

Backtracking



Backtracking is a recursive algorithm technique for solving problems by exploring all potential solution sequentially and backtracking when it determines that the current path can not lead to a valid solution.

Application like:- N - Queens problem
Sudoku
Subset Sum.

Characteristics :

Depth-first-Search: Backtracking uses a depth first search strategy to explore potential solution.

State maintenance: The algorithm maintains the current state of the solution path and modifies it as it explores different choices.

Completeness and optimality: Backtracking ensures completeness (finding all solution, if required) and can sometimes be optimized for finding optimal solution (using pruning techniques).



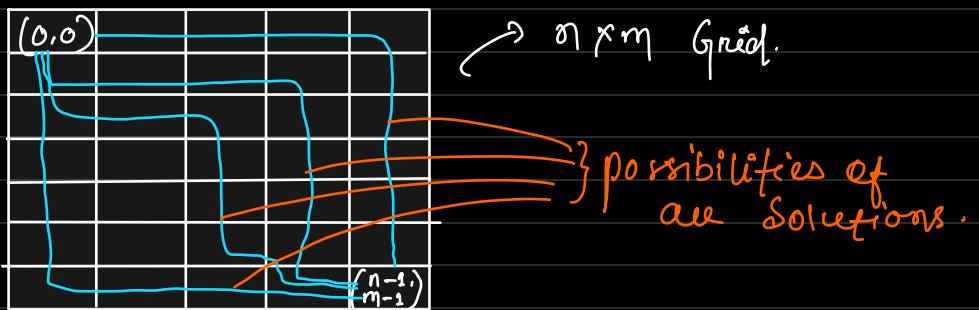
Types Of Backtracking

1 - Decision.

2 - Optimization

3 - Enumeration.

* Examples of Decision backtracking



* Examples of Optimization backtracking

There may be multiple ways to find out the destination. But we have to find the most optimized way to reach.

for example, there is multiple ways to reach from (0,0) to (n-1, m-1)

But we have to find out the shortest path. So here Optimization comes handy.

* Examples of Enumeration backtracking

Means total number of possibilities to reach the destination / to find out the solution.

So maybe we just have to print all the possible ways, or to count no. of possible solution.

Backtracking - arrays

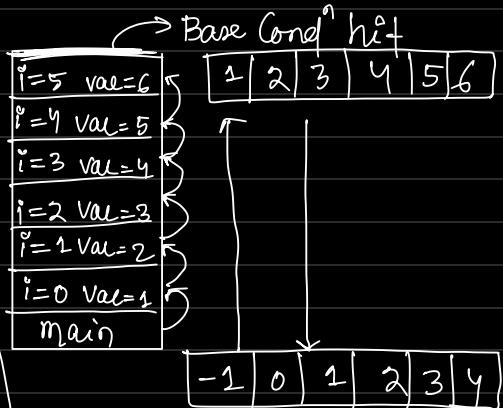
1	2	3	4	5	6
---	---	---	---	---	---

main $i=0$ $i=1$ $i=2$ $i=3$ $i=4$ $i=5$

Goal is to assign the index value +1 to the respective position in array using recursion.

But after base condition hit then we want an array that should contain this

-1	0	1	2	3	4
----	---	---	---	---	---



```

import java.util.Arrays;
public class BacktrackingArrays { new *
    public static void main(String[] args) { new *
        int[] array = new int[5];
        backtracking(array, 0);
        System.out.println(Arrays.toString(array));
    }
    public static void backtracking(int[] array, int i){ usages new *
        if (i == array.length){ usages new *
            System.out.println(Arrays.toString(array));
            return;
        }
        array[i] = i+1;
        backtracking(array, i+1);
        array[i] = array[i] - 2;
    }
}

```

value assigning to array
recursion.
After base Condition hit
Can Stack decrease to
main, then Array Element
decrease value by 2

$$\begin{aligned}
 \text{time Complexity} &= O(n) + O(n) \\
 &= O(2n) \\
 &= O(n)
 \end{aligned}$$

