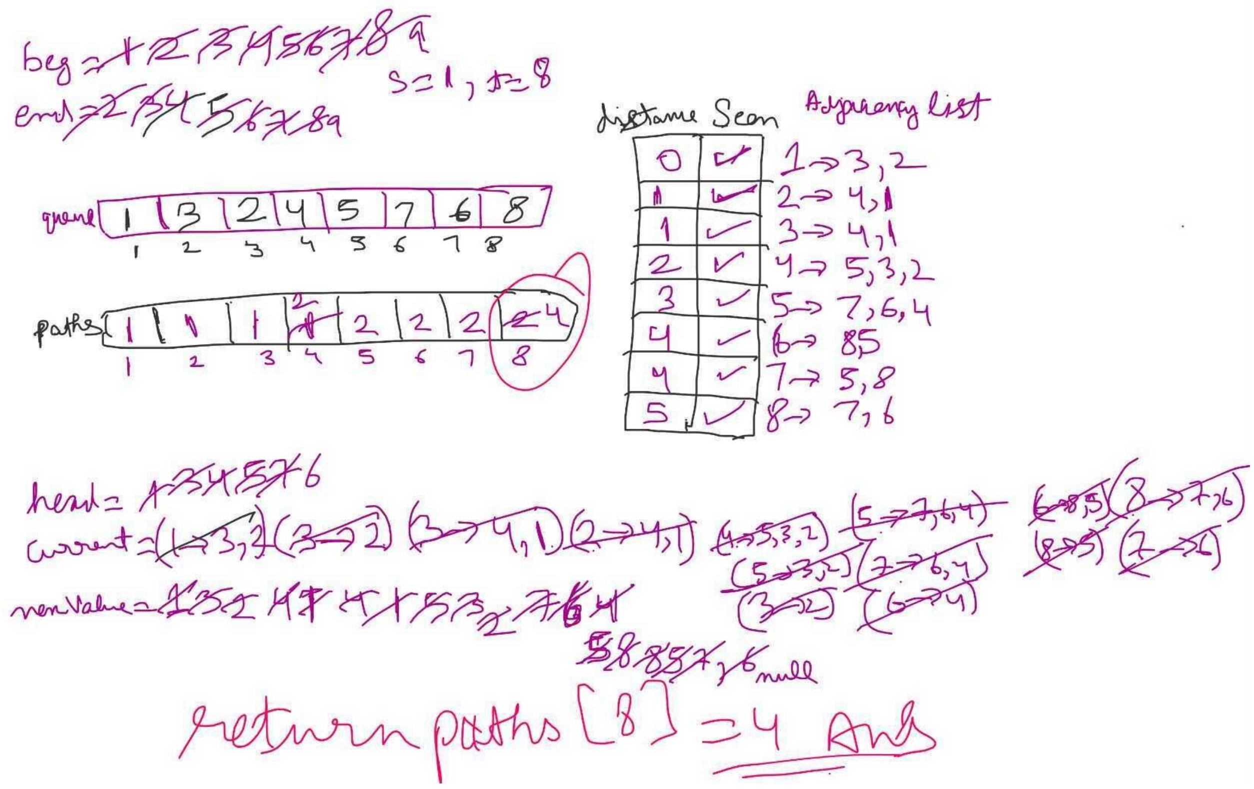
Problem 1

Description: The problem given was to find the number of paths there are to reach from a starting node to an ending node in an undirected graph in the minimum number of steps. I used BFS to find the minimum path and counted the total number of paths, where I reached the ending node in the minimum steps during the traversal.

