it allows us to break down the problem. It helps in efficiently finding the optimal strategy over time.

FACTORS:-

1) Inventory Levels:- prices are adjusted based on remaining inventory to avoid stockouts & excess stock.

2) Competitor pricing:- prices are aligned with competitor pricing strategies to remain competitive

3) Demand Elasticity:- Adjustments are made considering how sensitive demand on to price changes.

ALGORITHM:-

Dynamic programming is an Algorithmic technique that solves complex problems by breaking them down into smaller subproblems and storing their solutions for future use. It is particularly effective for problems that contains overlapping subproblems & optimal substructure.

* Longest Common Subsequence (LCS): Finds the longest common subsequence blw 2 strings.
* Shortest path in a Graph: Finds the shortest path blw 2 Nodes.
* KNAPSACK PROBLEM: Determines the maximum value of items that can be placed in a knapsack with a given capacity.
* FIBONACCI SEQUENCE: Calculates the nth fibonacci numbers.

TECHNIQUES TO SOLVE DYNAMIC PROGRAMMING PROBLEMS:.

1) Top-Down (memoization):

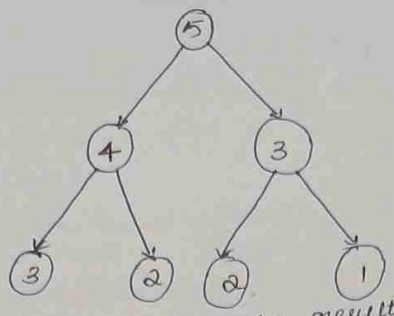
Break down the given problem in order to begin solving it. If you see that the problem has already been solved, return the saved answer. This is usually easy to think of and very intutive. This is referred to memoization.

2) BOTTOM-UP (TABULATION):

Analyze the problem and see in what order the subproblems are solved, and work your way up from the trivial subproblem to the given problem. This is referred to as a DYNAMIC PROGRAMMING.

GRAPH:-

Let's look at the Graph for finding the 5th fibonacci number.



Let us say we know the result to:

State (n=1), State (n=2), State (n=3).... State (n=6)

We can only add 1, 3 and 5 we have 7 steps

1) Adding 1 to all the possible combinations of state (n=6)

Eg: [(1+1+1+1+1+1)+1]

[(1+1+1+3)+1]

[(1+3+1+1)+1]

[(3+1+1+1)+1]

[(3+3)+1]

[(1+5)+1]

[(5+1)+1]

2) Adding 3 to all possible combinations of state (n=4):

[(1+1+1+1)+3]

[(1+3)+3]

[(3+1)+3]

3) Adding 5 to all possible combinations of state (n=2):

[(1+1)+5]

Therefore, we can say that result to

State (7) = State (6) + State (4) + State (2)

State (7) = State (7-1) + State (7-3) + State (7-5)

In general;

State (n) = State (n-1) + State (n-3) + State (n-5).

PYTHON IMPLEMENTATION:-

```
def Solve(n):
    if (n<0):
        return 0;
    if (n==0):
        return 1;
    return solve(n-1) + Solve(n-3) + solve(n-5)
```

DRAWBACKS:-

* COMPLEXITY:- Implementing dynamic pricing requires sophisticated algorithms & real-time data it mde complex & resource-intensive.

* Customer preception:- Frequent price changes can dead to customer dissatisfaction & a preceptions unfairness.

* Market Volatility:- Over-reliance on competitis pricing might dead to price wars, market stability.

3) SCENARIO: A Social media company wants to identify influential users within its network to target for marketing campaigns.

TASKS:

1) Model the Social network as a graph where users are nodes and connections are edges.

2) Implement the pageRank algorithm to identify the most influential users.

3) Compare the results of the pageRank with a simple degree centrality measure.

DELIVERABLES:

* Graph model of the Social network.
* PSuedocode & implementation of the pageRank algorithm.
* Comparison of pageRank and degree centrality results.

REASONING:- Discuss why pageRank is an effective measure for identifying influential users. Explain the differences between the pageRank and degree centrality and why one might be preffered over the other in different scenarios.

Ans:-

-GRAPH MODEL:-

* Nodes: users in the Social network.
* Edges: Connections b/w users. An edge exists if 2 users are friends in some other way.
* Weights: The weight of an edge can represent the strength of the connection frequency of interactions.

-GRAPH REPRESENTATION:

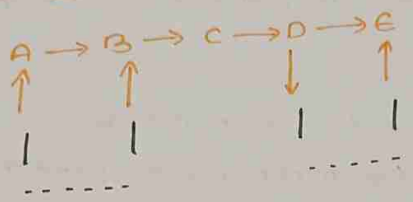
-Adjacency Matrix: A square matrix where rows & columns represents users. The value at cell (i,j) indicates the weight of the edge b/w user i and user j.

-ADJACENCY LIST:- A list where each element represents a user and contains a list of its connected neighbours.

