main.py

Share Run

```python
import numpy as np
def floyd_warshall(n, edges):

    distance = np.full((n, n), np.inf)
    for i in range(n):
        distance[i][i] = 0
    for u, v, w in edges:
        distance[u][v] = w
        distance[v][u] = w

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

    for k in range(n):
        for i in range(n):
            for j in range(n):
                if distance[i][j] > distance[i][k] + distance[k][j]:
                    distance[i][j] = distance[i][k] + distance[k][j]

    print("Distance matrix after applying Floyd's Algorithm:")
    print(distance)

```

Output

Cle

```
Distance matrix before applying Floyd's Algorithm:
[[ 0.  3. inf inf]
 [ 3.  0.  1.  4.]
 [inf  1.  0.  1.]
 [inf  4.  1.  0.]]
Distance matrix after applying Floyd's Algorithm:
[[0. 3. 4. 5.]
 [3. 0. 1. 2.]
 [4. 1. 0. 1.]
 [5. 2. 1. 0.]]
Shortest path within distance threshold: 0.0

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

main.py Share Run

```python
import numpy as np
routers = ['A', 'B', 'C', 'D', 'E', 'F']
distance_matrix = np.array([[0, 3, np.inf, 7, np.inf, np.inf],
                            [3, 0, 2, np.inf, np.inf, np.inf],
                            [np.inf, 2, 0, 1, 4, np.inf],
                            [7, np.inf, 1, 0, 2, 3],
                            [np.inf, np.inf, 4, 2, 0, 1],
                            [np.inf, np.inf, np.inf, 3, 1, 0]]
                              )
def floyd_warshall(dist):
    num_routers = len(dist)
    for k in range(num_routers):
        for i in range(num_routers):
            for j in range(num_routers):
                if dist[i][j] > dist[i][k] + dist[k][j]:
                    dist[i][j] = dist[i][k] + dist[k][j]
    return dist
shortest_paths = floyd_warshall(distance_matrix.copy())
print(f"Router A to Router F = {shortest_paths[0][5]}")
distance_matrix[1][3] = np.inf
distance_matrix[3][1] = np.inf
shortest_paths_after_failure = floyd_warshall(distance_matrix
    .copy())
print(f"Router A to Router F =
```

Output Clear

```
Router A to Router F = 9.0
Router A to Router F = 9.0

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

```python
import numpy as np
def floyd_warshall(n, edges):
    dist = np.full((n, n), np.inf)
    for i in range(n):
        dist[i][i] = 0
    for u, v, w in edges:
        dist[u][v] = w
        dist[v][u] = w
    print("Distance matrix before applying Floyd's Algorithm:")
    print(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]
    print("Distance matrix after applying Floyd's Algorithm:")
    print(dist)
    return dist
n = 5
edges = [[0, 1, 2], [0, 4, 8], [1, 2, 3], [1, 4, 2], [2, 3, 1],
    [3, 4, 1]]
distance_matrix = floyd_warshall(n, edges)
shortest_path = np.min(distance_matrix)
print("Shortest path:", shortest_path)
```

```
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. 5. 4.]
 [2. 0. 3. 3. 2.]
 [5. 3. 0. 1. 2.]
 [5. 3. 1. 0. 1.]
 [4. 2. 2. 1. 0.]]
Shortest path: 0.0

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

```python
class OBST:
    def __init__(self, keys, freq):
        self.keys = keys
        self.freq = freq
        self.n = len(keys)
        self.cost = [[0] * self.n for _ in range(self.n)]
        self.root = [[0] * self.n for _ in range(self.n)]

    def optimal_bst(self):
        for i in range(self.n):
            self.cost[i][i] = self.freq[i]

        for length in range(2, self.n + 1):
            for i in range(self.n - length + 1):
                j = i + length - 1
                self.cost[i][j] = float('inf')
                for r in range(i, j + 1):
                    c = (self.cost[i][r - 1] if r > i else 0) + \
                        (self.cost[r + 1][j] if r < j else 0) + \
                        sum(self.freq[i:j + 1])
                    if c < self.cost[i][j]:
                        self.cost[i][j] = c
                        self.root[i][j] = r
    def print_cost_and_root(self):
        print("Cost Matrix:")
```

