**1. Implement Floyd's Algorithm to find the shortest path between all pairs of cities. Display the distance matrix before and after applying the algorithm. Identify and print the shortest path Input: n = 4, edges = [[0,1,3],[1,2,1],[1,3,4],[2,3,1]], distanceThreshold = 4 Output: 3.**

**Code :**

def floyd\_warshall(n, edges, distanceThreshold):

dist = [[float('inf')] \* n for \_ in range(n)]

for i in range(n):

dist[i][i] = 0

for u, v, w in edges:

dist[u][v] = w

dist[v][u] = w

for k in range(n):

for i in range(n):

for j in range(n):

if dist[i][j] > dist[i][k] + dist[k][j]:

dist[i][j] = dist[i][k] + dist[k][j]

result = -1

min\_neighbors = n

for i in range(n):

neighbors = 0

for j in range(n):

if dist[i][j] <= distanceThreshold:

neighbors += 1

if neighbors <= min\_neighbors:

min\_neighbors = neighbors

result = i

return result

n = 4

edges = [[0, 1, 3], [1, 2, 1], [1, 3, 4], [2, 3, 1]]

distanceThreshold = 4

output = floyd\_warshall(n, edges, distanceThreshold)

print("Output:", output)

**2. Write a Program to implement Floyd's Algorithm to calculate the shortest paths between all pairs of routers. Simulate a change where the link between Router B and Router D fails. Update the distance matrix accordingly. Display the shortest path from Router A to Router F before and after the link failure. Input as above Output : Router A to Router F = 5.**

**Code :**

def floyd\_warshall(n, edges):

dist = [[float('inf')] \* n for \_ in range(n)]

for i in range(n):

dist[i][i] = 0

for u, v, w in edges:

dist[u][v] = w

dist[v][u] = w

for k in range(n):

for i in range(n):

for j in range(n):

dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])

return dist

def simulate\_link\_failure(dist, u, v):

dist[u][v] = dist[v][u] = float('inf')

n = len(dist)

for k in range(n):

for i in range(n):

for j in range(n):

dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])

return dist

def print\_shortest\_path(dist, src, dest):

print(f" Router {chr(src + ord('A'))} to Router {chr(dest + ord('d'))} = {dist[src][dest]}")

n = 6

edges = [[0, 1, 2], [0, 2, 5], [1, 2, 4], [1, 3, 6], [2, 3, 2], [3, 4, 1], [4, 5, 3]]

dist = floyd\_warshall(n, edges)

dist\_after\_failure = simulate\_link\_failure(dist, 1, 5)

print\_shortest\_path(dist\_after\_failure, 0, 2)

**3. Implement Floyd's Algorithm to find the shortest path between all pairs of cities. Display the distance matrix before and after applying the algorithm. Identify and print the shortest path Input: n = 5, edges = [[0,1,2],[0,4,8],[1,2,3],[1,4,2],[2,3,1],[3,4,1]], distance threshold = 2 Output: 0.**

**Code :**

def floyd\_warshall(n, edges):

dist = [[float('inf')] \* n for \_ in range(n)]

for i in range(n):

dist[i][i] = 0

for u, v, w in edges:

dist[u][v] = w

dist[v][u] = w

for k in range(n):

for i in range(n):

for j in range(n):

if dist[i][j] > dist[i][k] + dist[k][j]:

dist[i][j] = dist[i][k] + dist[k][j]

return dist

def find\_city\_with\_max\_neighbors(dist, distanceThreshold):

n = len(dist)

result = -1

min\_neighbors = n

for i in range(n):

neighbors = 0

for j in range(n):

if dist[i][j] <= distanceThreshold:

neighbors += 1

if neighbors <= min\_neighbors:

min\_neighbors = neighbors

result = i

return result

n = 5

edges = [[0, 1, 2], [0, 4, 8], [1, 2, 3], [1, 4, 2], [2, 3, 1], [3, 4, 1]]

distanceThreshold = 2

dist = floyd\_warshall(n, edges)

output = find\_city\_with\_max\_neighbors(dist, distanceThreshold)

print("Output:", output)

