DAY 8

1.

```
# Floyd-Warshall algorithm implementation
def floyd_warshall(n, edges):
    # Initialize the distance matrix with infinity and 0 for diagonal elements
    dist = [[float('inf')] * n for _ in range(n)]

    # Distance to itself is always zero
    for i in range(n):
        dist[i][i] = 0

    # Initialize the distances based on the input edges
    for u, v, w in edges:
        dist[u][v] = w
        dist[v][u] = w  # As the graph is undirected, set both directions

    # Apply Floyd-Warshall algorithm
    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

# Function to find the number of cities within a given distance threshold
def count_cities_within_threshold(n, dist, threshold):
    city_counts = [0] * n
    for i in range(n):
        for j in range(n):
            if dist[i][j] <= threshold:
                city_counts[i] += 1
    return city_counts

# Test case 1: 4 Cities with given edges
n = 4
edges = [[0, 1, 3], [1, 2, 1], [1, 3, 4], [2, 3, 1]]
distanceThreshold = 4

# Apply Floyd-Warshall
```

```
dist_matrix = floyd_warshall(n, edges)
print("Distance Matrix before applying Floyd-Warshall:")
for row in dist_matrix:
    print(row)

# Count cities within distance threshold
city_counts = count_cities_within_threshold(n, dist_matrix, distanceThreshold)

# Find the city with the greatest number of neighboring cities within the threshold
max_city = max(range(n), key=lambda x: city_counts[x])
print(f"\nCity {max_city} has the greatest number of neighbors within the threshold distance.")

# Test case 2: Small network with 4 cities and different edges
edges2 = [
    [0, 1, 3], [0, 2, 8], [0, 3, -4],
    [1, 3, 1], [1, 2, 4], [2, 0, 2],
    [3, 2, -5], [3, 1, 6]
]
n2 = 4

# Apply Floyd-Warshall
dist_matrix2 = floyd_warshall(n2, edges2)
print("\nDistance Matrix after applying Floyd-Warshall:")
for row in dist_matrix2:
    print(row)

# Shortest path from City 1 to City 3
print(f"\nShortest path from City 1 to City 3 = {dist_matrix2[1][3]}")
```

2.

```
# Floyd-Warshall algorithm implementation
def floyd_warshall(n, dist):
    # Apply Floyd-Warshall algorithm to find the shortest paths between all pairs
    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

# Display the distance matrix in a readable format
def print_distance_matrix(dist):
    for row in dist:
        print(row)

# Simulate the link failure between Router B and Router D
def simulate_link_failure(dist, b, d):
    dist[b][d] = float('inf')  # No direct path
    dist[d][b] = float('inf')  # No direct path in the opposite direction
```

```
# Test case: 6 Routers and initial routing table
n = 6  # Number of routers (Router A, B, C, D, E, F)
edges = [
    [0, 1, 1],  # Router A to Router B
    [0, 2, 5],  # Router A to Router C
    [1, 2, 2],  # Router B to Router C
    [1, 3, 1],  # Router B to Router D
    [2, 4, 3],  # Router C to Router E
    [3, 4, 1],  # Router D to Router E
    [3, 5, 6],  # Router D to Router F
    [4, 5, 2]   # Router E to Router F
]

# Create the initial distance matrix
dist = [[float('inf')] * n for _ in range(n)]
for i in range(n):
    dist[i][i] = 0  # Distance to itself is zero

# Add edges to the distance matrix
for u, v, w in edges:
    dist[u][v] = w
    dist[v][u] = w  # As the graph is undirected, set both directions

# Apply Floyd-Warshall to compute the shortest paths
print("Distance Matrix before link failure:")
print_distance_matrix(dist)

# Calculate the shortest paths using Floyd-Warshall
dist_after_floyd = floyd_warshall(n, [row[:] for row in dist])  # Clone the matrix to preserve original
print("\nShortest paths before link failure (from A to F):", dist_after_floyd[0][5])

# Simulate the link failure between Router B (1) and Router D (3)
simulate_link_failure(dist, 1, 3)

# Apply Floyd-Warshall again to update the shortest paths after the failure
dist_after_failure = floyd_warshall(n, [row[:] for row in dist])  # Re-run Floyd-Warshall

print("\nDistance Matrix after link failure:")
print_distance_matrix(dist_after_failure)

# Shortest path from Router A (0) to Router F (5) after link failure
print("\nShortest path from Router A to Router F after link failure:", dist_after_failure[0][5])
```

