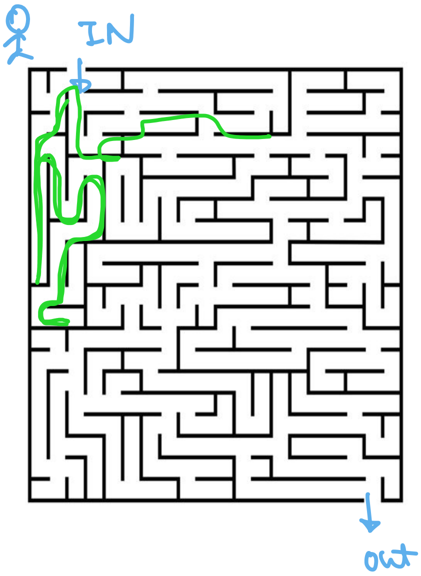
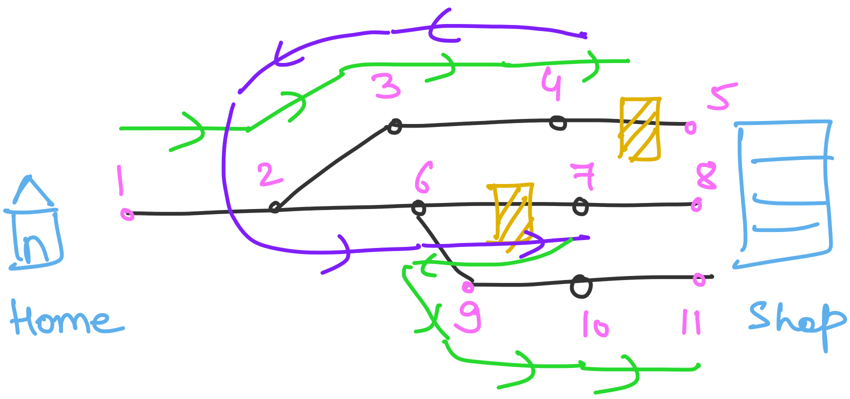
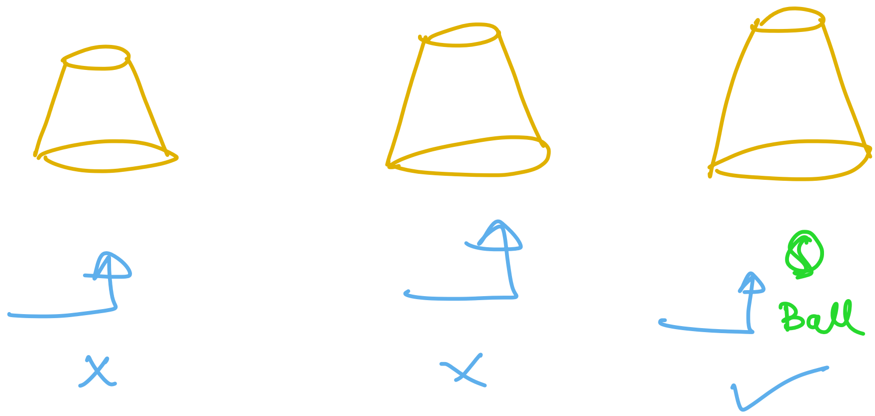
PW SKILLS

Backtracking





Home → 1 → 2 → 6 → 9 → 10 → 11 → Shop



It is a technique for solving problems recursively by trying to build the sol^n incrementally, one piece at a time, removing those sol^ns which fail to satisfy the constraints at any point of time.

Types of backtracking problems

- Decision problem
- Optimization problem
- Enumeration problem

Decision problem

In this we search for a feasible soln.

↓

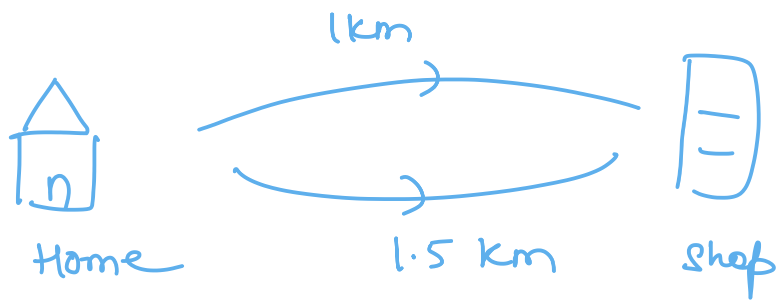
possible soln

↓

Something that will work for us.

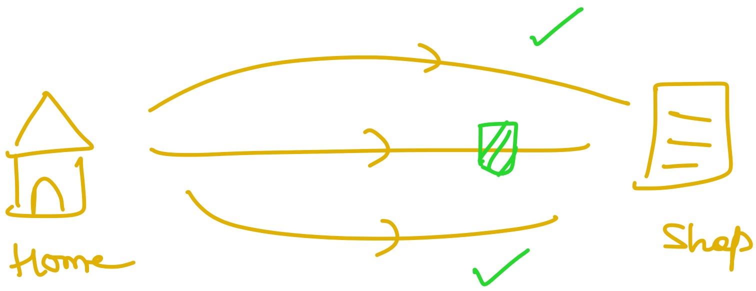
Optimization problem

In this we look for the soln which will work best for us.



