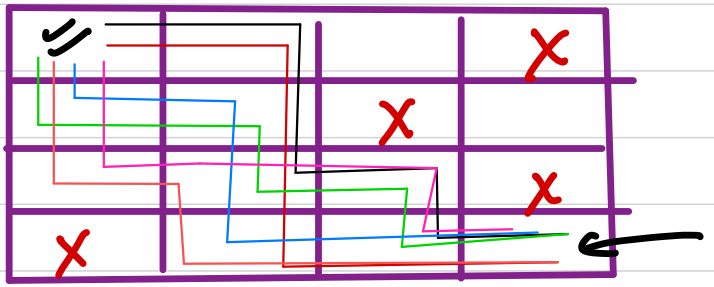
Q: Given a grid with few blocked cells and few open cells. We are standing on the top left of the grid and want to reach the bottom right. In one move from any cell, we can go to the cell on up, down, left or right direction. Count the no. of ways to reach bottom right.



6 ways

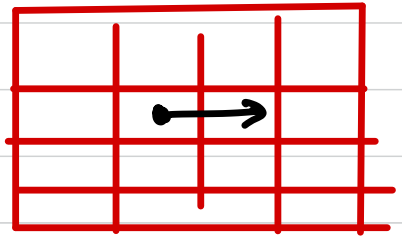
Ans → 6

R D D R D R
R D D D R R
D R D D R R
D R D R D R
D D R D R R
D D R R D R

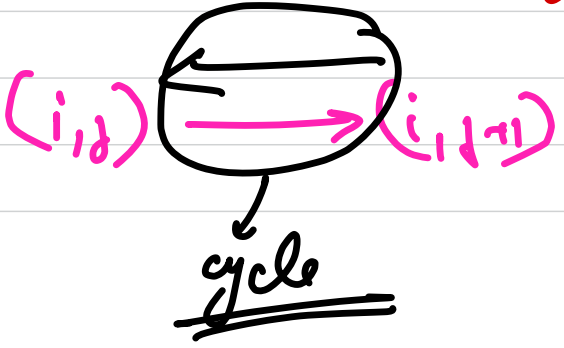
6 ways

⇒ Observations →

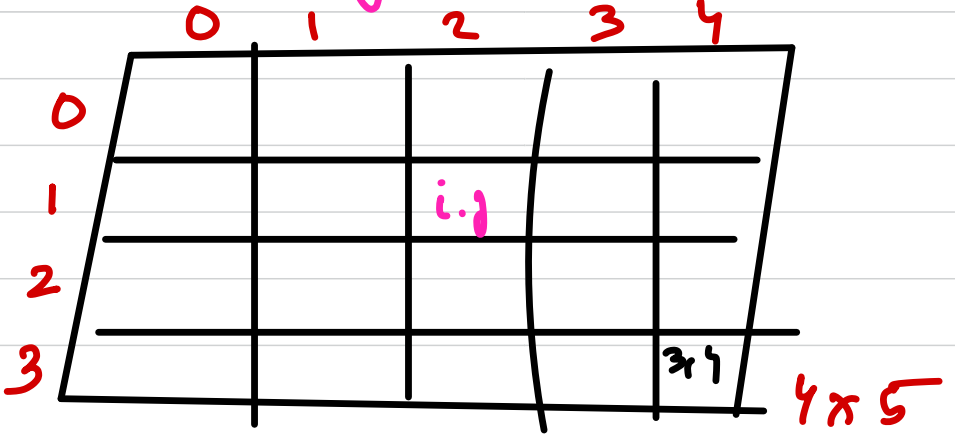
1) From a cell (i,j), assume we go right $(i,j+1)$



Now from $(i,j+1)$ we cannot go left. otherwise a cycle will be generated. So, we will not go back to the parent cell. How to achieve this ??



We can say that, if we keep a grid where every cell that we touch in a path is marked visited, then for that path we will never go back again on a visited cell.



$f(i,j,m,n)$ = $\begin{cases} f(i,j+1,m,n) & \rightarrow \text{right} \\ + & \\ f(i+1,j,m,n) & \rightarrow \text{down} \\ + & \\ f(i,j-1,m,n) & \rightarrow \text{left} \\ + & \\ f(i-1,j,m,n) & \rightarrow \text{up} \end{cases}$

$f(i,j,m,n)$ ↓

gives us no. of ways to reach m,n from i,j with up, down, left right all moves available.

