DAY-8

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
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.

CODE:

```
import sys

def floyd_warshall(n, edges, distanceThreshold):

    # Initialize the distance matrix

    dist = [[sys.maxsize] * n for _ in range(n)]

        for i in range(n):

        dist[i][i] = 0

    for edge in edges:

        u, v, w = edge

        dist[u][v] = w

        dist[v][u] = w  # Since the graph is undirected

    print("Distance matrix before applying Floyd's algorithm:")

    print_matrix(dist)

    for k in range(n):

        for i in range(n):

            for j in range(n):

                if dist[i][k] != sys.maxsize and dist[k][j] != sys.maxsize:

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

    print("\nDistance matrix after applying Floyd's algorithm:")

    print_matrix(dist)

    neighboring_cities = []

    for i in range(n):

        count = 0
```

```
        for j in range(n):
            if dist[i][j] <= distanceThreshold and i != j:
                count += 1
        neighboring_cities.append((i, count))
        city_with_max_neighbors = max(neighboring_cities, key=lambda x: (x[1], -x[0]))[0]
    print("\nCity with the most neighbors within distance threshold =", distanceThreshold, ":",
city_with_max_neighbors)
  return city_with_max_neighbors
def print_matrix(matrix):
    for row in matrix:
        print(row)
n = 4
edges = [[0, 1, 3], [1, 2, 1], [1, 3, 4], [2, 3, 1]]
distanceThreshold = 4
result = floyd_warshall(n, edges, distanceThreshold)
print("Output:", result)
```

OUTPUT:

Distance matrix after applying Floyd's algorithm:

[0, 3, 4, 5]

[3, 0, 1, 2]

[4, 1, 0, 1]

[5, 2, 1, 0]

City with the most neighbors within distance threshold = 4 : 3

Output: 3

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:
```
import sys

def floyd_warshall(n, edges):

    # Initialize the distance matrix

    dist = [[sys.maxsize] * n for _ in range(n)]

        for i in range(n):

        dist[i][i] = 0

        for edge in edges:

        u, v, w = edge

        dist[u][v] = w

        dist[v][u] = w

    print("Distance matrix before applying Floyd's algorithm:")

    print_matrix(dist)

    for k in range(n):

        for i in range(n):

            for j in range(n):

                if dist[i][k] != sys.maxsize and dist[k][j] != sys.maxsize:

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

    return dist

def simulate_link_failure(dist, routerB, routerD):

    dist[routerB][routerD] = sys.maxsize

    dist[routerD][routerB] = sys.maxsize

    print("\nSimulated link failure between Router B and Router D.")

def print_matrix(matrix):

    for row in matrix:

        print(row)

def find_shortest_path(dist, routerA, routerF):

    if dist[routerA][routerF] == sys.maxsize:
```

```
        return "No path available"
    return dist[routerA][routerF]
n = 6
edges = [
    [0, 1, 2],
    [0, 2, 4],
    [1, 2, 1],
    [1, 3, 7],
    [2, 4, 3],
    [3, 4, 2],
    [3, 5, 1
    [4, 5, 5]
]routerA = 0
routerF = 5
routerB = 1
routerD = 3
dist = floyd_warshall(n, edges)
print("\nShortest path from Router A to Router F before link failure:")
shortest_path_before = find_shortest_path(dist, routerA, routerF)
print("Router A to Router F =", shortest_path_before)
simulate_link_failure(dist, routerB, routerD)
for k in range(n):
    for i in range(n):
        for j in range(n):
            if dist[i][k] != sys.maxsize and dist[k][j] != sys.maxsize:
                dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])
print("\nShortest path from Router A to Router F after link failure:")
shortest_path_after = find_shortest_path(dist, routerA, routerF)
print("Router A to Router F =", shortest_path_after)
```

OUTPUT:

Distance matrix before applying Floyd's algorithm:

[0, 2, 4, 9, 7, 10]

[2, 0, 1, 7, 4, 8]

[4, 1, 0, 6, 3, 7]

[9, 7, 6, 0, 2, 1]

[7, 4, 3, 2, 0, 5]

[10, 8, 7, 1, 5, 0]

Shortest path from Router A to Router F before link failure:

Router A to Router F = 5

Simulated link failure between Router B and Router D.

Shortest path from Router A to Router F after link failure:

Router A to Router F = 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.

