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 Explanation: The figure above describes the graph. The neighboring cities at a distanceThreshold = 4 for each city are: City 0 -> [City 1, City 2] City 1 -> [City 0, City 2, City 3] City 2 -> [City 0, City 1, City 3] City 3 -> [City 1, City 2] Cities 0 and 3 have 2 neighboring cities at a distanceThreshold = 4, but we have to return city 3 since it has the greatest number. Test cases : a) You are given a small network of 4 cities connected by roads with the following distances: City 1 to City 2: 3 City 1 to City 3: 8 City 1 to City 4: -4 City 2 to City 4: 1 City 2 to City 3: 4 City 3 to City 1: 2 City 4 to City 3: -5 City 4 to City 2: 6 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 from City 1 to City 3. Input as above Output : City 1 to City 3 = -9 b. Consider a network with 6 routers. The initial routing table is as follows: Router A to Router B: 1 Router A to Router C: 5 Router B to Router C: 2 Router B to Router D: 1 Router C to Router E: 3 Router D to Router E: 1 Router D to Router F: 6 Router E to Router F: 2

import sys

# Floyd-Warshall Algorithm

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], dist[v][u] = w, w # Undirected graph

print("Before Floyd-Warshall:\n", \*dist, sep="\n")

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])

print("\nAfter Floyd-Warshall:\n", \*dist, sep="\n")

return dist

# Find shortest path between two specific nodes

def shortest\_path(dist, u, v):

return dist[u][v] if dist[u][v] != float('inf') else "No path"

# Test Case 1a: Small network of 4 cities

n1 = 4

edges1 = [[0, 1, 3], [0, 2, 8], [0, 3, -4], [1, 3, 1], [1, 2, 4], [2, 0, 2], [3, 2, -5], [3, 1, 6]]

dist1 = floyd\_warshall(n1, edges1)

print("\nShortest path from City 1 to City 3:", shortest\_path(dist1, 0, 2))

# Test Case 1b: Network of 6 routers

n2 = 6

edges2 = [[0, 1, 1], [0, 2, 5], [1, 2, 2], [1, 3, 1], [2, 4, 3], [3, 4, 1], [3, 5, 6], [4, 5, 2]]

dist2 = floyd\_warshall(n2, edges2)

print("\nShortest path from Router A to Router F:", shortest\_path(dist2, 0, 5))

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

import sys

# Floyd-Warshall Algorithm

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], dist[v][u] = w, 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

# Remove a link from the graph (Simulate failure)

def remove\_link(dist, u, v):

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

# Display shortest path

def shortest\_path(dist, u, v):

return dist[u][v] if dist[u][v] != float('inf') else "No path"

# Initial setup

n = 6

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

# Floyd-Warshall before link failure

dist\_before = floyd\_warshall(n, edges)

print("Shortest path from Router A to Router F before failure:", shortest\_path(dist\_before, 0, 5))

# Simulate link failure between Router B and Router D

remove\_link(dist\_before, 1, 3)

# Floyd-Warshall after link failure

dist\_after = floyd\_warshall(n, edges)

print("Shortest path from Router A to Router F after failure:", shortest\_path(dist\_after, 0, 5))

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]], distanceThreshold = 2 Output: 0 Explanation: The figure above describes the graph. The neighboring cities at a distanceThreshold = 2 for each city are: City 0 -> [City 1] City 1 -> [City 0, City 4] City 2 -> [City 3, City 4] City 3 -> [City 2, City 4] City 4 -> [City 1, City 2, City 3] The city 0 has 1 neighboring city at a distanceThreshold = 2. a) Test cases : B to A 2 A TO C 3 C TO D 1 D TO A 6 C TO B 7 Find shortest path from C to A Output = 7 b) Find shortest path from E to C C TO A 2 A TO B 4 B TO C 1 B TO E 6 E TO A 1 A TO D 5 D TO E 2 E TO D 4 D TO C 1 C TO D 3 Output : E to C = 5

def floyd\_warshall\_cities(n, edges, threshold):

