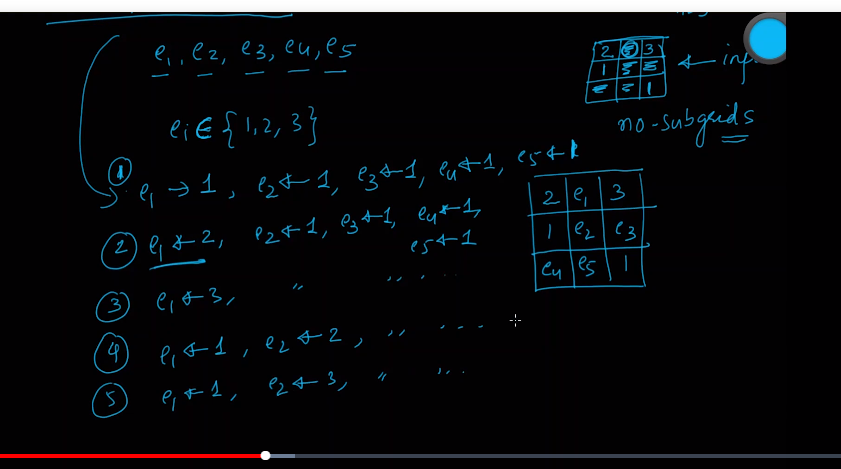
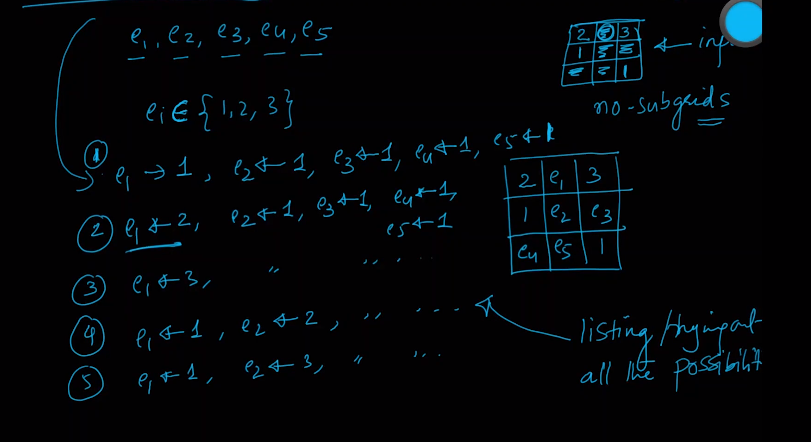
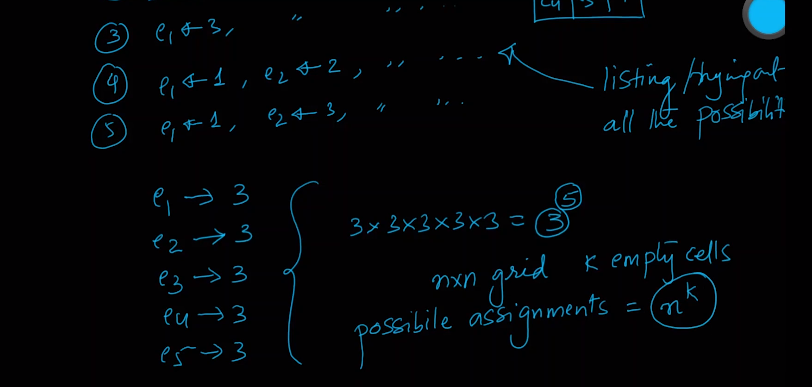


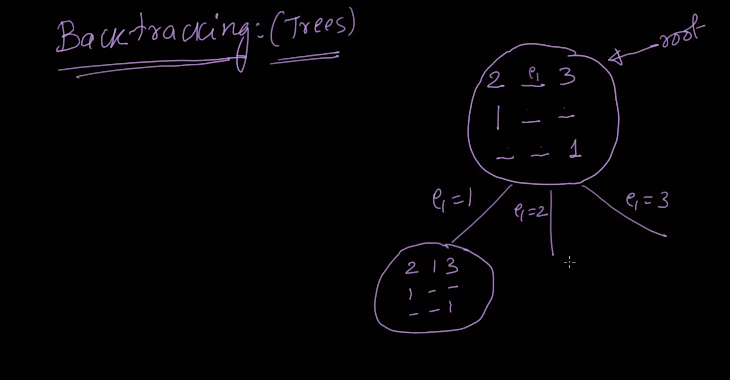
Keep listing all the possibilities.