3.

```
# Floyd-Warshall Algorithm to calculate shortest paths
def floyd_warshall(n, dist):
    # Applying the Floyd-Warshall algorithm
```

```
    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

# Display the distance matrix in a readable format
def print_distance_matrix(dist):
    for row in dist:
        print(row)

# Function to count neighboring cities within the given distance threshold
def count_neighbors(n, dist, threshold):
    neighbors = [0] * n
    for i in range(n):
        for j in range(n):
            if dist[i][j] <= threshold:
                neighbors[i] += 1
    return neighbors

# Test Case 1: Shortest path calculation between cities
def test_case_1():
    # Number of cities (n = 5)
    n = 5
    edges = [
        [0, 1, 2],  # City 0 to City 1: distance 2
        [0, 4, 8],  # City 0 to City 4: distance 8
        [1, 2, 3],  # City 1 to City 2: distance 3
        [1, 4, 2],  # City 1 to City 4: distance 2
        [2, 3, 1],  # City 2 to City 3: distance 1
        [3, 4, 1],  # City 3 to City 4: distance 1
    ]
    distanceThreshold = 2

    # Initialize the distance matrix
    dist = [[float('inf')] * n for _ in range(n)]
    for i in range(n):
        dist[i][i] = 0  # Distance from a city to itself is 0

    # Populate the distance matrix with the edges information
    for u, v, w in edges:
        dist[u][v] = w
        dist[v][u] = w  # The graph is undirected

    print("Distance Matrix before applying Floyd-Warshall:")
    print_distance_matrix(dist)

    # Apply Floyd-Warshall to find the shortest paths
```

```
    dist = floyd_warshall(n, dist)

    print("\nDistance Matrix after applying Floyd-Warshall:")
    print_distance_matrix(dist)

    # Identify neighboring cities at the given distance threshold
    neighbors = count_neighbors(n, dist, distanceThreshold)

    # Find the city with the greatest number of neighbors
    city_with_max_neighbors = max(range(n), key=lambda x: neighbors[x])

    print(f"\nCity {city_with_max_neighbors} has the greatest number of neighbors within the threshold.")

    return dist

# Test Case 2: Shortest path from C to A (for given test case)
def test_case_2():
    # Number of cities (5 cities: A, B, C, D, E)
    n = 5
    edges = [
        [1, 0, 2],  # B to A: 2
        [0, 2, 3],  # A to C: 3
        [2, 3, 1],  # C to D: 1
        [3, 0, 6],  # D to A: 6
        [2, 1, 7],  # C to B: 7
    ]

    # Initialize the distance matrix
    dist = [[float('inf')] * n for _ in range(n)]
    for i in range(n):
        dist[i][i] = 0  # Distance to itself is zero

    # Add edges to the matrix
    for u, v, w in edges:
        dist[u][v] = w
        dist[v][u] = w  # Undirected graph

    # Apply Floyd-Warshall algorithm
    dist = floyd_warshall(n, dist)

    # Shortest path from C (2) to A (0)
    print("\nShortest path from C to A:", dist[2][0])
    return dist

# Test Case 3: Shortest path from E to C
def test_case_3():
    n = 5
    edges = [
        [4, 2, 2],  # E to C: 2
```

```
        [0, 1, 4],  # A to B: 4
        [1, 2, 1],  # B to C: 1
        [1, 4, 6],  # B to E: 6
        [4, 0, 1],  # E to A: 1
        [0, 3, 5],  # A to D: 5
        [3, 4, 2],  # D to E: 2
        [4, 3, 4],  # E to D: 4
        [3, 2, 1],  # D to C: 1
        [2, 3, 3],  # C to D: 3
    ]

    # Initialize the distance matrix
    dist = [[float('inf')] * n for _ in range(n)]
    for i in range(n):
        dist[i][i] = 0  # Distance to itself is zero

    # Add edges to the matrix
    for u, v, w in edges:
        dist[u][v] = w
        dist[v][u] = w  # Undirected graph

    # Apply Floyd-Warshall algorithm
    dist = floyd_warshall(n, dist)

    # Shortest path from E (4) to C (2)
    print("\nShortest path from E to C:", dist[4][2])
    return dist

# Run the test cases
print("\nTest Case 1 Output (City with max neighbors within threshold):")
dist_1 = test_case_1()

print("\nTest Case 2 Output (Shortest path from C to A):")
test_case_2()

print("\nTest Case 3 Output (Shortest path from E to C):")
test_case_3()
```

