Back tracking. - Arrays.

1	2	3	4	5	⇒	-1	0	1	2	3

↳ Normal Recursion step. ↳ while Backtracking.

arr[i] = val

func arr, index, val).
or
i

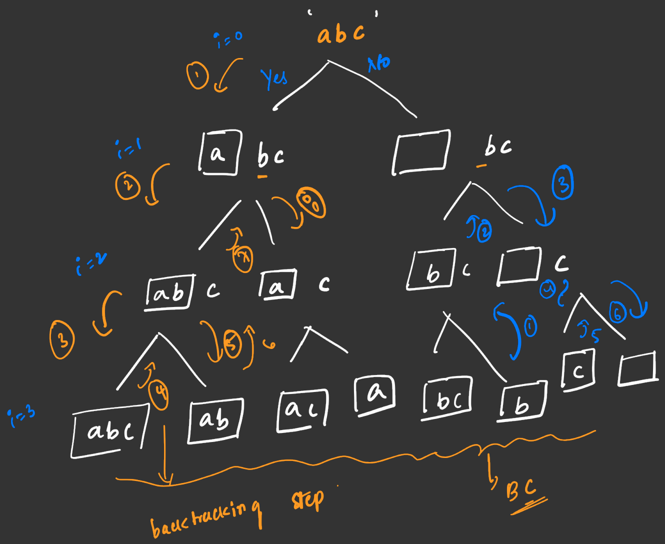
```
func (arr, 0, 1)        -1  0 1 2 3
 |-- [1, ~~0,0,0~~, 0]
 |-- func (arr, 1, 2)
 |-- arr[0] -= 2
        0 1 2 3
      |-- [1, 2, ~~0,0~~, 0]
      |-- func ( arr, 2, 3)
      |-- arr[1] -= 2
                1  2  3
            |-- [1, 2, 3, ~~0~~, ~~0~~]
            |-- func (arr, 3, 4)
            |-- arr[2] -= 2
                              2  3
                  |-- [1, 2, 3, 4, ~~0~~]
                  |-- func (arr, 4, 5)
                  |-- arr[3] -= 2
                                    3
                        |-- [1, 2, 3, 4, 5]
                        |-- func ( arr, 5, 6).
                                base case
                                if i == len(arr):
                                      print(arr)
                                      return.
                        |-- arr[i] = arr[i] -2
```

print(arr)

= -1, 0, 1, 2, 3.

Finding * Subsets. (2^n) subsets of length n.

abc => '', a, b, c, ab, bc, ac, abc.



abc

ab

ac

a

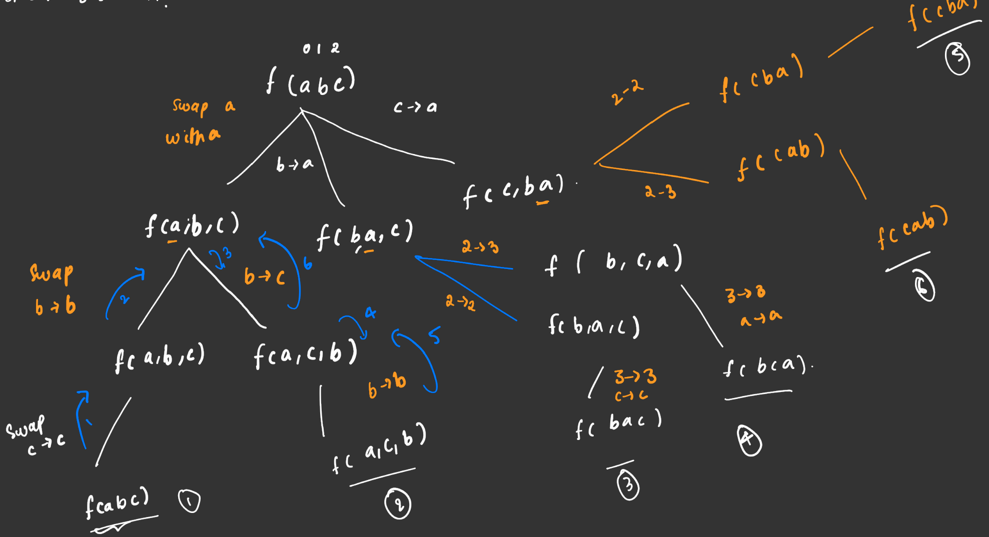
bc

b

c

'' Null

Permutations : n!



print it

and backtrack.