<u>CODE:</u>

```
import sys

def floyd_warshall(n, edges):

    dist = [[sys.maxsize] * n for _ in range(n)]

    for i in range(n):

        dist[i][i] = 0

        for edge in edges:

        u, v, w = edge

        dist[u][v] = w

        dist[v][u] = w  # The graph is undirected

    print("Distance matrix before applying Floyd's algorithm:")

    print_matrix(dist)

    for k in range(n):

        for i in range(n):

            for j in range(n):

                if dist[i][k] != sys.maxsize and dist[k][j] != sys.maxsize:

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

    return dist

def print_matrix(matrix):

    for row in matrix:

        print(row)

def count_neighbors(dist, distanceThreshold):

    neighbors_count = [0] * len(dist)

    for i in range(len(dist)):
```

```
        for j in range(len(dist)):
            if i != j and dist[i][j] <= distanceThreshold:
                neighbors_count[i] += 1
    return neighbors_count
def find_city_with_fewest_neighbors(neighbors_count):
    min_neighbors = sys.maxsize
    city_with_min_neighbors = -1
    for i, count in enumerate(neighbors_count):
        if count < min_neighbors:
            min_neighbors = count
            city_with_min_neighbors = i

    return city_with_min_neighbors
n = 5
edges = [
    [0, 1, 2],  # City 0 -> City 1
    [0, 4, 8],  # City 0 -> City 4
    [1, 2, 3],  # City 1 -> City 2
    [1, 4, 2],  # City 1 -> City 4
    [2, 3, 1],  # City 2 -> City 3
    [3, 4, 1]   # City 3 -> City 4
]
distanceThreshold = 2
dist = floyd_warshall(n, edges)
print("\nDistance matrix after applying Floyd's algorithm:")
print_matrix(dist)
neighbors_count = count_neighbors(dist, distanceThreshold)
print("\nNumber of neighboring cities within distance threshold:")
for i in range(n):
    print(f"City {i} -> {neighbors_count[i]} neighbors")
city_with_min_neighbors = find_city_with_fewest_neighbors(neighbors_count)
print(f"\nCity with the fewest neighboring cities within the distance threshold: City {city_with_min_neighbors}")
```

OUTPUT:

Distance matrix before applying Floyd's algorithm:

[0, 2, inf, inf, 8]

[2, 0, 3, inf, 2]

[inf, 3, 0, 1, inf]

[inf, inf, 1, 0, 1]

[8, 2, inf, 1, 0]

Distance matrix after applying Floyd's algorithm:

[0, 2, 5, 6, 4]

[2, 0, 3, 4, 2]

[5, 3, 0, 1, 2]

[6, 4, 1, 0, 1]

[4, 2, 2, 1, 0]

Number of neighboring cities within distance threshold:

City 0 -> 1 neighbors

City 1 -> 2 neighbors

City 2 -> 2 neighbors

City 3 -> 2 neighbors

City 4 -> 3 neighbors

City with the fewest neighboring cities within the distance threshold: City 0

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

CODE:

```
import sys

def optimal_bst(keys, freq, n):

    # Initialize the cost and root tables

    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 L in range(2, n + 1):  # L is the chain length

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

            j = i + L - 1

            cost[i][j] = sys.maxsize

            sum_freq = sum(freq[i:j+1])  # Sum of frequencies from i to j

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

                # Calculate cost when r is the root

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

                if c < cost[i][j]:

                    cost[i][j] = c

                    root[i][j] = r

    return cost, root
```

```
def print_matrix(matrix, name):
    print(f"\n{name} Table:")
    for row in matrix:
        print(row)
keys = ['A', 'B', 'C', 'D']
freq = [0.1, 0.2, 0.4, 0.3]
n = len(keys)
cost, root = optimal_bst(keys, freq, n)
print_matrix(cost, "Cost")
print_matrix(root, "Root")
print(f"\nThe minimum cost of the OBST is: {cost[0][n-1]}")
```

OUTPUT:

```
Cost Table:
[0.1, 0.4, 1.1, 1.7]
[0, 0.2, 0.8, 1.4]
[0, 0, 0.4, 1.0]
[0, 0, 0, 0.3]
Root Table:
[0, 1, 2, 2]
[0, 1, 2, 2]
[0, 0, 2, 3]
[0, 0, 0, 3]
 The minimum cost of the OBST is: 1.7
```

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

<u>CODE:</u>
```
import sys

def optimal_bst(keys, freq, n):

    # Initialize the cost and root tables

    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 L in range(2, n + 1):  # L is the chain length

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

            j = i + L - 1

            cost[i][j] = sys.maxsize

            sum_freq = sum(freq[i:j+1])  # Sum of frequencies from i to j

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

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

                if c < cost[i][j]:

                    cost[i][j] = c

                    root[i][j] = r

    return cost, root

def print_matrix(matrix, name):

    print(f"\n{name} Table:")

    for row in matrix:
```

```
        print(row)
keys = [10, 12, 16, 21]
freq = [4, 2, 6, 3]
n = len(keys)

cost, root = optimal_bst(keys, freq, n)
print_matrix(cost, "Cost")
print_matrix(root, "Root")
print(f"\nThe minimum cost of the OBST is: {cost[0][n-1]}")
```

OUTPUT:

```
Cost Table:
[4, 10, 26, 46]
[0, 2, 14, 28]
[0, 0, 6, 15]
[0, 0, 0, 3]

Root Table:
[0, 0, 2, 2]
[0, 1, 2, 2]
[0, 0, 2, 3]
[0, 0, 0, 3]

The minimum cost of the OBST is: 26
```

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

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 cat in range(1, n):

        dp[0][cat][0] = 1  # Mouse's turn, Mouse wins

        dp[0][cat][1] = 1  # Cat's turn, Mouse wins

        queue.append((0, cat, 0))

