**DESIGN AND ANALYSIS OF ALGORITHM**

**DAY -08 PROGRAMS**

**COURSE CODE: CSA0689**

**NAME: G. GANESH**

**REGISTER NO:192373008**

**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**

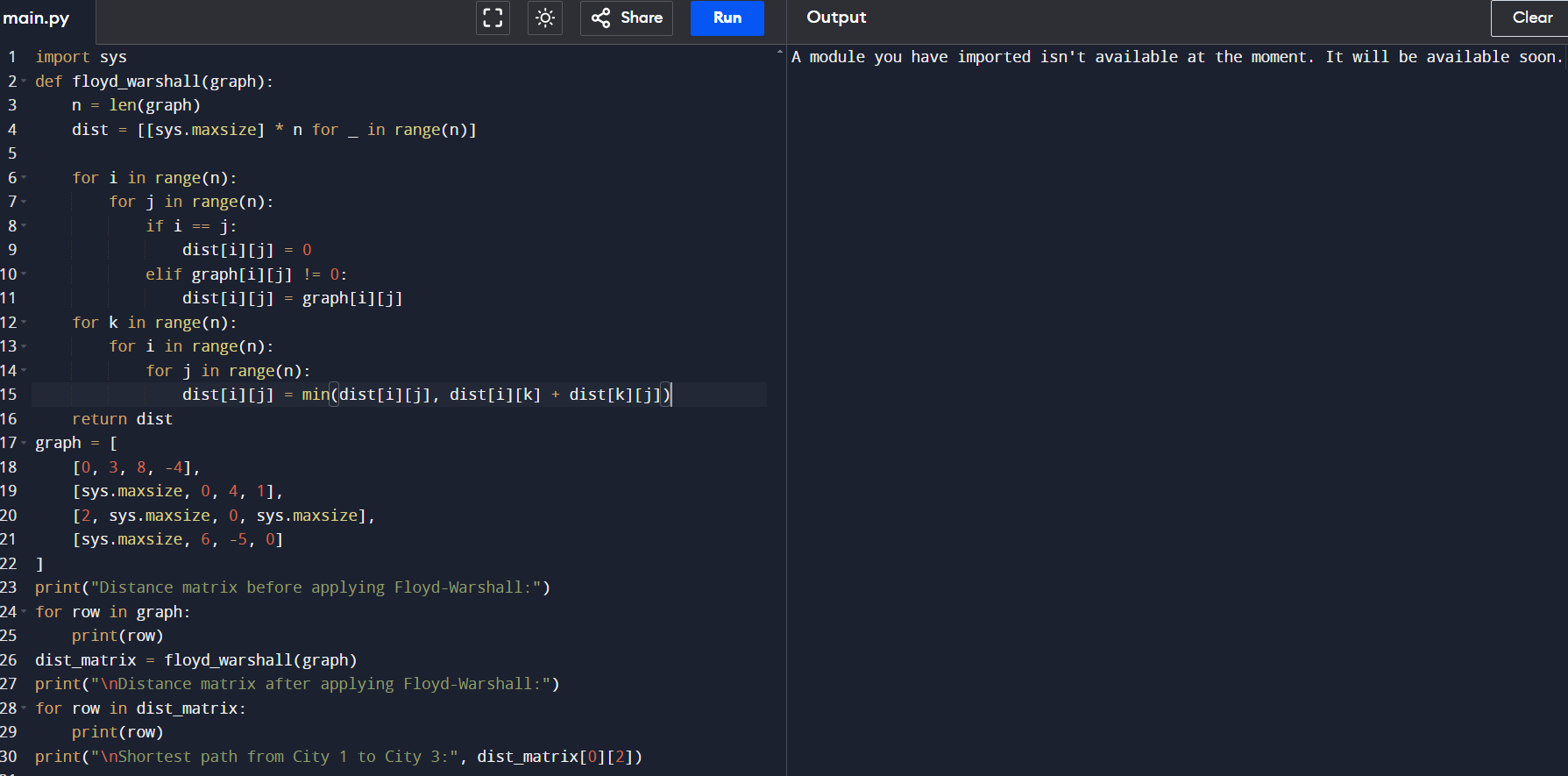
**Router B to Router D: 1**

**Router C to Router E: 3**

**Router D to Router