Find subsets

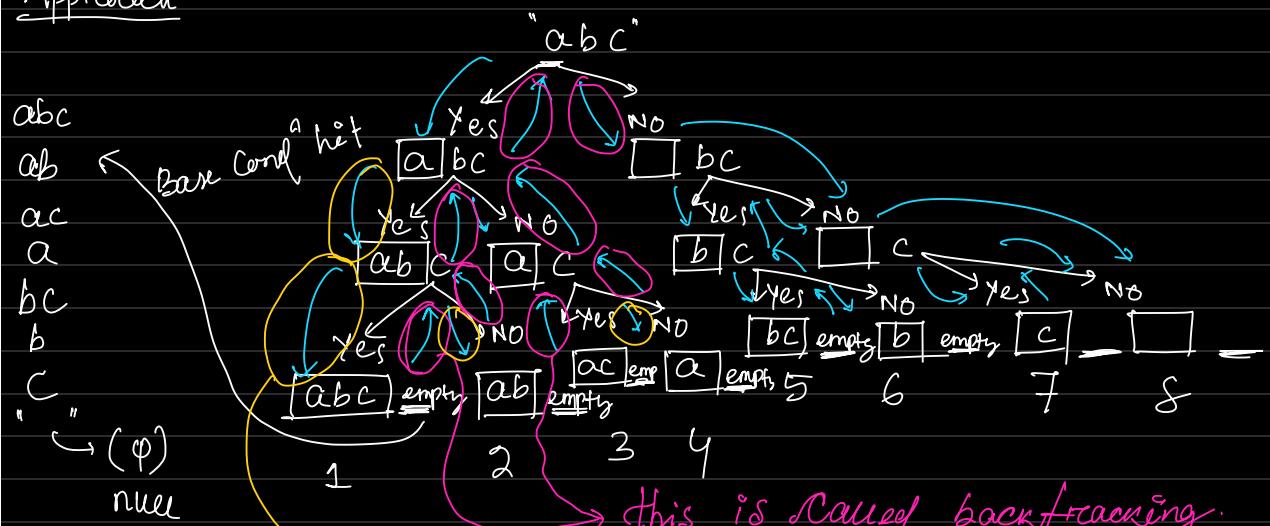
(find and print all Subsets of a given string)

"abc" → a, b, c, ab, bc, ac, abc. " " empty set / null set
total 8 Subsets.

* If String length = n
then total no of subset = 2^n

} Also Applicable
in Arrays.

Approach



This is caused recursion.

```

public class BacktrackingArrays {
    public static void findSubsets(String str, String answer, int i) {
        if (i == str.length()) {
            if (answer.isEmpty())
                System.out.println("null");
            else
                System.out.println(answer);
            return;
        }
        choice yes
        findSubsets(str, answer + str.charAt(i), i+1);
        choice no
        findSubsets(str, answer, i+1);
    }
}

```

At one recursion
Stack grows from
 n levels

Space Complexity = $O(n)$
time Complexity = $O(n * 2^n)$

(\because to find out the subset of a string the
time complexity = n
to find out 2^n subset = $(n * 2^n)$)

$a b c = n = 3$
 $2^3 = 8$ Subsets.

{ "a b c d e f" → String }

a have 2 choices
b have 2 choices
c have 2 choices
d have 2 choices
e have 2 choices
f have 2 choices
 $2 * 2 * 2 * \dots * n$ times = 2^n subsets. }

Find permutations

(Find and print all permutations of a string)

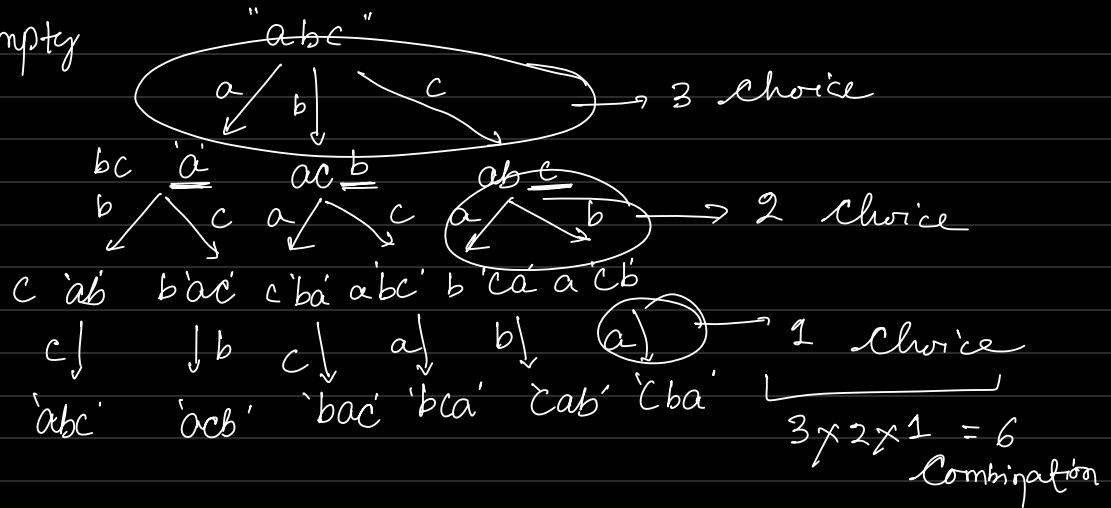
* If there is a string of length - 'n'
then no of Combination would be = $[n!]$