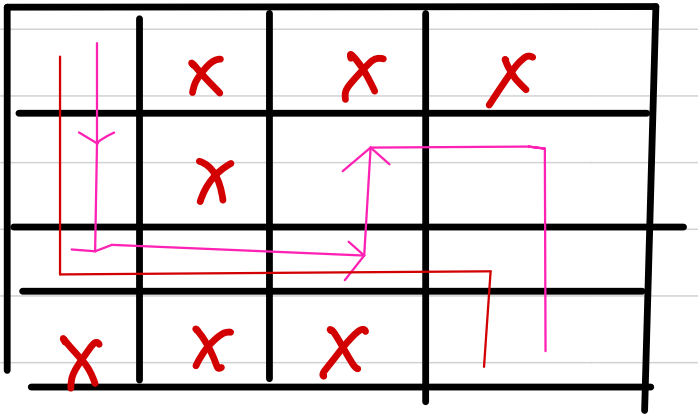
Base Case → `if (i == m-1 && j == n-1) return 1;`

↳ if while traversing we reached the destination we got one way

```
if (i<0 or j<0 or i≥m or j≥n or
    visited[i][j] == true or grid[i][j]==0){
      return 0;
}
```

If any of the above conditions hold true, then we will be outside of the grid. or on a unwanted cell, so no. of ways to reach dest will be 0;

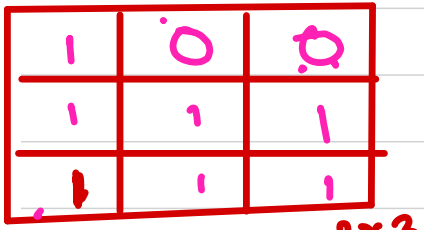
grid[i][j] → 0 → blocked
→ 1 → open



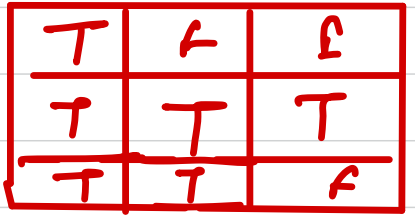
```javascript
function f(i, j, m, n) {
    if(i == m-1 && j == n-1) {
        // while travelling we reached the destination
        return 1;
    }
    if(i < 0 || j < 0 || i ≥ m || j ≥ m || grid[i][j] == 0 || visited[i][j] == true) {
        /**
         * All of the above cases are unwanted, either they make us go
         * out of grid, or make us go to a blocked cell or make us go to
         * an already visited, if we reach in any one of the situation
         * we return 0, as there will no path from a position like this
         */
        return 0;
    }
    // whenever we will visit a cell we will mark it as visited
    visited[i][j] = true;

    let ans = 0;
    ans += f(i, j+1, m, n);
    ans += f(i+1, j, m, n);
    ans += f(i, j-1, m, n);
    ans += f(i-1, j, m, n);
    return ans;
}
```

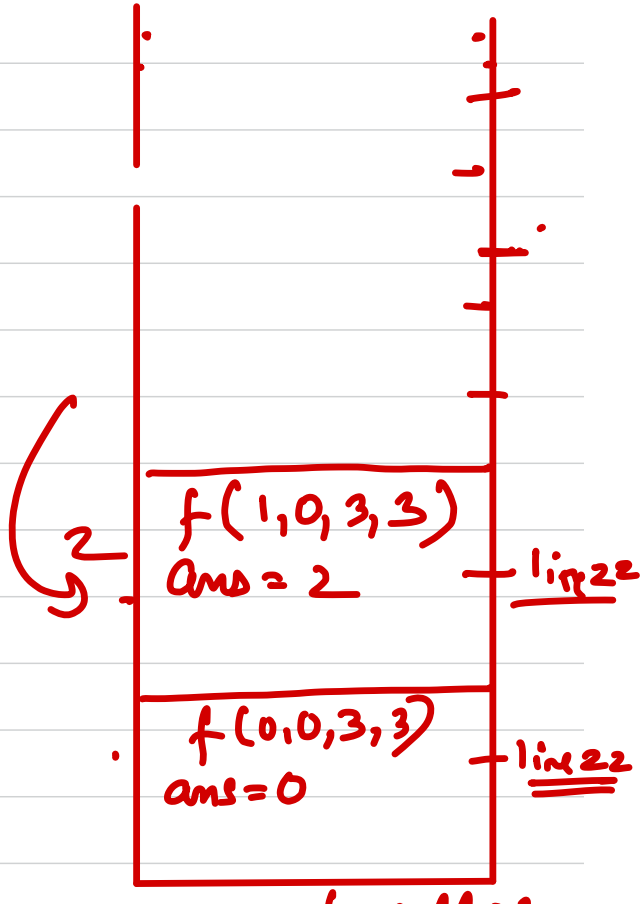
f(0,0,m,n)