E: 1**

**Router D to Router F: 6**

**Router E to Router F: 2**

****

**2.** **Write a Program to implement Floyd's Algorithm to calculate the shortest paths between all**

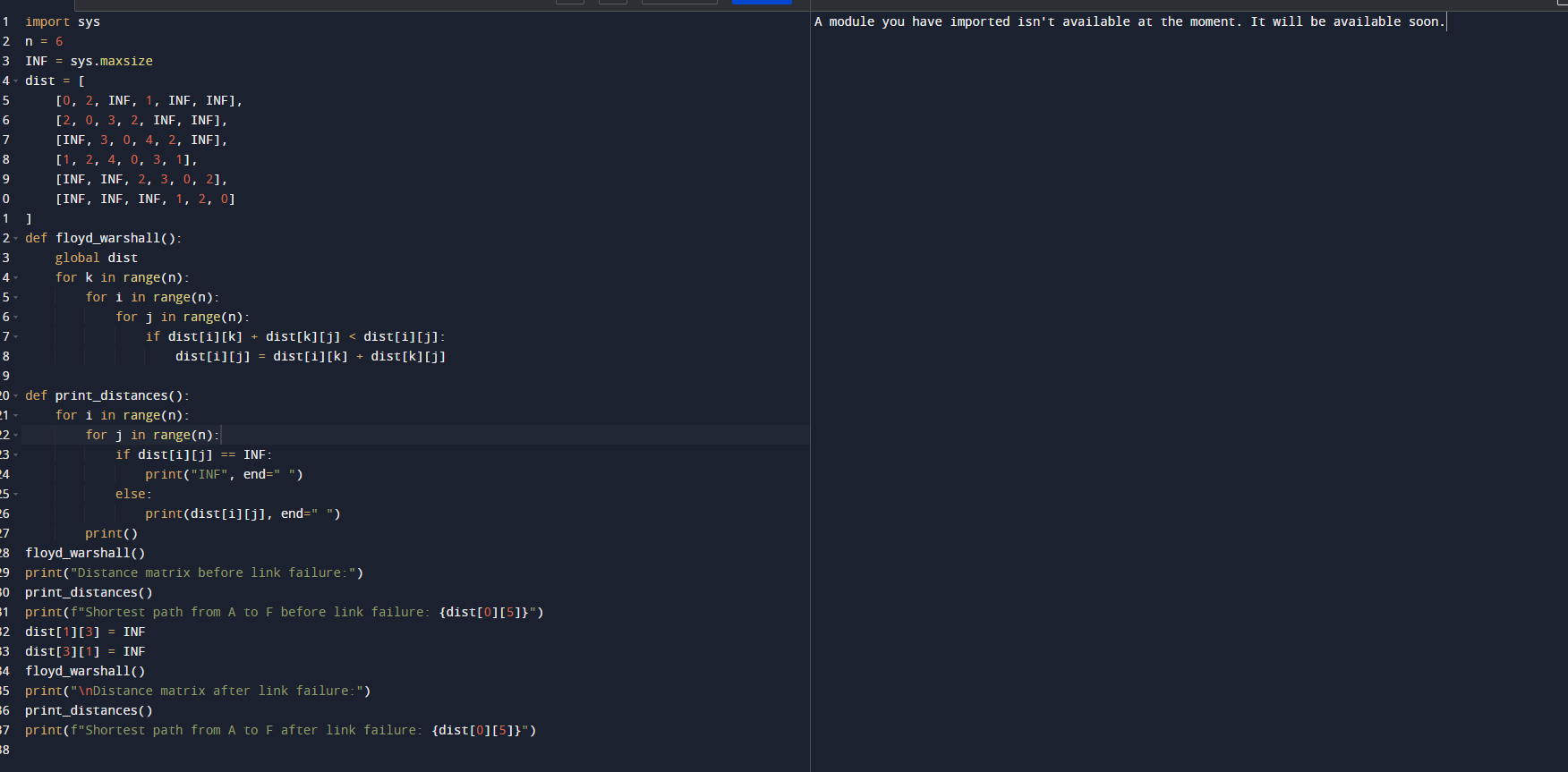
**pairs of routers. Simulate a change where the link between Router B and Router D fails.**

