1. INF = float('inf')

def floyd\_warshall(n, edges):

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 Floyd's Algorithm:")

for row in dist:

print(row)

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])

print("\nDistance Matrix After Floyd's Algorithm:")

for row in dist:

print(row)

n = 4

edges = [

[0, 1, 3],

[1, 2, 1],

[1, 3, 4],

[2, 3, 1]

]

floyd\_warshall(n, edges)

output:

Distance Matrix Before Floyd's Algorithm:

[0, 3, inf, inf]

[inf, 0, 1, 4]

[inf, inf, 0, 1]

[inf, inf, inf, 0]

Distance Matrix After Floyd's Algorithm:

[0, 3, 4, 5]

[inf, 0, 1, 2]

[inf, inf, 0, 1]

[inf, inf, inf, 0]

2. INF = float('inf')

def floyd\_warshall(n, edges):

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] = 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 = 6 # Routers A-F

edges = [

[0, 1, 1], [0, 2, 5], [1, 2, 2], [1, 3, 1],

[2, 4, 3], [3, 4, 1], [3, 5, 6], [4, 5, 2]

]

print("Before Link Failure:")

dist\_before = floyd\_warshall(n, edges)

print("Shortest Path A to F:", dist\_before[0][5])

# Simulate link failure between B and D

edges = [e for e in edges if e[:2] != [1, 3] and e[:2] != [3, 1]]

print("\nAfter Link Failure:")

dist\_after = floyd\_warshall(n, edges)

print("Shortest Path A to F:", dist\_after[0][5])

output:

Before Link Failure:

Shortest Path A to F: 6

After Link Failure:

Shortest Path A to F: 6

3. import numpy as np

def floyd\_warshall(graph):

num\_vertices = len(graph)

distance = np.array(graph)

for k in range(num\_vertices):

for i in range(num\_vertices):

for j in range(num\_vertices):

if distance[i][j] > distance[i][k] + distance[k][j]:

distance[i][j] = distance[i][k] + distance[k][j]

return distance

graph = [

[0, 3, float('inf'), 7],

[8, 0, 2, float('inf')],

[5, float('inf'), 0, 1],

[2, float('inf'), float('inf'), 0]

]

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

print(np.array(graph))

shortest\_paths = floyd\_warshall(graph)

print("\nDistance matrix after applying Floyd's Algorithm:")

print(shortest\_paths)

# Identifying the shortest path from city 0 to city 2

print("\nShortest path from city 0 to city 2:", shortest\_paths[0][2])

output:

Distance matrix before applying Floyd's Algorithm:

[[ 0. 3. inf 7.]

[ 8. 0. 2. inf]

[ 5. inf 0. 1.]

[ 2. inf inf 0.]]

Distance matrix after applying Floyd's Algorithm:

[[ 0. 3. 5. 6.]

[ 8. 0. 2. 3.]

[ 5. 8. 0. 1.]

[ 2. 5. 7. 0.]]

Shortest path from city 0 to city 2: 5.0

4. 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')

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) + \

sum(freq[i:j + 1])

if c < cost[i][j]:

cost[i][j] = c

root[i][j] = r

return cost, root

keys = ['A', 'B', 'C', 'D']

freq = [0.1, 0.2, 0.4, 0.3]

cost, root = optimal\_bst(keys, freq)

print("Cost Matrix:")

print(cost)

print("\nRoot Matrix:")

print(root)

print("\nOptimal Cost:", cost[0][len(keys) - 1])

output:

Cost Matrix:

[[0.1 0.4 1.1 1.7]

[0. 0.2 0.8 1.4]

[0. 0. 0.4 1. ]

[0. 0. 0. 0.3]]

Root Matrix:

[[0 1 2 2]

[0 1 2 2]

[0 0 2 3]

[0 0 0 3]]

Optimal Cost: 1.7

5. 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')

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) + \

sum(freq[i:j + 1])

if c < cost[i][j]:

cost[i][j] = c

root[i][j] = r

return cost, root

keys = [10, 12, 16, 21]

freq = [4, 2, 6, 3]

cost, root = optimal\_bst(keys, freq)

print("Cost Matrix:")

print(cost)

print("\nRoot Matrix:")

print(root)

print("\nOptimal Cost:", cost[0][len(keys) - 1])

output:

26

6. def canMouseWin(graph, mouseStart, catStart):

from functools import lru\_cache

@lru\_cache(None)

def dfs(mouse, cat, turn):

if mouse == 0: return True # Mouse wins

if cat == mouse: return False # Cat wins

if turn % 2 == 0: # Mouse's turn

return any(dfs(next\_mouse, cat, turn + 1) for next\_mouse in graph[mouse])

else: # Cat's turn

return all(dfs(mouse, next\_cat, turn + 1) for next\_cat in graph[cat] if next\_cat != 0)

return dfs(mouseStart, catStart, 0)

graph = {1: [2, 3], 2: [1, 0], 3: [1]}

print(canMouseWin(graph, 1, 2))

Output: 1

7. 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)] # Store negative for max-heap behavior

visited = set()

while max\_heap:

prob, node = heapq.heappop(max\_heap)

prob = -prob # Convert back to positive

if node in visited:

continue

visited.add(node)

if node == end:

return prob

for neighbor, edge\_prob in graph[node]:

if neighbor not in visited:

heapq.heappush(max\_heap, (-(prob \* edge\_prob), neighbor))

return 0.0

output:

No path found

8. import math

def unique\_paths(m, n):

return math.factorial(m + n - 2) // (math.factorial(m - 1) \* math.factorial(n - 1))

print(unique\_paths(3, 7))

Output:

28

9. def count\_good\_pairs(nums):

count = {}

good\_pairs = 0

for num in nums:

if num in count:

good\_pairs += count[num]

count[num] += 1

else:

count[num] = 1

return good\_pairs

nums = [1, 2, 3, 1, 1, 3]

print(count\_good\_pairs(nums))

Output:

4

10. 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\_count = dijkstra(city)

if (reachable\_count < min\_reachable) or (reachable\_count == min\_reachable and city > city\_with\_min\_reachable):

min\_reachable = reachable\_count

city\_with\_min\_reachable = city

return city\_with\_min\_reachable

output:3

11. 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, node)

time\_to\_receive = {i: float('inf') for i in range(1, n + 1)}

time\_to\_receive[k] = 0

while min\_heap:

current\_time, node = heapq.heappop(min\_heap)

if current\_time > time\_to\_receive[node]:

continue

for neighbor, travel\_time in graph[node]:

new\_time = current\_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))

Output:

2