Design And Analysis Of Algorithms

Project Using Backtracking Team Members :

A.R.Arun Roshan(RA2011027010150)
M.Vignan Reddy(RA2011027010154)
M.Sandeep Reddy(RA2011027010165)

CONTRIBUTION TABLE

S.NO.	NAME	REGISTRATION NO.	CONTRIBUTION
1	A.R.ARUN ROSHAN	RA2011027010150	ALGORITHM ANALYSIS
2	M.SANDEEP REDDY	RA2011027010165	PLANNING AND REPORT WRITING
3	M.VIGNAN REDDY	RA2011027010154	DEVELOPER

roblem Statement : Rat In A Maze

You are given a maze in the form of a matrix of size n * m. Each cell is either clear or blocked denoted by 1 and 0 respectively. A rat sits at the top-left cell and there exists a block of cheese at the bottom-right cell. Both these cells are guaranteed to be clear. You need to find if the rat can get the cheese if it can move only in one of the two directions - down and right. It can’t move to blocked cells.

Gets Cheese

Does Not Get Cheese

Approach: Backtracking

1. Backtracking is an algorithmic-technique for solving problems recursively by trying to build a solution incrementally. Solving one piece at a time and removing those solutions that fail to satisfy the constraints of the problem at any point of time (by time, here, is referred to the time elapsed till reaching any level of the search tree) is the process of backtracking.

2. Form a recursive function, which will follow a path and check if the path reaches the destination or not. If the path does not reach the destination, then backtrack and try other paths.

ALGORITHM

1.Firstly, create a matrix to store your solution. It should contain all zeros and should be of the same size as the maze matrix.

2.Make a recursive call with the position of the rat in the matrix: initial maze matrix, solution matrix

3.If the position provided is invalid, return from the function.

4. Otherwise, mark the position as 1 in the solution.

matrix.