**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.**

****

**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**

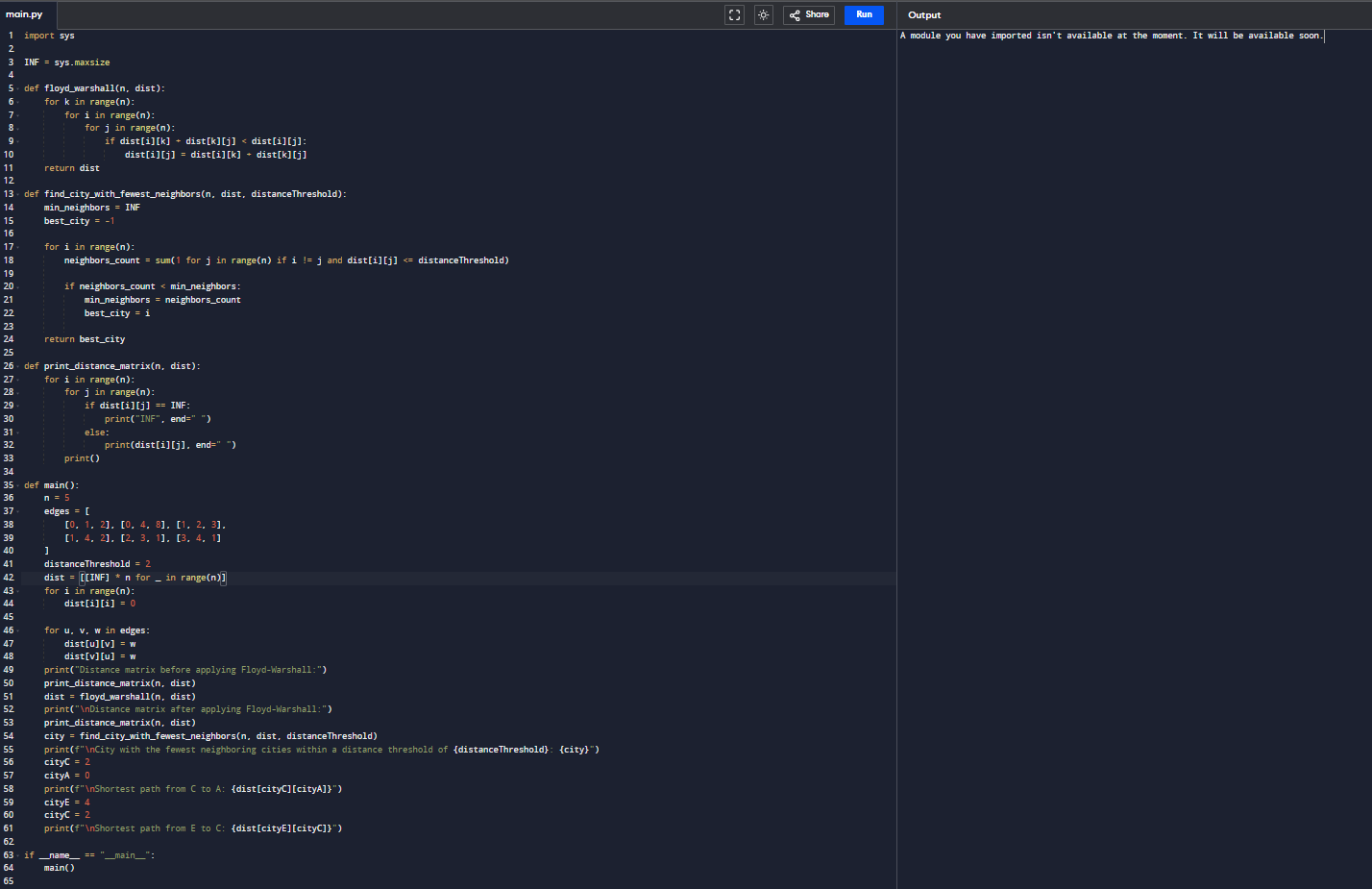
**D TO E 2**

**E TO D 4**

**D TO C 1**

**C TO D 3**

**Output : E to C = 5**

****

**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 = {01.02.,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**

