**BACK TRACK**

Backtracking is an algorithmic-technique for solving problems recursively by trying to build a solution incrementally, one piece at a time, removing those solutions that fail to satisfy the constraints of the problem at any point of time.

**There are three types of problems in backtracking –**

1. Decision Problem – In this, we search for a feasible solution.
2. Optimization Problem – In this, we search for the best solution.
3. Enumeration Problem – In this, we find all feasible solutions.

**Example**

**RATE IN MAZE**

****

Now look at the diagram. Suppose a rat have to reach the destination from the source via a path. Now the dark boxes are the block ones. Means rat can not go there. And it has to reach the destination by following the white paths.

So this is the question now we have to solve it using back track. A Maze is given as N\*N binary matrix of blocks where source block is the upper left most block i.e., maze[0][0] and destination block is lower rightmost block i.e., maze[N-1][N-1]. A rat starts from source and has to reach the destination. The rat can move only in two directions: forward and down. In the maze matrix, 0 means the block is a dead end and 1 means the block can be used in the path from source to destination. Note that this is a simple version of the typical Maze problem.

And this will be the following representation of the diagram –

{1, 0, 0, 0}

{1, 1, 0, 1}

{0, 1, 0, 0}

{1, 1, 1, 1}

And the output should give us like –

{1, 0, 0, 0}

{1, 1, 0, 0}

{0, 1, 0, 0}

{0, 1, 1, 1}

The path is clearly visible. So the path will be this –



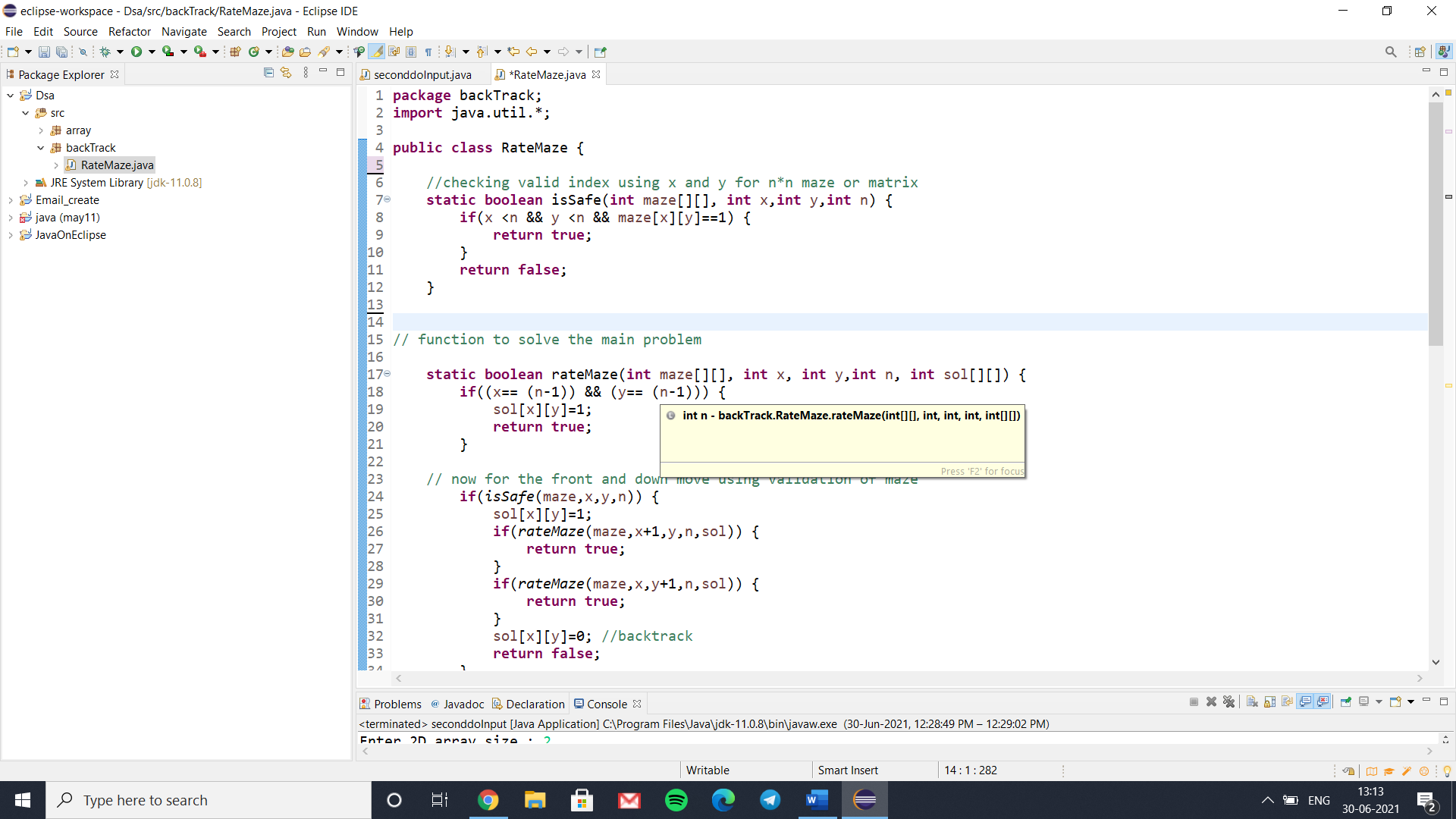
**Sol.**

**Approach:**

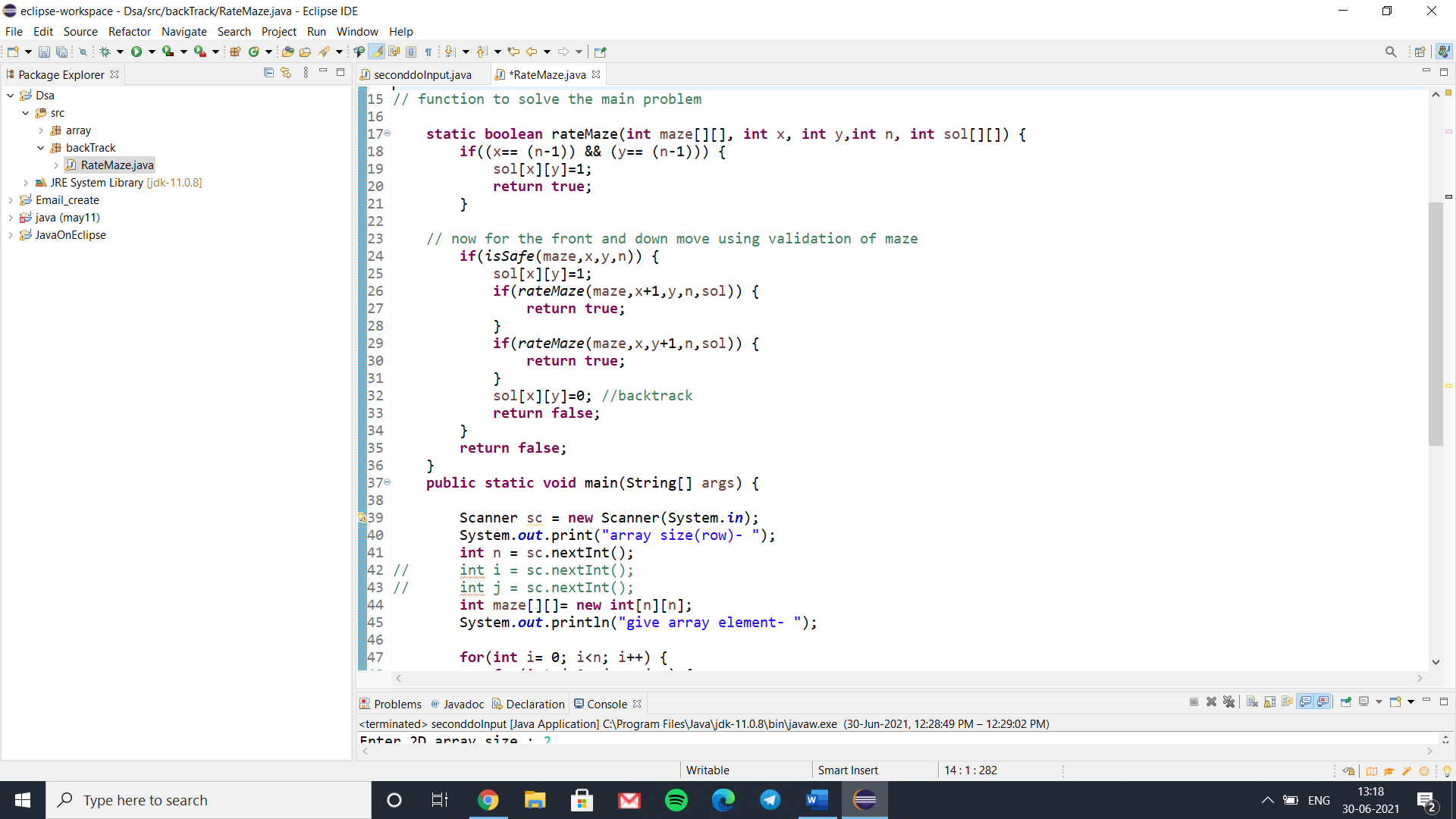
* **So we will use recursive function here.**
* **Wo will make some Boolean functions and we will call the functions later in our main code. So this will make the code more clear and understable.**
* **So the function will check the path. And it will decide that the rat reaches the destination or not.**
* **If the path does not exist on the both side then it will backtrack and try other paths.**