In this case, 1km will be the optimal soln (or best soln) because it takes lesser time.

Enumeration problem
↓
Collection



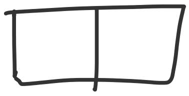
Here we are targetting to find all the feasible $sol^n s$.

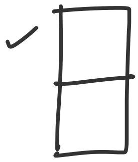
Q Tiling problem — Amazon

Given a board of size "2 * n" and tile of size "2 * 1", count the number of ways to tile the given board.

Sol^n For placing a tile, we have

2 placements
|

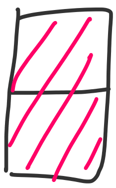
Horizontally  $1 * 2$

Vertically  $2 * 1$

<u>$n = 1$</u>

size of board $= 2 * n$

$= 2 * 1$



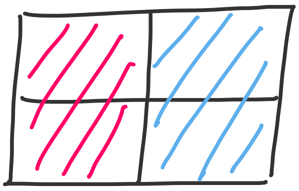
Placed the tile vertically

Total no of ways $= 1$

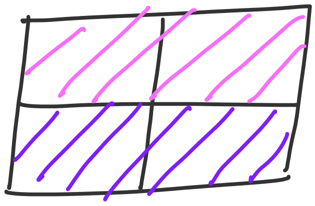
<u>$n = 2$</u>

Board size $= 2 * n$

$= 2 * 2$



Both vertically



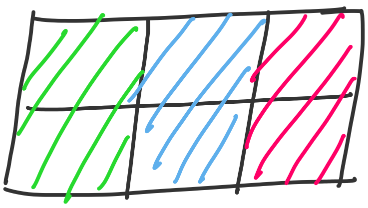
Both horizontally

Total no of ways $= 2$

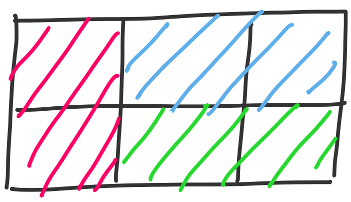
$n = 3$

Board size $= 2 * n$

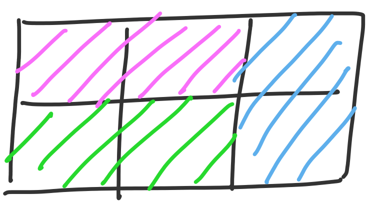
$= 2 * 3$



All vertically



one vertically,
two horizontally

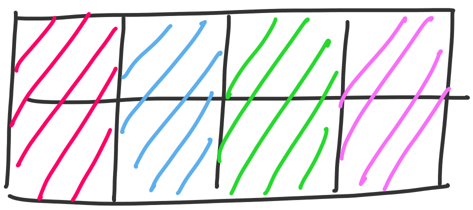


Two horizontally,
one vertically

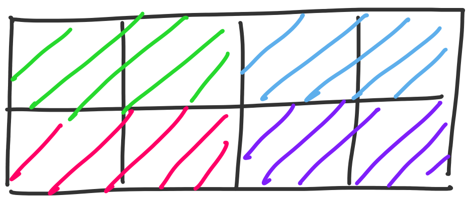
Total no of ways = 3

$\underline{n = 4}$

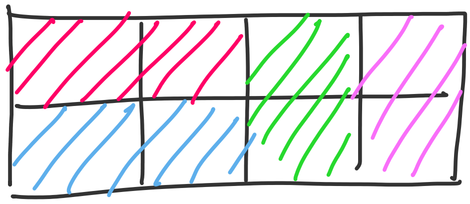
Board size = $2 * 4$



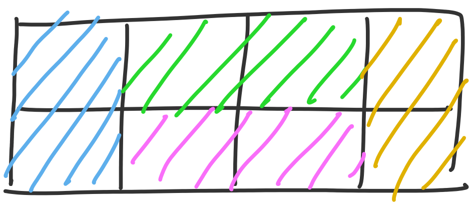
All vertically



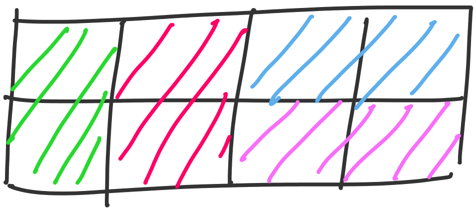
All horizontally



2H + 2V



1V + 2H + 1V



2V + 2H

Total no of ways = 5

n	Total no of ways	
1	1	Base case
2	2	
3	3	
4	5	Recursive call
5	8	
	⋮	

```
Getways(n)
{
    Base case

    Recursive call
}
```