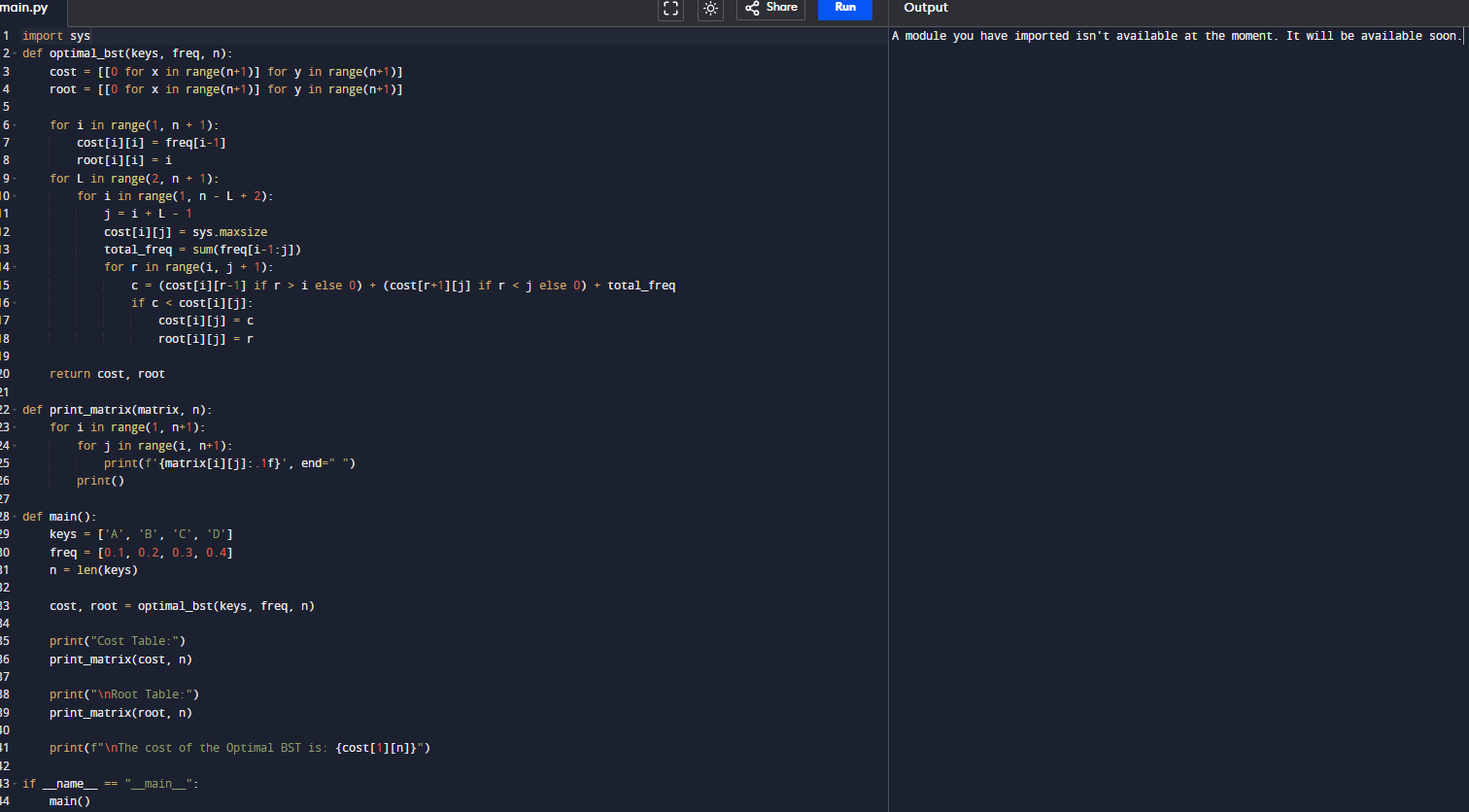
**Input: keys[] = {10, 12}, freq[] = {34, 50}**