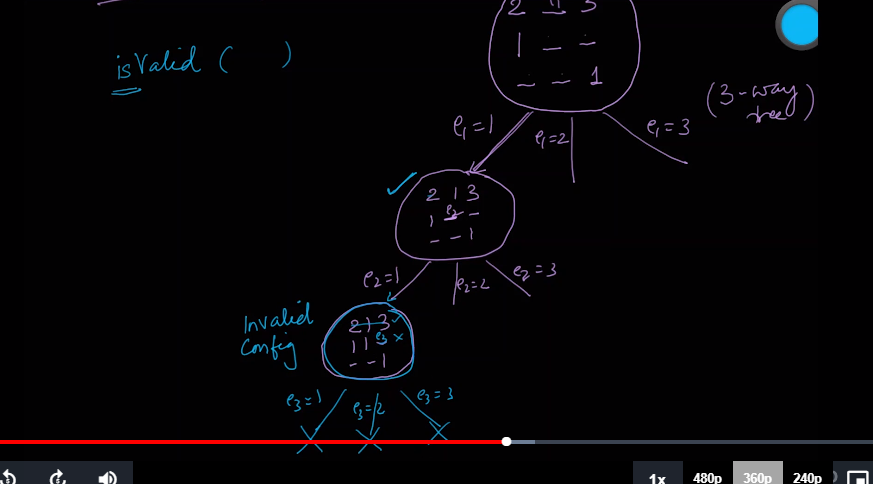




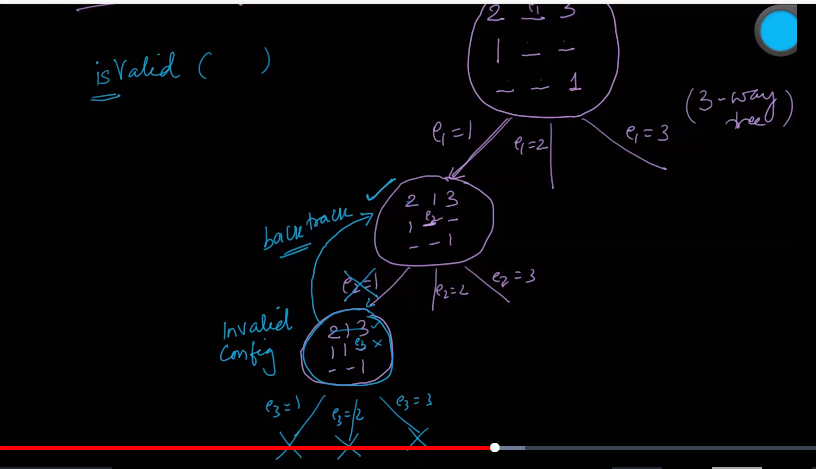


it is massive. So we will use a concept called Backtracking which uses Tress internally.