(Also applicable in case of Array)

for example = for string 'abc' → length = 3

$$abc, acb, bac, bca, cab, cba \} 3! = 3 \times 2 \times 1 = 6 \text{ Combination}$$

let's fix an empty
String



```

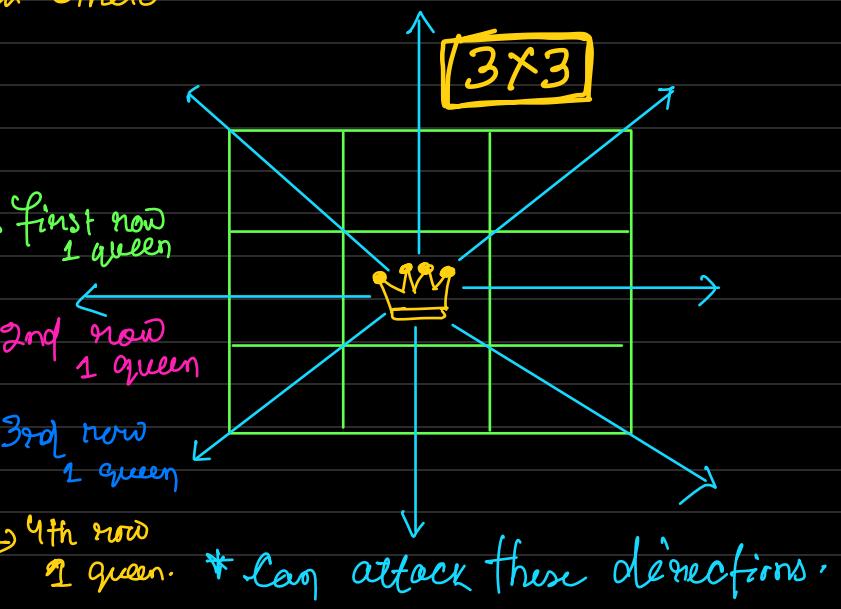
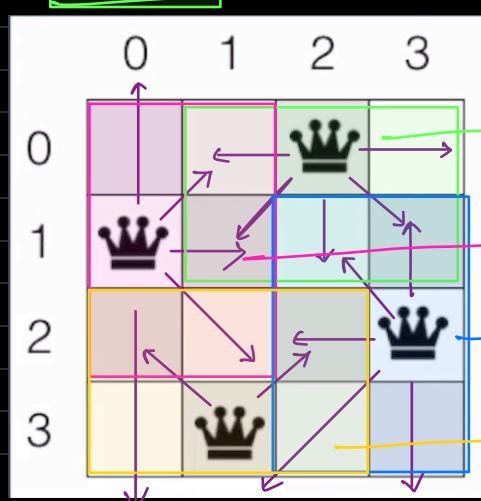
public class BacktrackingArrays {
    public static void findSubsets(String str, String answer, int i) {
        if (str.isEmpty()) {
            System.out.println(answer);
            return;
        }
        for (int j = 0; j < str.length(); j++) {
            char cur = str.charAt(j);
            String newStr = str.substring(0, j) + str.substring(j+1);
            permutations(newStr, answer + cur);
        }
    }
}

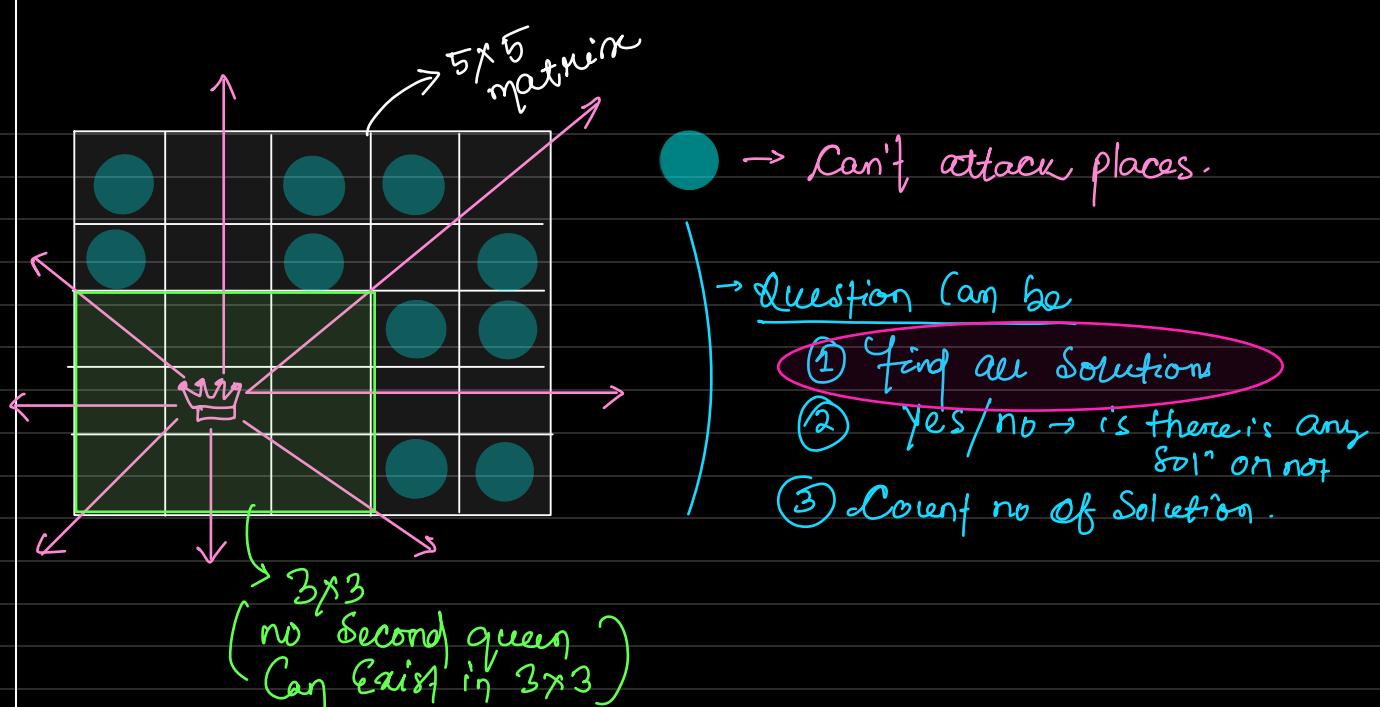
```