**Output = 118**

**b)**

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

**Output = 142**

****

**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**

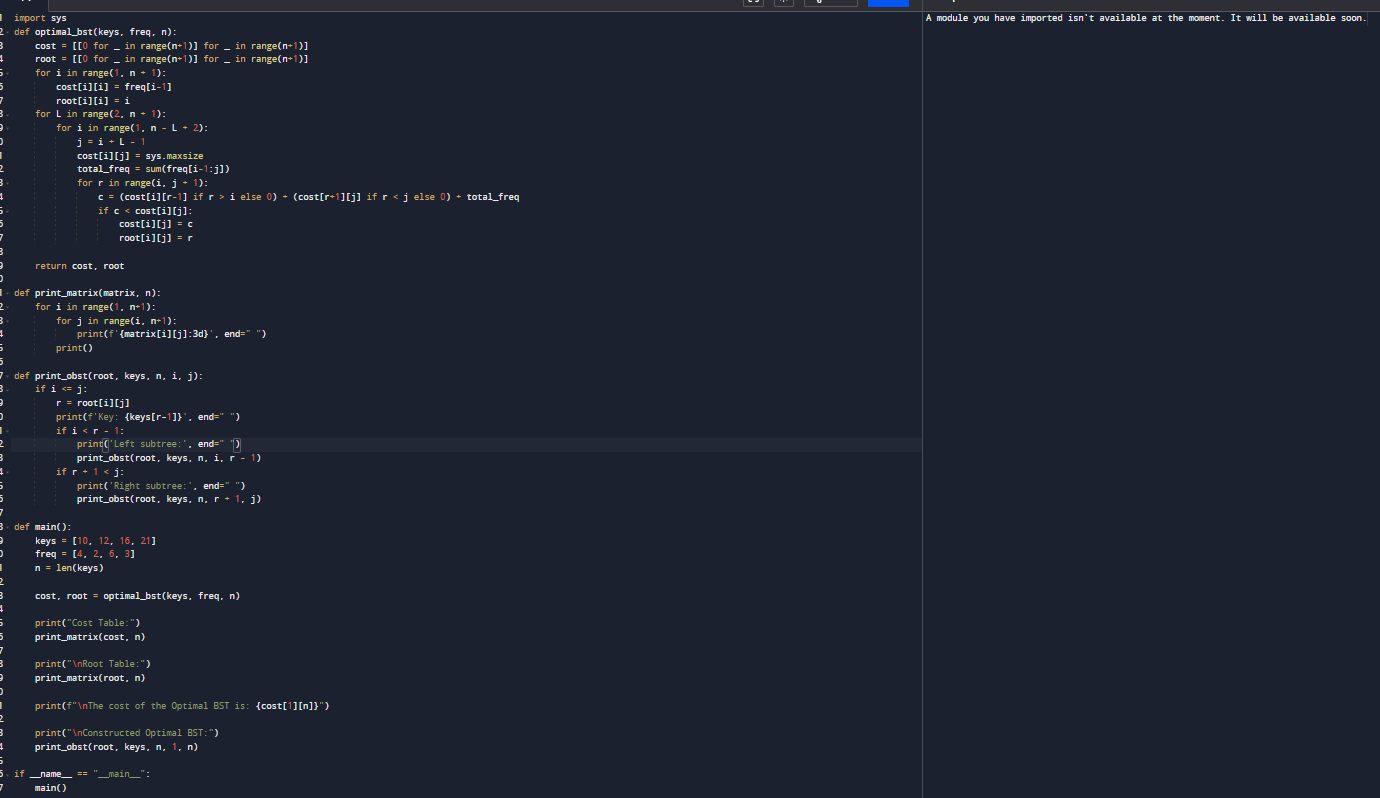
**a) Test cases**

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

**Output = 118**

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

**Output = 142**

****

**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:**

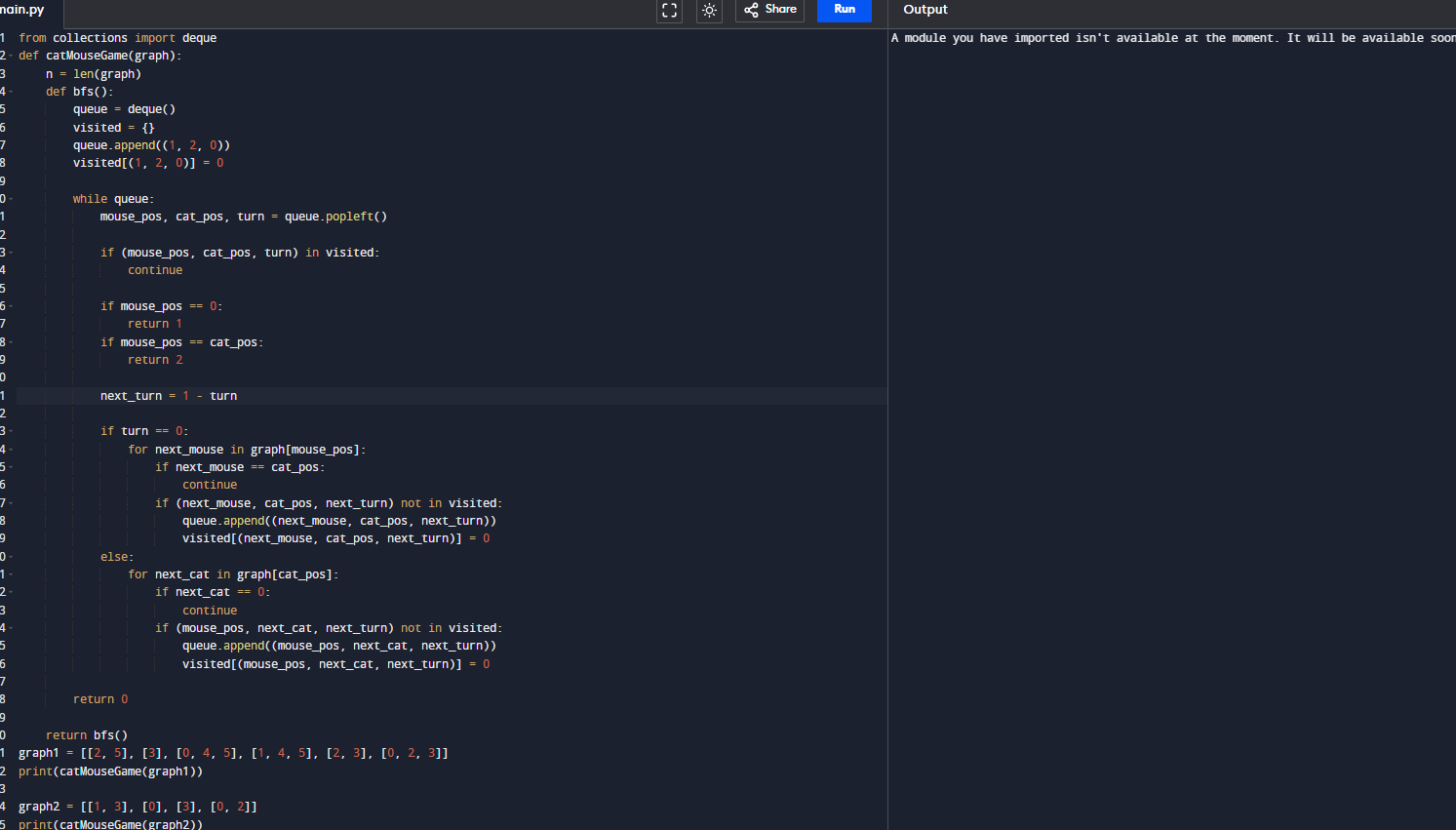
**Input: graph = [[2,5],[3],[0,4,5],[1,4,5],[2,3],[0,2,3]]**