**4. Implement the Optimal Binary Search Tree algorithm for the keys A,B,C,D with frequencies 0.1,0.2,0.4,0.3 Write the code using any programming language to construct the OBST for the given keys and frequencies. Execute your code and display the resulting OBST and its cost. Print the cost and root matrix. Input N =4, Keys = {A,B,C,D} Frequencies = {01.02.,0.3,0.4} Output : 1.7.**

**Code :**

import numpy as np

def optimal\_bst(keys, freq):

n = len(keys)

cost = np.zeros((n, n))

root = np.zeros((n, n), dtype=int)

for i in range(n):

cost[i][i] = freq[i]

for L in range(2, n + 1):

for i in range(n - L + 1):

j = i + L - 1

cost[i][j] = float('inf')

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

c = 0

if r > i:

c += cost[i][r - 1]

if r < j:

c += cost[r + 1][j]

c += sum(freq[i:j + 1])

if c < cost[i][j]:

cost[i][j] = c

root[i][j] = r

return cost, root

def print\_optimal\_bst(root, keys, i, j, parent="root", is\_left=True):

if i > j:

return

r = root[i][j]

if parent == "root":

print(f"Root: {keys[r]}")

else:

if is\_left:

print(f"{keys[r]} is left child of {parent}")

else:

print(f"{keys[r]} is right child of {parent}")

print\_optimal\_bst(root, keys, i, r - 1, keys[r], True)

print\_optimal\_bst(root, keys, r + 1, j, keys[r], False)

keys = ['A', 'B', 'C', 'D']

freq = [0.1, 0.2, 0.4, 0.3]

cost, root = optimal\_bst(keys, freq)

print(f"Optimal cost: {cost[0][len(keys) - 1]}")

**5. Consider a set of keys 10,12,16,21 with frequencies 4,2,6,3 and the respective probabilities. Write a Program to construct an OBST in a programming language of your choice. Execute your code and display the resulting OBST, its cost and root matrix. Input N =4, Keys = {10,12,16,21} Frequencies = {4,2,6,3} Output : 26.**

**Code :**

import numpy as np

def optimal\_bst(keys, freq):

n = len(keys)

cost = np.zeros((n, n))

root = np.zeros((n, n), dtype=int)

for i in range(n):

cost[i][i] = freq[i]

for L in range(2, n + 1):

for i in range(n - L + 1):

j = i + L - 1

cost[i][j] = float('inf')

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

c = 0

if r > i:

c += cost[i][r - 1]

if r < j:

c += cost[r + 1][j]

c += sum(freq[i:j + 1])

if c < cost[i][j]:

cost[i][j] = c

root[i][j] = r

return cost, root

def print\_optimal\_bst(root, keys, i, j, parent="root", is\_left=True):

if i > j:

return

r = root[i][j]

if parent == "root":

print(f"Root: {keys[r]}")

else:

if is\_left:

print(f"{keys[r]} is left child of {parent}")

else:

print(f"{keys[r]} is right child of {parent}")

print\_optimal\_bst(root, keys, i, r - 1, keys[r], True)

print\_optimal\_bst(root, keys, r + 1, j, keys[r], False)

keys = [10, 12, 16, 21]

freq = [4, 2, 6, 3]

cost, root = optimal\_bst(keys, freq)

print(f"Optimal cost: {cost[0][len(keys) - 1]}")

**6. A game on an undirected graph is played by two players, Mouse and Cat, who alternate turns. The graph is given as follows: graph[a] is a list of all nodes b such that ab is an edge of the graph. The mouse starts at node 1 and goes first, the cat starts at node 2 and goes second, and there is a hole at node 0. During each player's turn, they must travel along one edge of the graph that meets where they are. For example, if the Mouse is at node 1, it must travel to any node in graph[1]. Additionally, it is not allowed for the Cat to travel to the Hole (node 0).Then, the game can end in three ways: If ever the Cat occupies the same node as the Mouse, the Cat wins. If ever the Mouse reaches the Hole, the Mouse wins. If ever a position is repeated (i.e., the players are in the same position as a previous turn, and it is the same player's turn to move), the game is a draw. Given a graph, and assuming both players play optimally, return 1 if the mouse wins the game, 2 if the cat wins the game, or 0 if the game is a draw. Example 1: Input: graph = [[2,5],[3],[0,4,5],[1,4,5],[2,3],[0,2,3]] Output: 0 Example 2: Input: graph = [[1,3],[0],[3],[0,2]] Output: 1.**

