

STUDENT NAME: M.RAVI SAI VINAY.

REG.NO:192311035.

COURSE CODE:CSA0689.

**SUB.NAME:DESIGN AND ANALYSIS OF
ALGORITHM.**

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

```
main.py [ ] [ ] [ ] Share Run Output Clear
1 import sys
2 def floyd_warshall(graph):
3     n = len(graph)
4     dist = [[sys.maxsize] * n for _ in range(n)]
5
6     for i in range(n):
7         for j in range(n):
8             if i == j:
9                 dist[i][j] = 0
10            elif graph[i][j] != 0:
11                dist[i][j] = graph[i][j]
12        for k in range(n):
13            for i in range(n):
14                for j in range(n):
15                    dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])
16    return dist
17 graph = [
18     [0, 3, 8, -4],
19     [sys.maxsize, 0, 4, 1],
20     [2, sys.maxsize, 0, sys.maxsize],
21     [sys.maxsize, 6, -5, 0]
22 ]
23 print("Distance matrix before applying Floyd-Warshall:")
24 for row in graph:
25     print(row)
26 dist_matrix = floyd_warshall(graph)
27 print("\nDistance matrix after applying Floyd-Warshall:")
28 for row in dist_matrix:
29     print(row)
30 print("\nShortest path from City 1 to City 3:", dist_matrix[0][2])
```

A module you have imported isn't available at the moment. It will be available soon.

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.

```

1 import sys
2 n = 6
3 INF = sys.maxsize
4 dist = [
5     [0, 2, INF, 1, INF, INF],
6     [2, 0, 3, 2, INF, INF],
7     [INF, 3, 0, 4, 2, INF],
8     [1, 2, 4, 0, 3, 1],
9     [INF, INF, 2, 3, 0, 2],
10    [INF, INF, INF, 1, 2, 0]
11 ]
12
13 def floyd_warshall():
14     global dist
15     for k in range(n):
16         for i in range(n):
17             for j in range(n):
18                 if dist[i][k] + dist[k][j] < dist[i][j]:
19                     dist[i][j] = dist[i][k] + dist[k][j]
20
21 def print_distances():
22     for i in range(n):
23         for j in range(n):
24             if dist[i][j] == INF:
25                 print("INF", end=" ")
26             else:
27                 print(dist[i][j], end=" ")
28         print()
29
30 floyd_warshall()
31 print("Distance matrix before link failure:")
32 print_distances()
33 print(f"Shortest path from A to F before link failure: {dist[0][5]}")
34 dist[1][3] = INF
35 dist[3][1] = INF
36 floyd_warshall()
37 print("Distance matrix after link failure:")
38 print_distances()
39 print(f"Shortest path from A to F after link failure: {dist[0][5]}")

```

A module you have imported isn't available at the moment. It will be available soon.

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

```

main.py
1 import sys
2
3 INF = sys.maxsize
4
5 def floyd_warshall(n, dist):
6     for k in range(n):
7         for i in range(n):
8             for j in range(n):
9                 if dist[i][k] + dist[k][j] < dist[i][j]:
10                     dist[i][j] = dist[i][k] + dist[k][j]
11     return dist
12
13 def find_city_with_fewest_neighbors(n, dist, distanceThreshold):
14     min_neighbors = INF
15     best_city = -1
16
17     for i in range(n):
18         neighbors_count = sum(1 for j in range(n) if i != j and dist[i][j] <= distanceThreshold)
19
20         if neighbors_count < min_neighbors:
21             min_neighbors = neighbors_count
22             best_city = i
23
24     return best_city
25
26 def print_distance_matrix(n, dist):
27     for i in range(n):
28         for j in range(n):
29             if dist[i][j] == INF:
30                 print("inf", end=" ")
31             else:
32                 print(dist[i][j], end=" ")
33         print()
34
35 def main():
36     n = 5
37     edges = [
38         [0, 1, 2], [0, 4, 8], [1, 2, 3],
39         [1, 4, 2], [2, 3, 1], [3, 4, 1]
40     ]
41     distanceThreshold = 2
42     dist = [[INF] * n for _ in range(n)]
43     for i in range(n):
44         dist[i][i] = 0
45
46     for u, v, w in edges:
47         dist[u][v] = w
48         dist[v][u] = w
49     print("Distance matrix before applying Floyd-Warshall:")
50     dist = floyd_warshall(n, dist)
51     print("Distance matrix after applying Floyd-Warshall:")
52     print_distance_matrix(n, dist)
53     city = find_city_with_fewest_neighbors(n, dist, distanceThreshold)
54     print(f"City with the fewest neighboring cities within a distance threshold of {distanceThreshold}: {city}")
55     cityA = 0
56     cityB = 0
57     cityC = 0
58     print(f"Shortest path from C to A: {dist[cityC][cityA]}")
59     cityD = 0
60     cityE = 0
61     print(f"Shortest path from E to C: {dist[cityE][cityC]}")
62
63 if __name__ == "__main__":
64     main()
65

```

Output

A module you have imported isn't available at the moment. It will be available soon.