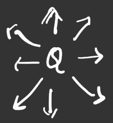
```
        0
f( index , str)
  |→     i → (0 → n-1)
               left / start
         |— swap ( a[i], a[index] (or) a[left?] )
         |→ f( index+1, str)
         |→ swap then back  => back tracking.
```

N Queen.

↳ n queens puzzle is the problem of placing n queens on an n x n chess board. such that no 2 queens attack each other.

N = 4.

Attack :



→ 8 directions.

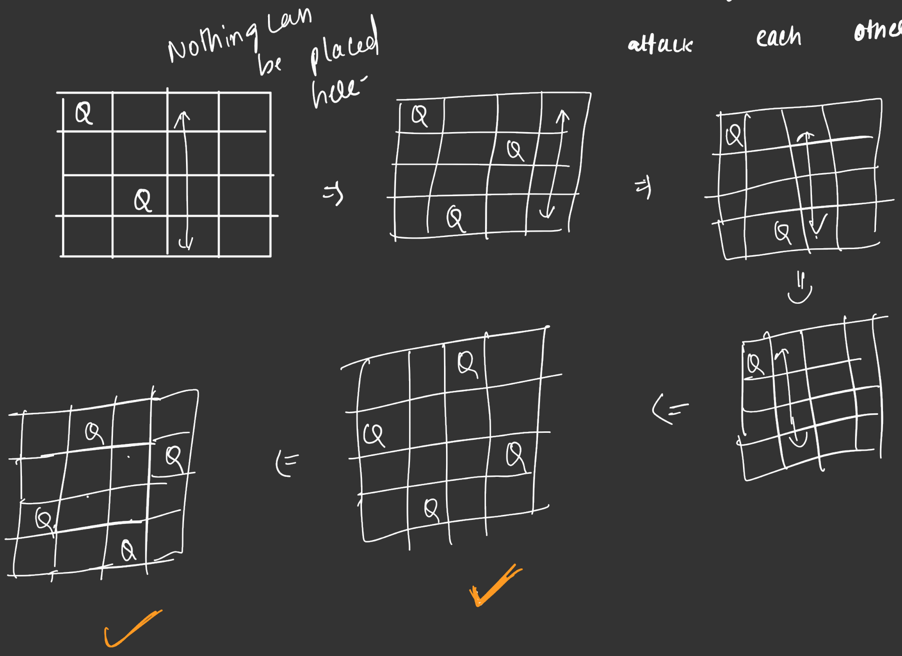
Rules :

① every row should have a queen

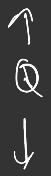
② every col should have a queen

③ None of them should attack each other.

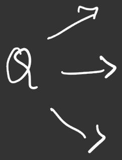
①



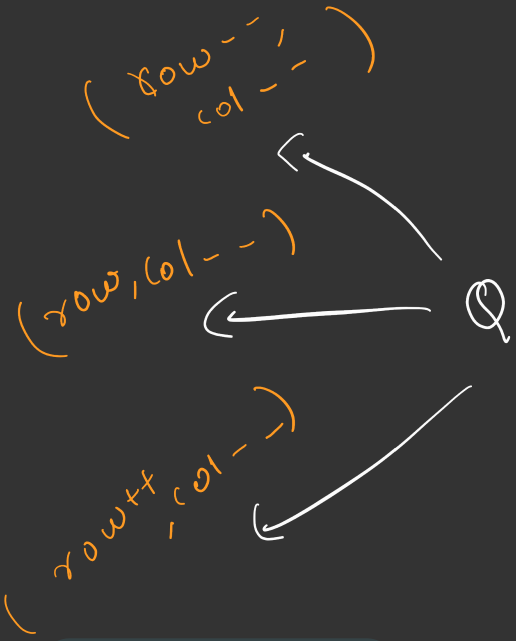
How & where to Place Queen ??



This isnt req to check as we are going through col wise.