**Output: 0**

**Example 2:**

**Input: graph = [[1,3],[0],[3],[0,2]]**

**Output: 1**

****

**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**

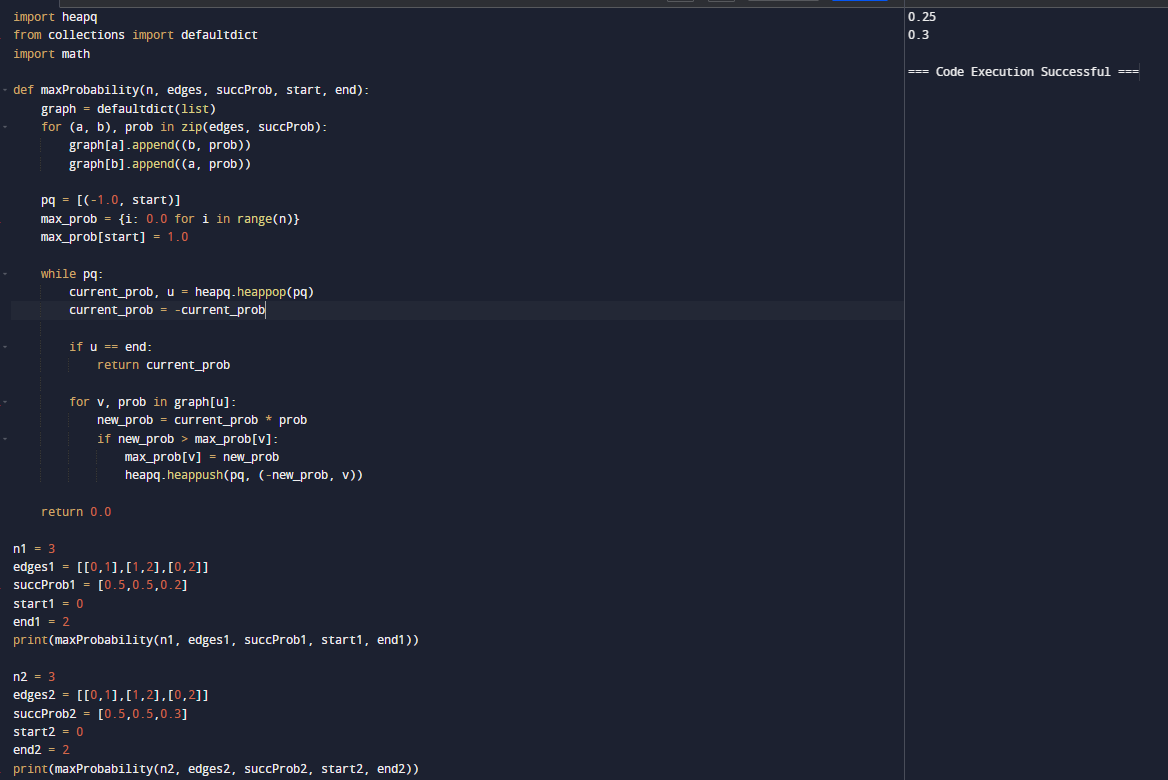
**Explanation: There are two paths from start to end, one having a probability of success =**