4.

```
# Function to construct the Optimal Binary Search Tree
def optimal_bst(keys, freq, n):
    # cost[i][j] will store the minimum cost for keys i to j
    cost = [[0] * n for _ in range(n)]

    # root[i][j] will store the index of the root for the keys from i to j
    root = [[0] * n for _ in range(n)]

    # We need to calculate the sum of frequencies for keys i to j
```

```python
    sum_freq = [[0] * n for _ in range(n)]

    # Initialize sum_freq with frequencies
    for i in range(n):
        sum_freq[i][i] = freq[i]
        for j in range(i + 1, n):
            sum_freq[i][j] = sum_freq[i][j - 1] + freq[j]

    # Fill the DP tables
    for length in range(1, n + 1):  # length of the subarray (1 to n)
        for i in range(n - length + 1):  # i is the starting point of the subarray
            j = i + length - 1  # j is the endpoint of the subarray
            if length == 1:
                cost[i][j] = freq[i]
                root[i][j] = i
            else:
                min_cost = float('inf')
                # Try every root between i and j
                for r in range(i, j + 1):
                    # Left subtree: cost[i][r-1], Right subtree: cost[r+1][j]
                    left_cost = cost[i][r - 1] if r > i else 0
                    right_cost = cost[r + 1][j] if r < j else 0
                    total_cost = left_cost + right_cost + sum_freq[i][j]
                    if total_cost < min_cost:
                        min_cost = total_cost
                        root[i][j] = r
                cost[i][j] = min_cost

    # Print the cost and root matrices
    print("Cost Table:")
    for row in cost:
        print(row)

    print("\nRoot Table:")
    for row in root:
        print(row)

    return cost[0][n-1]  # The cost of the optimal BST for all keys

# Test case 1: Keys = {10, 12}, Frequencies = {34, 50}
keys1 = [10, 12]
freq1 = [34, 50]
n1 = len(keys1)
print("Test case 1 Output:")
result1 = optimal_bst(keys1, freq1, n1)
print("\nOptimal BST cost:", result1)

# Test case 2: Keys = {10, 12, 20}, Frequencies = {34, 8, 50}
keys2 = [10, 12, 20]
```

```
freq2 = [34, 8, 50]
n2 = len(keys2)
print("\nTest case 2 Output:")
result2 = optimal_bst(keys2, freq2, n2)
print("\nOptimal BST cost:", result2)

# Test case 3: Keys = {A, B, C, D}, Frequencies = {0.1, 0.2, 0.4, 0.3}
keys3 = ['A', 'B', 'C', 'D']
freq3 = [0.1, 0.2, 0.4, 0.3]
n3 = len(keys3)
print("\nTest case 3 Output:")
result3 = optimal_bst(keys3, freq3, n3)
print("\nOptimal BST cost:", result3)
```

5.

```
def optimal_bst(keys, freq, n):
    # Initialize matrices
    cost = [[0] * n for _ in range(n)]
    root = [[0] * n for _ in range(n)]

    # Sum of frequencies for keys i to j
    sum_freq = [[0] * n for _ in range(n)]

    # Initialize sum_freq: sum of frequencies from i to j
    for i in range(n):
        sum_freq[i][i] = freq[i]
        for j in range(i + 1, n):
            sum_freq[i][j] = sum_freq[i][j - 1] + freq[j]

    # Fill the cost and root tables
    for length in range(1, n + 1):  # length of subarray (1 to n)
        for i in range(n - length + 1):  # starting point i
            j = i + length - 1  # endpoint j
            if length == 1:
                cost[i][j] = freq[i]
                root[i][j] = i
            else:
                min_cost = float('inf')
                # Try every root between i and j
                for r in range(i, j + 1):
                    # Left subtree cost (i to r-1), Right subtree cost (r+1 to j)
                    left_cost = cost[i][r - 1] if r > i else 0
                    right_cost = cost[r + 1][j] if r < j else 0
                    total_cost = left_cost + right_cost + sum_freq[i][j]
                    if total_cost < min_cost:
                        min_cost = total_cost
                        root[i][j] = r
                cost[i][j] = min_cost
```

```
    # Print the cost and root matrices
    print("Cost Table:")
    for row in cost:
        print(row)

    print("\nRoot Table:")
    for row in root:
        print(row)

    return cost[0][n-1]  # The cost of the optimal BST for all keys

# Test case 1: Keys = {10, 12}, Frequencies = {34, 50}
keys1 = [10, 12]
freq1 = [34, 50]
n1 = len(keys1)
print("Test case 1 Output:")
result1 = optimal_bst(keys1, freq1, n1)
print("\nOptimal BST cost:", result1)

# Test case 2: Keys = {10, 12, 20}, Frequencies = {34, 8, 50}
keys2 = [10, 12, 20]
freq2 = [34, 8, 50]
n2 = len(keys2)
print("\nTest case 2 Output:")
result2 = optimal_bst(keys2, freq2, n2)
print("\nOptimal BST cost:", result2)

# Test case 3: Keys = {10, 12, 16, 21}, Frequencies = {4, 2, 6, 3}
keys3 = [10, 12, 16, 21]
freq3 = [4, 2, 6, 3]
n3 = len(keys3)
print("\nTest case 3 Output:")
result3 = optimal_bst(keys3, freq3, n3)
print("\nOptimal BST cost:", result3)
```

