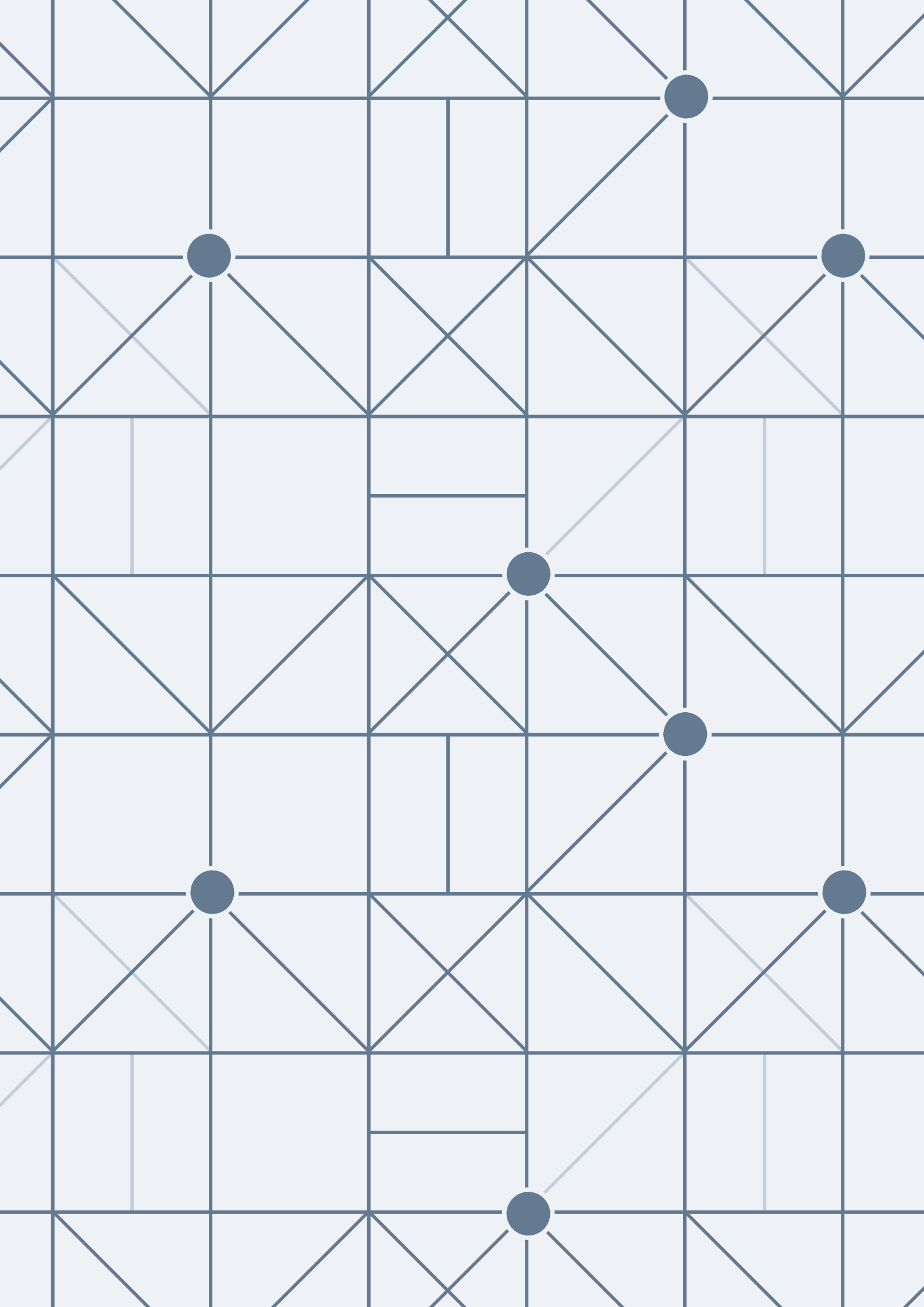


Backtracking

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)

→ means back from here

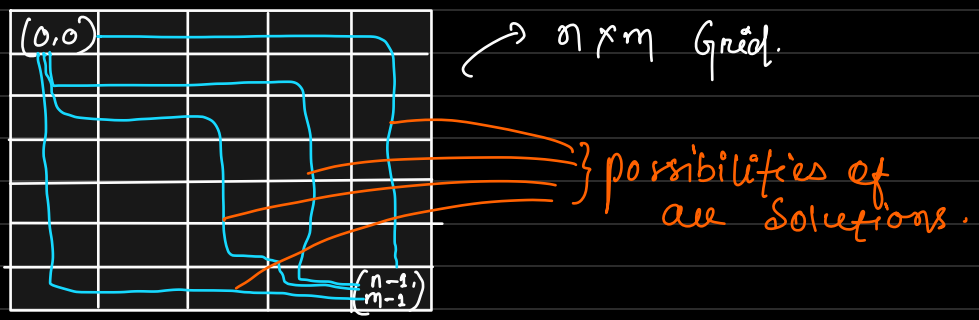
Types Of Backtracking

1 - Decission.

2 - Optimization

3 - Enumeration.

* Examples of Decission backtracking



* Example of Optimization backtracking

There may be multiple Way to findout the destination. But We have to find the most optimized Way to reach.