GRAPH MODEL:-

User A follows User B and User C
User B follows User C
User C follows User D and User D
User D follows User E
User E follows User C.

GRAPH REPRESENTATION:-



PAGERANK ALGORITHM

```
import networkx as nx
G = nx.DiGraph()
edges = [('A','B'),('A','C'),('B','C'),('C','A'),('C','D'),('D','E'),('E','C')]
G.add_edges_from(edges)
pagerank = nx.pagerank(G, alpha=0.85)
print("pageRank:", pagerank)
```

SCENARIO PREFERENCE:-

* **pageRank:** Is preffered when the influence of the connections is crucial (eg., marketing campaigns Targeting users who influence other influencers).

* **Degree Centrality:-** Is useful for scenarios where the number of connections alone determines importance.

Node size & color could represent the pageRank & degree centrality score.

4) SCENARIO: A financial institution wants to develop an algorithm to detect fraudulent transcations in real-time.

TASKS:

1) Design a greedy algorithm to flag potentially fraudulent transcations based on a set of predefined rules.

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

3) Suggest and implement potential improvements to the algorithm.

DELIVERABLES:

* Pseudocode & implementation of the fraud detection algorithm.
* performance evaluation using the historical data.
* Suggestions & implementations of improvements.

REASONING:-

Explain why a greedy algorithm is suitable for real-time fraud detection. Discuss the trade-offs b/w speed and accuracy and how your algorithm addresses them.

Ans: DESIGN A GREEDY ALGORITHM FOR FRAUD DETECTION:-

Input: The Algorithm takes a transaction & a set of predefined rules.

Process:- It checks the transcation against the each rule, large transactions, multiple transcations

Output: If any rule flags the transaction as potentially fraudulent, it is added to the list of flagged transactions.

... transaction. Check against the predefined

PERFORMANCE EVALUATION USING DATA:-

Precision:- Measures the accuracy of the positive predictions

$$\text{precision} = \frac{\text{True positives}}{\text{True positive} + \text{False positive}}$$

Recall: Measures how many actual frauds were identified.

$$\text{Recall} = \frac{\text{True positives}}{\text{True positives} + \text{False Negatives}}$$

F1Score:- Harmonic mean of precision and Recall.

$$\text{F1Score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

GREEDY ALGORITHM:-

A greedy Algorithm is a problem solving technique that makes the local choice at each step in the hope of finding the global solution.

However, it is important to note that not all the problems are suitable for greedy algorithms. They work best when the problem exhibits the following properties:

Greedy choice property:- The optimal solution can be constructed by making the best local choice at each step.

Optimal Substructure:- The optimal solution to be the problem that contains the optimal solutions to its subproblems.

CODE:-

```
def detect_fraud(transactions, rules):
    for transaction in transactions:
        for rule in rules:
            if rule.matches(transactions):
                flag_as_fraudulent(transaction)
                break
```

Iterate through transaction: For each transaction, check against the predefined rules.

Match Against Rules: If a transactions matches any rule, flag it as the potentially fraudulent

Break: Once a transaction is flagged, move on to the next transaction.

IMPROVING THE ALGORITHM:-

* RULE REFINEMENT:- Continuously update and refine the rules based on the new data and feedback.

* MACHINE LEARNING:- Incorporate ML techniques to learn patterns in fraudulent transactions & make more accurate predictions.

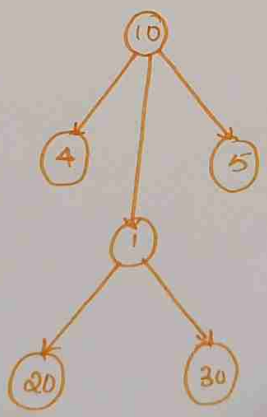
* ANOMALY DETECTION:- Use Anamoly detection algorithms to identify unusal patterns that might indicate fraud.

* CONTEXTUAL INFORMATION:- Consider, the additional factors such as user behaviour, location & time of day to improve the accuracy.

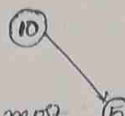
* FALSE +VE REDUCTION:- Implement the Techniques to reduce the number of false positives.

GRAPH VISUALIZATION (GRAPH):-

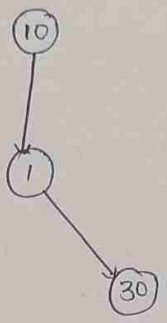
One such example where the Greedy Approach fails to the finding the maximum weighted Graph path of Nodes in the given graph.



S1:- Start from the root Node 10 if we greedily select the next node will be 5 that will total Sum to 15 & path end to 5 but the path 10-5 The max weight path



S2: In sides to find most weighted path to all the possible path Sum. It is visible that the most weighted path in the above graph 10-1-30 gives path sum 41.



SPEED:- Greedy Algorithms make local optimal choices, which allows for the Quick-decision making in real-time processing

EFFICIENCY:- By stopping checks once a transaction is flagged, the algorithm minimizes overhead, for the high-frequency transaction environments.