Base case

if (n == 1)

↳ return 1

if (n == 2)

↳ return 2

if (n == 3)

↳ return 3

combine them

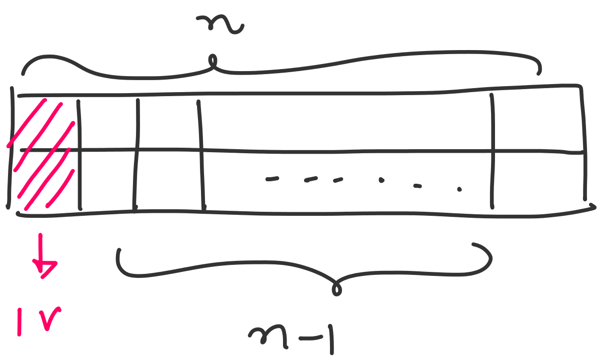
```
if (n == 1 || n == 2 || n == 3)
{
    return n;
}
```

$\boxed{\text{OR}}$

if (n ≤ 3)

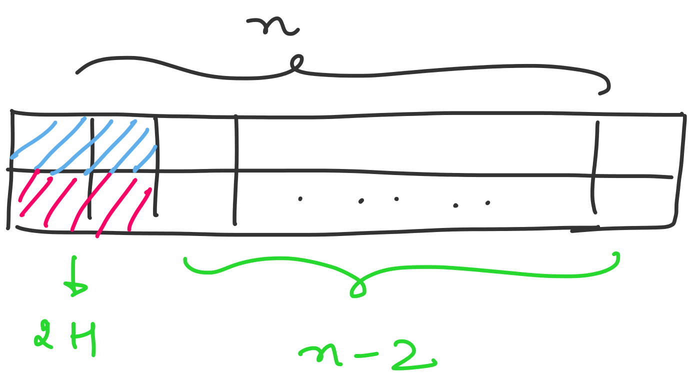
```
{
    return n;
}
```

Recursive call



$2 * n$

If we place one tile vertically then the problem reduces to $n-1$, so for this we will use Getways $(n-1)$.



$2 * n$

If we are placing one tile horizontally then we will have to place second tile also horizontally, so the problem reduces to $n-2$, so for this we can call Getways $(n-2)$

```
Getways (n)
{
    if (n <= 3)
    {
        return n;
    }

    return  Getways (n-1) +
            Getways (n-2);
}
```

Dry Run

$n = 4$

Getways (4)

$\downarrow$

$n = 4$

$n \leq 3$

$4 \leq 3$

$\downarrow$

no

return $\underbrace{\text{Getways}(3)} + \underbrace{\text{Getways}(2)}$

Getways (3)

$\downarrow$

$n = 3$

$n \leq 3$

$3 \leq 3$

$\downarrow$

yes

$\hookrightarrow$ return $n = 3$

Getways (2)

$\downarrow$

$n = 2$

$n \leq 3$

$2 \leq 3$

↓

yes

↪ return $n = 2$

$$\text{Getways}(4) = \text{Getways}(3) + \text{Getways}(2)$$
$$= 3 + 2$$
$$= 5$$

Q Print all the sequences of a string using recursion.