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

But We have to findout the shortest path., So Here Optimization Comes handy.

* Example of Enumeration backtracking

Means total number of possibilities to reach the destination / to findout the Solution.

So may be We just have to print all the possible Way, or to Count no. of possible Solution.

Backtracking - arrays

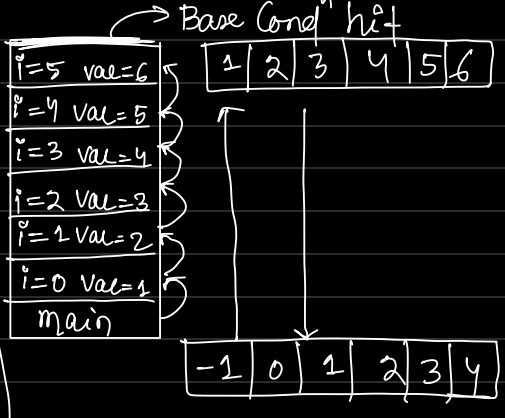
1	2	3	4	5	6

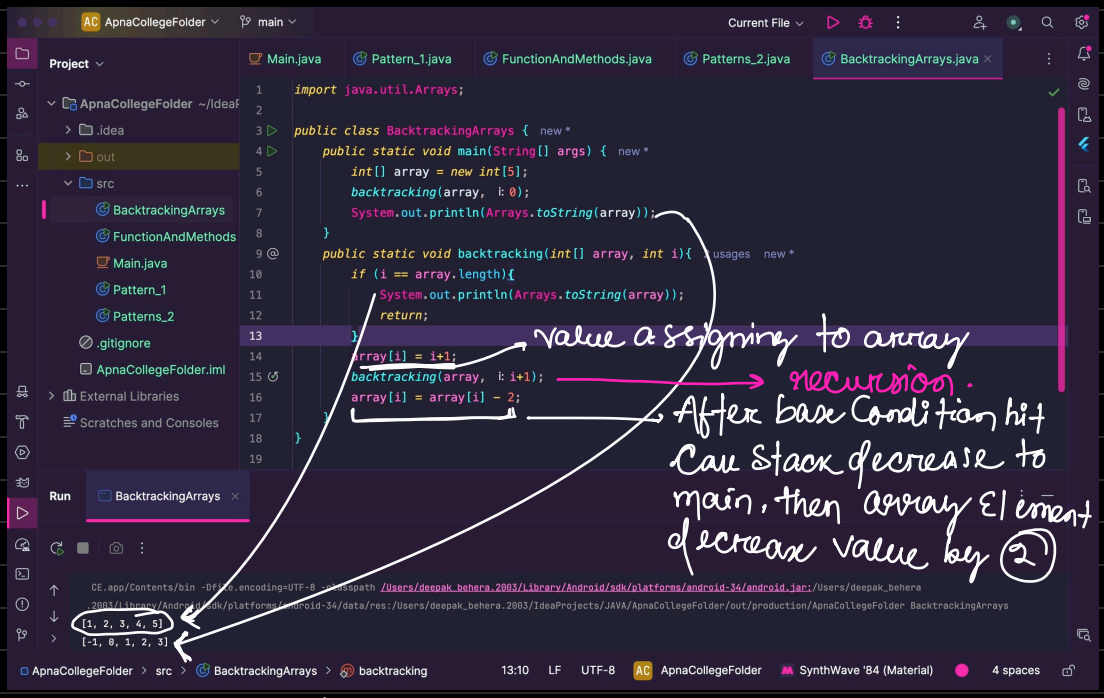
main i=0 i=1 i=2 i=3 i=4 i=5

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





time Complexity $= O(n) + O(n)$
$= O(2n)$
$= O(n)$

Find subsets

(find and print all subsets of a given string)

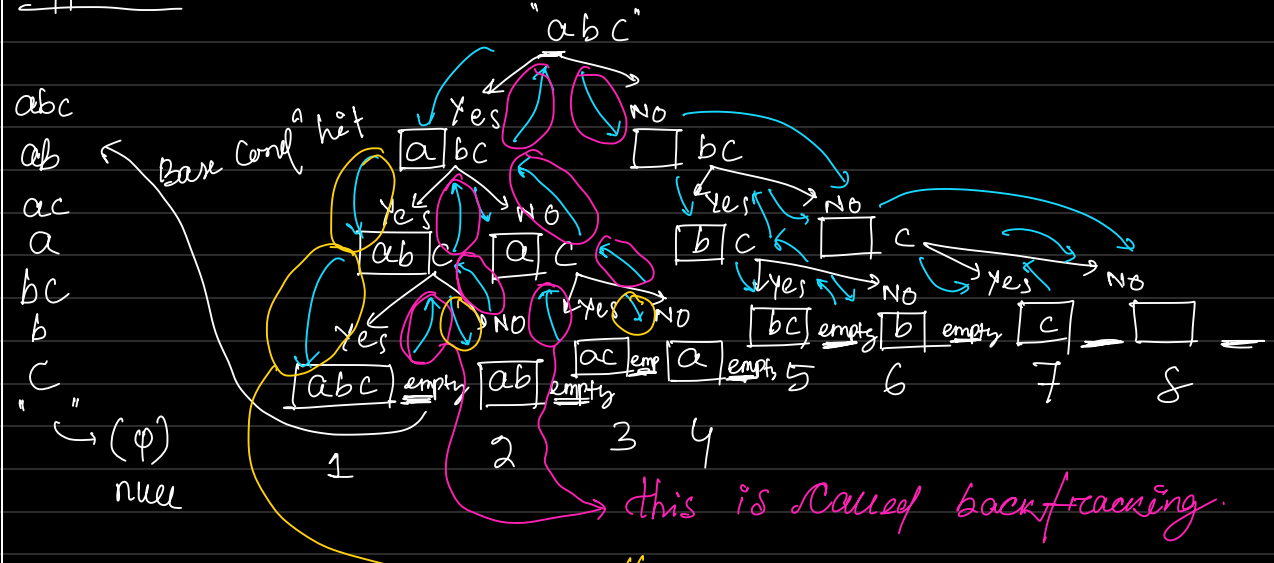
"abc" → a, b, c, ab, bc, ac, abc, "" ↳ empty set / null set = (φ)