5. Recursively call for position (i+1, j), (i, j+1), (i-1, j), and (i, j-1).

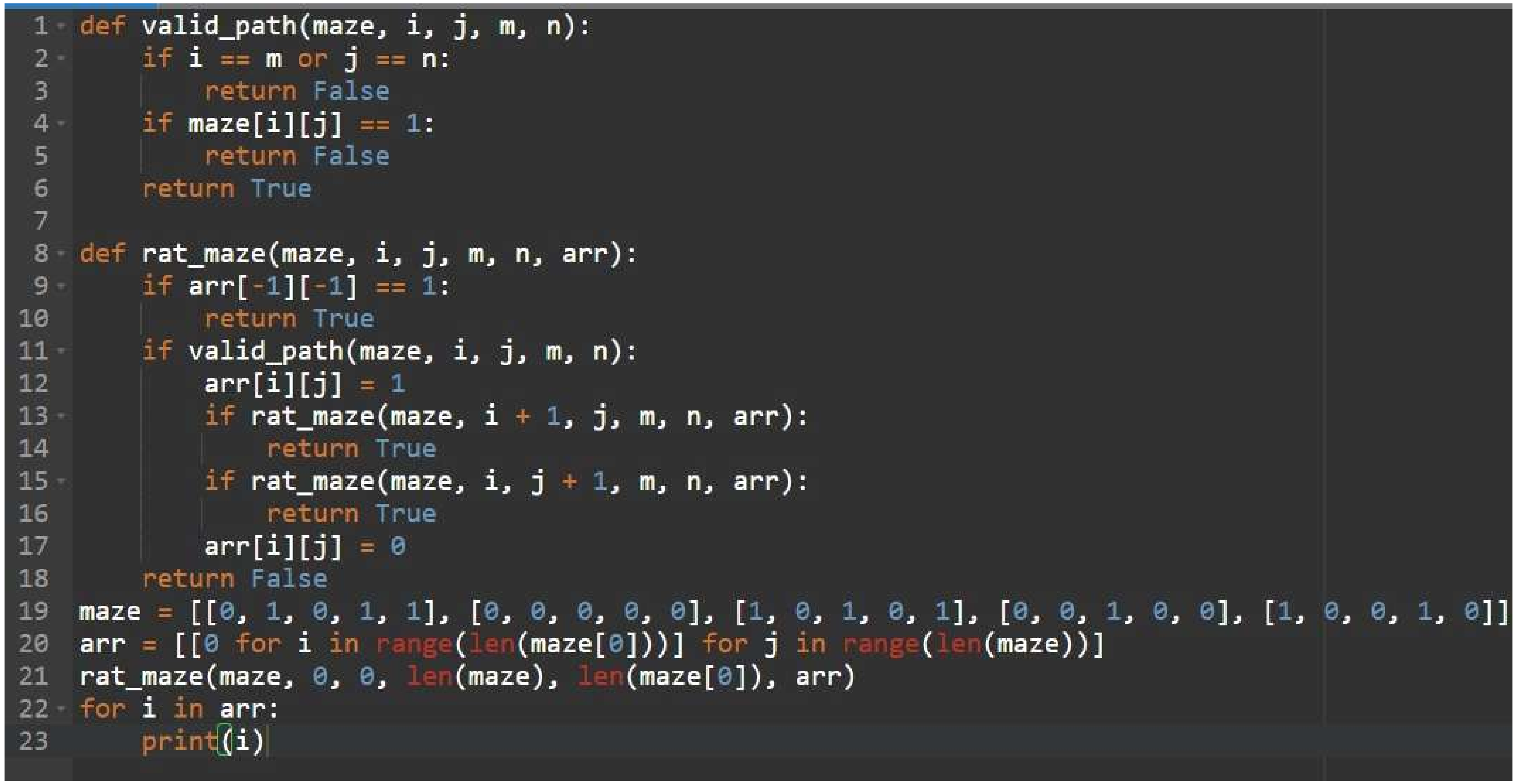
CODE:

```
def vAlid_pAth(mAze, i, j, m, n):         if i == m or
j == n:                  return FAlse         if mAze[i][j]
== 1:                  return FAlse        return True
def rAt_mAze(mAze, i, j, m, n, Arr):         if Arr[–
1][–1] == 1:                  return True         if
vAlid_pAth(mAze, i, j, m, n):                  Arr[i][j]
= 1                  if rAt_mAze(mAze, i + 1, j, m, n,
Arr):
                         return True                  if
rAt_mAze(mAze, i, j + 1, m, n, Arr):
                         return True
Arr[i][j] = 0            return FAlse
mAze = [[0, 1, 0, 1, 1],
[0, 0, 0, 0, 0],
                [1, 0, 1, 0, 1],
                [0, 0, 1, 0, 0],
[1, 0, 0, 1, 0]]
```

Arr = **[[0 for** i **in** rAnge(len(mAze**[0]**))] **for j in** rAnge(len(mAze))]

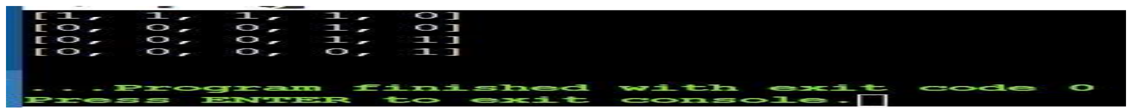
rAt_mAze(mA ZE, 0, 0, LEN(MAZE), LEN(MAZE[0]), ARR) FOR I IN ARR:

PRINT(I)