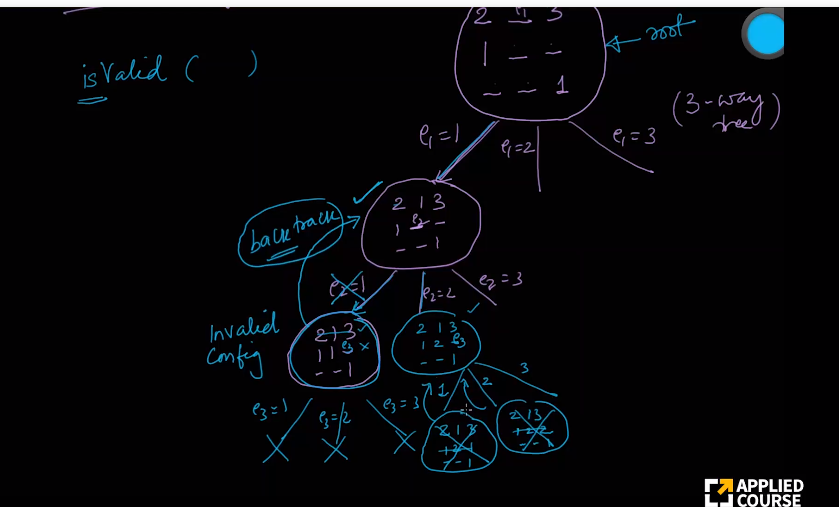


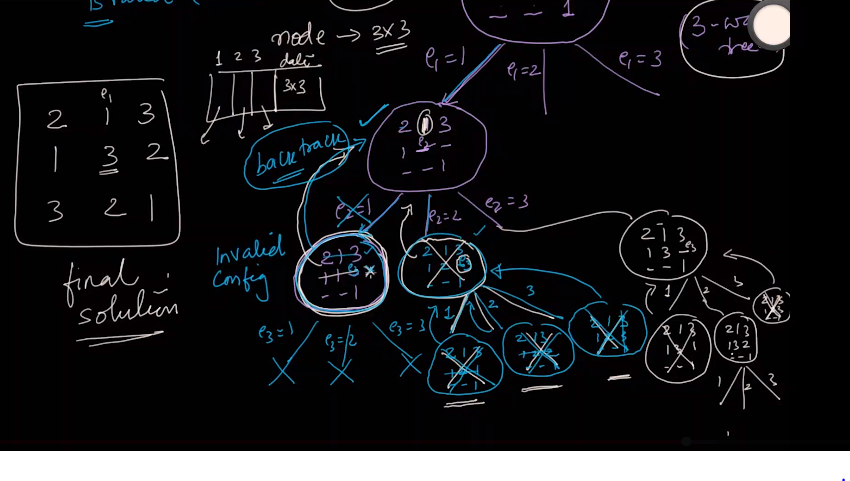


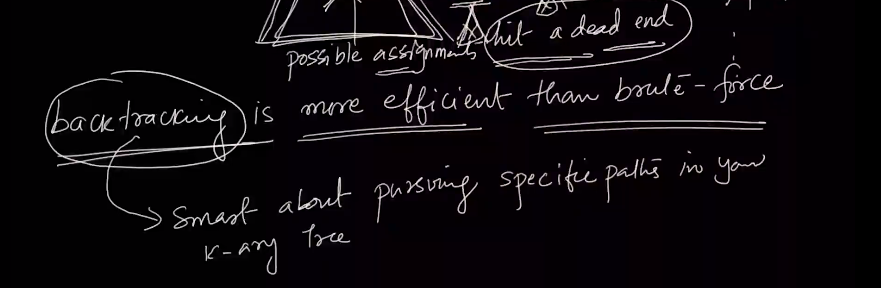
If is\_valid() function is False, there is no point of going down below.