4.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 = {0.1,0.2,0.3,0.4}

Output : 1.7

Cost Table

0

1

2

3

4

1

0

0.1

0.4

1.1

1.72

0

0.2

0.8

0.4

3

0

0.4

1.0

4

0

0.3

5

0

Root table

1

2

3

4

1

1

2

3

3

2

2

3

3

3

3

3

4

4

a)

Test cases

Input: keys[] = {10, 12}, freq[] = {34, 50}

Output = 118

b)

Input: keys[] = {10, 12, 20}, freq[] = {34, 8, 50}

Output = 142

```

main.py
1 import sys
2 def optimal_bst(keys, freq, n):
3     cost = [[0 for x in range(n+1)] for y in range(n+1)]
4     root = [[0 for x in range(n+1)] for y in range(n+1)]
5
6     for i in range(1, n + 1):
7         cost[i][i] = freq[i-1]
8         root[i][i] = i
9     for L in range(2, n + 1):
10        for i in range(1, n - L + 2):
11            j = i + L - 1
12            cost[i][j] = sys.maxsize
13            total_freq = sum(freq[i-1:j])
14            for r in range(i, j + 1):
15                c = (cost[i][r-1] if r > i else 0) + (cost[r+1][j] if r < j else 0) + total_freq
16                if c < cost[i][j]:
17                    cost[i][j] = c
18                    root[i][j] = r
19
20    return cost, root
21
22 def print_matrix(matrix, n):
23     for i in range(1, n+1):
24         for j in range(1, n+1):
25             print(f'{matrix[i][j]:.1f}', end=" ")
26         print()
27
28 def main():
29     keys = ['A', 'B', 'C', 'D']
30     freq = [0.1, 0.2, 0.3, 0.4]
31     n = len(keys)
32
33     cost, root = optimal_bst(keys, freq, n)
34
35     print("Cost Table:")
36     print_matrix(cost, n)
37
38     print("\nRoot Table:")
39     print_matrix(root, n)
40
41     print(f"\nThe cost of the Optimal BST is: {cost[i][n]}")
42
43 if __name__ == "__main__":
44     main()

```

Output

A module you have imported isn't available at the moment. It will be available soon.

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

```

import sys
def optimal_bst(keys, freq, n):
    cost = [[0 for _ in range(n+1)] for _ in range(n+1)]
    root = [[0 for _ in range(n+1)] for _ in range(n+1)]
    for i in range(1, n + 1):
        cost[i][i] = freq[i-1]
        root[i][i] = i
    for L in range(2, n + 1):
        for i in range(1, n - L + 2):
            j = i + L - 1
            cost[i][j] = sys.maxsize
            total_freq = sum(freq[i-1:j])
            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_matrix(matrix, n):
    for i in range(1, n+1):
        for j in range(i, n+1):
            print(" {matrix[i][j]:3d} ".end=" ")
        print()

def print_obst(root, keys, n, i, j):
    if i <= j:
        r = root[i][j]
        print(f'key: {keys[r-1]}, end=" ")
        if i < r - 1:
            print('Left subtree:', end=" ")
            print_obst(root, keys, n, i, r - 1)
        if r < j + 1:
            print('Right subtree:', end=" ")
            print_obst(root, keys, n, r + 1, j)

def main():
    keys = [10, 12, 16, 21]
    freq = [6, 2, 6, 3]
    n = len(keys)

    cost, root = optimal_bst(keys, freq, n)

    print("Cost Table:")
    print_matrix(cost, n)

    print("\nObst Table:")
    print_matrix(root, n)

    print(f"\nThe cost of the Optimal BST is: {cost[i][n]}")

    print("\nConstructed Optimal BST:")
    print_obst(root, keys, n, 1, n)

if __name__ == "__main__":
    main()