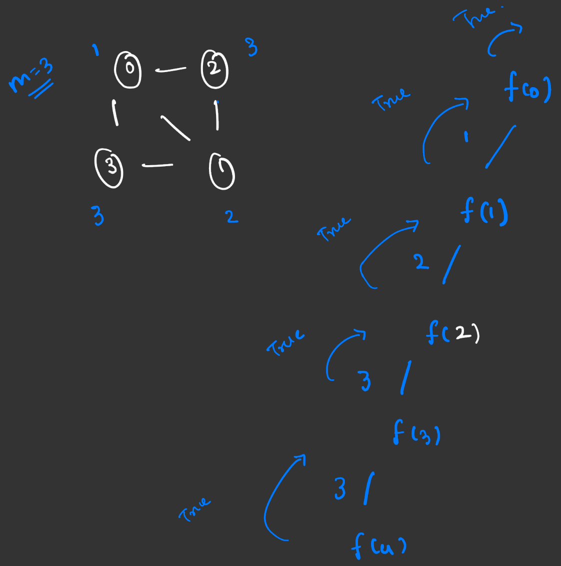


This side isnt req to check as we go col by col



M coloring | Graph coloring

↳ Decision , Enumeration problem.



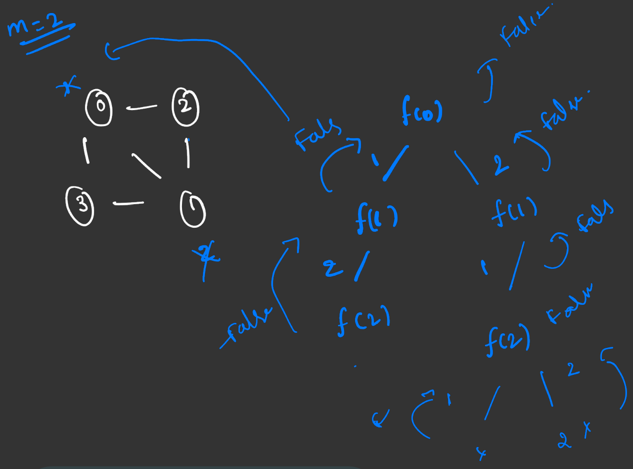
↳ if n == len(arr) :

✓

```
fun (node) {
    if node == N :
        return True.

    for ( col : 1 -> N ) :
        if possible -> V :
            col [node] = col
            if (f (node+1))
                return True
            col [node] = 0;

    return false.
```



Time Complexity :

(N^m)

SC -> $O(N) + O(N)$

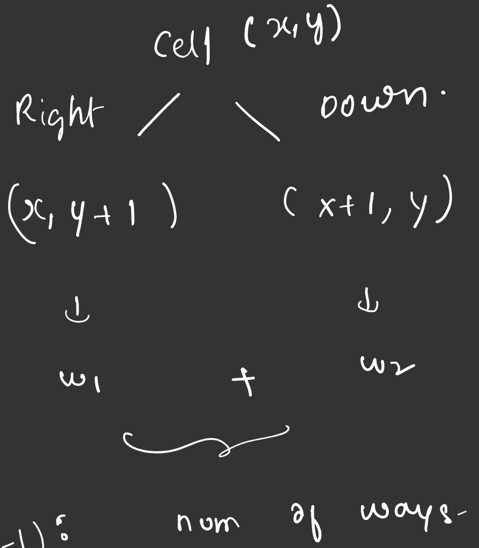
recursi col

Grid ways :

Find num of ways to reach from (0,0) to (N-1, m-1) in N x M grid.

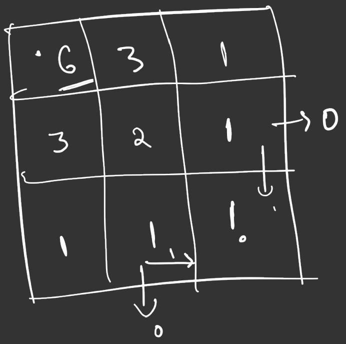
Allowed moves ⇒ right , down.

•			1
			1
4	3	1+1 =2	1+0=1
1	1	0+1 =1	•



if src = (n-1 , m-1):

$$\Rightarrow f(x,y) = \underbrace{f(x+1,y)}_{down} + \underbrace{f(x,y+1)}_{Right}$$



T.C = $2^{(n+m)}$.

→ Sudoku.

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

Rules:

① The digit 1-9 shd appeal only once in any / every row / col

② within 3x3 matrix it should appeal once.

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

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

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

(wait)

↳ After backtrack.

false

remove

1

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

False

remove the elem

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

True.

```
for i → (0 - 9):
    for j → ( 0-9):
        if board [i] [j] = = '.':
            for k ( 1 - 9] :
                if is valid ( board, i, j, k):
                    board [i] [j] = k
                    if solve (board ):
                        return True.
                    else :
                        board [i) [j] = '.'
    return False.
```