Pseudo Code:

```
1. int getNumberOfPaths(Node[] input, s, t):
    a. beg=1
    b. end=2
    c. paths=int [ length of input ]
    d. seen=boolean [ length of input ]
    e. queue = int [ length of input ]
    f. distance = int [ length of input ] initialize this to Integer.MAX_VALUE
    g. queue[1]=s
    h. paths[s]=1
    i. seen[s]=true
    j. distance[s]=0
    k. while (beg<end):
        i. head= queue[beg]
        ii. current = input[head]
        iii. while current is not null :
            1. nextValue= current.value
            2. if not seen[nextValue]:
                a. queue[end]= nextValue
                b. seen[nextValue] =true
                c. end++
            3. if distance[nextValue] > distance[head] + 1:
                a. paths[nextValue] = paths[head]
                b. distance[nextValue] = distance[head]+1
            4. else if distance[nextValue] == distance[head]+1:
                a. paths[nextValue]+=paths[head]
            5. current = current.next // moving to the next node
        iv. beg++
    l. return paths[t]
```



Running Time:

- Taking the input and creating the adjacency matrix would be O(n + m)
- getNumberOfPaths:
    - Initializing distance array would just be O(n)
    - BFS traversal inside the while loops would again be O(n + m)

Total running time estimate would be O(n + m)+ O(n)+ O(n + m) which is simply O(n + m)

Problem 2

Description: The problem was to find the longest chain from the given input of the adjacency list of courses and their prerequisites. To find the longest chain in a DAG we can use topological sort and dfs and then count the number of courses in the chain which would be our answer.

Pseudo Code:

```
graph[ ] of type Node // initialize the adjacency list from the standard input

stack of type Stack

1. static void TopOrder()
    a. boolean[] seen = new boolean[graph.length]
    b. int[] dist = new int[graph.length]
    c. for i from 0 to length of graph:
        i. seen[i] = false    //set all seens to fals
        ii. dist[i] = 1   //Initialize dists to zero
    d. for i = 1 to length of graph:
        i. if not seen[i]:
            1. TopHelper(graph[i], seen)  //call top helper on each graph head
    e. While stack.getSize() > 0
        i. Node n = stack.peek()  //look at next node
        ii. stack.pop()    //pop node
        iii. for i = 0 to n.getNumberNodes()    //for each neighboring node
            1. if dist[n.getNode(i).val] < dist[n.val] + 1   //distance is more
                a. dist[n.getNode(i).val] = dist[n.val] + 1   //update distance
    f. int max = 0
    g. for i = 0 to graph.length
        i. if dist[i] > max
            1. max = dist[i]
    h. print(max)

2. static void TopHelper(Node n, boolean[] seen)
    a. seen[n.val] = true //set current node is seen
    b. for i = 0 to n.getNumberNodes()    // for each node connected to this node
        i. Node tempNode = n.getNode(i)
        ii. If not seen[tempNode.val]:   //if node is not seen
            1. TopHelper(graph[tempNode.val], seen)   //move to this node
    c. stack.push(n)  //push node
```

Running Time:

- Taking the input and creating the adjacency matrix would be O(n + m)
- TopOrder():
    - For loop on line c is O(n)
    - For loop on line d is O(n)
        - Calling the topHelper() for all the nodes which will be combined O(n+m)
    - While loop will be O(n) size of the stack
        - The for loop will be for all the nodes attached to the current node in the graph which will be O(n+m)
    - For loop for the length of the graph array which will be O(n)

Total running time estimate would be O(n)+O(n)+O(n + m)+ O(n) +O(n + m) +O(n) which is same as

O(n + m)

Problem 3

Description: The problem says to find the number of minimum steps needed to exit a maze for both thing1 and thing2 and they both must exit the maze at the same time. I used BFS and created a state object that would store data for each configuration and after traversing through the 2d matrix, we can check whether or not it has reached the end and count the number of steps needed to do that.

