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
Explanation: The figure above describes the graph. 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.
Test cases :
a) You are given a small network of 4 cities connected by roads with the following distances:
City 1 to City 2: 3
City 1 to City 3: 8
City 1 to City 4: -4
City 2 to City 4: 1
City 2 to City 3: 4
City 3 to City 1: 2
City 4 to City 3: -5
City 4 to City 2: 6
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 from City 1 to City 3.
Input as above
Output : City 1 to City 3 = -9
b. Consider a network with 6 routers. The initial routing table is as follows:
Router A to Router B: 1
Router A to Router C: 5
Router B to Router C: 2
Router B to Router D: 1
Router C to Router E: 3
Router D to Router E: 1
Router D to Router F: 6
Router E to Router F: 2

```python
import sys
def floyd_algorithm(n, edges):
    distance = [[sys.maxsize for _ in range(n)] for _ in range(n)]
    for i in range(n):
        distance[i][i] = 0
    for edge in edges:
        distance[edge[0]][edge[1]] = edge[2]
    for k in range(n):
        for i in range(n):
            for j in range(n):
                distance[i][j] = min(distance[i][j], distance[i][k] + distance[k][j])
    return distance
n = 4
edges = [[0,1,3],[1,2,1],[1,3,4],[2,3,1]]
distanceThreshold = 4
result = floyd_algorithm(n, edges)
print("Distance Matrix Before Applying Floyd's Algorithm:")
for row in result:
    print(row)
print("\nDistance Matrix After Applying Floyd's Algorithm:")
for row in result:
    print(row)
shortest_path = min(range(n), key=lambda x: sum(1 for d in result[x] if d <= distanceThreshold))
print(f"\nShortest Path City: {shortest_path}")
```

```
[9223372036854775807, 9223372036854775807, 0, 1]
[9223372036854775807, 9223372036854775807, 9223372036854775807, 0]

Distance Matrix After Applying Floyd's Algorithm:
[0, 3, 4, 5]
[9223372036854775807, 0, 1, 2]
[9223372036854775807, 9223372036854775807, 0, 1]
[9223372036854775807, 9223372036854775807, 9223372036854775807, 0]

Shortest Path City: 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

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

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


...Program finished with exit code 0
Press ENTER to exit console.
```

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.
a) Test cases :
B to A 2
A TO C 3
C TO D 1
D TO A 6
C TO B 7
Find shortest path from C to A
Output = 7
b) Find shortest path from E to C
C TO A 2
A TO B 4
B TO C 1
B TO E 6
E TO A 1
A TO D 5
D TO E 2
E TO D 4
D TO C 1
C TO D 3
Output : E to C = 5

```python
import numpy as np
def floyd_warshall(n, edges):
    # Initialize distance matrix
    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)
    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
def find_shortest_path(dist, start, end):
    return dist[start][end]
n = 5
edges = [[0, 1, 2], [0, 4, 8], [1, 2, 3], [1, 4, 2], [2, 3, 1], [3, 4, 1]]
distanceThreshold = 2
distance_matrix = floyd_warshall(n, edges)
shortest_path_C_to_A = find_shortest_path(distance_matrix, 2, 0)
print(f"Shortest path from C to A: {shortest_path_C_to_A}")
shortest_path_E_to_C = find_shortest_path(distance_matrix, 4, 2)
print(f"Shortest path from E to C: {shortest_path_E_to_C}")
```

input

```
 [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: 2.0
```

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

```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 optimal_bst(self):
        for i in range(self.n):
            self.cost[i][i + 1] = self.freq[i]
            self.root[i][i + 1] = i
        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, j):
                    c = self.cost[i][r] + self.cost[r + 1][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))
    def print_cost(self):
        for i in range(self.n + 1):
            for j in range(i + 1, self.n + 1):
                print(f"{self.cost[i][j]:.1f}", end=" ")
            print()
    def print_root(self):
        for i in range(self.n):
```

```python
        for i in range(self.n):
            for j in range(i + 1, self.n + 1):
                print(self.root[i][j] + 1, end=" ")
            print()
keys = ['A', 'B', 'C', 'D']
freq = [0.1, 0.2, 0.4, 0.3]
obst = OptimalBinarySearchTree(keys, freq)
obst.optimal_bst()
print("Cost:", obst.cost[0][len(keys)])
print("Cost Table")
obst.print_cost()
print("Root Table")
obst.print_root()
test_cases = [
    (['10', '12'], [34, 50]),
    (['10', '12', '20'], [34, 8, 50])
]
for keys, freq in test_cases:
    obst = OptimalBinarySearchTree(keys, freq)
    obst.optimal_bst()
    print("Output =", obst.cost[0][len(keys)])
```

```
Cost Table
0.1 0.4 1.1 1.7
0.2 0.8 1.4
0.4 1.0
0.3

Root Table
1 2 3 3
2 3 3
3 3
4
Output = 118
Output = 142
```

