Question 1 *ConnectGraph.java*

VERBAL DESCRIPTION

- This program finds the minimum number of edges required to make the given graph connected.
- To do that depth first approach was used. DFS finds all the nodes of the given graph that couldn't be reached in one call. If we know the number of nodes of the graph that weren't reachable, this means we need a total of total count - 1 edges to make the graph connected.
- Linked list was implemented to make an adjacency list of the given edges.
- A boolean array named visited is used to avoid any cycles that might exist in the graph.

PSEUDOCODE

1. The main method takes the input from the command line.
2. And creates an adjacency list of all the edges to make a graph.

```
public static void main(String[] args){

        takes command line inputs;

        calls the constructor for ConnectGraph and creates an
        adjacency list of size V(vertices) LinkedList[] adj;

        calls g.addEdge(u, v) to add the edges;

        calls the main DFS() function.
}


g.addEdge(u, v) {
        adj[v].add(u);
        adj[v].add(v); // since it is a undirected graph.
```

```
}


DFS(){
        creates an boolean array visited[] and initializes it to false;
        int count = 0;
        for ( int j from 1 till V+1){
                if visited[j] is false:
                        count += 1;
                        dfs(j, visited);
        }
Output : count - 1;
}


dfs(int k, boolean[] visited){
        mark the visited at k as true;
        neighbors = get k's neighbors from adj[];
        size_neighbors = get the size of neighbors;
        for (j from 1 till size_neighbors) {
                n = get the node at the position j from neighbors;
                if not visited:
                        dfs(n, visited);
        }
}
```

- DFS() is the main function that calls the recursive function dfs(). dfs() runs until the entire path has been traversed and then it goes back to the DFS() function where the count is incremented by 1 which indicates the number of nodes that were still not marked as visited and the dfs() function had to be called again from the main DFS() function.
- In the end we find the minimum number of edges required to make the undirected graph connected, which is the total count-1.

PROOF OF CORRECTNESS

This algorithm gives the correct output in every case since it finds the number of nodes that couldn't be marked as true in just one iteration. This

implies that there are in total total_nodes-1 edges required to make the graph connected.

TIGHT RUNNING TIME ESTIMATE

This algorithm runs in O(m+n) time complexity.

BRIEF REASONING BEHIND THE RUNNING TIME ESTIMATE

It runs in O(m+n) time complexity since there are in total m nodes and n edges in the given graph. The algorithm iterates m times and checks n edges to check the connectivity.

Question 2 *NumPaths.java*

VERBAL DESCRIPTION

- In this program we have to find the total number of shortest paths possible in the graph from the given start node till the given end node. To find those paths breadth first search approach was used and the number path and the total distance was maintained from the start node till all it's neighbors.
- distnace[] denotes the distance from the start node to the current node.
- Linked list and queue was implemented for this.
- Linked list to make the adjacency list and the queue to keep track of all the nodes whose neighbors were to be checked.
- Dynamically it gets the total number of shortest paths possible in the given graph.
- During the BFS traversal, the current node's neighbors are checked simultaneously and if they have not been visited then it is assigned a value of current node's distance + 1. Further, the paths of those neighbors are updated with the path of the current node.

PSEUDOCODE

```
NumPaths(vertices){
        LinkedList adj[vertices+1]
        boolean visited[vertices+1] // set it to false
        total_path[vertices+1] // set it to 0
        distnace[vertices+1] // set it to 100000
}

Public static void main(String[] args){
        takes the input from command line;
        created a graph of edges using an adjacency list;
        calls the totalShortestPath( start_node, dest_node);
}

totalShortestPath( start_node, dest_node){
        mark the visited[] as false;
        mark distance at start node as 0;
        total_path at start_node 1;
```

```
        visited at start_node as true;
        add the start node to the queue;

        while queue is not empty:

                s = remove the start node from the queue;

                neigh = get all the neighbors of s from the adj[];

                for (k =0 till length of neigh) {

                        neigh_1 = get the value of node k;

                        if neigh_1 is not visited:
                                mark neigh_1 as as true in visited[];
                                add neigh_1 to the queue.

                        if distance of neigh_1 > distance of start_node+1:
                                distance of neigh_1 = distance of start_node+1;
                                total_path of neigh_1 = total_path of start_node;

                        else if distance of neigh_1 == distance of start_node+1:
                                total_path of neigh_1 = total_path of start_node +1;
        }

        return the total_path at destination node;

}
```

The output will be stored in total_path[] at position dest_node.

PROOF OF CORRECTNESS

This algorithm works dynamiclally during which all the neighbors of the current node gets the total number of paths. In the end the total number of path at the destination node will be the total number of shortest paths in the graph.

TIGHT RUNNING TIME ESTIMATE

This algorithm runs in O(m+n) time complexity since it uses breadth first approach.

BRIEF REASONING BEHIND THE RUNNING TIME ESTIMATE

It runs in O(m+n) time complexity since there are in total m nodes and n edges in the given graph. There a total of m nodes therefore it will check each node and m edges of the graph.