Eg- $\{1, 2\}$

↓

$\{1\}$, $\{2\}$, $\{1, 2\}$

Eg- $\{1, 2, 3\}$

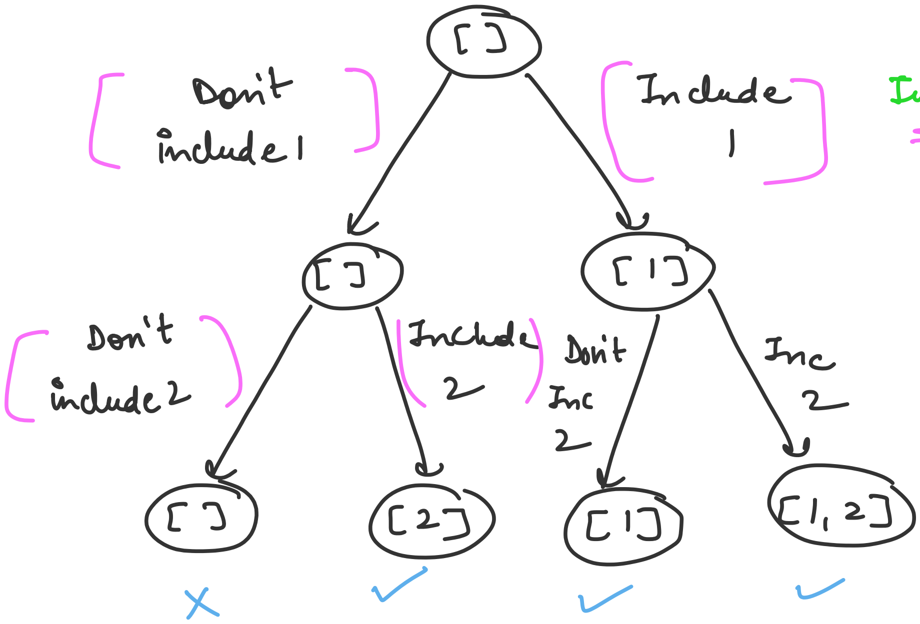
↓

$\{1\}$, $\{2\}$, $\{3\}$, $\{1, 2\}$,

$\{1, 3\}$, $\{2, 3\}$, $\{1, 2, 3\}$

0 1

Eg - {1, 2}



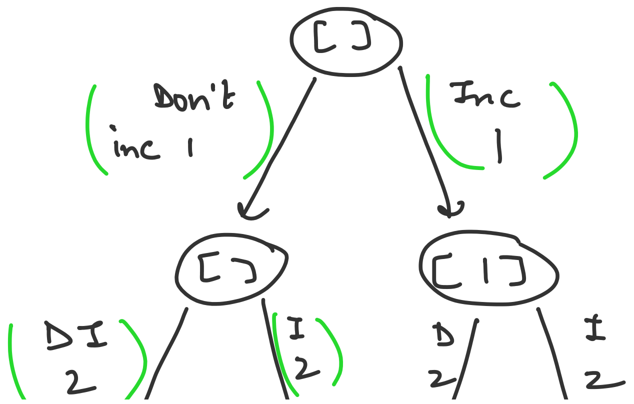
Index = 0

Index=1

output = [2], [1], [1, 2]

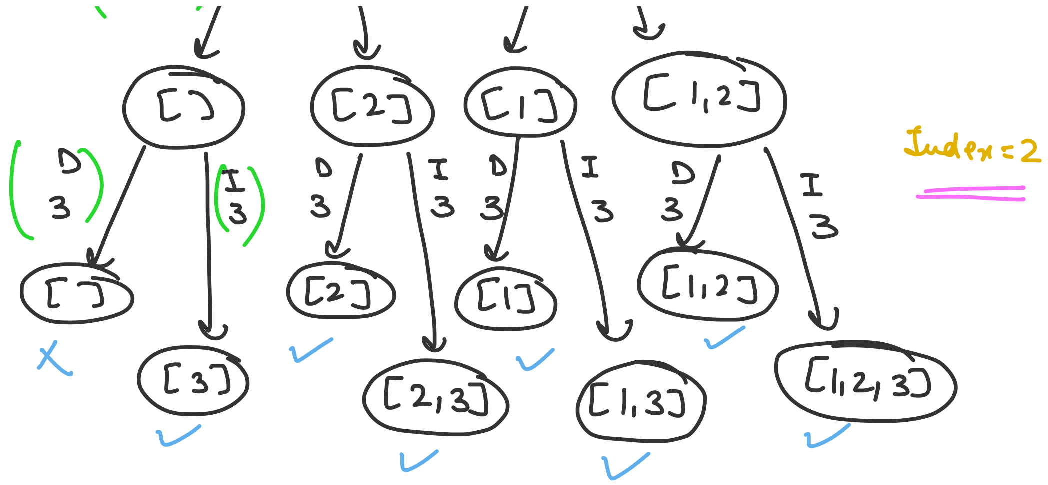
Another eg-

0 1 2
{1, 2, 3}



Index = 0

Index = 1



output - [3], [2], [2,3], [1], [1,3], [1,2], [1,2,3]

for every element in the array, we have 2 choices -

1. To include in the subsequence
2. To not include in the subsequence.

Apply this on all the elements in the array starting with index 0, and do this recursively until we reach the last index.

Base case -

if (index == length of array)

↓

print all the subsequences.

Recursive call

PS(arr, index, tempArr)

↳ To store subsequences

{

PS(arr, index+1, tempArr) → Include

Store in tempArr ← Add the value in tempArr

↓

arr [index]

PS (arr, index+1, tempArr) → Don't include

Remove the last value

from tempArr

}

Note -

Here tempArr is dynamic array – which is called as array list, because arrays are of fixed length, means they cannot grow or shrink in size. But arraylists can grow or shrink according to the requirement.

Dry Run –

arr : {1, 2}

1. PS (arr, 0, tempArr)

tempArr = []

index = 0

arr.length = 2

if (0 == 2)

no

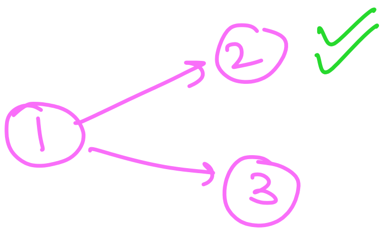
fails

2 PS (arr , 1, tempArr) ;

tempArr. add (1)

3 PS (arr, 1, tempArr);

Remove last value from tempArr.