**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**

****

**8.** **There is a robot on an m x 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**

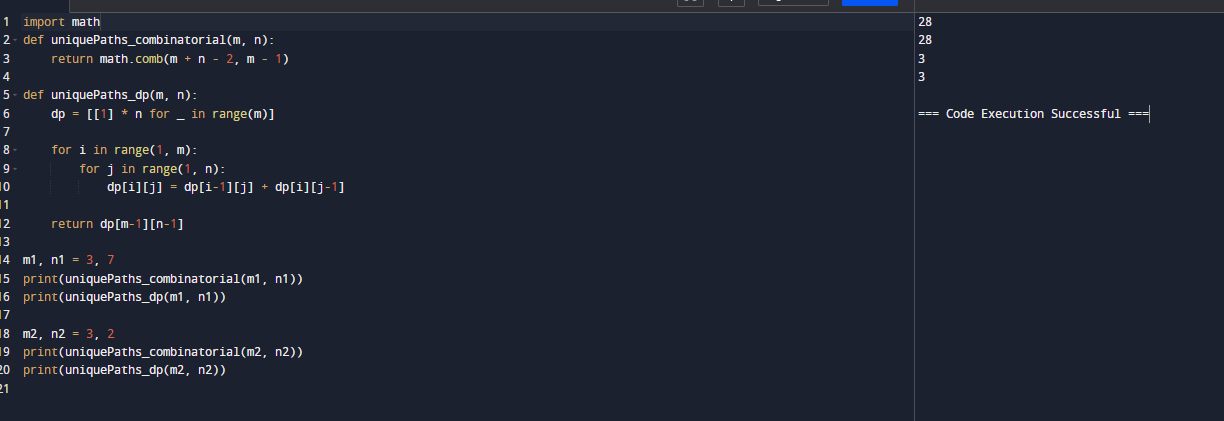
**Explanation: From the top-left corner, there are a total of 3 ways to reach the**

**bottom-right corner:**

**1. Right -> Down -> Down**

**2. Down -> Down -> Right**

**3. Down -> Right -> Down**

****

**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**

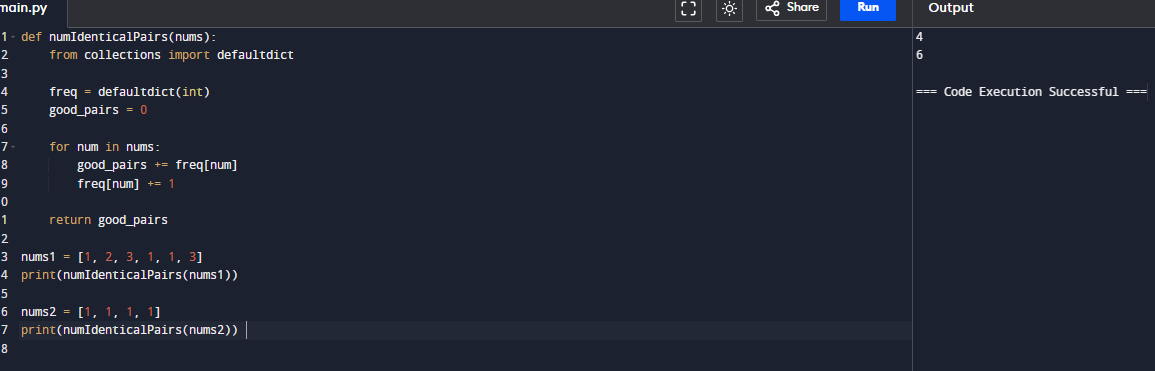
**Explanation: There are 4 good pairs (0,3), (0,4), (3,4), (2,5) 0-indexed.**