N - QUEENS

Place N Queens on an $N \times N$ Chessboard such that no two queens can attack each other

$$N = 4$$





logic : Vertical - $\text{row}++ / \text{row--}$, Column Constant

horizontal - Row Constant, Column++, Column--

diagonal - $\text{row}--$, Column--, $\text{row}--$, Column++;

$\text{row}++$, Column--;

$\text{row}++$, Column++;

We don't have to check this condition (top-down approach)

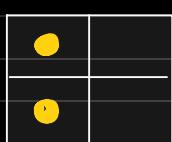
(* No two queens can place at one row)
(Only one queen at one row)

(do if there is queen of length = 2,
then we need 2 row)

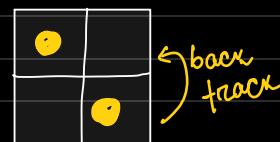
(∴ No of rows = no of queens)

(let's forget that 2 queens attack each other for some times)

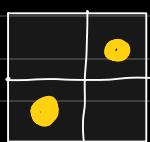
if $n = 2$ possible solution



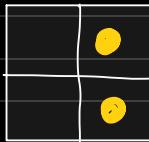
(1st Case)



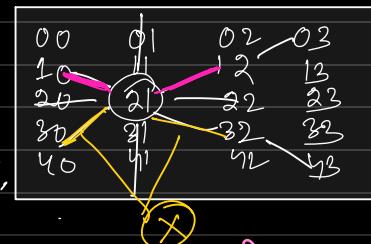
(2nd Case)



(back track
(3rd Case))



(back track
(4th Case))



```
cd ApnaCollegeFolder
cd src
cd Nqueens
javac Nqueens.java
java Nqueens
```