2 PS (arr, 1, tempArr)

tempArr = [ ]

index = 1

if (1 = = 2)

↓ no

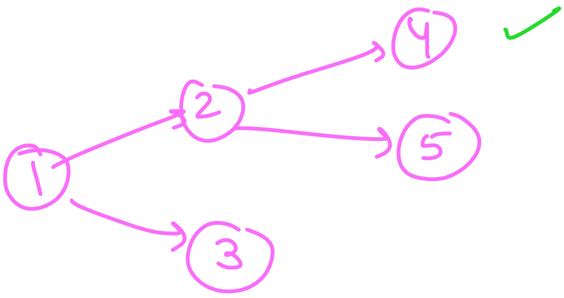
fails

4 PS (arr , 2 , tempArr) ✓

tempArr. add (2) $\boxed{2}$

5 PS (arr, 2, tempArr)

Remove the last value from tempArr



4 PS (arr, 2, tempArr)

tempArr = [ ]

index = 2

if (2 = = 2)

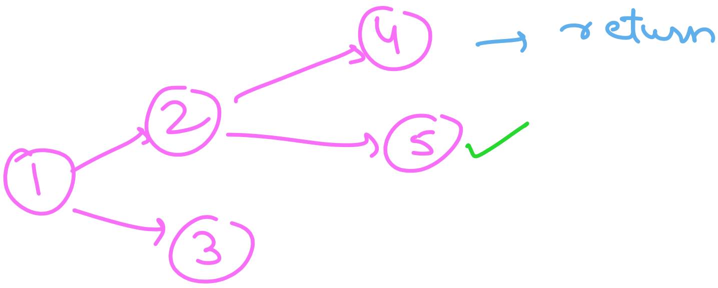
↓ yes

print

tempArr.size() = 0

↓

Return



5 PS (arr, 2, tempArr)

tempArr $\boxed{2}$

index = 2

if (2 = = 2)

yes

print → [2]

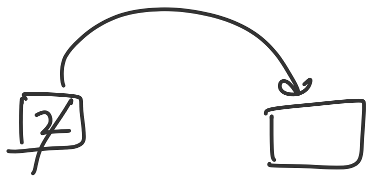
Remove the last value from tempArr

tempArr $\boxed{2}$

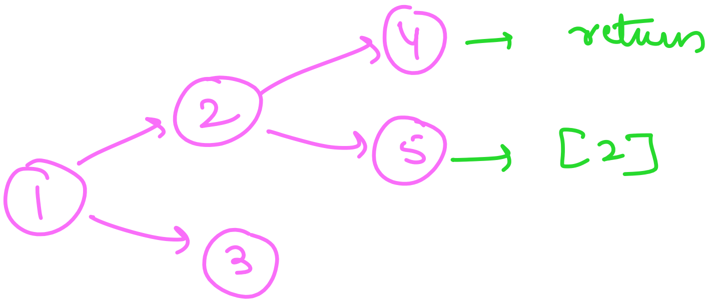
tempArr. remove (tempArr. size() - 1)

tempArr . remove (1 - 1)

tempArr. remove (0)



tempArr



② PS (arr, 1, tempArr);

tempArr.add (1) $\boxed{1}$ tempArr

③ PS (arr, 1, tempArr);

Remove last value from tempArr.

③ PS (arr, 1, tempArr)

↓

tempArr = [1]

index = 1

if (1 = = 2)

↓ no

fails

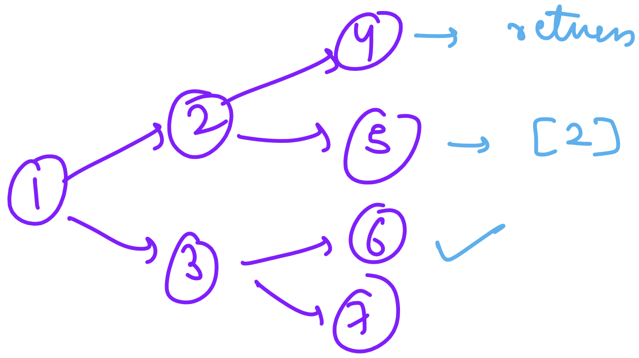
6 PS (arr, 2, tempArr)

tempArr. add (2)

$\boxed{1 \mid 2}$

7 PS (arr, 2, tempArr)

Remove last value



6 PS (arr, 2, tempArr)

tempArr [1]

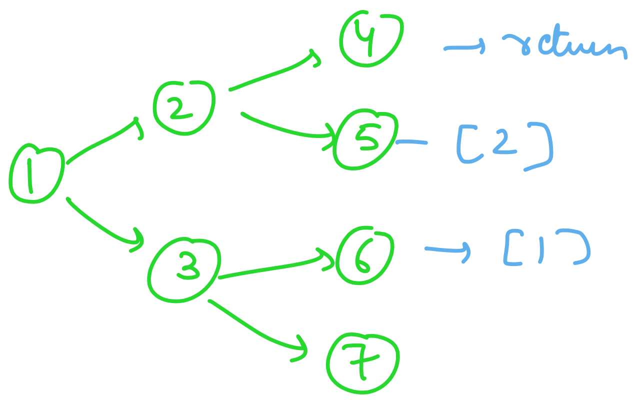
index = 2

if (index = = arr. length)

if (2 = = 2)