**Code :**

from collections import deque

def catMouseGame(graph):

n = len(graph)

dp = [[[0] \* 2 for \_ in range(n)] for \_ in range(n)]

queue = deque()

for i in range(1, n):

dp[0][i][0] = dp[0][i][1] = 1

queue.append((0, i, 0))

queue.append((0, i, 1))

for j in range(1, n):

dp[j][j][0] = dp[j][j][1] = 2

queue.append((j, j, 0))

queue.append((j, j, 1))

while queue:

mouse, cat, turn = queue.popleft()

result = dp[mouse][cat][turn]

if turn == 0:

for prev\_cat in graph[cat]:

if prev\_cat == 0:

continue

if dp[mouse][prev\_cat][1] == 0:

if result == 2:

dp[mouse][prev\_cat][1] = 2

queue.append((mouse, prev\_cat, 1))

elif all(dp[mouse][prev\_cat\_next][1] == 1 for prev\_cat\_next in graph[cat]):

dp[mouse][prev\_cat][1] = 1

queue.append((mouse, prev\_cat, 1))

else:

for prev\_mouse in graph[mouse]:

if dp[prev\_mouse][cat][0] == 0:

if result == 1:

dp[prev\_mouse][cat][0] = 1

queue.append((prev\_mouse, cat, 0))

elif all(dp[prev\_mouse\_next][cat][0] == 2 for prev\_mouse\_next in graph[mouse]):

dp[prev\_mouse][cat][0] = 2

queue.append((prev\_mouse, cat, 0))

return dp[1][2][0]

graph1 = [[2,5],[3],[0,4,5],[1,4,5],[2,3],[0,2,3]]

print("output:",catMouseGame(graph1))

**7. You are given an undirected weighted graph of n nodes (0-indexed), represented by an edge list where edges[i] = [a, b] is an undirected edge connecting the nodes a and b with a probability of success of traversing that edge succProb[i]. Given two nodes start and end, find the path with the maximum probability of success to go from start to end and return its success probability. If there is no path from start to end, return 0. Your answer will be accepted if it differs from the correct answer by at most 1e-5. Example 1: Input: n = 3, edges = [[0,1],[1,2],[0,2]], succProb = [0.5,0.5,0.2], start = 0, end = 2 Output: 0.25000.**

**Code :**

import heapq

from collections import defaultdict

def maxProbability(n, edges, succProb, start, end):

graph = defaultdict(list)

for i, (a, b) in enumerate(edges):

graph[a].append((b, succProb[i]))

graph[b].append((a, succProb[i]))

max\_heap = [(-1.0, start)]

probabilities = [0.0] \* n

probabilities[start] = 1.0

while max\_heap:

prob, node = heapq.heappop(max\_heap)

prob = -prob

if node == end:

return prob

for neighbor, edge\_prob in graph[node]:

new\_prob = prob \* edge\_prob

if new\_prob > probabilities[neighbor]:

probabilities[neighbor] = new\_prob

heapq.heappush(max\_heap, (-new\_prob, neighbor))

return 0.0

n = 3

edges = [[0,1],[1,2],[0,2]]

succProb = [0.5,0.5,0.2]

start = 0

end = 2

print(maxProbability(n, edges, succProb, start, end))

**8. There is a robot on an m x n grid. The robot is initially located at the top-left corner (i.e., grid[0][0]). The robot tries to move to the bottom-right corner (i.e., grid[m - 1][n - 1]). The robot can only move either down or right at any point in time. Given the two integers m and n, return the number of possible unique paths that the robot can take to reach the bottom-right corner. The test cases are generated so that the answer will be less than or equal to 2 \* 10 9. Example 1: START FINISH Input: m = 3, n = 7 Output: 28.**