AC ApnaCollegeFolder

Project

- ApnaCollegeFolder ~/IdeaF
 - .idea
 - out
 - src
 - BackTracking
 - FunctionAndMethods
 - Main.java
 - Nqueens
 - Pattern_1
 - Patterns_2
 - .gitignore
 - ApnaCollegeFolder.iml
- External Libraries
- Scratches and Consoles

```

1 public class Nqueens {
2     public static void initialization(int n){
3         char[][] matrix = new char[n][n];
4         for (int i = 0; i < matrix.length; i++){
5             for (int j = 0; j < matrix.length; j++){
6                 matrix[i][j] = '.';
7             }
8         }
9         nQueens(matrix, row: 0);
10    }
11
12    public static void nQueens(char[][] matrix, int row){
13        if (row == matrix.length){
14            printChess(matrix);
15            return;
16        }
17        for (int i = 0; i < matrix.length; i++){
18            matrix[row][i] = 'Q';
19            nQueens(matrix, row: row+1);
20            matrix[row][i] = '.';
21        }
22    }
23
24    public static void printChess(char[][] matrix){
25        System.out.println("---- N queens ----");
26        for (int i = 0; i < matrix.length; i++){
27            for (int j = 0; j < matrix.length; j++){
28                System.out.print(matrix[i][j]);
29            }
30            System.out.println();
31        }
32    }
33}
34

```

2-D array initialization with char - '.'

place 'Q' at respective position then call for next row

When backtracking we again assign '.' char to matrix

(See the Output)

(* No check that, is it safe to place a queen at a position)
 (We have to check the condition)

isSafe (matrix, row, j)

