Task 1a

This code takes the input from the provided file, from which it takes the number of vertices and edges from the first line. It uses the vertices to generate a square matrix array of size n+1. It then uses the first two numbers in the following lines to populate the generated array with the third number, using the first two numbers, labelled u and v as indexes of the square array. After this is done, the whole matrix is shipped off line by line into an output file.

Task 1b

This code takes the input in the same manner as it does in Task 1a, although this code is mainly run through the function present known as "graphadj", which takes the whole list as well as the number of vertices and edges. Within this function it at first makes an empty dictionary that stores each node 'u' as the key, and then stores the node 'v' and the weight as a tuple pair, appending them into existing keys as they come into the key.

Task 2

To begin with, the code takes the number of vertices and edges from the first line of the input, and then makes an empty dictionary like in task 1b to create an adjacency list of each vertex present. The list is then shipped off to the BFS function which starts off from the starting vertex, defined '1' and then goes through the adjacency list in breadth first order, assigning levels to vertices based on their distance from the vertex assigned. Finally it writes the vertices and their assigned levels in the level dictionary into the output file.

Task 3

Similarly to task 2, the code takes the number of vertices and edges from the first line of the input, and then makes an empty dictionary like in task 1b to create an adjacency list of each vertex present. The dictionary, called here a graph, is then shipped off the DFS function alongside the edge count, which goes through the dictionary recursively marking visited nodes in a list and recording their parents in another list as well. After the whole process is done the code goes on to write the nodes in order of visit into an output file.

Task 4

Like the other tasks above it starts off the number of vertices and edges from the first line of the input, and then makes an empty dictionary like in task 1b to create an adjacency list, also known as a graph, of each vertex present. Next, it uses a DFS function that recursively traverses the graph, marking visited vertices and checking for cycles. The function returns true if a cycle is detected writing "Yes" to the output file else "No".

Task 5

First off, the algorithm starts by taking out the number of nodes, edges and the final destination node from the first line, after which the remainder is used to create an undirected graph in the form of a dictionary where each node is mapped to a list of its adjacent nodes. It then uses the BFS function to go through this graph, marking nodes visited and their parents in two lists like in Task 4. Using the parents list, it then calculates the shortest path from node 1 to the destination node, writing it alongside the time taken into an output file

Task 6

This algorithm at first extracts information about the dimensions of the grid from the first line and initialises and writes up a numpy array to represent the grid. Then using the 'jumanji_max_diamond' to find the max number of diamonds that can be collected in a connected region within the grid, and then uses the DFS subfunction to traverse the grid and count diamonds while avoiding obstacles, and tracking visited cells without revisiting them. It then goes on to write the max number of diamonds founds onto an output file.