↓ yes

print [1]



7 PS (arr, 2, tempArr)

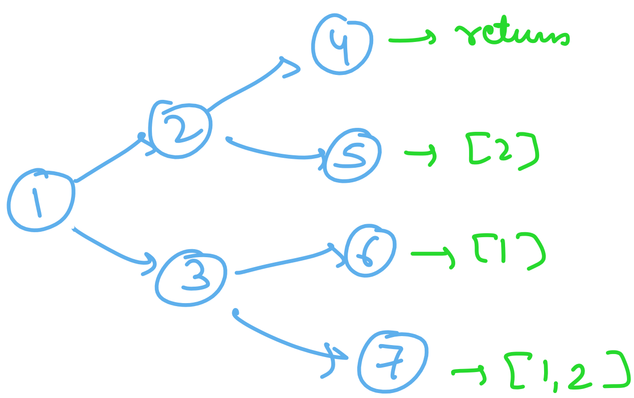
tempArr = [1, 2]

index = 2

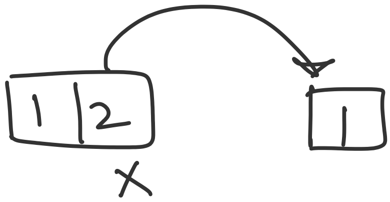
if (2 = = 2)

↓ yes

print [1, 2]



Remove last value from tempArr



2 PS (arr, 1, tempArr); ✓

tempArr.add(1) ✓

3 PS (arr, 1, tempArr); ✓

Remove last value from tempArr.

↓

Remove last value

[1] → [ ]

↓

Empty

Q Print all permutations of the given string

↓

Rearrangement of values

Eg -

string = " xy "

Permutations = { "XY", "YX" }

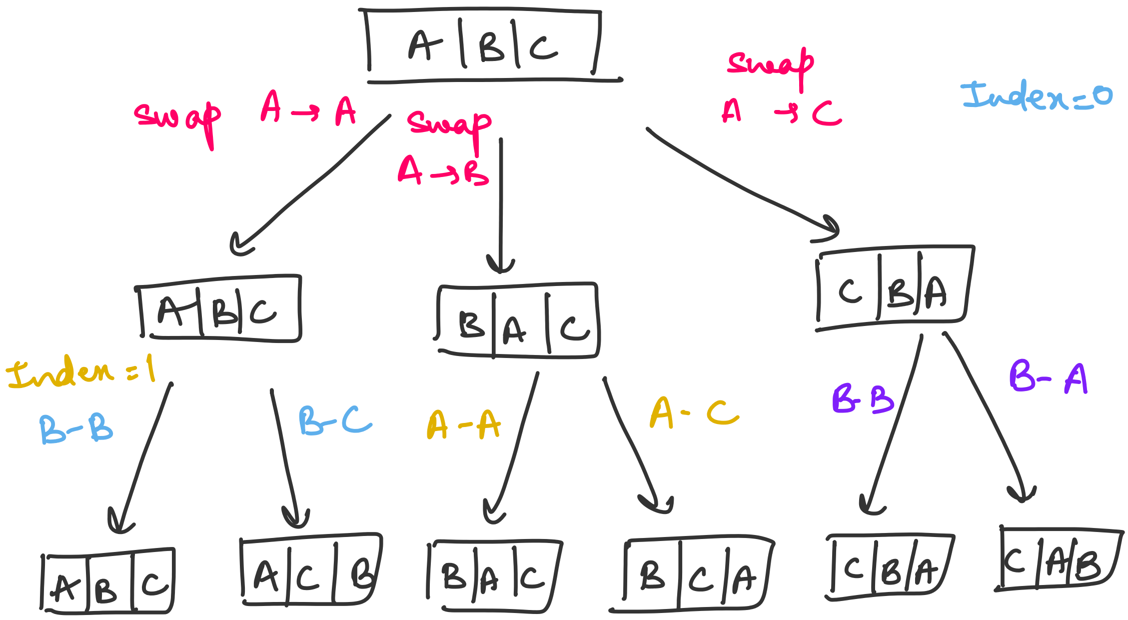
Another eg -

str = " ABC"

Permutations = { "ABC" , "ACB" ,

"BAC" , "BCA" , "CAB" , "CBA" }

For string = ' ABC"



Steps -

1. Create a f^n permute with parameters -

str - input string

l - starting index

r - ending index.

2. Call the permute f^n with values -

str = value of your string

l = 0

r = str.length - 1

3. If values of L and R are equal then print the string.

4. Run a loop from L to R and swap the current element of the string with inputstring [L].

5. call the same permute f^n by incrementing the value of L by 1.

6. After this again swap the previously swapped values to initiate backtracking.

Rat in a maze

↓

Count all the ways to reach the destination in a maze.