```
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, 0, 2, 2]
[0, 0, 0, 2]
[0, 0, 0, 0]

Optimal Cost: 1.7

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

```python
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 length in range(2, n + 1):
        for i in range(n - length + 1):
            j = i + length - 1
            cost[i][j] = float('inf')
            total_freq = sum(freq[i:j + 1])
            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) + total_freq
                if c < cost[i][j]:
                    cost[i][j] = c
                    root[i][j] = r
    return cost, root
def print_bst(root, keys, i, j):
    if i > j:
        return
    r = root[i][j]
    print(f"Key: {keys[r]} (Root)")
    print_bst(root, keys, i, r - 1)
```

```
Optimal Binary Search Tree:
Key: 16 (Root)
Key: 10 (Root)
Key: 10 (Root)
Key: 10 (Root)
Key: 10 (Root)
Key: 10 (Root)
Key: 10 (Root)
Key: 10 (Root)
Key: 10 (Root)
Key: 10 (Root)
Key: 10 (Root)
Key: 10 (Root)
Key: 10 (Root)
Key: 10 (Root)
Key: 10 (Root)
Key: 10 (Root)
Key: 10 (Root)
Key: 10 (Root)
Key: 10 (Root)
Key: 10 (Root)
Key: 10 (Root)
Key: 10 (Root)
Key: 10 (Root)
Key: 10 (Root)
```

```python
def catMouseGame(graph):
    from collections import deque
    n = len(graph)
    queue = deque()
    visited = set()
    queue.append((1, 2, 0))
    visited.add((1, 2, 0))
    result = {}
    while queue:
        mouse, cat, turn = queue.popleft()
        if mouse == 0:
            result[(mouse, cat, turn)] = 1
            continue
        if mouse == cat:
            result[(mouse, cat, turn)] = 2
            continue
        if (mouse, cat, turn) in result:
            continue
        if turn == 0:
            for next_mouse in graph[mouse]:
                if next_mouse == cat:
                    continue
                if (next_mouse, cat, 1) not in result:
                    queue.append((next_mouse, cat, 1))
                    visited.add((next_mouse, cat, 1))
```

```
0

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

```python
import heapq
from collections import defaultdict
def maxProbability(n, edges, succProb, start, end):
    graph = defaultdict(list)
    for (a, b), prob in zip(edges, succProb):
        graph[a].append((b, prob))
        graph[b].append((a, prob))
    max_heap = [(-1.0, start)]
    probabilities = {i: 0 for i in range(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
```

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

```python
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
output = uniquePaths(m, n)
print(output)
```

```
28

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

```python
def numIdenticalPairs(nums):
    count = 0
    freq = {}
    for num in nums:
        if num in freq:
            count += freq[num]
            freq[num] += 1
        else:
            freq[num] = 1

    return count

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

```
4

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

```python
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:
            curr_dist, node = heapq.heappop(min_heap)
            if curr_dist > distances[node]:
                continue
            for neighbor, weight in graph[node]:
                distance = curr_dist + weight
                if distance < distances[neighbor]:
                    distances[neighbor] = distance
                    heapq.heappush(min_heap, (distance, neighbor))
        return distances
    city_count = float('inf')
    result_city = -1
    for i in range(n):
        reachable_cities = sum(1 for d in dijkstra(i) if d <=
            distanceThreshold)
```

```
3


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

```python
import heapq
def networkDelayTime(times, n, k):
    graph = {}
    for u, v, w in times:
        if u not in graph:
            graph[u] = []
        graph[u].append((v, w))
    min_heap = [(0, k)]
    time_to_receive = {i: float('inf') for i in range(1, n + 1)}
    time_to_receive[k] = 0
    while min_heap:
        curr_time, node = heapq.heappop(min_heap)
        if curr_time > time_to_receive[node]:
            continue
        for neighbor, travel_time in graph.get(node, []):
            new_time = curr_time + travel_time
            if new_time < time_to_receive[neighbor]:
                time_to_receive[neighbor] = new_time
                heapq.heappush(min_heap, (new_time, neighbor))
    max_time = max(time_to_receive.values())
    return max_time if max_time < float('inf') else -1
times = [[2,1,1],[2,3,1],[3,4,1]]
n = 4
k = 2
print(networkDelayTime(times, n, k))
```

```
2

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