6.

```
def catMouseGame(graph):
    # Memoization: dp[mouse_pos][cat_pos][turn] = result
    # turn = 0 -> Mouse's turn, turn = 1 -> Cat's turn
    dp = {}

    def dfs(mouse, cat, turn):
        # Base cases
        if mouse == 0:
            return 1  # Mouse wins (reaches the hole)
        if mouse == cat:
            return 2  # Cat wins (catches the mouse)

        # If we have already calculated this state, return it
```

```python
        if (mouse, cat, turn) in dp:
            return dp[(mouse, cat, turn)]

        # Current player tries all possible moves
        if turn == 0:  # Mouse's turn
            result = 2  # Assume worst-case: Cat wins
            for next_mouse in graph[mouse]:
                if next_mouse == cat:  # Mouse can't move to the same place as the Cat
                    continue
                result = min(result, dfs(next_mouse, cat, 1))  # Switch turn to Cat
            dp[(mouse, cat, turn)] = result
            return result
        else:  # Cat's turn
            result = 1  # Assume worst-case: Mouse wins
            for next_cat in graph[cat]:
                if next_cat == 0:  # Cat can't move to the hole
                    continue
                result = max(result, dfs(mouse, next_cat, 0))  # Switch turn to Mouse
            dp[(mouse, cat, turn)] = result
            return result

    # Start the game with mouse at 1 and cat at 2, Mouse's turn (turn = 0)
    return dfs(1, 2, 0)

# Example 1:
graph1 = [[2,5],[3],[0,4,5],[1,4,5],[2,3],[0,2,3]]
print(catMouseGame(graph1))  # Output: 0 (draw)

# Example 2:
graph2 = [[1,3],[0],[3],[0,2]]
print(catMouseGame(graph2))  # Output: 1 (Mouse wins)
```

7.

```python
import heapq

def maxProbability(n, edges, succProb, start, end):
    # Graph adjacency list, where graph[i] = [(neighbor, probability), ...]
    graph = [[] for _ in range(n)]

    for i, (a, b) in enumerate(edges):
        graph[a].append((b, succProb[i]))
        graph[b].append((a, succProb[i]))

    # Max-heap to store (probability, node) and start with node `start`
    max_heap = [(-1.0, start)]  # We use negative probability because heapq is a min-heap by default
    prob = [0.0] * n  # Stores maximum probability to each node
    prob[start] = 1.0  # Probability to start node is 1

    while max_heap:
```

```
        current_prob, node = heapq.heappop(max_heap)
        current_prob = -current_prob  # Convert back to positive probability

        # If we reached the end, return the probability
        if node == end:
            return current_prob

        # Explore the neighbors
        for neighbor, edge_prob in graph[node]:
            new_prob = current_prob * edge_prob
            if new_prob > prob[neighbor]:  # If a better path is found
                prob[neighbor] = new_prob
                heapq.heappush(max_heap, (-new_prob, neighbor))  # Push the updated probability into the
heap

    # If we reach here, there's no path from start to end
    return 0.0

# Example 1:
n1 = 3
edges1 = [[0, 1], [1, 2], [0, 2]]
succProb1 = [0.5, 0.5, 0.2]
start1, end1 = 0, 2
print(maxProbability(n1, edges1, succProb1, start1, end1))  # Output: 0.25000

# Example 2:
n2 = 3
edges2 = [[0, 1], [1, 2], [0, 2]]
succProb2 = [0.5, 0.5, 0.3]
start2, end2 = 0, 2
print(maxProbability(n2, edges2, succProb2, start2, end2))  # Output: 0.30000
```

8.

```
import math

def uniquePaths(m, n):
    # Calculate C(m+n-2, m-1), which is the number of unique paths
    return math.comb(m + n - 2, m - 1)

# Example 1:
m1, n1 = 3, 7
print(f"Unique paths for m={m1}, n={n1}: {uniquePaths(m1, n1)}")  # Output: 28

# Example 2:
m2, n2 = 3, 2
print(f"Unique paths for m={m2}, n={n2}: {uniquePaths(m2, n2)}")  # Output: 3
```