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], dist[v][u] = w, 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])

print("Distance matrix after Floyd-Warshall:\n", \*dist, sep="\n")

# Count neighbors within the threshold for each city

reachable\_cities = [sum(1 for d in dist[i] if d <= threshold) for i in range(n)]

# Find the city with the smallest number of neighbors, breaking ties by largest index

result\_city = max(range(n), key=lambda x: (reachable\_cities[x], -x))

return result\_city

# Test Case

n = 5

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

threshold = 2

result = floyd\_warshall\_cities(n

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 Cost Table 0 1 2 3 4 1 0 0.1 0.4 1.1 1.7 2 0 0.2 0.8 0.4 3 0 0.4 1.0 4 0 0.3 5 0 Root table 1 2 3 4 1 1 2 3 3 2 2 3 3 3 3 3 4 4 a) Test cases Input: keys[] = {10, 12}, freq[] = {34, 50} Output = 118 b) Input: keys[] = {10, 12, 20}, freq[] = {34, 8, 50} Output = 142

import sys

def obst(keys, freq, n):

cost = [[0 for \_ in range(n)] for \_ in range(n)]

root = [[0 for \_ in range(n)] for \_ in range(n)]

for i in range(n):

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

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

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

j = i + length - 1

cost[i][j] = sys.maxsize

total\_freq = sum(freq[i:j + 1])

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

c = (0 if r == i else cost[i][r - 1]) + (0 if r == j else cost[r + 1][j])

if c + total\_freq < cost[i][j]:

cost[i][j] = c + total\_freq

root[i][j] = r

return cost, root

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 0 1 2 3 0 4 80 202 262 1 2 102 162 2 6 12 3 3 a) Test cases Input: keys[] = {10, 12}, freq[] = {34, 50} Output = 118 b) Input: keys[] = {10, 12, 20}, freq[] = {34, 8, 50} Output = 142

# Test Case

keys = [10, 12, 16, 21]

freq = [4, 2, 6, 3]

n = len(keys)

cost, root = obst(keys, freq, n)

print("Cost Table:")

for row in cost:

print(row)

print("\nRoot Table:")

for row in root:

print(row)

print("Optimal cost of OBST:", cost[0][n - 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

from collections import deque

def catMouseGame(graph):

n = len(graph)

# 3D memoization: 0 -> mouse wins, 1 -> cat wins, 2 -> draw

dp = [[[-1] \* 3 for \_ in range(n)] for \_ in range(n)]

# Breadth-first search to determine game states

def search(mouse, cat, turn):

# Mouse wins by reaching the hole

if mouse == 0: return 1

# Cat wins by catching the mouse

if cat == mouse: return 2

# If the game is stuck in a draw

if dp[mouse][cat][turn] != -1:

return dp[mouse][cat][turn]

if turn == 0: # Mouse's turn

for next\_pos in graph[mouse]:

if search(next\_pos, cat, 1 - turn) == 1:

dp[mouse][cat][turn] = 1

return 1

dp[mouse][cat][turn] = 2

return 2

else: # Cat's turn

for next\_pos in graph[cat]:

if next\_pos != 0 and search(mouse, next\_pos, 1 - turn) == 2:

dp[mouse][cat][turn] = 2

return 2

dp[mouse][cat][turn] = 1

return 1

result = search(1, 2, 0)

return result if result != -1 else 0

# Example usage

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

graph2 = [[1,3],[0],[3],[0,2]]

print(catMouseGame(graph1)) # Output: 0

print(catMouseGame(graph2)) # Output: 1

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 Explanation: There are two paths from start to end, one having a probability of success = 0.2 and the other has 0.5 \* 0.5 = 0.25. Example 2: Input: n = 3, edges = [[0,1],[1,2],[0,2]], succProb = [0.5,0.5,0.3], start = 0, end = 2 Output: 0.30000

import heapq

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

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

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

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

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

# Priority queue to maximize probability

pq = [(-1, start)] # Initialize with start node and probability of 1

probabilities = [0] \* n

probabilities[start] = 1

while pq:

prob, node = heapq.heappop(pq)

