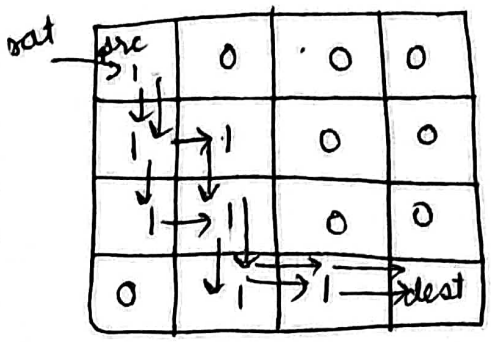
DnC - Level-3

22/08/23

Backtracking → Ques → Rat in a maze →

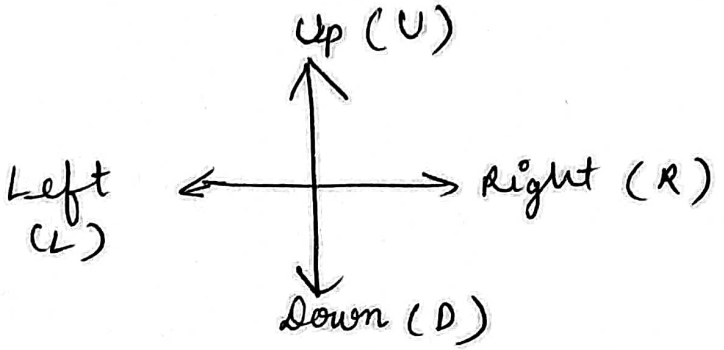


0 → path is blocked

1 → path opened and rat can move in this block.

, find all solution to reach sol^n.

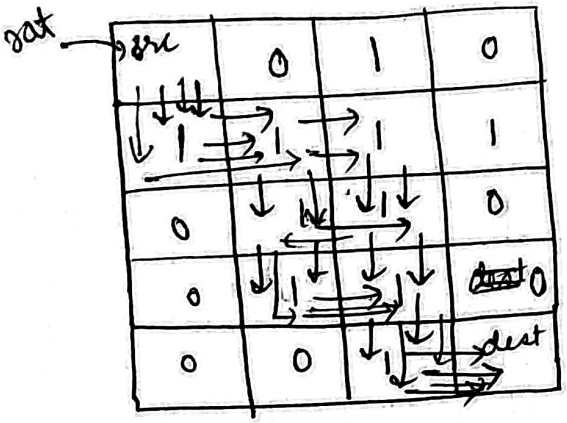
rat can move in four directions.



Possible solutions

→ DDRDRR

→ DRDDRR



ek case → 1 movement in all four directions.