**ALGORITHM**

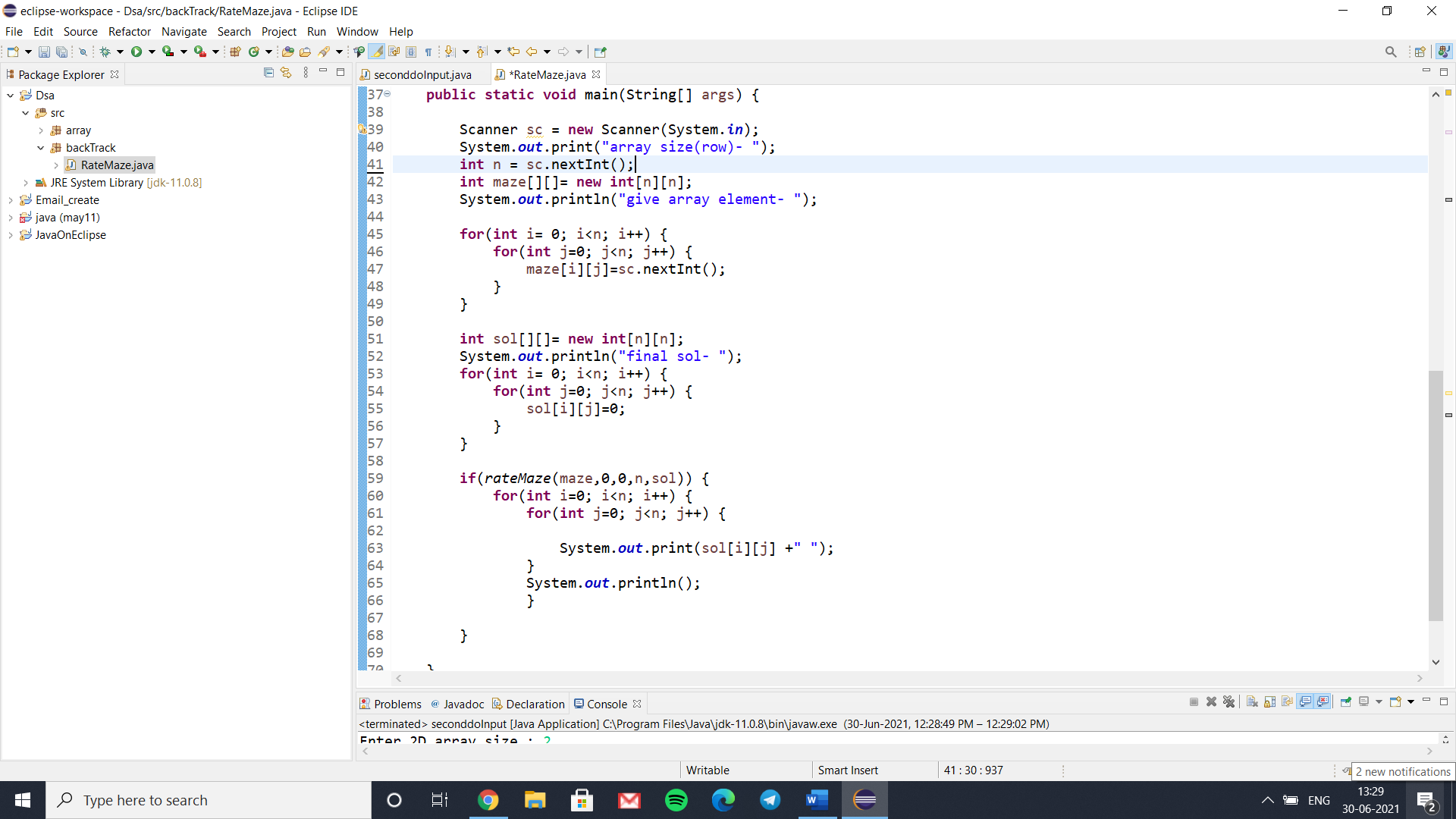
1. Creating a function which will determine weather the path is safe or not.
2. If it is safe mark it as maze[ i ][ j ] = 1. Where i is row and j is column.
3. Then create a solution matrix that will store the final result. And that will initially filled with 0.
4. Creating a recursive matrix that will take the input and can call the maze solver function.
5. Check if the position is out of the matrix. If it is then return not valid.
6. Then call the safe mark function and mark the position output [ i ][ j ] = 1.and check if the current position is the destination or not. If it reach the destination then print the matrix.
7. For incrementing the position recursively call and set the position as ( i + 1 , j) [ for row increment ] and ( i + j +1 ) [ for column increment ].
8. If it cannot go any path further then set the [i][j] as 0 for reaching the initial position it came from. ( main back tracking )
9. Then unmark and position and move forward.
10. And print the output