9.

```
from collections import Counter
```

```
def numIdenticalPairs(nums):
    # Step 1: Count the frequency of each element in nums
    freq = Counter(nums)

    # Step 2: Calculate the number of good pairs using the formula C(k, 2) = k * (k-1) / 2
    good_pairs = 0
    for count in freq.values():
        if count > 1:
            good_pairs += count * (count - 1) // 2

    return good_pairs

# Example 1:
nums1 = [1, 2, 3, 1, 1, 3]
print(f"Good pairs count for nums1: {numIdenticalPairs(nums1)}")  # Output: 4

# Example 2:
nums2 = [1, 1, 1, 1]
print(f"Good pairs count for nums2: {numIdenticalPairs(nums2)}")  # Output: 6
```

10.

```
import heapq

def dijkstra(n, graph, start, distanceThreshold):
    # Initialize distances to infinity
    distances = [float('inf')] * n
    distances[start] = 0
    min_heap = [(0, start)]  # (distance, node)

    while min_heap:
        current_distance, node = heapq.heappop(min_heap)

        # If the current distance is greater than the threshold, stop
        if current_distance > distanceThreshold:
            continue

        # Explore the neighbors of the current node
        for neighbor, weight in graph[node]:
            distance = current_distance + weight
            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(min_heap, (distance, neighbor))

    return distances

def findCity(n, edges, distanceThreshold):
    # Step 1: Build the graph as an adjacency list
    graph = {i: [] for i in range(n)}
```

```
    for u, v, w in edges:
        graph[u].append((v, w))
        graph[v].append((u, w))

    # Step 2: Find the city with the smallest number of reachable cities
    min_reachable = float('inf')
    city_with_min_reachable = -1

    for city in range(n):
        distances = dijkstra(n, graph, city, distanceThreshold)
        # Count how many cities are reachable
        reachable_count = sum(1 for distance in distances if distance <= distanceThreshold)

        # Check if this city should be the new answer
        if reachable_count <= min_reachable:
            min_reachable = reachable_count
            city_with_min_reachable = city

    return city_with_min_reachable

# Example 1
n1 = 4
edges1 = [[0,1,3],[1,2,1],[1,3,4],[2,3,1]]
distanceThreshold1 = 4
print(f"Output for example 1: {findCity(n1, edges1, distanceThreshold1)}")  # Expected output: 3

# Example 2
n2 = 5
edges2 = [[0,1,2],[0,4,8],[1,2,3],[1,4,2],[2,3,1],[3,4,1]]
distanceThreshold2 = 2
print(f"Output for example 2: {findCity(n2, edges2, distanceThreshold2)}")  # Expected output: 0
```

11.

```
import heapq

def networkDelayTime(times, n, k):
    # Step 1: Build the graph as an adjacency list
    graph = {i: [] for i in range(1, n + 1)}
    for u, v, w in times:
        graph[u].append((v, w))

    # Step 2: Initialize distances and the priority queue for Dijkstra's algorithm
    distances = {i: float('inf') for i in range(1, n + 1)}
    distances[k] = 0
    pq = [(0, k)]  # (distance, node)

    # Step 3: Perform Dijkstra's algorithm
    while pq:
        current_distance, node = heapq.heappop(pq)
```

```python
        # If the current distance is already greater than the shortest known distance, skip it
        if current_distance > distances[node]:
            continue

        # Explore all the neighbors of the current node
        for neighbor, weight in graph[node]:
            distance = current_distance + weight
            # If a shorter path to the neighbor is found, update the distance
            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(pq, (distance, neighbor))

    # Step 4: Find the maximum distance in the distances dictionary
    max_distance = max(distances.values())

    # If there are any nodes that are still unreachable (distance = inf), return -1
    if max_distance == float('inf'):
        return -1

    return max_distance

# Example 1:
times1 = [[2,1,1],[2,3,1],[3,4,1]]
n1 = 4
k1 = 2
print(networkDelayTime(times1, n1, k1))  # Output: 2

# Example 2:
times2 = [[1,2,1]]
n2 = 2
k2 = 1
print(networkDelayTime(times2, n2, k2))  # Output: 1

# Example 3:
times3 = [[1,2,1]]
n3 = 2
k3 = 2
print(networkDelayTime(times3, n3, k3))  # Output: -1
```