**Code :**

def uniquePaths(m, n):

dp = [[1] \* n for \_ in range(m)]

for i in range(1, m):

for j in range(1, n):

dp[i][j] = dp[i - 1][j] + dp[i][j - 1]

return dp[m - 1][n - 1]

m = 3

n = 7

print(uniquePaths(m, n))

**9. Given an array of integers nums, return the number of good pairs. A pair (i, j) is called good if nums[i] == nums[j] and i < j. Example 1: Input: nums = [1,2,3,1,1,3] Output: 4.**

**Code :**

from collections import defaultdict

def numIdenticalPairs(nums):

count = defaultdict(int)

good\_pairs = 0

for num in nums:

good\_pairs += count[num]

count[num] += 1

return good\_pairs

nums = [1,2,3,1,1,3]

print(numIdenticalPairs(nums))

**10. There are n cities numbered from 0 to n-1. Given the array edges where edges[i] = [fromi, toi, weighti] represents a bidirectional and weighted edge between cities fromi and toi, and given the integer distanceThreshold. Return the city with the smallest number of cities that are reachable through some path and whose distance is at most distanceThreshold, If there are multiple such cities, return the city with the greatest number. Notice that the distance of a path connecting cities i and j is equal to the sum of the edges' weights along that path. Example 1: Input: n = 4, edges = [[0,1,3],[1,2,1],[1,3,4],[2,3,1]], distanceThreshold = 4 Output: 3.**

**Code :**

import heapq

from collections import defaultdict

def findTheCity(n, edges, distanceThreshold):

graph = defaultdict(list)

for u, v, w in edges:

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

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

def dijkstra(start):

distances = [float('inf')] \* n

distances[start] = 0

min\_heap = [(0, start)]

while min\_heap:

current\_dist, u = heapq.heappop(min\_heap)

if current\_dist > distances[u]:

continue

for v, weight in graph[u]:

distance = current\_dist + weight

if distance < distances[v]:

distances[v] = distance

heapq.heappush(min\_heap, (distance, v))

return sum(1 for dist in distances if dist <= distanceThreshold)

min\_reachable\_count = float('inf')

result\_city = -1

for city in range(n):

reachable\_count = dijkstra(city)

if (reachable\_count < min\_reachable\_count) or (reachable\_count == min\_reachable\_count and city > result\_city):

min\_reachable\_count = reachable\_count

result\_city = city

return result\_city

n = 4

edges = [[0,1,3],[1,2,1],[1,3,4],[2,3,1]]

distanceThreshold = 4

print(findTheCity(n, edges, distanceThreshold))

**11. You are given a network of n nodes, labeled from 1 to n. You are also given times, a list of travel times as directed edges times[i] = (ui, vi, wi), where ui is the source node, vi is the target node, and wi is the time it takes for a signal to travel from source to target. We will send a signal from a given node k. Return the minimum time it takes for all the n nodes to receive the signal. If it is impossible for all the n nodes to receive the signal, return -1. Example 1: Input: times = [[2,1,1],[2,3,1],[3,4,1]], n = 4, k = 2 Output: 2.**

**Code :**

import heapq

import math

def networkDelayTime(times, n, k):

graph = {i: [] for i in range(1, n + 1)}

for u, v, w in times:

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

dist = {i: math.inf for i in range(1, n + 1)}

dist[k] = 0

pq = [(0, k)]

while pq:

d, node = heapq.heappop(pq)

if d > dist[node]:

continue

for neighbor, weight in graph[node]:

new\_dist = d + weight

if new\_dist < dist[neighbor]:

dist[neighbor] = new\_dist

heapq.heappush(pq, (new\_dist, neighbor))

max\_dist = max(dist.values())

return max\_dist if max\_dist < math.inf else -1

times = [[2,1,1],[2,3,1],[3,4,1]]

n = 4

k = 2

print(networkDelayTime(times, n, k))