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(pq, (-new\_prob, neighbor))

return 0.0

# Example usage

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)) # Output: 0.258. 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 Example 2: Input: m = 3, n = 2 Output: 3 Explanation: From the top-left corner, there are a total of 3 ways to reach the bottom-right corner: 1. Right -> Down -> Down 2. Down -> Down -> Right 3. Down -> Right -> Down

def uniquePaths(m, n):

dp = [[1] \* n for \_ in range(m)] # Initialize DP array with 1s

# Update DP table

for i in range(1, m):

for j in range(1, n):

dp[i][j] = dp[i-1][j] + dp[i][j-1] # Sum of paths from above and left

return dp[m-1][n-1] # Return the bottom-right corner value

# Example usage

print(uniquePaths(3, 7)) # Output: 28

print(uniquePaths(3, 2)) # Output: 3

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 Explanation: There are 4 good pairs (0,3), (0,4), (3,4), (2,5) 0-indexed. Example 2: Input: nums = [1,1,1,1] Output: 6 Explanation: Each pair in the array are good.

from collections import Counter

def numIdenticalPairs(nums):

count = Counter(nums) # Count occurrences of each number

result = 0

# For each unique number, count the number of good pairs

for value in count.values():

if value > 1:

result += (value \* (value - 1)) // 2 # nC2 pairs formula

return result

# Example usage

print(numIdenticalPairs([1, 2, 3, 1, 1, 3])) # Output: 4

print(numIdenticalPairs([1, 1, 1, 1])) # Output: 6

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 Explanation: The figure above describes the graph. The neighboring cities at a distanceThreshold = 4 for each city are: City 0 -> [City 1, City 2] City 1 -> [City 0, City 2, City 3] City 2 -> [City 0, City 1, City 3] City 3 -> [City 1, City 2] Cities 0 and 3 have 2 neighboring cities at a distance Threshold = 4, but we have to return city 3 since it has the greatest number. Example 2: 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 Explanation: The figure above describes the graph. The neighboring cities at a distance Threshold = 2 for each city are: City 0 -> [City 1] City 1 -> [City 0, City 4] City 2 -> [City 3, City 4] City 3 -> [City 2, City 4] City 4 -> [City 1, City 2, City 3] The city 0 has 1 neighboring city at a distanceThreshold = 2.

import math

def findTheCity(n, edges, distanceThreshold):

# Initialize distance matrix with infinity

dist = [[math.inf] \* n for \_ in range(n)]

# Distance to itself is 0

for i in range(n):

dist[i][i] = 0

# Fill initial distances based on edges

for u, v, w in edges:

dist[u][v] = w

dist[v][u] = w

# Apply Floyd-Warshall to calculate all pairs shortest paths

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])

# Find the city with the fewest reachable cities within the distanceThreshold

minReachableCities = n

resultCity = -1

for i in range(n):

reachableCities = sum

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 Example 2: Input: times = [[1,2,1]], n = 2, k = 1 Output: 1 Example 3: Input: times = [[1,2,1]], n = 2, k = 2 Output: -1.

import heapq

def networkDelayTime(times, n, k):

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

# Build the graph

for u, v, w in times:

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

# Priority queue for Dijkstra's algorithm

pq = [(0, k)] # (time, node)

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

dist[k] = 0

while pq:

time, node = heapq.heappop(pq)

if time > dist[node]:

continue

# Relax edges

for neighbor, weight in graph[node]:

new\_time = time + weight

if new\_time < dist[neighbor]:

dist[neighbor] = new\_time

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

# Find the maximum time it takes to reach all nodes

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

return max\_dist if max\_dist < float('inf') else -1

# Example usage

times1 = [[2,1,1],[2,3,1],[3,4,1]]

print(networkDelayTime(times1, 4, 2)) # Output: 2

times2 = [[1,2,1]]

print(networkDelayTime(times2, 2, 1)) # Output: 1

times3 = [[1,2,1]]

print(networkDelayTime(times3, 2, 2)) # Output: -1