Pseudo Code:

```
Static class Coords:
        Has properties x1,x2,y1,y2, distance

canExitMaze ( char[][] input, Coords iniCoords):
    a. rows= length of input
    b. cols= length of first row the input
    c. seen = [rows][cols][rows][cols] Boolean array initialized to false
    d. queue = [ rows*rows*cols*cols ] of type Coords
    e. beg=1
    f. end=2
    g. seen[iniCoords.x1][ iniCoords.y2][ iniCoords.x2][ iniCoords.y2]= true
    h. queue[1]= iniCoords
    i. while(begin<end):
        i. Coords headCoord= queue[beg]
        ii. x1,x2,y1,y2 initialize from headCoord object
        iii. Thing1= input[x1][y1]
        iv. Thing2= input[x2][y2]
        v. If thing1== # and thing2 == # and they both are not at the same position:
            1. Return headCoord.distance
        vi. If (either thing == # and the other thing is not) or they are at the same position :
            1. Continue
        vii. If not seen[x1+1][y1][x2+1][y2] and input[x1+1][y1]!='x' and
             input[x2+1][y2]!='x':
            1. queue[end]=new Coords(x1+1,y1,x2+1,y2, headCoord.distance+1)
            2. seen[x1+1][y1][x2+1][y2]=true
            3. end++
        viii. else if not seen[x1+1][y1][x2][y2] and  input[x1+1][y1]!='x':
            1. queue[end]= new Coords(x1+1,y1,x2,y2, headCoord.distance+1)
            2. seen[x1+1][y1][x2][y2]=true
            3. end++
        ix. else if not seen[x1][y1][x2+1][y2] and input[x2+1][y2]!='x':
            1. queue[end]=new Coords(x1,y1,x2+1,y2, headCoord.distance+1)
            2. seen[x1][y1][x2+1][y2]=true
            3. end++
        x. if  not seen[x1-1][y1][x2-1][y2] and input[x1-1][y1]!='x' and input[x2-
           1][y2]!='x':
            1. queue[end]=new Coords(x1-1,y1,x2-1,y2, headCoord.distance+1)
            2. seen[x1-1][y1][x2-1][y2]=true
```

```
                3. end++
        xi. else if not seen[x1-1][y1][x2][y2] and input[x1-1][y1]!='x':
                1. queue[end]=new Coords(x1-1,y1,x2,y2, headCoord.distance+1)
                2. seen[x1-1][y1][x2][y2]=true
                3. end++
        xii. else if not seen[x1][y1][x2-1][y2] and input[x2-1][y2]!='x':
                1. queue[end]=new Coords(x1,y1,x2-1,y2, headCoord.distance+1)
                2. seen[x1][y1][x2-1][y2]=true
                3. end++
        xiii. if not seen[x1][y1+1][x2][y2+1] and input[x1][y1+1]!='x' and
              input[x2][y2+1]!='x'
                1. queue[end]= new Coords(x1,y1+1,x2,y2+1, headCoord.distance+1)
                2. seen[x1][y1+1][x2][y2+1]=true
                3. end++
        xiv. else if not seen[x1][y1+1][x2][y2] and input[x1][y1+1]!='x'
                1. queue[end]= new Coords(x1,y1+1,x2,y2, headCoord.distance+1)
                2. seen[x1][y1+1][x2][y2]=true
                3. end++
        xv. else if not seen[x1][y1][x2][y2+1] and input[x2][y2+1]!='x'
                1. queue[end]=new Coords(x1,y1,x2,y2+1, headCoord.distance+1)
                2. seen[x1][y1][x2][y2+1]=true
                3. end++
        xvi. if not seen[x1][y1-1][x2][y2-1] and input[x1][y1-1]!='x' and input[x2][y2-1]!='x'
                1. queue[end]=new Coords(x1,y1-1,x2,y2-1, headCoord.distance+1)
                2. seen[x1][y1-1][x2][y2-1]=true
                3. end++
        xvii. else if not seen[x1][y1-1][x2][y2] and input[x1][y1-1]!='x':
                1. queue[end]=new Coords(x1,y1-1,x2,y2, headCoord.distance+1)
                2. seen[x1][y1-1][x2][y2]=true
                3. end++
        xviii. else if seen[x1][y1][x2][y2-1] and input[x2][y2-1]!='x':
                1. queue[end]=new Coords(x1,y1,x2,y2-1, headCoord.distance+1)
                2. seen[x1][y1][x2][y2-1]=true
                3. end++
2. return -1;
```

Running Time:

- We were given a a*b matrix, so O(a*b) to take the standard input
- In the canExitMaze function we are creating configurations for two things, thing1 and thing2, for each thing we have a O(a*b) and for 2 things we'll have O((a*b)^2)
    - While loop will run for all the possible configurations until we have explored the matrix completely, for changing values of configurations for each thing going in all 4 directions which is equal to the square of the size of the matrix, since we can have multiple configurations.

The overall running time estimate of the program will be given by O((a*b)^2).