59.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]], distance Threshold = 4 Output: 3

Aim: Aim is to find the shortest paths between all pairs of vertices in a weighted graph with positive or negative edge weights.

Algorithm:

step1: Initialize the distance matrix.

step2: Apply Floyd's Algorithm to find the shortest paths.

step3: Display the distance matrix before and after the algorithm.

Step4: Display the distance matrix before and after the algorithm.

Program:

```
def floyd_warshall(n, edges):
    inf = float('inf')
    dist = [[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
    print("Distance matrix before applying Floyd's Algorithm:")
    for row in dist:
        print(row)
    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]
    print("\nDistance matrix after applying Floyd's Algorithm:")
    for row in dist:
        print(row)

    for i in range(n):
        for j in range(n):
            if i != j:
                print(f"The shortest path from {i} to {j} is {dist[i][j]}")
    return dist
n = 4
edges = [[0,1,3],[1,2,1],[1,3,4],[2,3,1]]
distance_threshold = 4

shortest_paths = floyd_warshall(n, edges)
```

Input:

```
n = 4
edges = [[0,1,3],[1,2,1],[1,3,4],[2,3,1]]
distance_threshold = 4
```

Output:

Output

```
Distance matrix before applying Floyd's Algorithm:
[0, 3, inf, inf]
[inf, 0, 1, 4]
[inf, inf, 0, 1]
[inf, inf, inf, 0]

Distance matrix after applying Floyd's Algorithm:
[0, 3, 4, 5]
[inf, 0, 1, 2]
[inf, inf, 0, 1]
[inf, inf, inf, 0]
The shortest path from 0 to 1 is 3
The shortest path from 0 to 2 is 4
The shortest path from 0 to 3 is 5
The shortest path from 1 to 0 is inf
The shortest path from 1 to 2 is 1
The shortest path from 1 to 3 is 2
The shortest path from 2 to 0 is inf
The shortest path from 2 to 1 is inf
The shortest path from 2 to 3 is 1
The shortest path from 3 to 0 is inf
The shortest path from 3 to 1 is inf
The shortest path from 3 to 2 is inf

=== Code Execution Successful ===
```

Time Complexity: T(n)=O(n^3)

Result: When you run this code, it will display the initial distance matrix, the final distance matrix after applying Floyd's Algorithm, and the shortest paths between all pairs of cities.

60.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

Aim: To implement Floyd's Algorithm to calculate the shortest paths between all pairs of routers and simulate a scenario where a link between two routers fails.

Algorithm:

Step1: Create a 2D array to store the shortest distance between every pair of routers.

Step2: Use three nested loops to update the distance matrix by considering all possible paths between each pair of routers.

Step3: Update the distance matrix to reflect the failure of the link between Router B and Router D.

Step4: Calculate and display the shortest path from Router A to Router F before and after the link failure.

Program:

```
INF = float('inf')
def initialize_distance_matrix():
    dist = [
        [ 0,  2, INF,  1, INF, INF],
        [ 2,  0,  3,  2, INF, INF],
        [INF,  3,  0,  4,  5, INF],
        [ 1,  2,  4,  0,  3,  2],
        [INF, INF,  5,  3,  0,  2],
        [INF, INF, INF,  2,  2,  0]
    ]
    return dist
def floyd_warshall(dist):
    n = len(dist)
    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 simulate_link_failure(dist):
    dist[1][3] = INF
    dist[3][1] = INF
    return dist
def display_shortest_path(dist, from_router, to_router):
    routers = ['A', 'B', 'C', 'D', 'E', 'F']
    from_index = routers.index(from_router)
    to_index = routers.index(to_router)
    return dist[from_index][to_index]
def main():
    dist = initialize_distance_matrix()
    dist_before_failure = floyd_warshall([row[:] for row in dist])
    print("Shortest path from Router A to Router F before link failure:", display_shortest_path(dist_before_failure, 'A', 'F'
        ))
    dist_after_failure = simulate_link_failure(dist_before_failure)
    dist_after_failure = floyd_warshall(dist_after_failure)
    print("Shortest path from Router A to Router F after link failure:", display_shortest_path(dist_after_failure, 'A', 'F'))
if __name__ == "__main__":
    main()

```

Input:

The program does not require any user input as the distance matrix is initialized within the code.