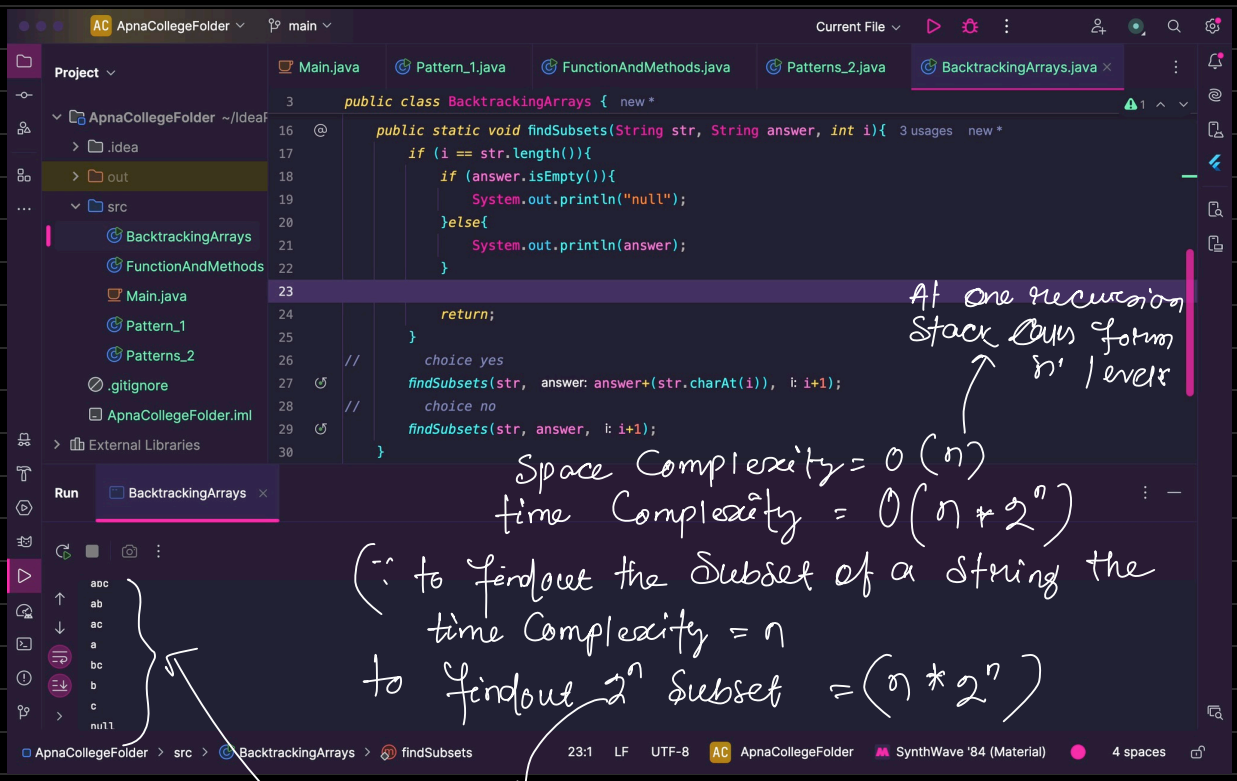
total 8 subsets.

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

Also Applicable in Arrays.

Approach





At one recursion Stack Goes form n' level

Space Complexity = O(n)

time Complexity = O(n * 2^n)

(∵ to findout the Subset of a string the time Complexity = n

to findout 2^n Subset = (n * 2^n))

abc = n = 3

2^3 = 8 Subsets.

"abcdef" → String

a have 2 Choices
b have 2 choices
c have 2 Choices
:
f have 2 Choices

2 * 2 * 2 * ---- * n times = 2^n Subsets.