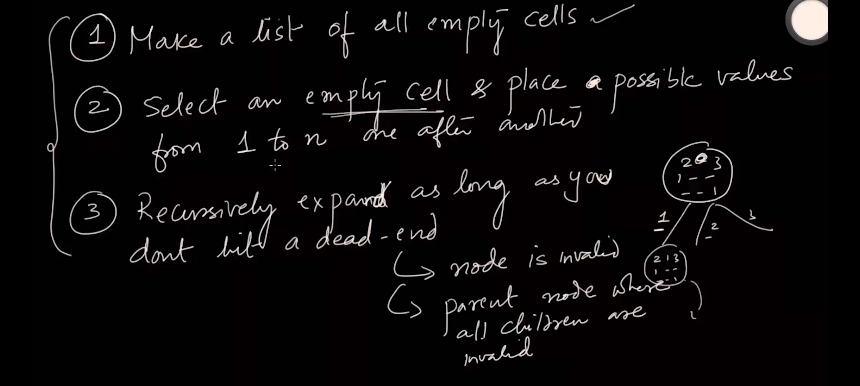


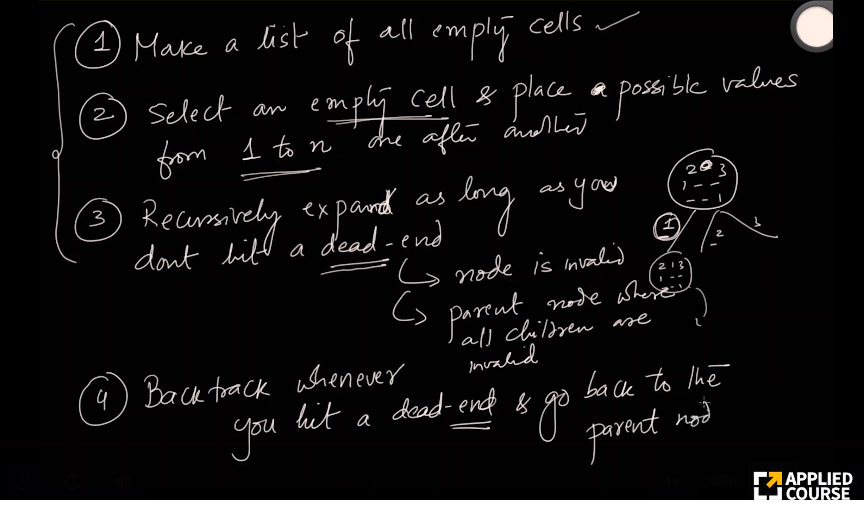
Then we need to go back to the parent and tell them that this is not right.











In part 1 of this **Sudoku solver with python tutorial** I explain how we are going to go about solving the problem and discuss the algorithm known as **backtracking**. Backtracking is simply reverting back to the previous step or solution as soon as we determine that our current solution cannot be continued into a complete one. We will use this principle of backtracking to implement the following algorithm.

***Algorithm***

Starting with an incomplete board:

1. Find some empty space
2. Attempt to place the digits 1-9 in that space
3. Check if that digit is valid in the current spot based on the current board
4. a. If the digit is valid, recursively attempt to fill the board using steps 1-3.  
   b. If it is not valid, reset the square you just filled and go back to the previous step.
5. Once the board is full by the definition of this algorithm we have found a solution.

We will finish about half of the algorithm in part 1. In part 2 (look below) we will implement the entire algorithm.