Question 3 *Prerequisites.java*

VERBAL DESCRIPTION

- This program aims to find the longest chain of prerequisites from the given set of prerequisites.
- To find that it uses the depth first approach to find the longest path by contructing a directed graph of the given inputs.
- It uses dynamic approach to find the longest sequence possible.
  dp[node] = max(dp[node], 1 + max(dp[child1], dp[child2], dp[child3].

- dp[i] will be the length of the longest path from the start node i.
- Everytime the recursive dfs is run the dp of that node is updated by taking the maximum of the dp[node] and dp[node]+1.
  dp[n] = Math.max(dp[n], 1 + dp[ele]);
- The heart of the dp is:
  dp[n] = the longest length of path at node(vertex)

```
                1,              if node has no edges as in no child
dp[n] = {
                Math.max(dp[n], 1 + dp[ele]),    if node has child
```

PSEUDOCODE

```
Prerequisites{

    LinkedList adj[node+1];
    boolean visited[node+1];
    int dp[node+1];
}


addEdge(u,v){

    adj[u].add(v); // adds the edges to the adjacency list.
}
```

The main function calls the longestPath() method.

```
longestPath(int nodes){

        for (w = 0 till nodes){

                if w is not visited:
                        dfs(w, visited);
        }

        int longest_path = 0;

        for(j = 0 till length of dp[]){

                // finds the longest path from the dp[].
                longest_path = Math.max(longest_path, dp[j]);
        }

        outputs the longest_path+1 found which is the answer.
}

dfs(w, visited){

        mark w as true in visited[];

        neighbors = get all neighbors of w;

        for(k = 0 till length of neighbors){

                ele = get value of node k;

                if Ele is false in visited:
                        dfs(ele, visited);
        }

        compute the dp[n] by taking Math.max(dp[n], 1 + dp[ele]);
}
```

Since, it finds the length of the longest path we add 1 to it and output it because the total number of nodes will total length + 1.

PROOF OF CORRECTNESS

- The naive aproach for this program would take $O(n^2)$ time where the length of the longest path will be calculated from every node using dfs.

- But, the program uses depth first approach along with dynamic approach to compute the longest path it will be executed in O(m+n) time complexity since, dp[node] will be computed using the recursive formula dp[node] = max(dp[node], 1 + max(dp[child1], dp[child2], dp[child3].
- Therefore, it will always calculate the longest possible length in the given time complexity.

TIGHT RUNNING TIME ESTIMATE

This algorithm runs in O(m+n) time complexity since it uses depth first approach along with dynamic approach.

BRIEF REASONING BEHIND THE RUNNING TIME ESTIMATE

It runs in O(m+n) time complexity since there are in total m nodes and n edges in the given graph. There a total of m nodes therefore it will check each node and m edges of the graph and to find the longest path it uses dynamic approach which ascertains that the algorithm runs in the O(m+n) time complexity.

Question 4 *DoubleTrouble.java*

VERBAL DESCRIPTION

- This program uses the concept of breadth first search to find the minimum number moves to take to ensure that THING1 and THING2 exit the environment they are in.
- To ensure that the conditions where and when THING1 and Thing2 can move the State.java has been implemented. It checks all the conditions and then moves accordingly. The conditions handles the empty spot, walls and positions THING1 and THING2 can move to.
- While traversing, the THINGS can move in four directions in total: North, South, East and West keeping the following conditions in mind:-
    1) Both the things stay at one position since they can't go anywhere else.
    2) Thing1 moves to a new position but Thing2 stays put.
    3) Thing2 moves to a new position but Thing1 stays put.
    4) Both the things can move to a new position since there are no obstacles as of yet.

PSEUDOCODE

```
Public static void main(String[] args){

        creates an object for State called state.

        takes input from command lines and sends to the method
        configure_state() which sets the values as 0 or 1 depending on the
         type(".", "x", "1", "2").

        Calls the BreadthFirstSearch(state)
}


BreadthFirstSearch(State state){

        set current_position = state;
        create a boolean visited 4-d axb array ;
        create an int move 4-d axb array;
```

```
        set visited to true at the current position;
        create an object for Queue;
        append current positon to queue;

        while size of queue is not 0:

                dequeue the node;
                neigh = get all the neighbors of node;

                for(j=0 till length of neigh){

                        get node at j and check if it visited;

                        if not visited:
                                mark as true;

                        moves[s.a_1][s.b_1][s.a_2][s.b_2] = moves[state.a_1]
                                                [state.b_1][state.a_2][state.b_2] + 1;

                        if node has exited the environment:
                                output the minimum moves taken;

        else:
                output "STUCK"

                }

}
```

PROOF OF CORRECTNESS

TIGHT RUNNING TIME ESTIMATE

This algorithm will run in O((ab)^2) time complexity for worst case when there are absolutely no obstacles and the THINGS will have to go through all the locations in the environment and the exiting state of the environment will be the last state breadth first search traversal will reach.

BRIEF REASONING BEHIND THE RUNNING TIME ESTIMATE

There are in total four for loops to check all the four directions it can move in. Two of them will run till dimension1 and the other two will run till dimension2. Therefore, the worst case time complexity will be O((ab)^2). In the worst case, THINGS will have to go through all the locations in the environment and the exiting state of the environment will be the last state breadth first search traversal will reach