Find permutations

(find and print all permutations of a String)

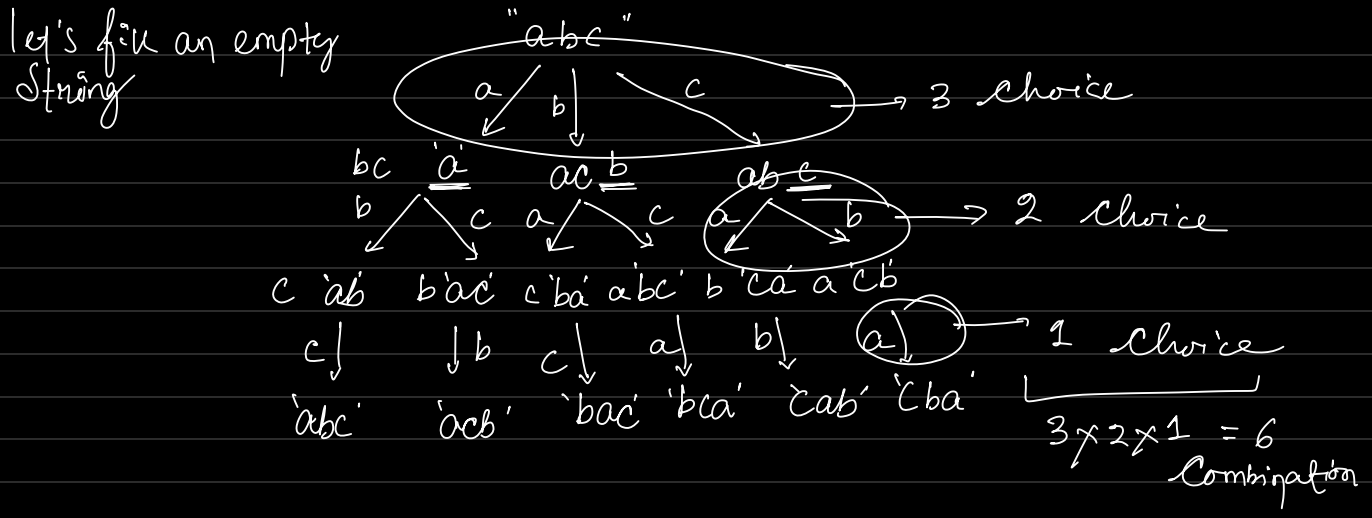
* If there is a string of length = 'n'
then no of Combination would be = $n!$

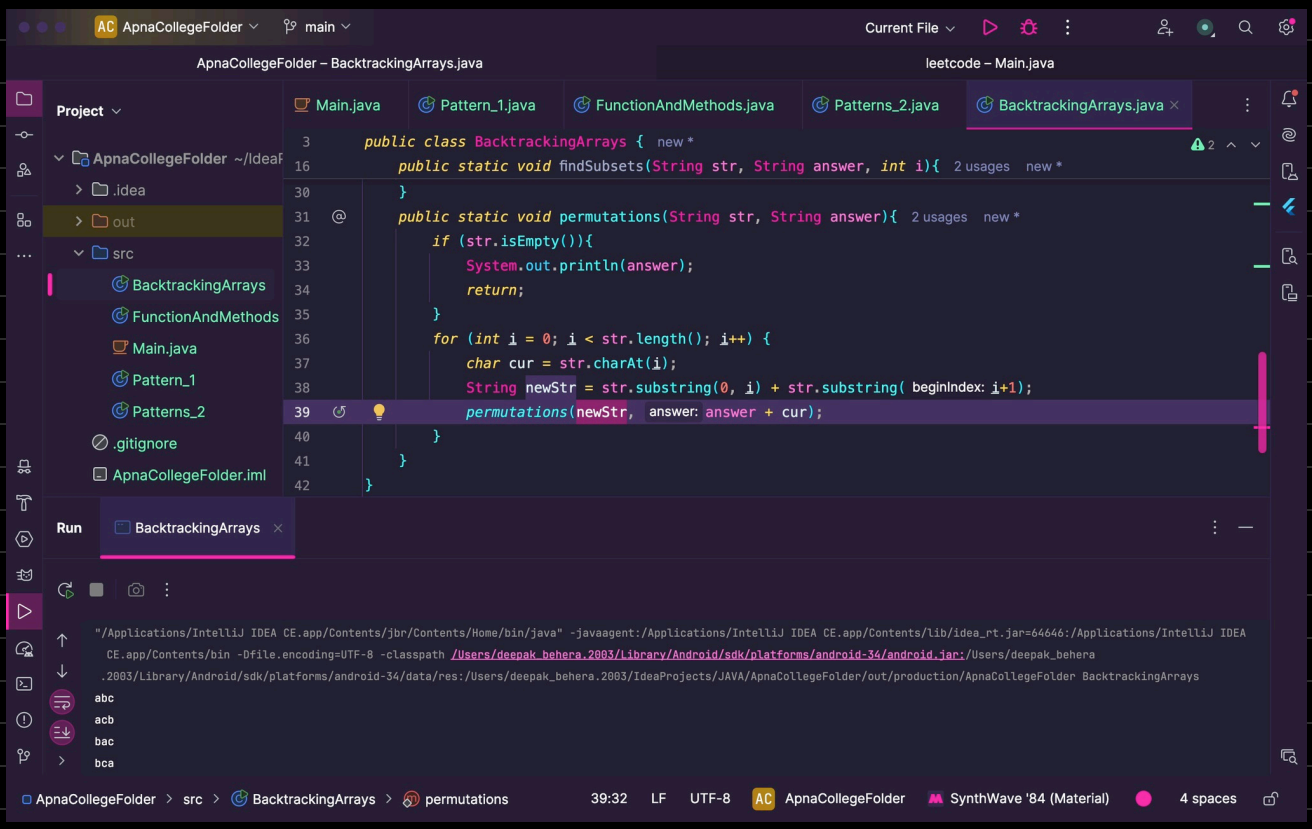
(Also applicable in case of Array)