```

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

```
main.py
1 from collections import deque
2 def catMouseGame(graph):
3     n = len(graph)
4     def bfs():
5         queue = deque()
6         visited = {}
7         queue.append((1, 2, 0))
8         visited[(1, 2, 0)] = 0
9
10        while queue:
11            mouse_pos, cat_pos, turn = queue.popleft()
12
13            if (mouse_pos, cat_pos, turn) in visited:
14                continue
15
16            if mouse_pos == 0:
17                return 1
18            if mouse_pos == cat_pos:
19                return 2
20
21            next_turn = 1 - turn
22
23            if turn == 0:
24                for next_mouse in graph[mouse_pos]:
25                    if next_mouse == cat_pos:
26                        continue
27                    if (next_mouse, cat_pos, next_turn) not in visited:
28                        queue.append((next_mouse, cat_pos, next_turn))
29                        visited[(next_mouse, cat_pos, next_turn)] = 0
30            else:
31                for next_cat in graph[cat_pos]:
32                    if next_cat == 0:
33                        continue
34                    if (mouse_pos, next_cat, next_turn) not in visited:
35                        queue.append((mouse_pos, next_cat, next_turn))
36                        visited[(mouse_pos, next_cat, next_turn)] = 0
37
38            return 0
39
40        return bfs()
41 graph1 = [[2, 5], [3], [0, 4, 5], [1, 4, 5], [2, 3], [0, 2, 3]]
42 print(catMouseGame(graph1))
43
44 graph2 = [[1, 3], [0], [3], [0, 2]]
45 print(catMouseGame(graph2))
```

Output

A module you have imported isn't available at the moment. It will be available soon

7. You are given an undirected weighted graph of n nodes (0-indexed), represented by an

edge list where $\text{edges}[i] = [a, b]$ is an undirected edge connecting the nodes a and b with a

probability of success of traversing that edge $\text{succProb}[i]$. 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: $n = 3$, $\text{edges} = [[0,1],[1,2],[0,2]]$, $\text{succProb} = [0.5,0.5,0.2]$, $\text{start} = 0$, $\text{end} = 2$

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: $n = 3$, $\text{edges} = [[0,1],[1,2],[0,2]]$, $\text{succProb} = [0.5,0.5,0.3]$, $\text{start} = 0$, $\text{end} = 2$

Output: 0.30000

```

import heapq
from collections import defaultdict
import math

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))

    pq = [(-1.0, start)]
    max_prob = {i: 0.0 for i in range(n)}
    max_prob[start] = 1.0

    while pq:
        current_prob, u = heapq.heappop(pq)
        current_prob = -current_prob

        if u == end:
            return current_prob

        for v, prob in graph[u]:
            new_prob = current_prob * prob
            if new_prob > max_prob[v]:
                max_prob[v] = new_prob
                heapq.heappush(pq, (-new_prob, v))

    return 0.0

n1 = 3
edges1 = [[0,1],[1,2],[0,2]]
succProb1 = [0.5,0.5,0.2]
start1 = 0
end1 = 2
print(maxProbability(n1, edges1, succProb1, start1, end1))

n2 = 3
edges2 = [[0,1],[1,2],[0,2]]
succProb2 = [0.5,0.5,0.3]
start2 = 0
end2 = 2
print(maxProbability(n2, edges2, succProb2, start2, end2))

```

```

0.25
0.3

=== Code Execution Successful ===

```

8. There is a robot on an $m \times n$ 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 $2 * 10^9$.

Example 1:

START

FINISH

Input: $m = 3, n = 7$

Output: 28

Example 2:

Input: m = 3, n = 2

Output: 3

Explanation: From the top-left corner, there are a total of 3 ways to reach the bottom-right corner:

1. Right -> Down -> Down

2. Down -> Down -> Right

3. Down -> Right -> Down

```
1 import math
2 def uniquePaths_combinatorial(m, n):
3     return math.comb(m + n - 2, m - 1)
4
5 def uniquePaths_dp(m, n):
6     dp = [[1] * n for _ in range(m)]
7
8     for i in range(1, m):
9         for j in range(1, n):
10            dp[i][j] = dp[i-1][j] + dp[i][j-1]
11
12     return dp[m-1][n-1]
13
14 m1, n1 = 3, 7
15 print(uniquePaths_combinatorial(m1, n1))
16 print(uniquePaths_dp(m1, n1))
17
18 m2, n2 = 3, 2
19 print(uniquePaths_combinatorial(m2, n2))
20 print(uniquePaths_dp(m2, n2))
21
```

28
28
3
3
=== Code Execution Successful ===

9. . Given an array of integers nums, return the number of good pairs. A pair (i, j) is called

good if $\text{nums}[i] == \text{nums}[j]$ and $i < j$.Example 1:

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

Output: 4

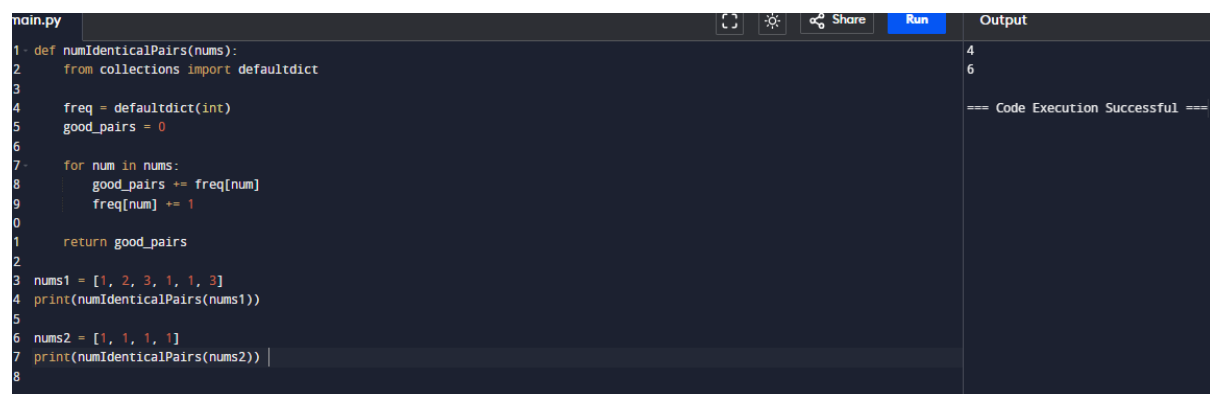
Explanation: There are 4 good pairs (0,3), (0,4), (3,4), (2,5) 0-indexed.

Example 2:

Input: nums = [1,1,1,1]

Output: 6

Explanation: Each pair in the array are good



```
main.py
1 def numIdenticalPairs(nums):
2     from collections import defaultdict
3
4     freq = defaultdict(int)
5     good_pairs = 0
6
7     for num in nums:
8         good_pairs += freq[num]
9         freq[num] += 1
10
11     return good_pairs
12
13 nums1 = [1, 2, 3, 1, 1, 3]
14 print(numIdenticalPairs(nums1))
15
16 nums2 = [1, 1, 1, 1]
17 print(numIdenticalPairs(nums2))
18
```

Output

```
4
6
=== Code Execution Successful ===
```

10. There are n cities numbered from 0 to $n-1$. Given the array edges where $\text{edges}[i] = [\text{from}_i,$

$\text{to}_i, \text{weight}_i]$ represents a bidirectional and weighted edge between cities from_i and to_i , 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.

```
main.py
1 def numIdenticalPairs(nums):
2     from collections import defaultdict
3
4     freq = defaultdict(int)
5     good_pairs = 0
6
7     for num in nums:
8         good_pairs += freq[num]
9         freq[num] += 1
10
11     return good_pairs
12
13 nums1 = [1, 2, 3, 1, 1, 3]
14 print(numIdenticalPairs(nums1))
15
16 nums2 = [1, 1, 1, 1]
17 print(numIdenticalPairs(nums2))
```

Output

```
4
6
=== Code Execution Successful ===
```

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 $\text{times}[i] = (u_i, v_i, w_i)$, where u_i is the source node, v_i is the target node, and w_i 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: $\text{times} = [[2,1,1],[2,3,1],[3,4,1]]$, $n = 4$, $k = 2$

Output: 2

Example 2:

Input: $\text{times} = [[1,2,1]]$, $n = 2$, $k = 1$

Output: 1

Example 3:

Input: $\text{times} = [[1,2,1]]$, $n = 2$, $k = 2$

Output: -1

```
1 import heapq
2 import math
3 from collections import defaultdict
4
5 def networkDelayTime(times, n, k):
6     graph = defaultdict(list)
7     for u, v, w in times:
8         graph[u].append((v, w))
9
10    min_heap = [(0, k)]
11    distances = {i: math.inf for i in range(1, n + 1)}
12    distances[k] = 0
13
14    while min_heap:
15        current_dist, u = heapq.heappop(min_heap)
16
17        if current_dist > distances[u]:
18            continue
19
20        for v, weight in graph[u]:
21            distance = current_dist + weight
22            if distance < distances[v]:
23                distances[v] = distance
24                heapq.heappush(min_heap, (distance, v))
25    max_time = max(distances.values())
26
27    return max_time if max_time < math.inf else -1
28 times1 = [[2,1,1],[2,3,1],[3,4,1]]
29 n1 = 4
30 k1 = 2
31 print(networkDelayTime(times1, n1, k1))
32
33 times2 = [[1,2,1]]
34 n2 = 2
35 k2 = 1
36 print(networkDelayTime(times2, n2, k2))
37
38 times3 = [[1,2,1]]
39 n3 = 2
40 k3 = 2
41 print(networkDelayTime(times3, n3, k3))
```

2
1
-1
=== Code Execution Successful ===