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

```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 construct_obst(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_obst(self):
        print("Cost Matrix:")
        for row in self.cost:
            print(" ".join(map(str, row)))
        print("\nRoot Matrix:")
        for row in self.root:
            print(" ".join(map(str, row)))
        print("\nTotal Cost:", self.cost[0][self.n - 1])
keys1 = [10, 12, 16, 21]
freq1 = [4, 2, 6, 3]
obst1 = OBST(keys1, freq1)
obst1.construct_obst()
obst1.print_obst()
```

```
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 0 2 2
0 0 0 2
0 0 0 0

Total Cost: 26
Cost Matrix:
34 118
0 50

Root Matrix:
0 1
0 0

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

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

Total Cost: 142
```

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
Example 2:
Input: graph = [[1,3],[0],[3],[0,2]]
Output: 1

```python
from collections import deque
def catMouseGame(graph):
    n = len(graph)
    dp = [[[0] * 3 for _ in range(n)] for _ in range(n)]
    for mouse in range(n):
        for cat in range(n):
            if mouse == cat:
                dp[mouse][cat][0] = 2  # Cat wins
            elif mouse == 0:
                dp[mouse][cat][0] = 1  # Mouse wins
    for turn in range(2, -1, -1):
        for mouse in range(n):
            for cat in range(n):
                if dp[mouse][cat][turn] != 0:
                    continue
                if turn == 0:  # Mouse's turn
                    for next_mouse in graph[mouse]:
                        if next_mouse == 0:
                            dp[mouse][cat][turn] = 1
                            break
                        if dp[next_mouse][cat][1] == 1:
                            dp[mouse][cat][turn] = 1
                            break
                        if dp[next_mouse][cat][1] == 0:
                            dp[mouse][cat][turn] = 0
                else:  # Cat's turn
                    for next_cat in graph[cat]:
                        if next_cat == 0:
                            continue
                        if dp[mouse][next_cat][0] == 2:
```

```
                        dp[mouse][cat][turn] = 1
                        break
                    if dp[next_mouse][cat][1] == 0:
                        dp[mouse][cat][turn] = 0
            else:  # Cat's turn
                for next_cat in graph[cat]:
                    if next_cat == 0:
                        continue
                    if dp[mouse][next_cat][0] == 2:
                        dp[mouse][cat][turn] = 2
                        break
                    if dp[mouse][next_cat][0] == 0:
                        dp[mouse][cat][turn] = 0
    return dp[1][2][0]
print(catMouseGame([[2,5],[3],[0,4,5],[1,4,5],[2,3],[0,2,3]]))
print(catMouseGame([[1,3],[0],[3],[0,2]]))  # Output: 1
```

input

```
0
1
```

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

Explanation: There are two paths from start to end, one having a probability of success = 0.2 and the other has 0.5 * 0.5 = 0.25.

Example 2:

Input: n = 3, edges = [[0,1],[1,2],[0,2]], succProb = [0.5,0.5,0.3], start = 0, end = 2

Output: 0.30000

```
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:
        current_prob, node = heapq.heappop(max_heap)
        current_prob = -current_prob
        if node == end:
            return current_prob
        for neighbor, prob in graph[node]:
            new_prob = current_prob * prob
            if new_prob > probabilities[neighbor]:
                probabilities[neighbor] = new_prob
                heapq.heappush(max_heap, (-new_prob, neighbor))
    return 0.0
print(maxProbability(3, [[0,1],[1,2],[0,2]], [0.5,0.5,0.2], 0, 2))
print(maxProbability(3, [[0,1],[1,2],[0,2]], [0.5,0.5,0.3], 0, 2))
```

input

```
0.25
0.3
```

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
Explanation: There are 4 good pairs (0,3), (0,4), (3,4), (2,5) 0-indexed.
Example 2:
Input: nums = [1,1,1,1]
Output: 6
Explanation: Each pair in the array are good.

```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]))
```

```
4
6
```

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

Explanation: The figure above describes the graph.

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 distance Threshold = 4, but we have to return
city 3 since it has the greatest number.

Example 2:

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
Explanation: The figure above describes the graph.
The neighboring cities at a distance Threshold = 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.

```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):
        min_heap = [(0, start)]
        distances = {i: float('inf') for i in range(n)}
        distances[start] = 0
        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 sum(1 for d in distances.values() if d <= distanceThreshold)
    min_reachable = float('inf')
    city_with_min_reachable = -1
    for city in range(n):
        reachable_cities = dijkstra(city)
        if reachable_cities < min_reachable or (reachable_cities == min_reachable and city > city_with_min_reachable):
            min_reachable = reachable_cities
            city_with_min_reachable = city

    return city_with_min_reachable
print(findTheCity(4, [[0,1,3],[1,2,1],[1,3,4],[2,3,1]], 4))
print(findTheCity(5, [[0,1,2],[0,4,8],[1,2,3],[1,4,2],[2,3,1],[3,4,1]], 2))
```

input

```
3
0
```

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 = 2
Output: 2
Example 2:

Input: times = [[1,2,1]], n = 2, k = 1
Output: 1
Example 3:
Input: times = [[1,2,1]], n = 2, k = 2
Output: -1

```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))
```

input

```
2
1
-1
```