input



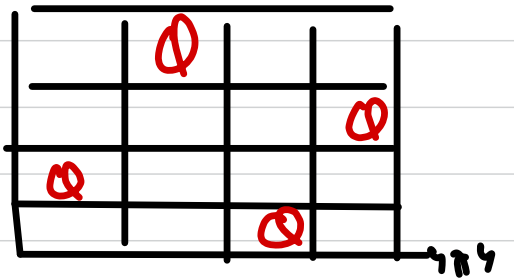
visited



Backtracking

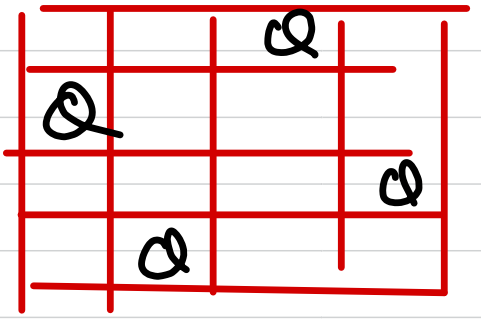
Backtracking is a way of solving problems in which we travel towards our ans & make changes in state. Irrespective of the fact that whether we got a valid ans or not, we will revert the changes done in the state.

Q. You're given a $n \times n$ grid, and you've n queens of chess. Count the no. of ways to arrange these queens so that none of them attacks the other.



4×4

1 ways

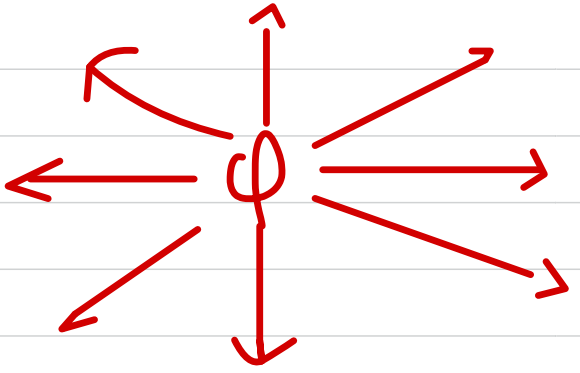


4×4

1 ways

n=4

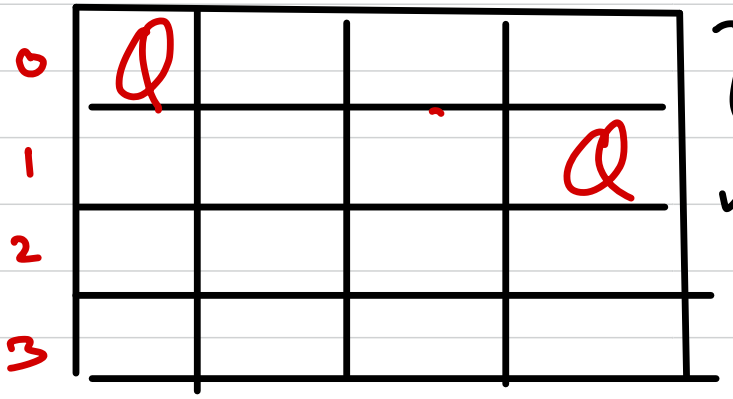
ans - 2



observations

1) if we have n queens that we have to place, on a n×n board, then on each row/col we can place at most 1 queen. that means on every row we are bound to place a queen.

f(0, n)



→ if few of the rows are already filled then we can fill a new row

We will go to the next row, if and only if we have placed the queen on the current row.