↓

Enumeration backtracking

We have a 2D maze where 0 represents clear path (path which we can travel) and −1 represents blockage.

S

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

D

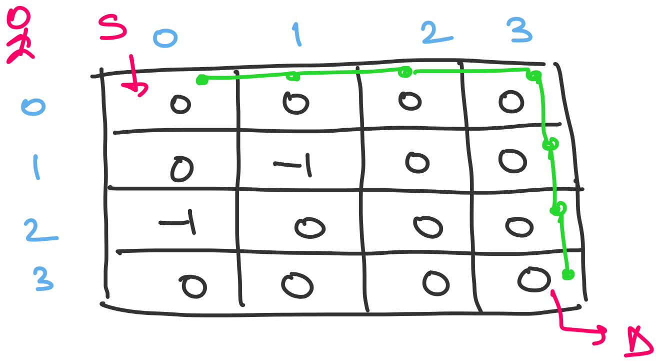
maze

Source – Starting point – maze[0][0]

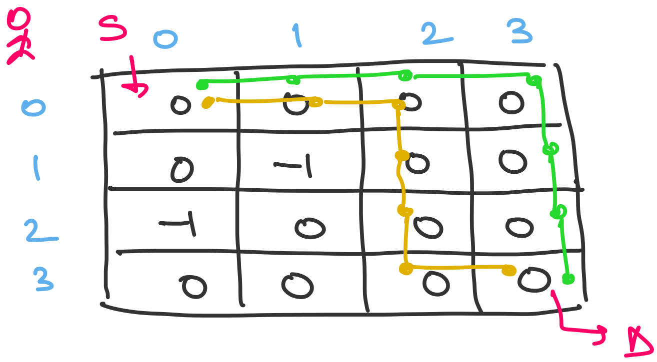
Destination – Exit point – maze[n-1][n-1]

Source – maze[0][0]

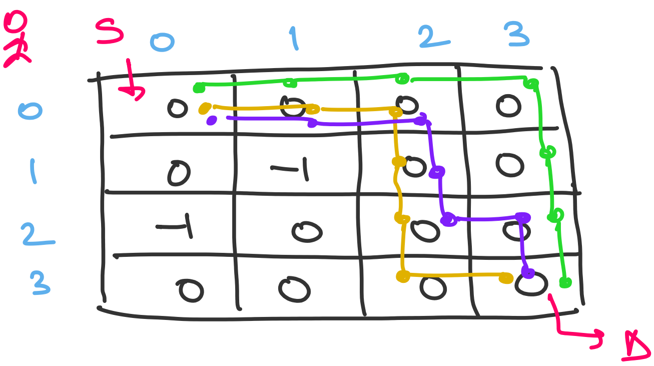
Destination – maze[3][3]



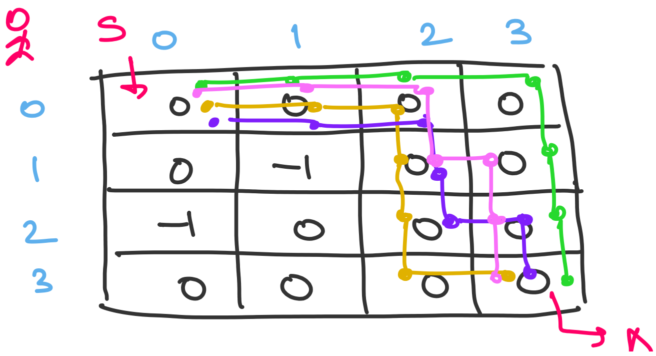
Green



Yellow



Purple



Pink

Total paths = 4
- Green
- Yellow
- Purple
- Pink

S

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

D

we can move in 2 directions
- Right
- Bottom

$(0, 0) \xrightarrow{\text{Right}} (0, 1)$

$(i, j) \longrightarrow (i, j+1)$

$(2, 0) \xrightarrow{\text{Right}} (2, 1)$

$(i, j) \longrightarrow (i, j+1)$

$(1, 2) \xrightarrow{\text{left}} (1, 1)$

$(i, j) \longrightarrow (i, j-1)$

$(1, 0) \xrightarrow{\text{Bottom}} (2, 0)$

$(i, j) \longrightarrow (i+1, j)$

$(0,3) \xrightarrow{\text{Bottom}} (1,3)$

$(i, j) \qquad\qquad (i+1, j)$

$(2,2) \xrightarrow{\text{Top}} (1,2)$

$(i, j) \qquad\qquad (i-1, j)$

we need to take care of 2 things—

1. Increment the count if path is possible in bottom or right direction.
   $(i, j+1) \qquad (i+1, j)$

2. Check for the blockage

$maze[i][j] == -1$

↓

Blockage

↓

we cannot move

if $(maze[i][j] > 0)$

↓

Clear path

Base case

1. Return if source is blocked

if (maze [0][0] == -1)