for example: for String 'abc' → length = 3

abc, acb, bac, bca, cab, cba } 3! = 3x2x1
= 6 Combination

let's fill an empty String

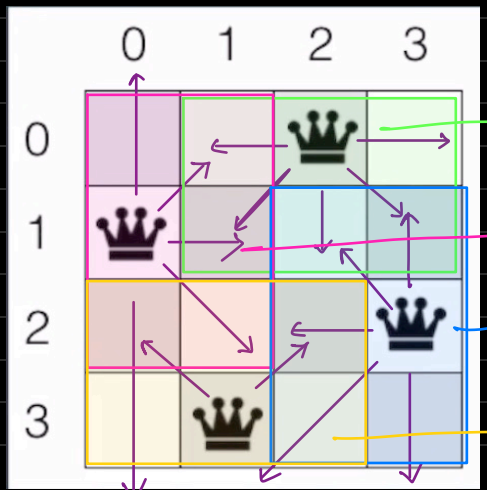




N - QUEENS

Place N Queens on an N x N chessboard such that no two queens can attack each other.

N = 4

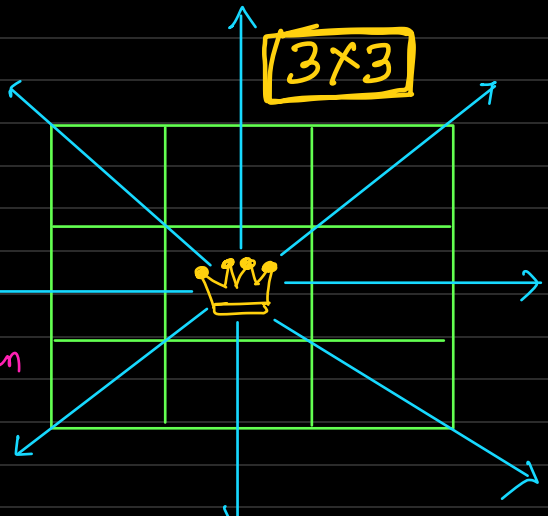


first row 1 queen

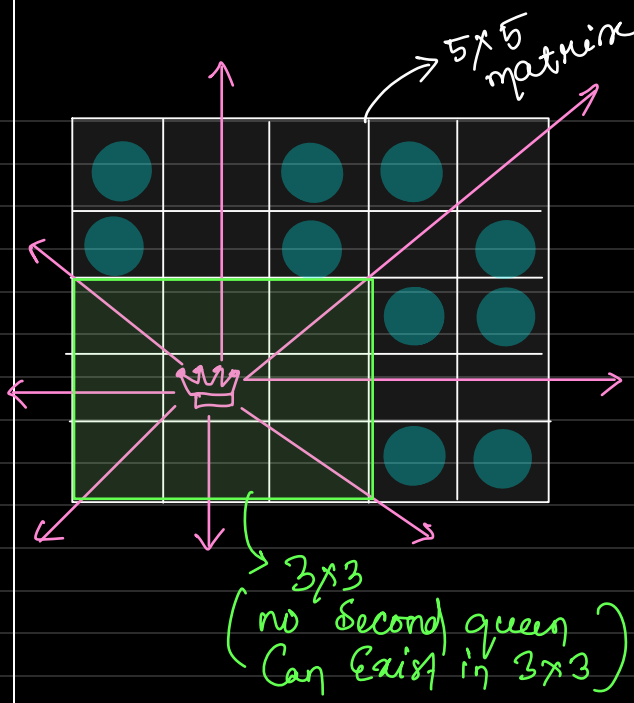
2nd row 1 queen

3rd row 1 queen

4th row 1 queen.



* Can attack these directions.



→ Can't attack places.

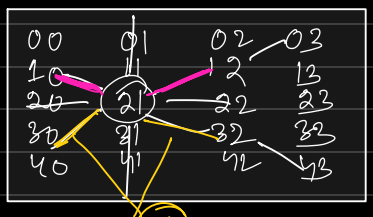
→ Question can be

1. Find all solutions
2. Yes/No → is there is any sol" or not
3. Count no of solution.

Logic : Vertical - row++ / row--, column constant

horizontal - row constant, column++, column--

diagonal - row--, column--,
row--, column++,
row++, column--,
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)

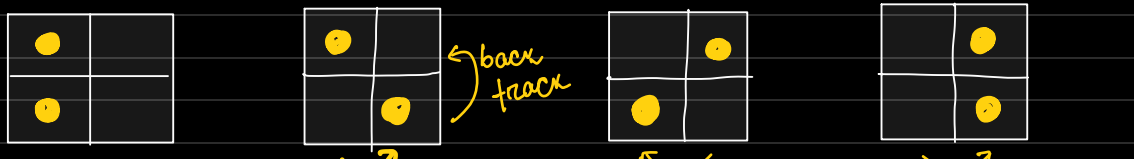
(So if there is queen of length = 2, then we need 2 rows)