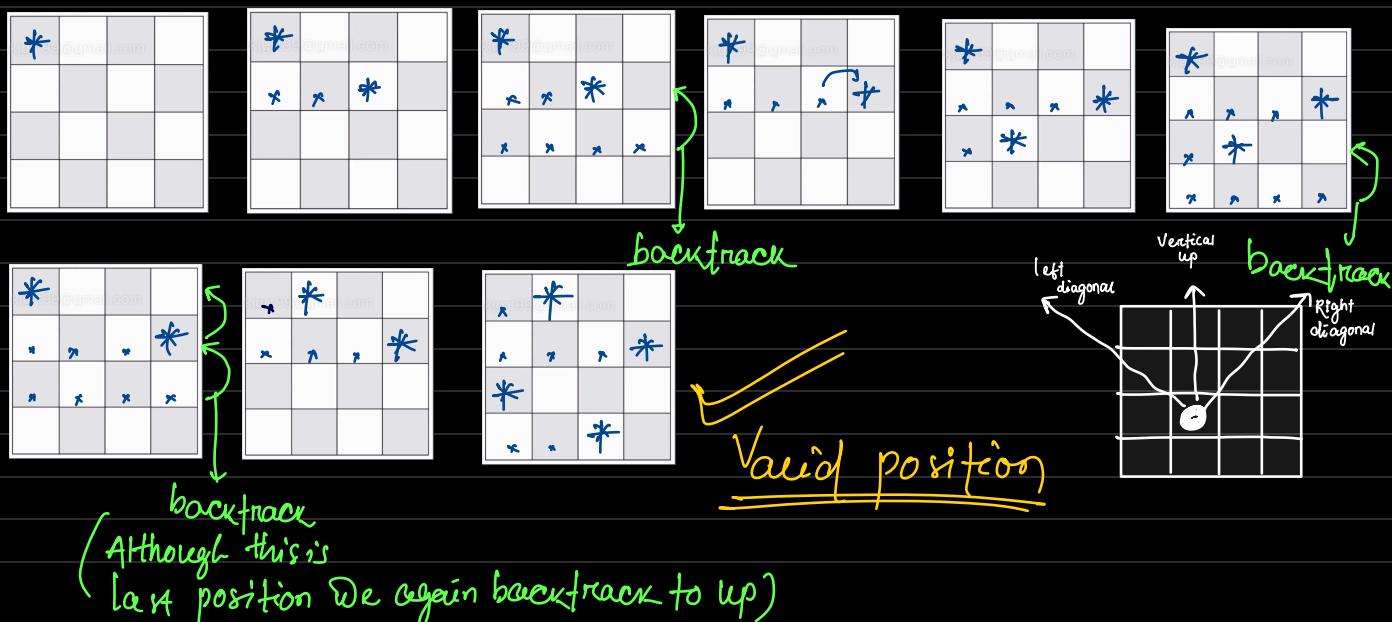
↳ If returns true, then we can place

Otherwise, we shouldn't place the Queen

using recursion

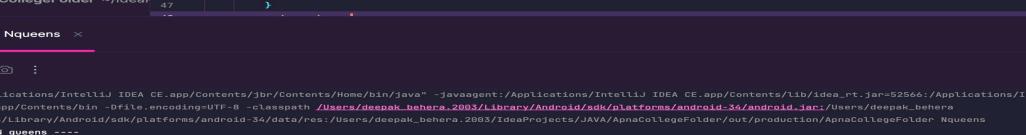
Process finished with exit code 0

Step-1



```
public class Nqueens {
    public static void nQueens(char[][] matrix, int row) {
        if (row == matrix.length) {
            for (int i = 0; i < matrix.length; i++) {
                System.out.print(matrix[i] + " ");
            }
            System.out.println();
        } else {
            for (int j = 0; j < matrix.length; j++) {
                if (isSafe(matrix, row, j)) {
                    matrix[row][j] = 'Q';
                    nQueens(matrix, row + 1);
                    matrix[row][j] = '0';
                }
            }
        }
    }

    public static boolean isSafe(char[][] matrix, int row, int column) {
        // vertical up (column constant)
        for (int i = row - 1; i >= 0; i--) {
            if (matrix[i][column] == 'Q') {
                return false;
            }
        }
        // left diagonal (row--, column--)
        for (int i = row - 1, j = column - 1; i >= 0 && j >= 0; i--, j--) {
            if (matrix[i][j] == 'Q') {
                return false;
            }
        }
        // right diagonal
        for (int i = row - 1, j = column + 1; i >= 0 && j < matrix.length; i--, j++) {
            if (matrix[i][j] == 'Q') {
                return false;
            }
        }
        return true;
    }
}
```



The screenshot shows the IntelliJ IDEA interface with the project 'ApnaCollegeFolder' open. The 'Run' configuration 'Nqueens' is selected. The terminal output shows the following for a 4x4 board:

```
//Applications/IntelliJ IDEA CE.app/Contents/jbr/Contents/Home/bin/java" -javaagent://Applications/IntelliJ IDEA CE.app/Contents/lib/idea_rt.jar=52566:/Applications/intellij IDEA CE.app/Contents/bin -Dfile.encoding=UTF-8 -classpath /Users/deepak_bhera.2003/Library/Android/sdk/platforms/android-34/data/res:/Users/deepak_bhera.2003/IdeaProjects/JAVA/ApnaCollegeFolder/out/production/ApnaCollegeFolder Nqueens
=====
----- N queens -----
x 0 x x
x x x 0
0 x x x
x x 0 x
-----
----- N queens -----
x x Q x
Q x x x
x x x Q
x Q x x
Process finished with exit code 0
```

A large handwritten note in yellow ink is overlaid on the terminal output, reading: } total possible solution for '4' queen = (2)

N. Queens (time complexity)

if there is 'n' number of queens. So

for 1st queen 2nd queen 3rd queen - - - , nth queen
 ↓ ↓ ↓ ↓
 n choices $n-1$ choice $n-2$ choice 1 choices

$$= n \times (n-1) \times (n-2) \times \dots \underline{1}$$

$$= n! = \Theta(n!)$$

Recurrence Equation ÷

$T(n) = \underbrace{1}_{O(n)} \text{ Queen place} * T(n-1) + \text{isSafe()}$

$T(n) = n * T(n-1) + \text{isSafe}()$

2nd Question : Count ways -(N-Queens)

Count total number of ways in which we can solve 'N' Queens problem.

So, Base Case

if (row == board.length)

instead of displaying/printing the chess board
here we should initialize a static variable
like a class Variable. So After Every base Condition
hit Count should increase by 1.

So Code would be.

Static int Count = 0;

// Base Condition

if (row == matrix.length) {

} Count++;
return;

3rd Question : Print 1 Solution

Check if problem can be solved and print only 1 solution to N Queens problems.