board **=** **[**

**[7,8,0,4,0,0,1,2,0],**

**[6,0,0,0,7,5,0,0,9],**

**[0,0,0,6,0,1,0,7,8],**

**[0,0,7,0,4,0,2,6,0],**

**[0,0,1,0,5,0,9,3,0],**

**[9,0,4,0,6,0,0,0,5],**

**[0,7,0,3,0,0,0,1,2],**

**[1,2,0,0,0,7,4,0,0],**

**[0,4,9,2,0,6,0,0,7]**

**]**

**def** print\_board**(**bo**):**

**for** i **in** range**(**len**(**bo**)):**

**if** i **%** **3** **==** **0** **and** i **!=** **0:**

**print(**"- - - - - - - - - - - - - "**)**

**for** j **in** range**(**len**(**bo**[0])):**

**if** j **%** **3** **==** **0** **and** j **!=** **0:**

**print(**" | "**,** end**=**""**)**

**if** j **==** **8:**

**print(**bo**[**i**][**j**])**

**else:**

**print(**str**(**bo**[**i**][**j**])** **+** " "**,** end**=**""**)**

**def** find\_empty**(**bo**):**

**for** i **in** range**(**len**(**bo**)):**

**for** j **in** range**(**len**(**bo**[0])):**

**if** bo**[**i**][**j**]** **==** **0:**

**return** **(**i**,** j**)** *# row, col*

**return** None

board **=** **[**

**[7,8,0,4,0,0,1,2,0],**

**[6,0,0,0,7,5,0,0,9],**

**[0,0,0,6,0,1,0,7,8],**

**[0,0,7,0,4,0,2,6,0],**

**[0,0,1,0,5,0,9,3,0],**

**[9,0,4,0,6,0,0,0,5],**

**[0,7,0,3,0,0,0,1,2],**

**[1,2,0,0,0,7,4,0,0],**

**[0,4,9,2,0,6,0,0,7]**

**]**

**def** solve**(**bo**):**

find **=** find\_empty**(**bo**)**

**if** **not** find**:**

**return** True

**else:**

row**,** col **=** find

**for** i **in** range**(1,10):**

**if** valid**(**bo**,** i**,** **(**row**,** col**)):**

bo**[**row**][**col**]** **=** i

**if** solve**(**bo**):**

**return** True

bo**[**row**][**col**]** **=** **0**

**return** False

**def** valid**(**bo**,** num**,** pos**):**

*# Check row*