→ DRDDRDR

→ DRRDDDR

→ DRDDRDR

→ DRRDLDRDR

Recursion tree

options/movements

(0,0) ~~D~~/L/R/U

D↓

(1,0) ~~D~~/~~L~~/~~R~~/U

R↓

(1,1) ~~D~~/L/R/U

D↓

(2,1) ~~D~~/L/R/U

D↓

(3,1) ~~D~~/~~L~~/~~R~~/U

R↓

(3,2) ~~D~~/~~L~~/R/U

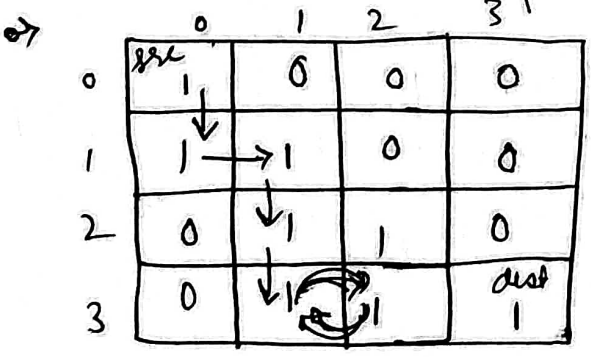
L↓

(3,1) ~~D~~/~~L~~/~~R~~/U

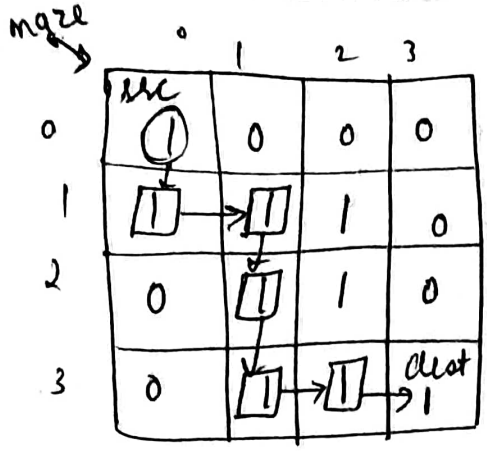
R↓

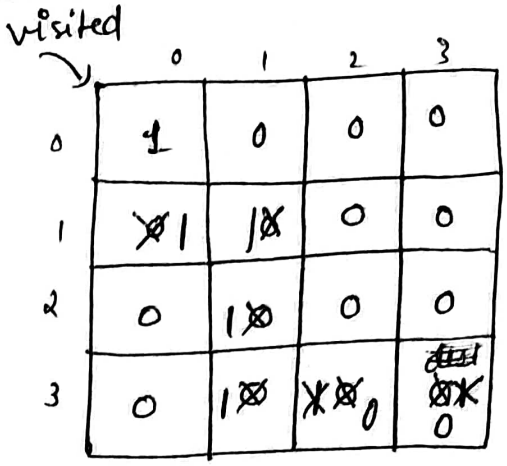
(3,2)

stucked in infinite loop.



So, to not stuck in this loop, we will use a visited array (2-D array) which stores the status of particular 1×1 cell that is visited or not.





(0,0) ✗/L/R/U

D / vis[1][0] = 1 ← 1 means true

(1,0) ✗/✗/✗/U

R / vis[1][1] = 1

(1,1) ✗/L/R/U

D / vis[2][1] = 1

(2,1) ✗/L/R/U

D / vis[3][1] = 1

(3,1) ✗/✗/R/U

vis[3][2]=0 → R / vis[3][2] = 1

(3,2) ✗/✗/✗/U

vis[3][3]=0 → R / vis[3][3] = 1

(3,3)

Destination

↳ Base case

print ans and return.

after returning we have to call for other possible movements.

X ← (Not possible to move up)

now we will go to that 1×1, which is not previously visited.

→ conditions to move to a particular cell →

① index should be inside array.

② Target cell's should have 1.

③ It should be marked ≠ false in visited array.

From here we didn't move to left because in visited array [3][1] is marked as 1 (means visited). That's how we didn't stucked in infinite loop).

This single line is backtracking.

We did this here because we have to find all the paths, so we have to explore all the possible paths to reach destination.

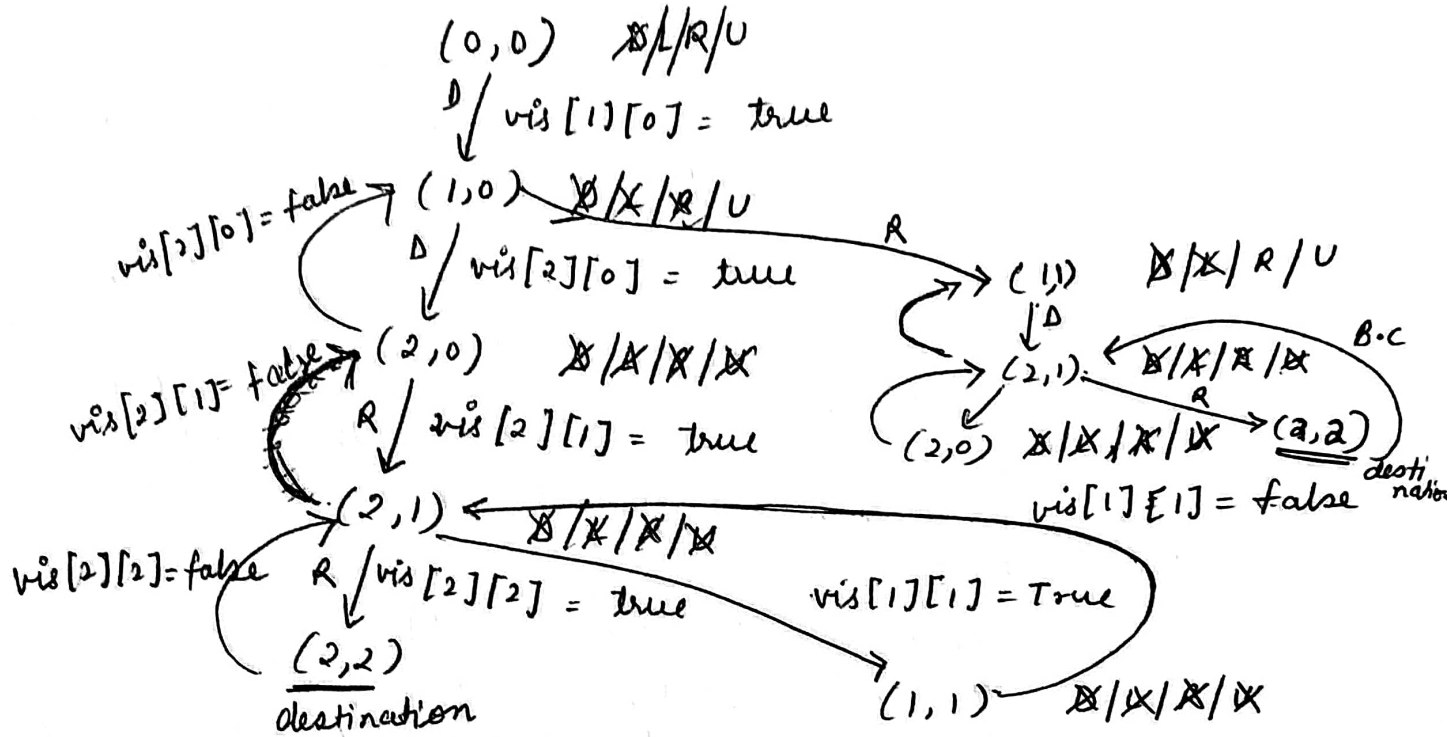
If we have to just print only one path then we don't need to do this.