↓

return

2. If the destination is blocked then return 0.

if (maze [n-1] [n-1] == -1)

Steps —

1. If current cell has a blockage then do not change.

if (maze [i] [j] == -1)

↓

continue

2. If we can move to maze [i][j] from maze [i-1] [j] then increment the count.

```
if (maze[i-1][j] > 0)
{
    maze[i][j] = maze[i][j]
                   + maze[i-1][j]
}
```

3. If we can move to maze[i][j] from maze[i][j-1] then increment the count.

```
if (maze[i][j-1] > 0)
{
    maze[i][j] = maze[i][j]
                   + maze[i][j-1]
}
```

n - Queen

Queen →
- Avoid same row
- Avoid same column
- Avoid keeping them diagonally

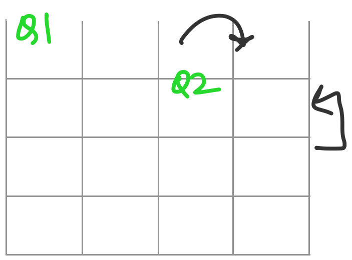
Given a chessboard of size "$n * n$", place n Queens in such a way that no two Queens should attack each other.

$n = 4$

↓

chessboard size $= 4 * 4$

No of Queens $= 4$



Q3 cannot be placed

↓

Backtrack Q2

Q1			
			Q2
	Q3		

Q4 cannot be placed

↓

Backtrack Q3

↓

Backtrack Q2

↓

Backtrack Q1

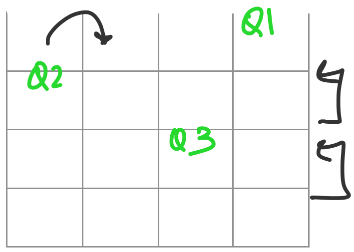
	Q1		
			Q2
Q3			
		Q4	
↓	↓	↓	↓
Q3	Q1	Q4	Q2

Solⁿ – Q3, Q1, Q4, Q2

Another solⁿ –

		Q1	
Q2			
			Q3
	Q4		
↓	↓	↓	↓
Q2	Q4	Q1	Q3

Solⁿ – Q2, Q4, Q1, Q3



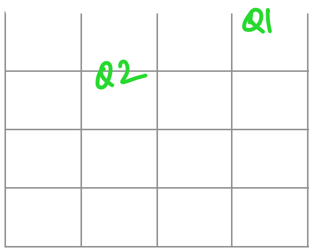
Q4 cannot be placed

↓

Backtrack Q3

↓

Backtrack Q2



Q3 cannot be placed

↓

Q2 cannot be backtracked

So for $n=4$, we have 2 sol^{ns} possible –

$$\left\{ \begin{array}{l} Q3,\ Q1,\ Q4,\ Q2 \\ Q2,\ Q4,\ Q1,\ Q3 \end{array} \right\}$$

n = 8

Chessboard size = 8 * 8

No of queens = 8

Q1							
		Q2					
				Q3			
						Q4	
	Q5						
			Q6				
					Q7		

Q8 cannot be placed

↓

Backtrack Q7

Algorithm for n - Queen

1. Initialize a 2-D array of size n*n.
2. Start with the leftmost column and place a queen in the first row of that column.

3. Move to the next column and place queen in the first row of that column.

4. Repeat step 3 until all queens have been placed or it is impossible to place a queen in the current column without violating the rules.

5. If n queens have been placed then print the solution.

6. If it is not possible to place all the n queens without violating the rules then we will backtrack.

7. Remove the queen from the previous column and move it to another row.

8. Repeat steps 4-7 untill all possible configurations have been explored.

3 methods —
- print Solution → void
- is safe → boolean
- solve n Queen → boolean

Solve Sudoku

Given a partially filled 2 D grid of size 9 * 9 , the goal is to assign numbers (1-9) to the empty in such a way that the instance of the number should be exactly once in the row, column and the subgrid.

	0	1	2	3	4	5	6	7	8
0	3		6	5		8	4		
1	5	2							
2	4	8	7					3	1
3			3		1			8	
4	9			8	6	3			5
5		5			9		6		
6	1	3					2	5	
7								7	4
8			5	2		6	3		

[1, 2, 3, 4, 5, 6, 7, 8, 9]

$\underbrace{2, 3, 5, 6, 7, 8}$

↓

1, 4, 9

Algorithm steps –

1. Create a function to check that after assigning a value to the empty cell will the grid becomes safe or not.

↓

For this we can use either a

hashmap or loops.

↓

If a number has a frequency greater than 1 then return false otherwise return true.

2. create a recursive function that operates on the given grid.

3. check for unassigned location –

(i) If present then assign a number from 1 to 9.

(ii) Check if after assigning the number if grid becomes safe or not.

(iii) If safe then call recursive call for all the safe cases from 0 to 9.

(iv) If any of the recursive call returns true then end the loop and return true.

If no recursive call returns true then return false.

3. If no unassigned location is left then return true.