**for** i **in** range**(**len**(**bo**[0])):**

**if** bo**[**pos**[0]][**i**]** **==** num **and** pos**[1]** **!=** i**:**

**return** False

*# Check column*

**for** i **in** range**(**len**(**bo**)):**

**if** bo**[**i**][**pos**[1]]** **==** num **and** pos**[0]** **!=** i**:**

**return** False

*# Check box*

box\_x **=** pos**[1]** **//** **3**

box\_y **=** pos**[0]** **//** **3**

**for** i **in** range**(**box\_y**\*3,** box\_y**\*3** **+** **3):**

**for** j **in** range**(**box\_x **\*** **3,** box\_x**\*3** **+** **3):**

**if** bo**[**i**][**j**]** **==** num **and** **(**i**,**j**)** **!=** pos**:**

**return** False

**return** True

**def** print\_board**(**bo**):**

**for** i **in** range**(**len**(**bo**)):**

**if** i **%** **3** **==** **0** **and** i **!=** **0:**

**print(**"- - - - - - - - - - - - - "**)**

**for** j **in** range**(**len**(**bo**[0])):**

**if** j **%** **3** **==** **0** **and** j **!=** **0:**

**print(**" | "**,** end**=**""**)**

**if** j **==** **8:**

**print(**bo**[**i**][**j**])**

**else:**

**print(**str**(**bo**[**i**][**j**])** **+** " "**,** end**=**""**)**

**def** find\_empty**(**bo**):**

**for** i **in** range**(**len**(**bo**)):**

**for** j **in** range**(**len**(**bo**[0])):**

**if** bo**[**i**][**j**]** **==** **0:**

**return** **(**i**,** j**)** *# row, col*

**return** None

print\_board**(**board**)**

solve**(**board**)**

**print(**"\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_"**)**

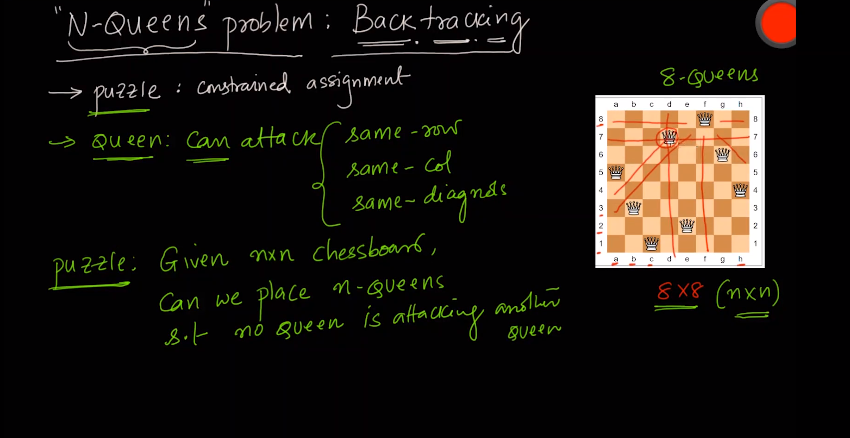
print\_board**(**board**)**

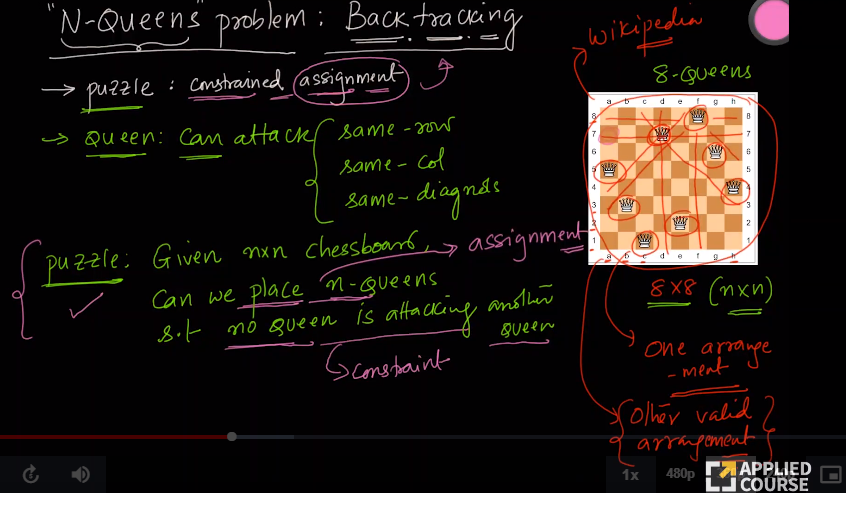
Full Program:-

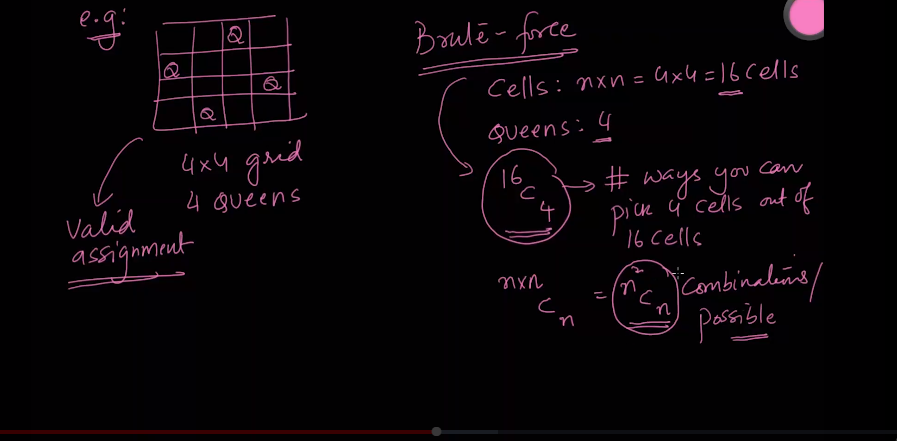
https://www.techwithtim.net/tutorials/python-programming/sudoku-solver-backtracking/