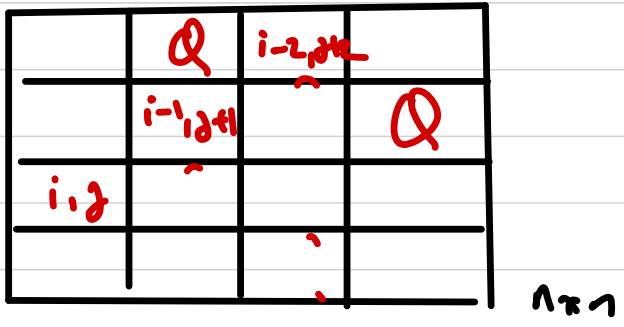
$f(r, n)$ = place queen on the r^{th} row

↳ $f(r+1, n)$

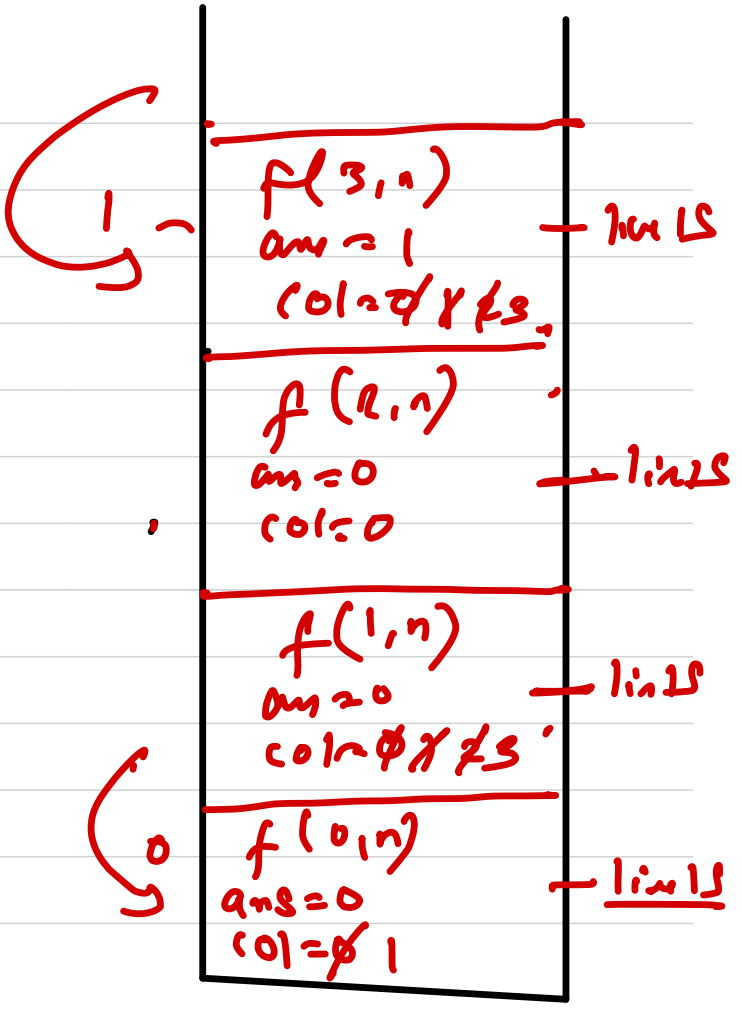
this func^ places the queen from the r^{th} row safely

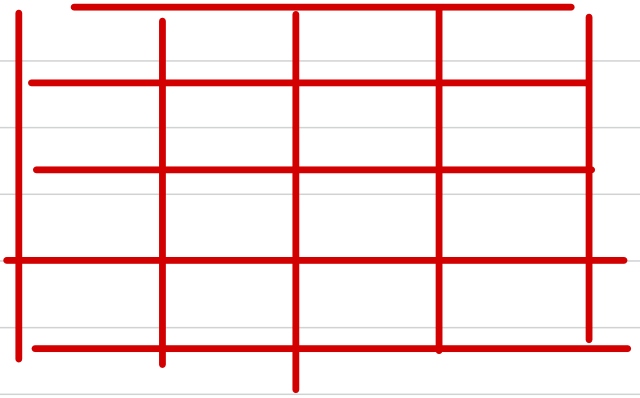
```
function f(r, n) {
    if(r == n) {
        return 1;
    }
    let ans = 0;
    for(let col = 0; col < n; col++) {
        if(canWePlaceQueen(r, col, n)) {
            grid[r][col] = 'Q';
            ans += f(r+1, n);
            grid[r][col] = '';
        }
    }
    return ans;
}
```

f(0,n)



n×n





$n \times n$

(1)

$n = 4$

$n^2 C_n$ combinations

$\hookrightarrow {}^{16}C_4$ combinations

$$\frac{16!}{4!(12!)} = \frac{16 \times 15 \times 14 \times 13}{4 \times 3 \times 2} = 1820$$

Backtracking solution are a bit more efficient than complete bruteforce.

Backtracking solutions prune some unwanted calls & then do recursion.