(∴ No of rows = no of queens)

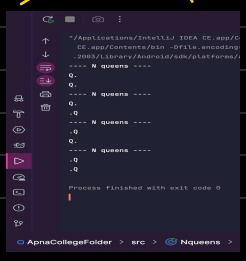
(Let's forgot that 2 queens attack each other for some times)

if n = 2

Possible solutions



(1 case) (2nd case) (back track (3rd case)) (4th case)



```java
public class Nqueens {
    public static void initialization(int n){ 1 usage
        char[][] matrix = new char[n][n];
        for (int i = 0; i < matrix.length; i++){
            for (int j = 0; j < matrix.length; j++){
                matrix[i][j] = '.';
            }
        }
        nQueens(matrix, row: 0);
    }
    public static void nQueens(char[][] matrix, int row){ 2 usages
        if (row == matrix.length){
            printChess(matrix);
            return;
        }
        for (int i = 0; i < matrix.length; i++){
            matrix[row][i] = 'Q';
            nQueens(matrix, row: row+1);
            matrix[row][i] = '.';
        }
    }
    public static void printChess(char[][] matrix){ 1 usage
        System.out.println("---- N queens ----");
        for (int i = 0; i < matrix.length; i++){
            for (int j = 0; j < matrix.length; j++){
                System.out.print(matrix[i][j]);
            }
            System.out.println();
        }
    }
}
```

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)

```
---- N queens ----
Q.
..
---- N queens ----
..
Q.
---- N queens ----
.Q
..
---- N queens ----
..
.Q
Process finished with exit code 0
```

using recursion → using backtrack

recursion

backtrack recursion

backtrack recursion

backtrack

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

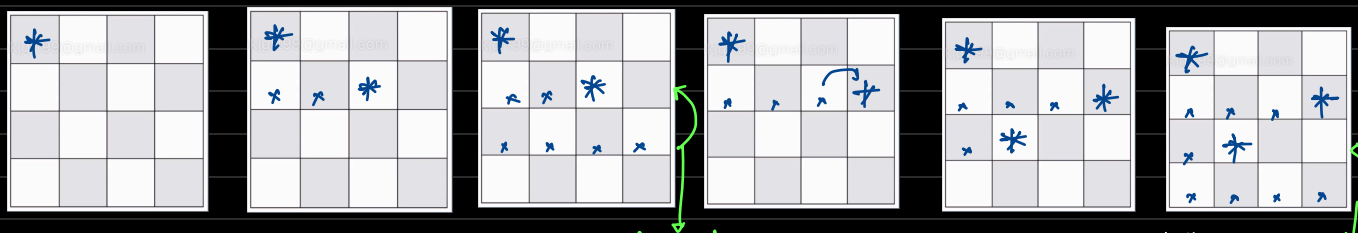
(We have to check the condition)

isSafe(matrix, row, j)

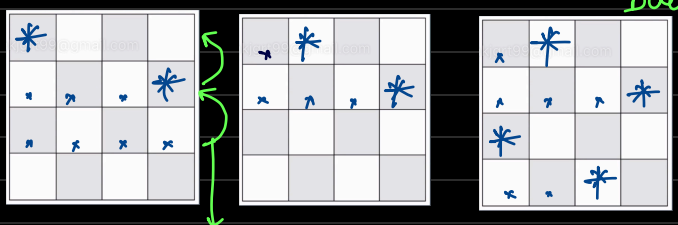
(if returns true, then we can place

Otherwise, we should'nt place the Queens)