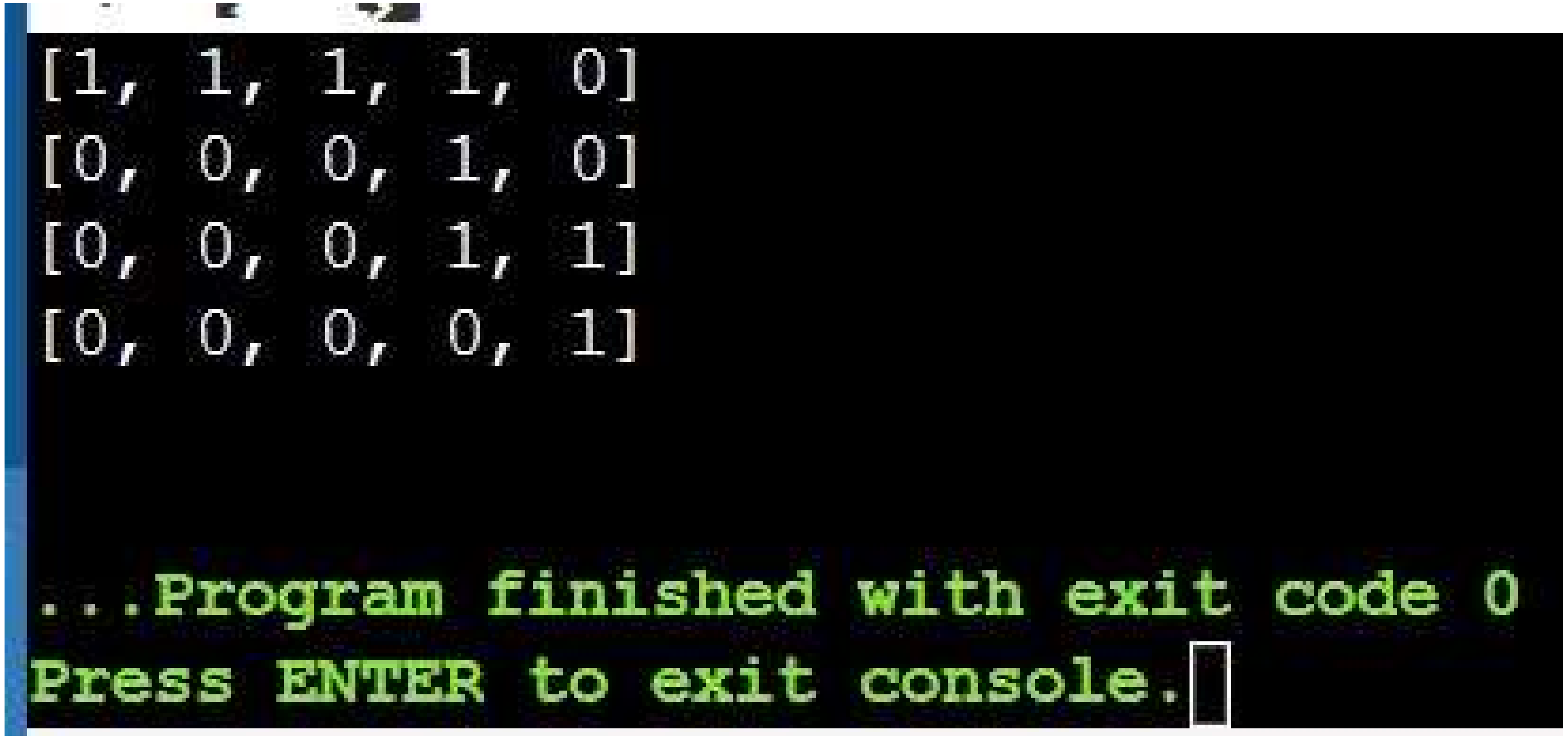
```python
def valid_path(maze, i, j, m, n):
    if i == m or j == n:
        return False
    if maze[i][j] == 1:
        return False
    return True

def rat_maze(maze, i, j, m, n, arr):
    if arr[-1][-1] == 1:
        return True
    if valid_path(maze, i, j, m, n):
        arr[i][j] = 1
        if rat_maze(maze, i + 1, j, m, n, arr):
            return True
        if rat_maze(maze, i, j + 1, m, n, arr):
            return True
        arr[i][j] = 0
    return False
maze = [[0, 1, 0, 1, 1], [0, 0, 0, 0, 0], [1, 0, 1, 0, 1], [0, 0, 1, 0, 0], [1, 0, 0, 1, 0]]
arr = [[0 for i in range(len(maze[0]))] for j in range(len(maze))]
rat_maze(maze, 0, 0, len(maze), len(maze[0]), arr)
for i in arr:
    print(i)
```



```
[1,  1,  1,  1,  0]
[0,  0,  0,  1,  0]
[0,  0,  0,  1,  1]
[0,  0,  0,  0,  1]

...Program finished with exit code 0
Press ENTER to exit console.
```



```
[1, 1, 1, 1, 0]
[0, 0, 0, 1, 0]
[0, 0, 0, 1, 1]
[0, 0, 0, 0, 1]


...Program finished with exit code 0
Press ENTER to exit console.
```

Time-Complexity and Auxiliary Space

Time-Complexity: **O(2^(n^2)). The recursion can run upper-bound 2^(n^2) times.**

Auxiliary Space: **O(n^2). Output matrix is required so an extra space of size n*n is needed.**

THANK YOU