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

    for mouse in range(1, n):

        dp[mouse][mouse][0] = 2  # Mouse's turn, Cat wins

        dp[mouse][mouse][1] = 2  # Cat's turn, Cat wins

        queue.append((mouse, mouse, 0))

        queue.append((mouse, mouse, 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][next_cat][0] == 1 for next_cat in graph[mouse]):
                    # If every possible move for the Cat leads to Mouse winning
                    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 the game hasn't been decided yet for this state
                if result == 1:  # Mouse wins this state
                    dp[prev_mouse][cat][0] = 1
                    queue.append((prev_mouse, cat, 0))
                elif all(dp[next_mouse][cat][1] == 2 for next_mouse in graph[prev_mouse]):
                    dp[prev_mouse][cat][0] = 2
                    queue.append((prev_mouse, cat, 0))
    return dp[1][2][0]
graph = [[2, 5], [3], [0, 4, 5], [1, 4, 5], [2, 3], [0, 2, 3]]
result = catMouseGame(graph)
print(result)
```

OUTPUT:

0

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

<u>CODE:</u>

```
import heapq
def maxProbability(n, edges, succProb, start, end):
    graph = [[] for _ in range(n)]
    for (a, b), prob in zip(edges, succProb):
        graph[a].append((b, prob))
        graph[b].append((a, prob))
    max_prob = [0.0] * n
    max_prob[start] = 1.0  # Start node has probability 1 to itself
        pq = [(-1.0, start)]  # We use -1.0 because heapq is a min-heap, and we want to maximize the
probability
        while pq:
        current_prob, node = heapq.heappop(pq)
        current_prob = -current_prob  # Convert back to positive
if node == end:
            return current_prob
                for neighbor, edge_prob in graph[node]:
            new_prob = current_prob * edge_prob
            if new_prob > max_prob[neighbor]:
                max_prob[neighbor] = new_prob
                heapq.heappush(pq, (-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
result = maxProbability(n, edges, succProb, start, end)
print(f"Output: {result:.5f}")
```

OUTPUT:

0.25000

8) 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

<u>CODE:</u>

```
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
result = uniquePaths(m, n)
print(f"Output: {result}")
```

OUTPUT:

1 1 1 1 1 1 1

1 2 3 4 5 6 7

1 3 6 10 15 21 28

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

<u>CODE:</u>

```
def numIdenticalPairs(nums):
    freq = {}
    good_pairs = 0
        for num in nums:
        if num in freq:
            good_pairs += freq[num]
            freq[num] += 1
        else:
            freq[num] = 1
    return good_pairs
nums = [1, 2, 3, 1, 1, 3]
result = numIdenticalPairs(nums)
print(f"Output: {result}")
```

OUTPUT:

4

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

<u>**CODE:**</u>

```
import heapq

def findTheCity(n, edges, distanceThreshold):

    graph = [[] for _ in range(n)]

    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)]  # (distance, node)


        while min_heap:

            current_distance, current_node = heapq.heappop(min_heap)

            if current_distance > distances[current_node]:

                continue


            for neighbor, weight in graph[current_node]:

                distance = current_distance + weight

                if distance < distances[neighbor]:

                    distances[neighbor] = distance


  heapq.heappush(min_heap, (distance, neighbor))

        return distances

    min_reachable_count = float('inf')
```

```
    city_with_min_reachable = -1
    for city in range(n):
        distances = dijkstra(city)
        reachable_count = sum(1 for dist in distances if dist <= distanceThreshold)
        if (reachable_count < min_reachable_count) or (
            reachable_count == min_reachable_count and city > city_with_min_reachable):
            min_reachable_count = reachable_count
            city_with_min_reachable = city
    return city_with_min_reachable




n = 4
edges = [[0, 1, 3], [1, 2, 1], [1, 3, 4], [2, 3, 1]]
distanceThreshold = 4
result = findTheCity(n, edges, distanceThreshold)
print(f"Output: {result}")
```

OUTPUT:

3

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
Output: 2

CODE:

```
import heapq
def networkDelayTime(times, n, k):
   # Step 1: Create the graph as an adjacency list
   graph = [[] for _ in range(n + 1)]

   for u, v, w in times:
      graph[u].append((v, w))  # u -> (v, w)
   distances = [float('inf')] * (n + 1)
   distances[k] = 0
   min_heap = [(0, k)]  # (time, node)
      while min_heap:
      current_time, current_node = heapq.heappop(min_heap)
            if current_time > distances[current_node]:
         continue
            for neighbor, travel_time in graph[current_node]:
         new_time = current_time + travel_time
                  if new_time < distances[neighbor]:
            distances[neighbor] = new_time
            heapq.heappush(min_heap, (new_time, neighbor))
   max_time = max(distances[1:])  # Ignore index 0 as nodes are 1-indexed
   return max_time if max_time != float('inf') else -1
times = [[2, 1, 1], [2, 3, 1], [3, 4, 1]]
n = 4
k = 2
result = networkDelayTime(times, n, k)
print(f"Output: {result}")
```

OUTPUT:

2