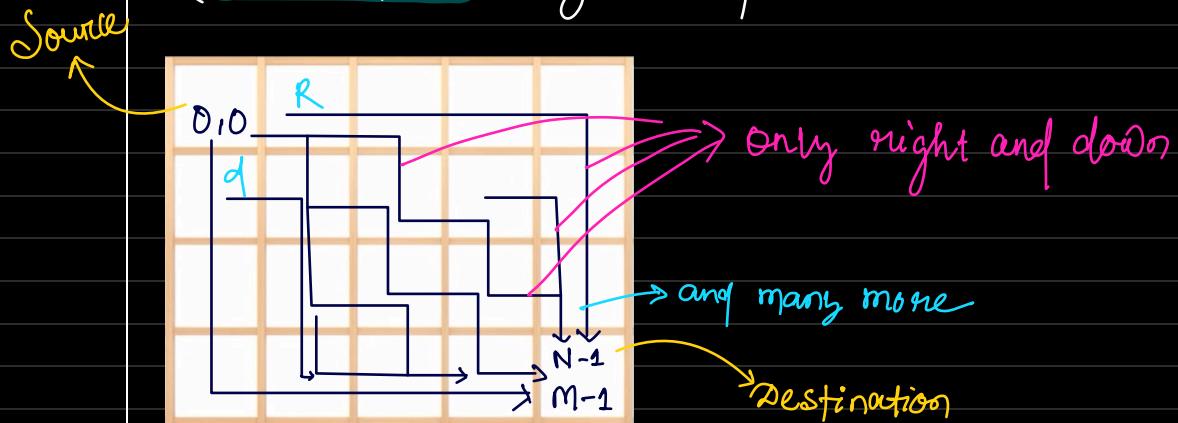
```
public static boolean nQueens(char[][] matrix, int row) { 2 usages
    if (row == matrix.length) {
        printChess(matrix);
        return true;
    }

    for (int j = 0; j < matrix.length; j++) {
        if (isSafe(matrix, row, j)) {
            matrix[row][j] = 'Q';
            if (nQueens(matrix, row + 1)) {
                return true;  { Solution exist for
            }                      next row
            matrix[row][j] = 'x';
        }
    }
    return false; } → no solution for n queens
}
```

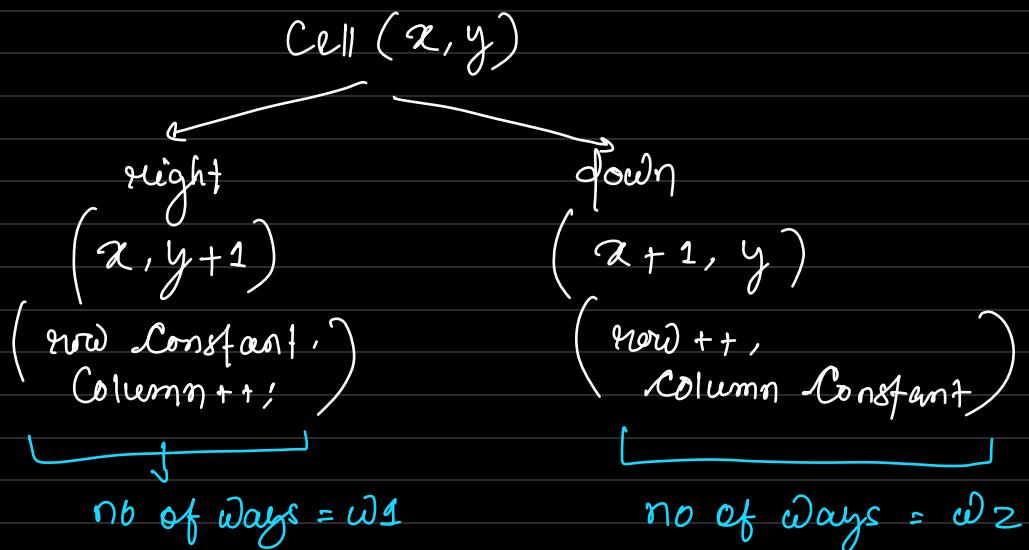
Grid ways

→ Find no. of ways to reach from $(0,0)$ to $(N-1, M-1)$ in a $N \times M$ Grid.

(Allowed moves - right or down)



Approach



$$f(x, y) = f(x, y+1) + f(x+1, y)$$

↳ Recursion Equation

Base Case

When $(0, 0)$ becomes $n-1$, and $m-1$ respectively
we have to return 1:

also there is another base condition, i.e

if we cross the right boundary or

Cross the bottom boundary the return 0;

AC ApnaCollegeFolder main

Project

- ApnaCollegeFolder ~/idea
- out
- src
 - BackTracking
 - FunctionAndMethods
 - Main.java
 - Nqueens
 - Pattern_1
 - Patterns_2
 - .gitignore
 - ApnaCollegeFolder.iml
- External Libraries
- Scratches and Consoles

```

public class Nqueens {
    // ...
    public static int gridWays(int ci, int cj, int row, int column) {
        if (row-1 == ci && column-1 == cj) { → first base condition
            return 1;
        } else if (ci == row || cj == column) { → second base condition
            return 0;
        }
        kaam recursion
        return gridWays(ci+1, cj, row, column) + gridWays(ci, cj+1, row, column); ← bottom ways + right ways
    }
}

