main.py

```python
import numpy as np

def floyd_warshall(n, edges):

    dist = np.full((n, n), float('inf'))
    for i in range(n):
        dist[i][i] = 0

    for u, v, w in edges:
        dist[u][v] = min(dist[u][v], w)
        dist[v][u] = min(dist[v][u], w)

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

    # Floyd's 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]

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

    return dist
```

Output

```
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.]]
Output: 2
Distance matrix before applying Floyd's Algorithm:
[[ 0.   1.   5. inf inf inf]
 [ 1.   0.   2.   1. inf inf]
 [ 5.   2.   0. inf  3. inf]
 [inf  1. inf  0.   1.   6.]
 [inf inf  3.   1.   0.   2.]
 [inf inf inf  6.   2.   0.]]
Distance matrix after applying Floyd's Algorithm:
[[0. 1. 3. 2. 3. 5.]
 [1. 0. 2. 1. 2. 4.]
 [3. 2. 0. 3. 3. 5.]
 [2. 1. 3. 0. 1. 3.]
 [3. 2. 3. 1. 0. 2.]
 [5. 4. 5. 3. 2. 0.]]
City 1 to City 3 = 1.0
```

main.py

```python
import numpy as np

routers = ['A', 'B', 'C', 'D', 'E', 'F']
distance_matrix = np.array([
    [0, 3, np.inf, np.inf, np.inf, 5],
    [3, 0, 1, 7, np.inf, np.inf],
    [np.inf, 1, 0, 2, 3, np.inf],
    [np.inf, 7, 2, 0, 1, 3],
    [np.inf, np.inf, 3, 1, 0, 2],
    [5, np.inf, np.inf, 3, 2, 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]}")
```

Output

```
Router A to Router F = 5.0
Router A to Router F = 5.0

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

main.py

```python
import numpy as np

def floyd_warshall(n, edges):
    dist = np.full((n, n), float('inf'))
    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):
                dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])

    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)

print("Distance Matrix Before:")
print(distance_matrix)

shortest_path_CA = distance_matrix[2][0]
```

Output

```
Distance Matrix Before:
[[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 from C to A: 5.0
Shortest path from E to C: 3.0

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

main.py

```python
class OptimalBinarySearchTree:
    def __init__(self, keys, freq):
        self.keys = keys
        self.freq = freq
        self.n = len(keys)
        self.cost = [[0] * (self.n + 1) for _ in range(self.n + 1)]
        self.root = [[0] * (self.n + 1) for _ in range(self.n + 1)]

    def construct_obst(self):
        for i in range(self.n):
            self.cost[i][i + 1] = self.freq[i]
            self.root[i][i + 1] = i + 1

        for length in range(2, self.n + 1):
            for i in range(self.n - length + 1):
                j = i + length
                self.cost[i][j] = float('inf')
                for r in range(i + 1, j + 1):
                    c = self.cost[i][r] + self.cost[r][j] + self.sum_freq(i, j)
                    if c < self.cost[i][j]:
                        self.cost[i][j] = c
                        self.root[i][j] = r

    def sum_freq(self, i, j):
        return sum(self.freq[k] for k in range(i, j))
```

Output

```
Output: 3.0
Cost Table
0.1 0.6 1.7 3.0
0.2 1.2 2.4
0.4 1.4
0.3

Root Table
1 1 2 2
2 2 3
3 3
4
Input: keys[] = ['10', '12'], freq[] = [34, 50]
Output = 168.0

Input: keys[] = ['10', '12', '20'], freq[] = [34, 8, 50]
Output = 226.0


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

main.py

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

    def construct_obst(self):
        for i in range(self.n):
            self.cost[i][i] = self.freq[i]
            self.root[i][i] = 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 display_results(self):
```

Output

```
0 0 2 2
0 0 0 3

Total Cost: 26
Cost Matrix:
34 118
0 50

Root Matrix:
0 1
0 1

Total Cost: 118
Cost Matrix:
34 50 142
0 8 66
0 0 50

Root Matrix:
0 0 2
0 1 2
0 0 2

Total Cost: 142

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

main.py

```python
from collections import deque

def catMouseGame(graph):
    n = len(graph)
    dp = [[[0] * 2 for _ in range(n)] for _ in range(n)]

    degree = [[[0] * 2 for _ in range(n)] for _ in range(n)]

    for mouse in range(n):
        for cat in range(n):
            degree[mouse][cat][0] = len(graph[mouse])
            degree[mouse][cat][1] = len(graph[cat]) - (0 in
                graph[cat])

    queue = deque()

    for cat in range(1, n):
        dp[0][cat][0] = 1
        dp[0][cat][1] = 1
        queue.append((0, cat, 0, 1))
        queue.append((0, cat, 1, 1))

    for mouse in range(1, n):
        dp[mouse][mouse][0] = 2
        dp[mouse][mouse][1] = 2
        queue.append((mouse, mouse, 0, 2))
        queue.append((mouse, mouse, 1, 2))
```

Output

```
0
1

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

main.py

```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)]
    visited = set()

    while max_heap:
        prob, node = heapq.heappop(max_heap)
        prob = -prob

        if node in visited:
            continue
        visited.add(node)

        if node == end:
            return prob

```

Output

```
0.25
0.3

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

main.py

```python
def uniquePaths(m: int, n: int) -> int:
    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]

print(uniquePaths(3, 7))
print(uniquePaths(3, 2))
```

Output

```
28
3

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

main.py

```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

print(numIdenticalPairs([1, 2, 3, 1, 1, 3]))
print(numIdenticalPairs([1, 1, 1, 1]))
```

Output

```
4
6

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

main.py

```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
```

Output

```
3
0

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

main.py

```python
import heapq
from collections import defaultdict

def networkDelayTime(times, n, k):
    graph = defaultdict(list)
    for u, v, w in times:
        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)

        for neighbor, travel_time in graph[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

print(networkDelayTime([[2,1,1],[2,3,1],[3,4,1]], 4, 2))
print(networkDelayTime([[1,2,1]], 2, 1))
print(networkDelayTime([[1,2,1]], 2, 2))
```

Output

```
2
1
-1

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