**Definition for Coding**

1. This is the safe function where we can check the box or the parameter give is safe or not by setting the column and row at 1. Because if it is one then the path we are going to take otherwise if it is 0 then not.

2. So this is the main function for solving this problem. So here we will take a input matrix as maze[][]. And check weather the solution matrix is less than the matrix we have entered. Other wise the matrix will go outside of the size. If it is in the matrix then set the solution matrix as 1.

Now call the safe function and checking the safe path. So if the path is safe we will set it to sol[x][y] = 1. And now check weather the next incrementation is row wise or column wise. If row wise then set x+1 to increment the row and if it is for column then set it to y+1. And in case if it will not able to go for any direction then backtrack the location as the initial position as sol[x][y] = 0.



Now time for the main body and calling the function. So here we take a int n for creating the matrix. And recursively call the maze function matrix and set it as the input matrix. Here we take the n\*n input. And then for getting the input we have to start a loop for row and column so here we start a for loop where i take the row input and j take the column.

And then we set the solution matrix and do the same as we did for input. And set the initial value as sol[i][j] = 0. Because for the output we want to see the main path as 1 and change the other values of the matrix with 0. And also that computer will not able to give us garbage value for this statement.

Finally we call the “ rateMaze “ or the main function and pass the value. And here notice that we set the initial value of x and y as 0. For starting the count from 0. And then we will print the output by again taking the row and column and finally print that using “sol” matrix.

**CODE-**

**package** backTrack;

**import** java.util.\*;

**public** **class** RateMaze {

//checking valid index using x and y for n\*n maze or matrix

**static** **boolean** isSafe(**int** maze[][], **int** x,**int** y,**int** n) {

**if**(x <n && y <n && maze[x][y]==1) {

**return** **true**;

}

**return** **false**;

}

// function to solve the main problem

**static** **boolean** rateMaze(**int** maze[][], **int** x, **int** y,**int** n, **int** sol[][]) {

**if**((x== (n-1)) && (y== (n-1))) {

sol[x][y]=1;

**return** **true**;

}

// now for the front and down move using validation of maze

**if**(*isSafe*(maze,x,y,n)) {

sol[x][y]=1;

**if**(*rateMaze*(maze,x+1,y,n,sol)) {

**return** **true**;

}

**if**(*rateMaze*(maze,x,y+1,n,sol)) {

**return** **true**;

}

sol[x][y]=0; //backtrack

**return** **false**;

}

**return** **false**;

}

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

Scanner sc = **new** Scanner(System.***in***);

System.***out***.print("array size(row)- ");

**int** n = sc.nextInt();

**int** maze[][]= **new** **int**[n][n];

System.***out***.println("give array element- ");

**for**(**int** i= 0; i<n; i++) {

**for**(**int** j=0; j<n; j++) {

maze[i][j]=sc.nextInt();

}

}

**int** sol[][]= **new** **int**[n][n];

System.***out***.println("final sol- ");

**for**(**int** i= 0; i<n; i++) {

**for**(**int** j=0; j<n; j++) {

sol[i][j]=0;