Step-1



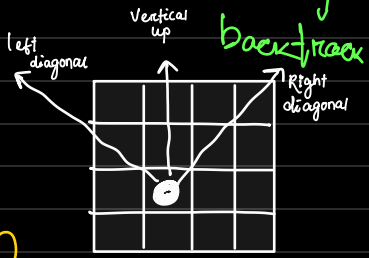
backtrack



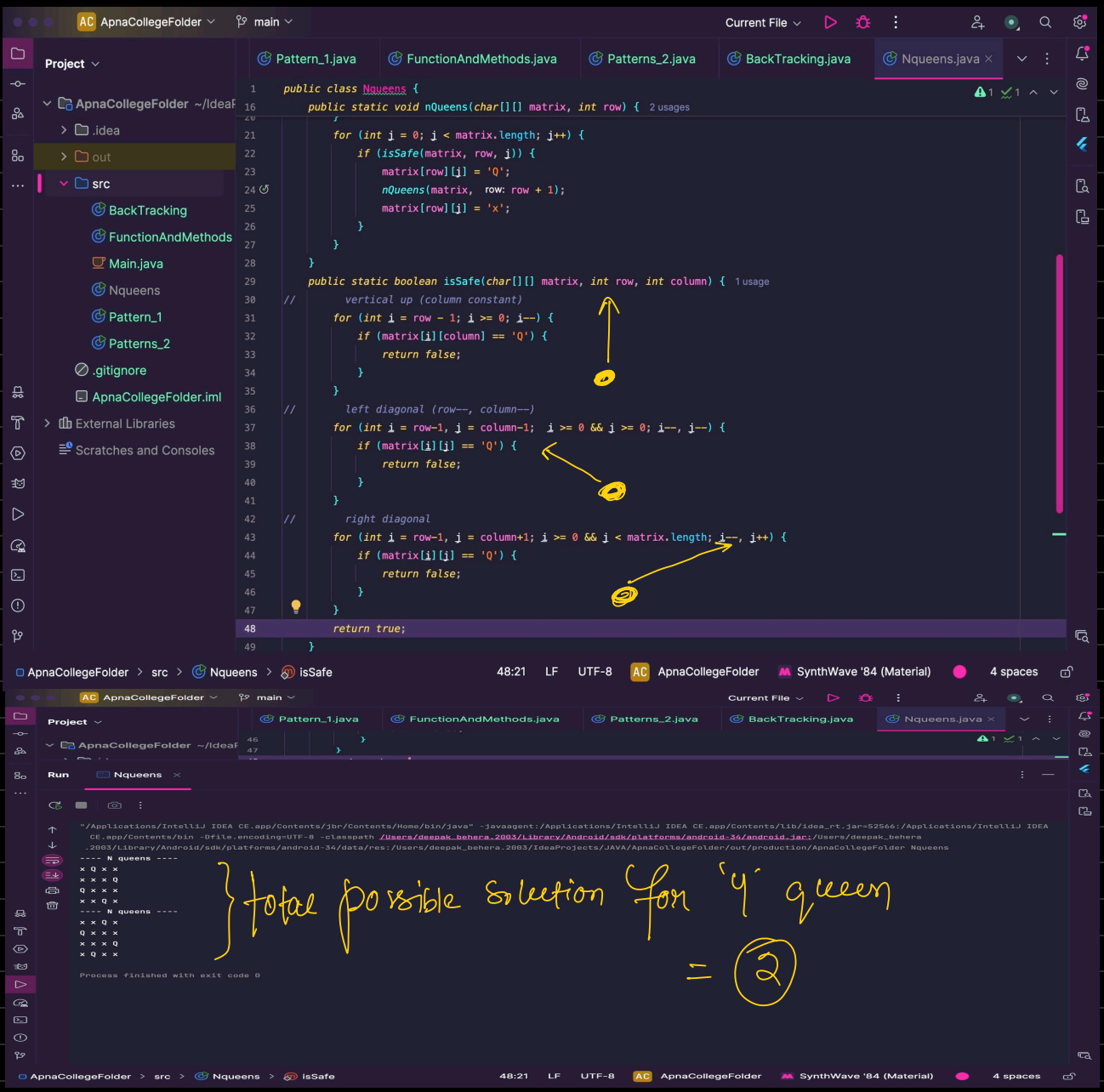
backtrack

(Although this is last position we again backtrack to up)