**Example 2:**

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

**Output: 6**

**Explanation: Each pair in the array are good**

****

**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]**

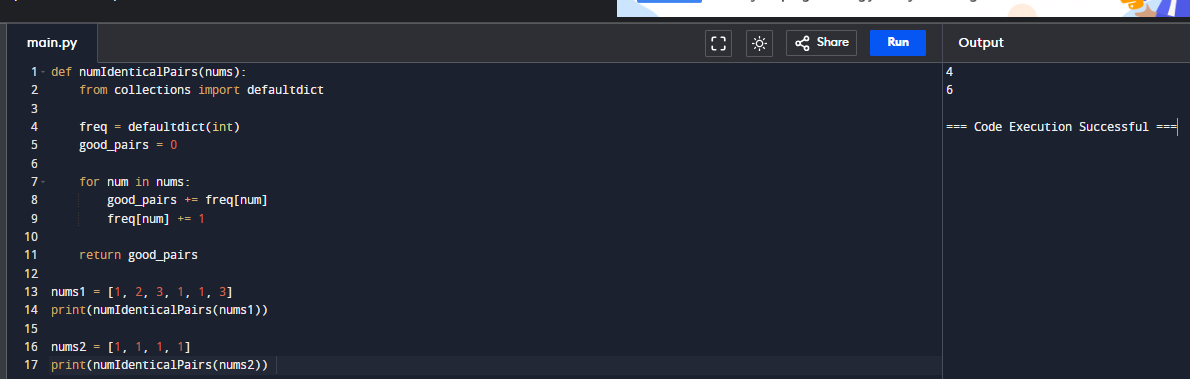
**City 1 -> [City 0, City 4]**

**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.**

****

**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:**

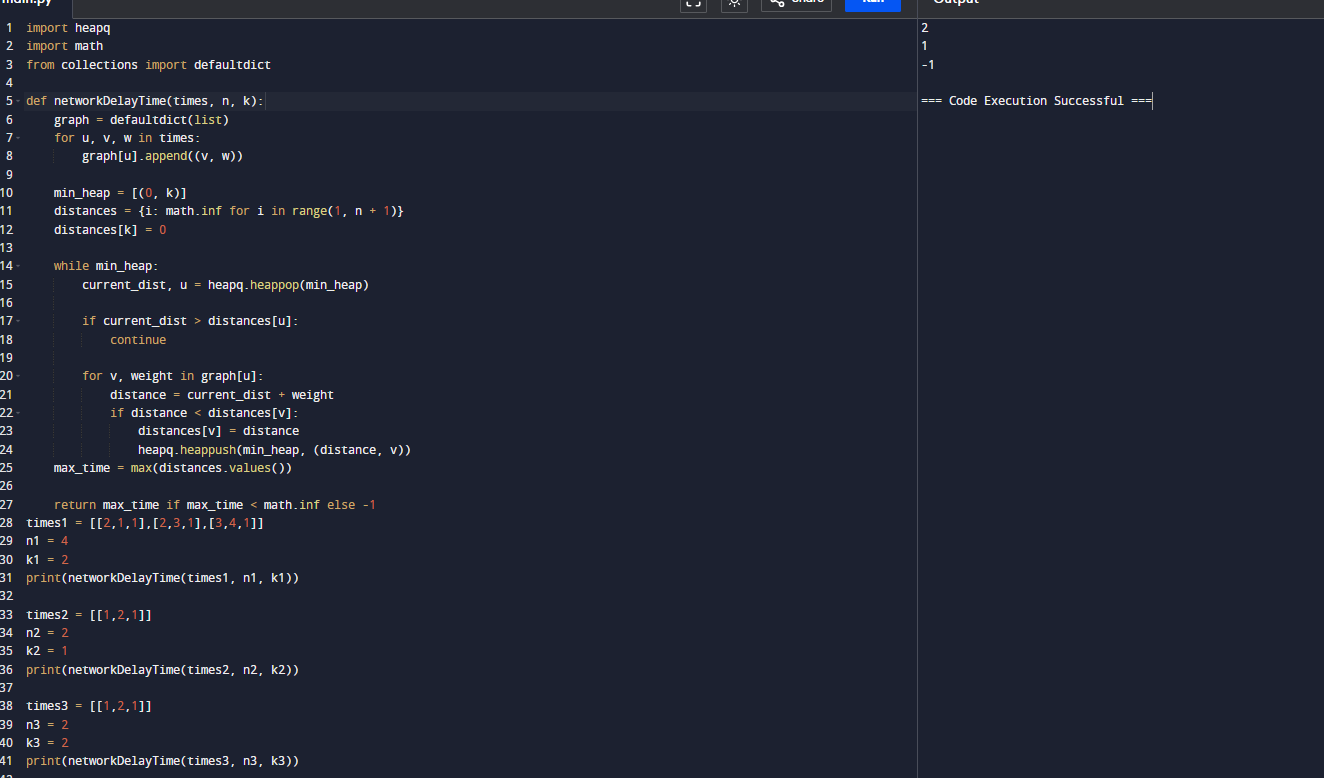
**Input: times = [[1,2,1]], n = 2, k = 1**

**Output: 1**

**Example 3:**

**Input: times = [[1,2,1]], n = 2, k = 2**

**Output: -1**

****