maze

	0	1	2
0	1	0	0
1	1	1	0
2	1	1	1

visited

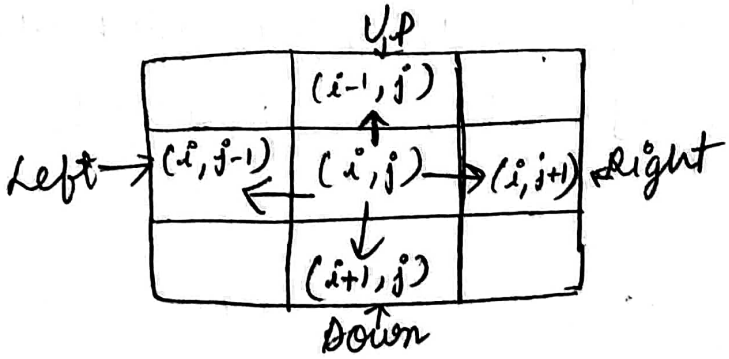
	0	1	2
0	1	0	0
1	1	0	0
2	0	0	0



2 answers ⇒ D D R R

⇒ D R D R

(i, j) → 4 movements (left, right, up, down)



Code →

```
main(){
    int maze[3][3] = { {1,0,0}, {1,1,0}, {1,1,1} };
    int rows = 3, cols = 3;
    vector<vector<bool>> visited(row, vector<bool>(col, false));
    visited[0][0] = true;
```

```
    vector <string> path;
    string output = " ";
    solveMaze(maze , row, col , 0 , 0 , visited , path , output);
    cout << "printing the results " << endl;
    for(auto  i: paths) { cout << i << " "; }
}

void solveMaze ( int arr[3][3], int row , int col,
        int i, int j, vector<vector<bool>>& visited , vector<string>
        & path, string output)
{
    // base case ← when we reached to the destination.
    if( i== row-1 && j== col-1){
        path.push_back(output); ← answer found, store in
                                  path string.
        return;
    } // Ek case solve → har movement me.
    // Down → i+1, j
    if(isSafe(i+1, j, row, col, arr, visited){
        visited[i+1][j] = true;     ← visited mark
        solveMaze(arr, row, col, i+1, j, visited, path, output+'D');
        //backtrack.
        visited[i+1][j] = false;
    }
    // Left → i, j-1
    if(isSafe(i, j-1, row, col, arr , visited){
        visited[i][j-1] = true;
        solveMaze(arr, row, col, i, j-1, visited, path, output+'L');
        visited[i][j-1] = false; ← Backtrack
    }

    // Right → i, j+1
    if(isSafe(i, j+1, row, col, arr, visited){
        visited[i][j+1] = true;
        solveMaze(arr , row, col, i, j+1, visited, path, output+'R');
        visited[i][j+1] = false;
    }
    // Up → i-1, j
    if(isSafe( i-1, j, row, col, arr , visited){
        visited[i-1][j] = true;
```

solveMaze(arr, row, col, i-1, j, visited, path, output + 'U');
visited[i-1][j] = false; ← backtrack

}

}

bool isSafe(int x, int y, int row, int col, int ~~arr~~ maze[3][3],
~~vector<int>~~ vector<vector<bool>> &visited) {

if ((x >= 0 && x < row) ① && (y >= 0 && y < col)) &&
(maze[x][y] == 1) ② && (visited[x][y] == false) ③) {
return true;

{

Here we are checking 3 conditions to move to the cell with index [x][y]
① → The cell's ~~ar~~ index are in bound.
② → The cell has 1.
③ → ~~The~~ In visited array cell's index are marked as 0 (not visited).
If these conditions are true we will return true else false;

return false;

}

Sumbit in gfg.