Valid position



backtrack



N. Queens (time complexity)

if there is 'n' number of queens. So

For 1st queen → n choices

2nd queen → n-1 choice

3rd queen → n-2 choice

— — — — nth queen → 1 choices

$= n \times (n-1) \times (n-2) \times \cdots 1$

$= n! = O(n!)$

Recurrence Equation :-

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

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

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 it 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: row + 1)){
                return true;
            }
            matrix[row][j] = 'x';
        }
    }
    return false;
}
```

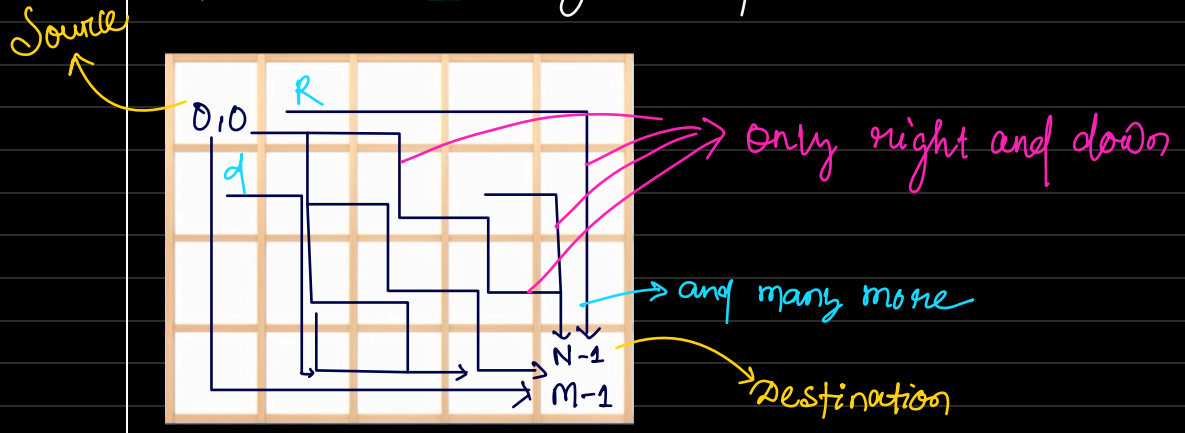
{ Solution exist for next row

} → no solution for n queens

Grid ways

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

(Allowed moves - right or down)



Approach

Cell (x, y)

right (x, y+1) (row Constant; Column++;) → no of ways = w1

down (x+1, y) (row++, column Constant) → no of ways = w2

total no of Ways = w1 + w2

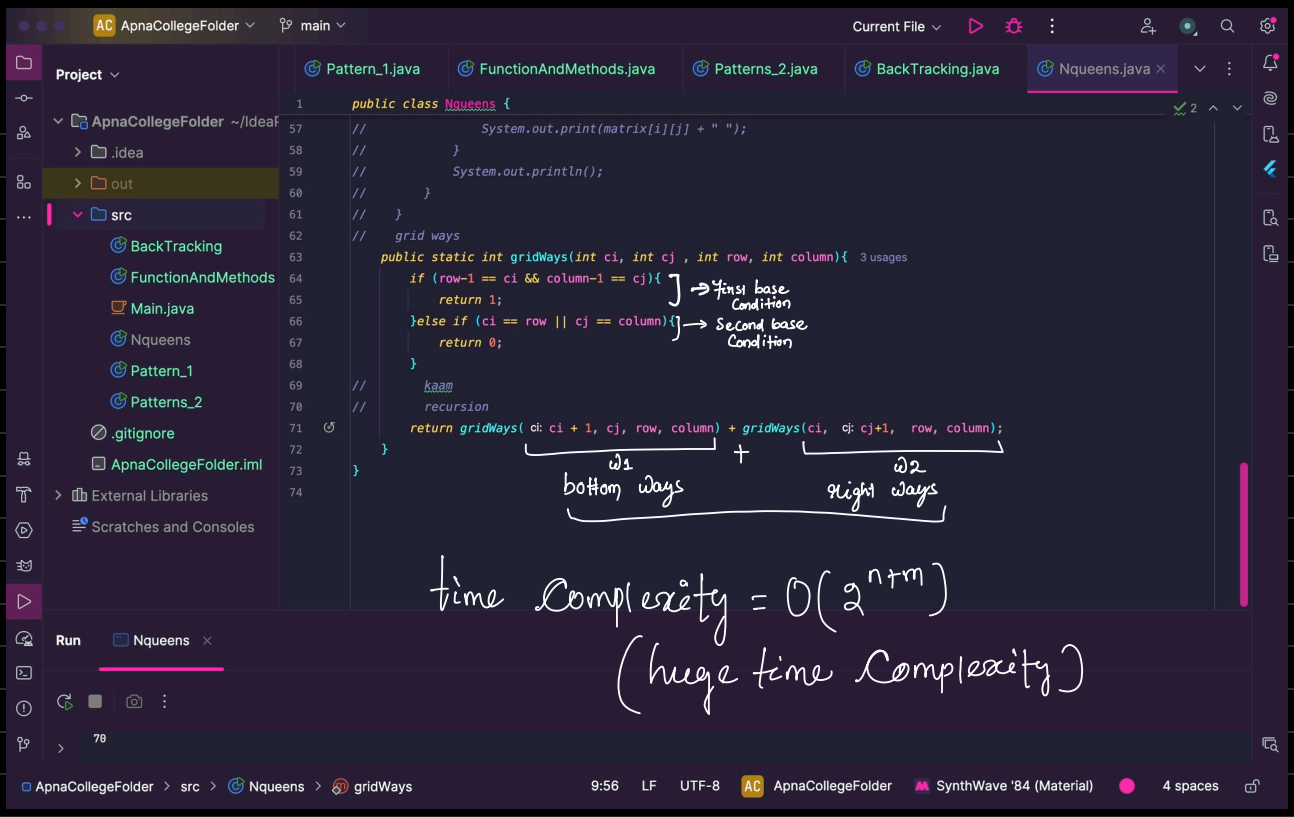
$$f(x,y) = f(x,y+1) + f(x+1,y)$$

↳ recursion Equation

Base Case

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

also their is another base condition, i.e if we cross the right boundary or cross the bottom boundary the return 0;



(Math trick for linear time complexity

way/path - (n-1) D
(m-1) R total characters = ((n-1)+(m-1))

if we found the permutation of these strings then we could found the no of ways

DD
RR

DDRR
DRDR
RRDD
RDRD
DRRD
RDDR
} there is also duplicacy

So Repeating → (n-1) D
(m-1) R

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

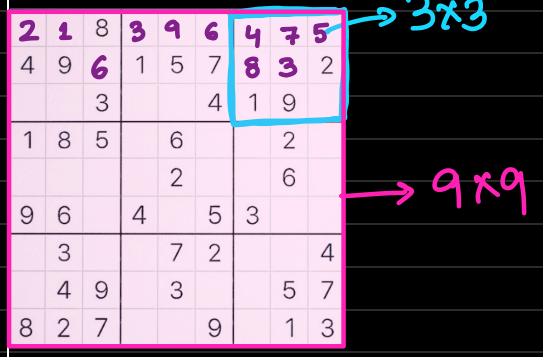
→ total number of ways

And the time Complexity will reduce to O(n+m)
= O(n)

if n=3, m=3

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

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 violets 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- Back tracking

- 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)

Starting row = (row/3) *3
Starting column = (column/3) *3
} for 3x3 grid Constraints

```java
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 row

if there is already some number exist at the place

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

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

```java
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 3×3 grid of the current cell