}

}

**if**(*rateMaze*(maze,0,0,n,sol)) {

**for**(**int** i=0; i<n; i++) {

**for**(**int** j=0; j<n; j++) {

System.***out***.print(sol[i][j] +" ");

}

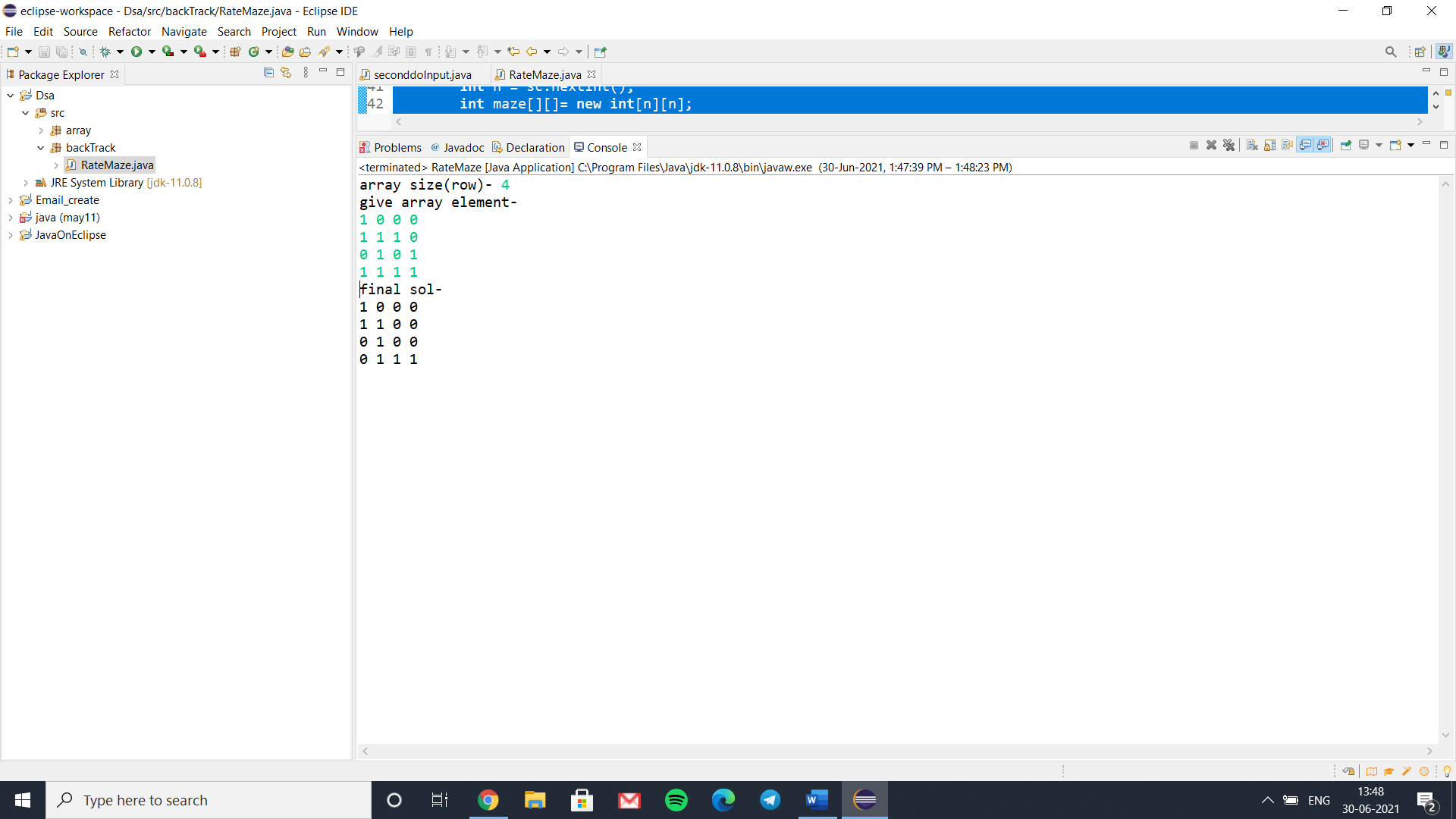
System.***out***.println();

}

}

}

}

**OUTPUT**

**References –**

* Geeksforgeeks (<https://www.geeksforgeeks.org/rat-in-a-maze-backtracking-2/>)
* Apnikaksha ratemaze problem.