Edge cases →

1. When ~~src~~ is 0. → no path exist.

if (maze[0][0] == 0) { cout << "No path exist";
return 0;
}

2. When destination is 0
⇒ no path exist.

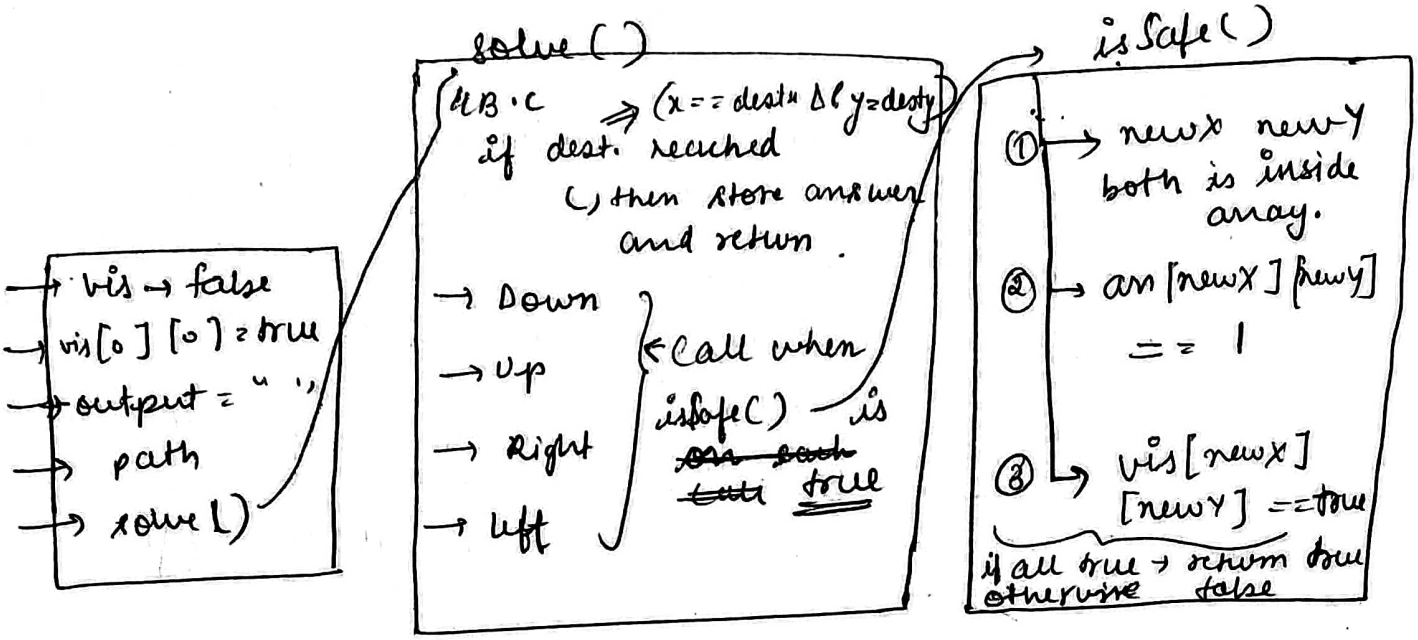
We can further reduce down, left, up, right codes to a single for loop. Because the only change is in i, j's value and adding 'D', 'R', 'L', 'U'.

int dx[] = {1, 0, 0, -1};
int dy[] = {0, -1, 1, 0};
char direction[] = {'D', 'L', 'R', 'U'};

now the loop →

for (int k = 0; k < 4; k++) {
int newX = i + dx[k];

```
int newY = j + dy[k];
char dir = direction[k];
if ( isSafe() ){
        visited[newX][newY] = true;
        solveMaze ( arr, row, col, newX, newY,
            visited, path, output + dir );
        // backtrack
        visited[newX][newY] = false;
}
```