Output:

```
Shortest path from Router A to Router F before link failure: 3
Shortest path from Router A to Router F after link failure: 3

=== Code Execution Successful ===
```

Time complexity: T(n): O(n3)

Result: The example input graph = [[2,5],[3], [0,4,5], [1,4,5],[2,3],[0,2,3]] returns 0, indicating that the game is a draw if both players play optimally.

61.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

Aim: To implement Floyd's Algorithm to find the shortest paths between all pairs of cities, and display the distance matrix before and after applying the algorithm.

Algorithm:

step1: Create an ‘n x n ’matrix where ‘n’ is the number of cities, and initialize the distances.

step2: Update the matrix with the given edges

step3: Use three nested loops to update the distance matrix with the shortest paths.

Step4: Print the distance matrix before and after applying the algorithm.

Program:

```python
def floyd_warshall(n, edges, distance_threshold):
    inf = float('inf')
    dist = [[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
    print("Distance matrix before applying Floyd's Algorithm:")
    for row in dist:
        print(row)
    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]
    print("\nDistance matrix after applying Floyd's Algorithm:")
    for row in dist:
        print(row)
    print("\nShortest paths within the distance threshold:")
    for i in range(n):
        for j in range(n):
            if i != j:
                if dist[i][j] <= distance_threshold:
                    print(f"The shortest path from {i} to {j} is {dist[i][j]}")
                else:
                    print(f"No path from {i} to {j} within the distance threshold")
n = 5
edges = [[0,1,2],[0,4,8],[1,2,3],[1,4,2],[2,3,1],[3,4,1]]
distance_threshold = 2
floyd_warshall(n, edges, distance_threshold)
```

Input:

```python
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:

```
Output
Distance matrix before applying Floyd's Algorithm:
[0, 2, inf, inf, 8]
[inf, 0, 3, inf, 2]
[inf, inf, 0, 1, inf]
[inf, inf, inf, 0, 1]
[inf, inf, inf, inf, 0]

Distance matrix after applying Floyd's Algorithm:
[0, 2, 5, 6, 4]
[inf, 0, 3, 4, 2]
[inf, inf, 0, 1, 2]
[inf, inf, inf, 0, 1]
[inf, inf, inf, inf, 0]

Shortest paths within the distance threshold:
The shortest path from 0 to 1 is 2
No path from 0 to 2 within the distance threshold
No path from 0 to 3 within the distance threshold
No path from 0 to 4 within the distance threshold
No path from 1 to 0 within the distance threshold
No path from 1 to 2 within the distance threshold
No path from 1 to 3 within the distance threshold
```

Time Complexity: T(n)=O(n^3)

Result: This program effectively demonstrates how Floyd's Algorithm works and how it can be used to find and display the shortest pa ths between all pairs of cities.

62. 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

Aim: To implement the Optimal Binary Search Tree (OBST) algorithm for a given set of keys and their frequencies, and display the resulting OBST and its cost.

Algorithm:

step1: **Initialize Matrices**: Create cost and root matrices.

step2: **Compute Costs**: Use dynamic programming to fill in the cost matrix.

step3: **Construct the OBST**: Use the root matrix to determine the structure of the OBST.

Step4: **Display Results**: Print the cost and root matrices and the resulting OBST.

Program:

```python
def optimal_bst(keys, freq, n):
    cost = [[0 for x in range(n)] for y in range(n)]
    root = [[0 for x in range(n)] for y in range(n)]

    for i in range(n):
        cost[i][i] = freq[i]
        root[i][i] = 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
    print("Cost Matrix:")
    for row in cost:
        print(row)
    print("\nRoot Matrix:")
    for row in root:
        print(row)
    return cost[0][n - 1], root
def print_obst(root, keys, i, j, parent, is_left):
    if i <= j:
        root_index = root[i][j]
        root_key = keys[root_index]
        if parent is None:
            print(f"Root: {root_key}")
        else:
            direction = "left" if is_left else "right"
            print(f"{root_key} is the {direction} child of {parent}")
        print_obst(root, keys, i, root_index - 1, root_key, True)
        print_obst(root, keys, root_index + 1, j, root_key, False)
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(f"\nThe cost of the optimal BST is: {cost}")
```

Input:

```
keys = ['A', 'B', 'C', 'D']
freq = [0.1, 0.2, 0.4, 0.3]
```

Output:

Output

```
Cost Matrix:
[0.1, 0.4, 1.1, 1.7]
[0, 0.2, 0.8, 1.4000000000000001]
[0, 0, 0.4, 1.0]
[0, 0, 0, 0.3]

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

The cost of the optimal BST is: 1.7

The structure of the optimal BST is:
Root: C
B is the left child of C
A is the left child of B
D is the right child of C
```

Time Complexity: T(n)=O(n^3)
Result: This program effectively demonstrates how to construct an Optimal Binary Search Tree using the OBST algorithm and display its cost and structure.

63. 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

Aim: To implement the Optimal Binary Search Tree (OBST) algorithm for a given set of keys and their frequencies, and display the resulting OBST, its cost, and root matrix.
Algorithm:
step1: **Initialize Matrices**: Create cost and root matrices.
step2: **Compute Costs**: Use dynamic programming to fill in the cost matrix.
step3: **Construct the OBST**: Use the root matrix to determine the structure of the OBST.
Step4: **Display Results**: Print the cost and root matrices and the resulting OBST.
Program:

```python
def optimal_bst(keys, freq, n):
    cost = [[0 for x in range(n)] for y in range(n)]
    root = [[0 for x in range(n)] for y in range(n)]
    for i in range(n):
        cost[i][i] = freq[i]
        root[i][i] = 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
    print("Cost Matrix:")
    for row in cost:
        print(row)
    print("\nRoot Matrix:")
    for row in root:
        print(row)
    return cost[0][n - 1], root
def print_obst(root, keys, i, j, parent, is_left):
    if i <= j:
        root_index = root[i][j]
        root_key = keys[root_index]
        if parent is None:
            print(f"Root: {root_key}")
        else:
            direction = "left" if is_left else "right"
            print(f"{root_key} is the {direction} child of {parent}")
        print_obst(root, keys, i, root_index - 1, root_key, True)
        print_obst(root, keys, root_index + 1, j, root_key, False)
keys = [10, 12, 16, 21]
freq = [4, 2, 6, 3]
n = len(keys)
cost, root = optimal_bst(keys, freq, n)
print(f"\nThe cost of the optimal BST is: {cost}")
print("\nThe structure of the optimal BST is:")
```

Input:

```
keys = [10, 12, 16, 21]
freq = [4, 2, 6, 3]
```

Output:

```
Output
Cost Matrix:
[4, 8, 20, 26]
[0, 2, 10, 16]
[0, 0, 6, 12]
[0, 0, 0, 3]

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

The cost of the optimal BST is: 26

The structure of the optimal BST is:
Root: 16
10 is the left child of 16
12 is the right child of 10
21 is the right child of 16
```

Time Complexity: T(n)=O(n^3)

Result: This program effectively demonstrates how to construct an Optimal Binary Search Tree using the OBST algorithm and display its cost and structure.

64. 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

Aim: To solve this problem, we can model it as a game theory problem with a BFS (Breadth-First Search) approach.

Algorithm:

step1: Initialize the DP table with the base cases

step2: Use BFS to propagate the results from the base cases to other states.

step3: Return the result of the initial state where the mouse starts at node 1 and the cat starts at node 2.

Program:

```python
from collections import deque
def cat_mouse_game(graph):
    n = len(graph)
    DRAW, MOUSE_WIN, CAT_WIN = 0, 1, 2
    dp = [[[DRAW] * 2 for _ in range(n)] for _ in range(n)]
    for i in range(n):
        for t in range(2):
            dp[0][i][t] = MOUSE_WIN
            if i > 0:
                dp[i][i][t] = CAT_WIN
    queue = deque()
    for i in range(1, n):
        for t in range(2):
            if i != 0:
                queue.append((0, i, t, MOUSE_WIN))
                queue.append((i, i, t, CAT_WIN))
    while queue:
        mouse, cat, turn, result = queue.popleft()
        dp[mouse][cat][turn] = result
        if turn == 1:
            prev_turn = 0
            for prev_mouse in graph[mouse]:
                if dp[prev_mouse][cat][prev_turn] == DRAW:
                    if result == MOUSE_WIN:
                        dp[prev_mouse][cat][prev_turn] = MOUSE_WIN
                        queue.append((prev_mouse, cat, prev_turn, MOUSE_WIN))
                    elif all(dp[next_mouse][cat][1] == CAT_WIN for next_mouse in graph[prev_mouse]):
                        dp[prev_mouse][cat][prev_turn] = CAT_WIN
                        queue.append((prev_mouse, cat, prev_turn, CAT_WIN))
        else:
            prev_turn = 1
            for prev_cat in graph[cat]:
                if prev_cat == 0:
                    continue
                if dp[mouse][prev_cat][prev_turn] == DRAW:
                    if result == CAT_WIN:
                        dp[mouse][prev_cat][prev_turn] = CAT_WIN
                        queue.append((mouse, prev_cat, prev_turn, CAT_WIN))
                    elif all(dp[mouse][next_cat][0] == MOUSE_WIN for next_cat in graph[prev_cat] if next_cat != 0):
                        dp[mouse][prev_cat][prev_turn] = MOUSE_WIN
                        queue.append((mouse, prev_cat, prev_turn, MOUSE_WIN))
    return dp[1][2][0]
graph = [[2,5],[3],[0,4,5],[1,4,5],[2,3],[0,2,3]]
result = cat_mouse_game(graph)
```

Input:

```
graph = [[2,5],[3],[0,4,5],[1,4,5],[2,3],[0,2,3]]
```

Output:

```
Output
Cost Matrix:
[4, 8, 20, 26]
[0, 2, 10, 16]
[0, 0, 6, 12]
[0, 0, 0, 3]

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

The cost of the optimal BST is: 26

The structure of the optimal BST is:
Root: 16
10 is the left child of 16
12 is the right child of 10
21 is the right child of 16
```

Time Complexity: T(n)=O(n^3)
Result: Finally, we return the result for the initial state (mouse at node 1, cat at node 2, mouse's turn).

65.**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 succ Prob[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.**

Aim: The aim is to compute the maximum probability of successfully traveling from a start node to an end node in an undirected graph, given the probabilities of successfully traversing each edge.

Algorithm:

1. the edge list into an adjacency list representation of the graph where each node points to its neighbors with the associated success probabilities of traversing the connecting edges.

2. Implement a priority queue-based algorithm to determine the maximum probability path.
3. After processing all reachable nodes, return the maximum probability of reaching the end node from the start node. If the end node is unreachable, return 0.

Program:

```python
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] * n
    max_prob[start] = 1
    heap = [(-1, start)]
    while heap:
        prob, node = heapq.heappop(heap)
        prob = -prob
        if node == end:
            return prob
        for neighbor, edge_prob in graph[node]:
            new_prob = prob * edge_prob
            if new_prob > max_prob[neighbor]:
                max_prob[neighbor] = new_prob
                heapq.heappush(heap, (-new_prob, neighbor))
    return 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))
```

Input:

```
n = 3
edges = [[0, 1], [1, 2], [0, 2]]
succProb = [0.5, 0.5, 0.2]
start = 0
end = 2
```

Output:

Time Complexity: **O ((V + E) log V)**

Result : The probability of reaching the end node after processing all reachable nodes is returned. If the end node is never reached, return **0.0**.

66.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.

Aim: To determine the number of distinct paths a robot can take to move from the top-left corner to the bottom-right corner of an **m x n** grid, given that it can only move right or down.

Algorithm:

1. **Calculate the Total Steps Required:** The robot needs to make a total of **m - 1** down moves and **n - 1** right moves, totaling **(m - 1) + (n - 1)** moves.
2. **Choose Down Moves:** Out of the total moves, choose **m - 1** positions for down moves (the remaining will be right moves). This is a combination problem where we calculate the binomial coefficient ((m−1)+(n−1)m−1)(*m*−1(*m*−1)+(*n*−1)).
3. **Compute the Binomial Coefficient:** Use the formula (m+n−2)!(m−1)!·(n−1)!(*m*−1)!·(*n*−1)!(*m*+*n*−2)! to calculate the number of unique paths.

Program:

```
import math
def uniquePaths(m, n):
    return math.comb(m + n - 2, m - 1)
m = 3
n = 7
print(uniquePaths(m, n))

```

Input:

```
m = 3
n = 7
print(uniquePaths(m, n))
```

Output:

```
Output
28
```

Time Complexity: O (min(m ,n))

Result: The program calculates that there are **28** unique paths for a robot to move from the top-left to the bottom-right corner of a **3 x 7** grid.

67.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

Aim:

To determine the number of good pairs (i,j)(*i*,*j*) in an array such that nums[i]=nums[j]nums[*i*]=nums[*j*] and i<*j**i*<*j*.

Algorithm:

1. **Count Occurrences:** Use a dictionary to count the occurrences of each integer in the array.
2. **Compute Good Pairs:** For each unique integer with a count greater than 1, compute the number of good pairs using the combination formula (count2)(2*count*), where **count** is the number of occurrences of the integer.
3. **Sum All Good Pairs:** Accumulate the results to get the total number of good pairs

Program:

```python
def numIdenticalPairs(nums):
    freq_map = {}
    good_pairs_count = 0
    for num in nums:
        if num in freq_map:
            good_pairs_count += freq_map[num]
            freq_map[num] += 1
        else:
            freq_map[num] = 1
    return good_pairs_count
nums = [1, 2, 3, 1, 1, 3]
print(numIdenticalPairs(nums))

```

Input:

```python
nums = [1, 2, 3, 1, 1, 3]
print(numIdenticalPairs(nums))
```

Output:

```
Output
4
```

Time Complexity: T(n)=O(n)

Result **:The program calculates that there are 4 good pairs in the given array.**

68. 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.

Aim:To identify the city with the smallest number of cities that can be reached within a specified distance threshold. If multiple cities have the same number of reachable cities, return the city with the greatest number.

Algorithm:

1. **Compute Shortest Paths:** Use the Floyd-Warshall algorithm to compute the shortest paths between all pairs of cities.
2. **Count Reachable Cities:** For each city, count the number of cities reachable within the **distance Threshold**.
3. **Select Optimal City:** Identify the city with the smallest number of reachable cities and in case of a tie, choose the city with the highest number of reachable cities.

Program:

```python
def findTheCity(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]
    min_count = float('inf')
    result_city = -1
    for i in range(n):
        count = sum(1 for j in range(n) if dist[i][j] <= distanceThreshold)
        if count < min_count or (count == min_count and i > result_city):
            min_count = count
            result_city = i

    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))
```

Input:

```python
n = 4
edges = [[0,1,3],[1,2,1],[1,3,4],[2,3,1]]
distanceThreshold = 4
print(findTheCity(n, edges, distanceThreshold))
```

Output:

Time Complexity: $O(n2\log n+n\cdot E)$.

Result: The program calculates that the city with the smallest number of reachable cities within the distance threshold is city **3**.

69. 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.

Aim: The aim is to determine the minimum time required for a signal, starting from a given node **k** in a directed graph, to reach all other nodes. If it is impossible for all nodes to receive the signal, the function should return **-1**.

Algorithm:

1. **Run Dijkstra's Algorithm:** Use Dijkstra's algorithm to compute the shortest path from the start node k*k* to all other nodes in the network.
2. **Check Reachability:** Determine the maximum time among all shortest paths. If any node is unreachable (i.e., its distance is infinite), return -1.
3. **Return the Maximum Time:** The result is the maximum time required to reach any node, as this represents the time it takes for the last node to receive the signal.

Program:

```python
import heapq
def network_delay_time(times, n, k):
    graph = {i: [] for i in range(1, n + 1)}
    for u, v, w in times:
        graph[u].append((v, w))
    min_heap = [(0, k)]
    visited = {}
    while min_heap:
        time, node = heapq.heappop(min_heap)
        if node in visited:
            continue
        visited[node] = time
        for neighbor, weight in graph[node]:
            if neighbor not in visited:
                heapq.heappush(min_heap, (time + weight, neighbor))
    if len(visited) == n:
        return max(visited.values())
    return -1
times1 = [[2, 1, 1], [2, 3, 1], [3, 4, 1]]
n1, k1 = 4, 2
print(network_delay_time(times1, n1, k1))
```

Input:

```python
times1 = [[2, 1, 1], [2, 3, 1], [3, 4, 1]]
n1, k1 = 4, 2
print(network_delay_time(times1, n1, k1))
```

Output:

```
Output
2
```

Time Complexity: T(n)= **O ((E + V) log V)**

Result: For each input, compute the shortest time for the signal to reach all nodes or determine if it's impossible.

vv