```

time Complexity = $O(2^{n+m})$
(huge time Complexity)

Run Nqueens

9:56 LF UTF-8 AC ApnaCollegeFolder SynthWave '84 (Material) 4 spaces

(Math trick for linear time complexity)

$$\text{way/path} - \frac{(n-1)D}{(m-1)R} \text{ total characters} = ((n-1) + (m-1))$$

if we found the permutation of these strings

then we could find the no of ways

DD
RR
↓
DPRR
DRDR
RRDD } there is
RDRD } also
DRRD } duplication
RDDR }

so Repeating $\Rightarrow (n-1)D$

$(m-1)R$

$$\Rightarrow \frac{\{(n-1) + (m-1)\}!}{(n-1)! * (m-1)!}$$

→ total number of ways

And the time Complexity you reduce to $O(n+m)$

$$= O(n)$$

? $n=3, m=3$

$$= \frac{\{(3-1) + (3-1)\}!}{(3-1)! * (3-1)!} = \frac{4!}{2! * 2!} = \frac{24}{4} = 6 \text{ ways.}$$

Sudoku solver

2	1	8	3	9	6	4	7	5
4	9	6	1	5	7	8	3	2
	3		4	1	9			
1	8	5	6			2		
		2			6			
9	6	4	5	3				
	3		7	2		4		
4	9		3		5	7		
8	2	7		9	1	3		

→ 3x3

→ 9x9

A Sudoku Solver is a backtracking algorithm that systematically attempts to fill in the empty cells of a Sudoku grid by exploring potential solution and backtracking when a conflict is found until the grid is fully and correctly filled.

Steps

1 - Identify an empty cell

- The Algorithm scans the grid to find an empty cell to fill.

2 - Try possible numbers

- For the identified empty cell, the Algorithm tries each number from 1 to 9.

3 - Check Constraints

- For each number, the Algorithm checks if placing number in the cell violates the Sudoku constraints (i.e. the number must not appear in the same row, column or 3x3 grid)

4 - Recursive call

- If a valid number is found, the Algorithm places the number in the cell and recursively attempts to solve the rest of the grid.

5 - Backtracking

- If placing a number leads to a dead end (no valid numbers can be placed in future steps) the Algorithm removes (backtracks) the number from the cell and tries the next possible number.

6 - Repeat until solved.

- The process repeats until the entire grid is filled correctly or all possibilities are exhausted (indicating no solution exist).

$$\text{Starting row} = (\text{row}/3) * 3 \quad \left. \right\} \text{for } 3 \times 3 \text{ grid}$$

$$\text{Starting Column} = (\text{Column}/3) * 3 \quad \left. \right\} \text{Constraints}$$

```

public static boolean sudokuSolver(int[][] grid, int row, int column){ 3 usages
    if (row == 9 && column == 0){
        return true;
    } else if (row == 9) {
        return false;
    }

    kaam
    int newRow = row, newColumn = column + 1;
    if (column+1 == 9){
        newRow = row+1;
        newColumn = 0;
    }
    if (grid[row][column] != 0){
        return sudokuSolver(grid, newRow, newColumn);
    }

    for (int digit = 1; digit <= 9; digit++){
        if (isSafe(grid, row, column, digit)){
            grid[row][column] = digit;
            if(sudokuSolver(grid, newRow, newColumn)) {
                return true;
            }grid[row][column] = 0;
        }
    }
    return false;
}

```

Base Condition

If Column Cross 8 then We have to change to next grid

If there is already some number exist at the place

Means the Cell is valid and there is no other number exist before

↳ backtracking if the grid is valid then this statement never execute

```

public static boolean isSafe(int[][] grid, int row, int column, int digit){ 1 usage
    checking column
    for (int i = 0; i <= 8; i++){
        if (grid[i][column] == digit){
            return false;
        }
    }

    checking row
    for (int j = 0; j <= 8; j++){
        if (grid[row][j] == digit){
            return false;
        }
    }

    for grid check
    int sr = (row / 3) * 3;
    int sc = (column / 3) * 3;
    for (int i = sr; i < sr+3; i++){
        for (int j = sc; j < sc+3; j++){
            if (grid[i][j] == digit){
                return false;
            }
        }
    }
    return true;
}

```

Check column horizontally

Check row vertically

Check 3x3 grid of the current cell