// Base Case →

```
if ( x == destx && y == desty ){   ← destination reached. (ans mil gaya).
    // ↳ store ans
        path.push_back(op);
    // ↳ and return
        return;
}
```

⇒ Visited array →

vector<vector<bool>> visited (row, vector<bool>(col, false));

let row = 4, col = 3 ⇒ visited

```
visited[0][0] = 1;
```

	0	1	2
0	1	0	0
1	0	0	0
2	0	0	0
3	0	0	0

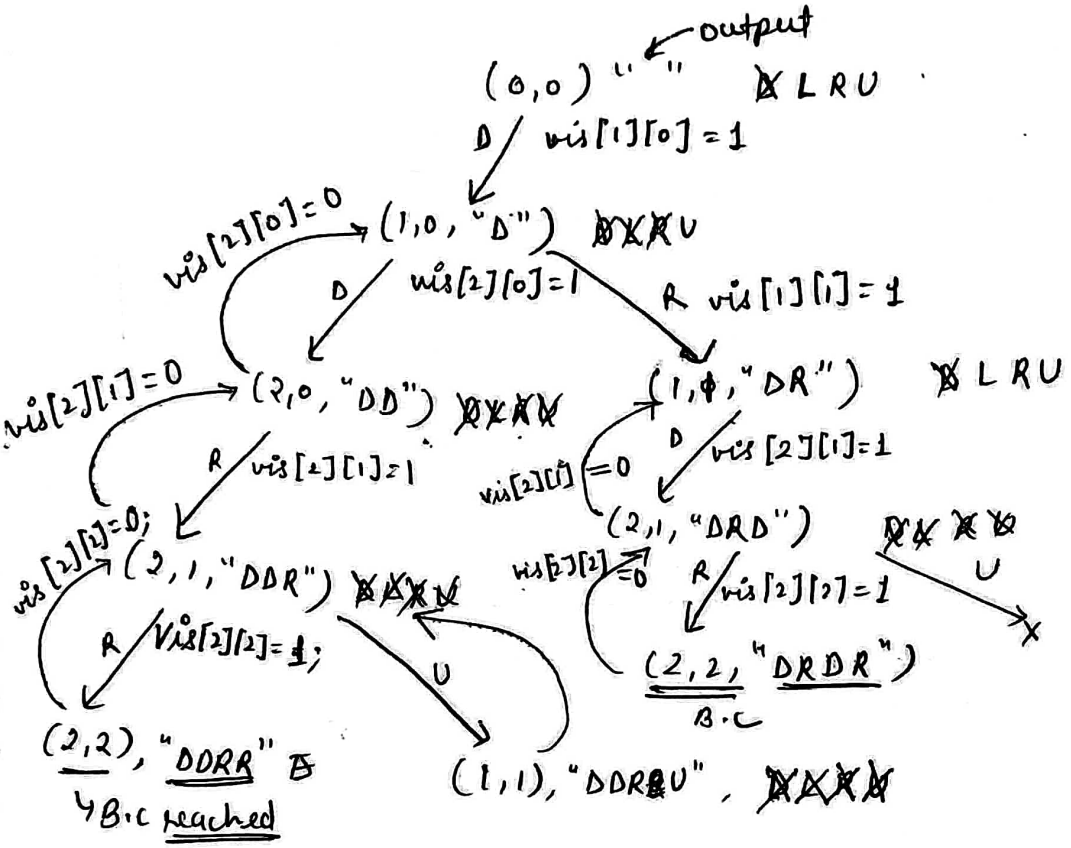
→ dry run →

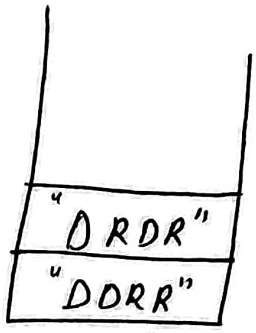
arr

	0	1	2
0	1	0	0
1	1	1	0
2	1	1	1

vis

	0	1	2
0	T	F	F
1	~~F~~ T	~~F~~ ~~F~~ F	F
2	~~F~~ T	~~F~~ T	~~F~~ T





func call →

solve (maze, row, col, srcx, srcy, visited, path, output)

- maze: I/p array
- row, col: rows and cols in array
- srcx, srcy: starting pos. of rat
- visited: 2D array to mark cells visited.
- path: to store ans
- output: to create ans.

T.C of this solution →

$$T.C = O(4^n)$$

max 4 calls (down, up, right, left) for each cell.