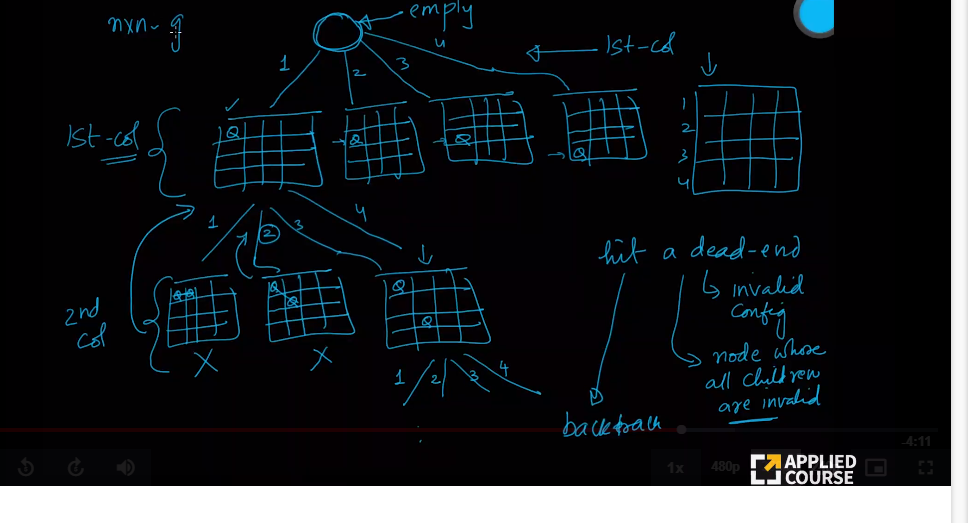
**Write a Program for N Queen Problem**

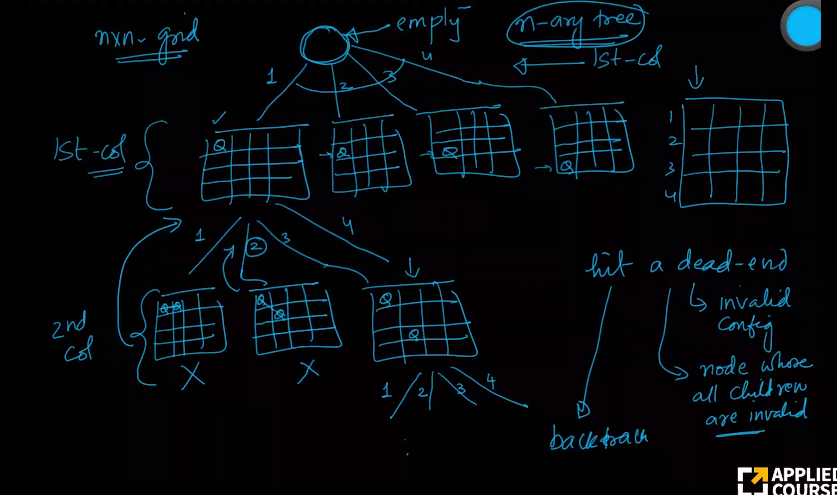


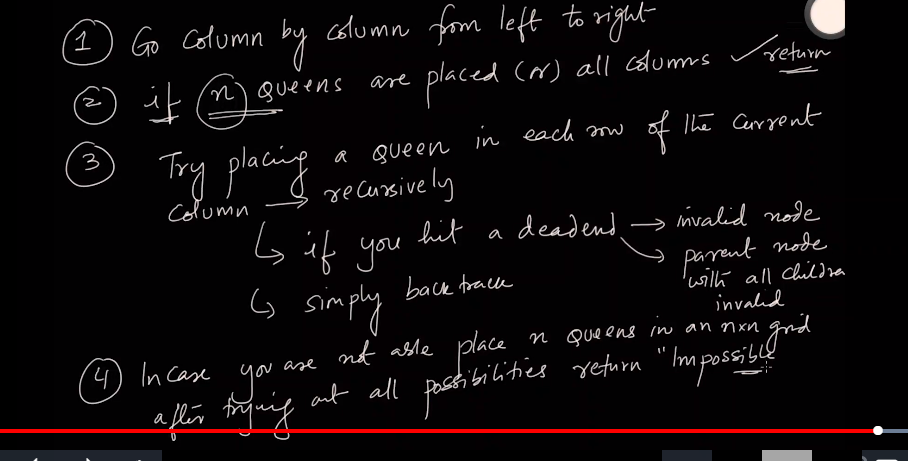






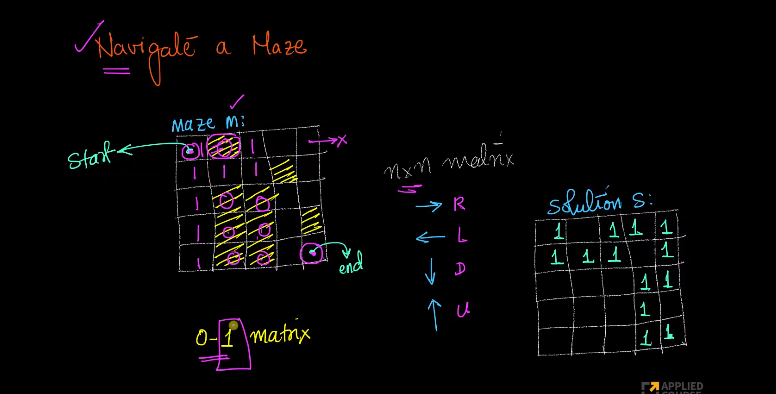


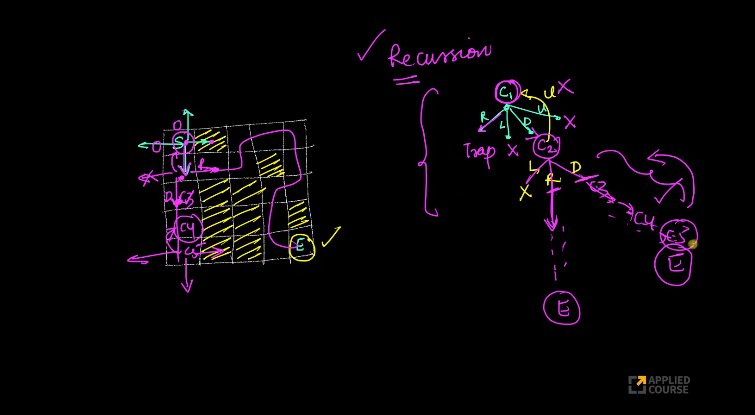


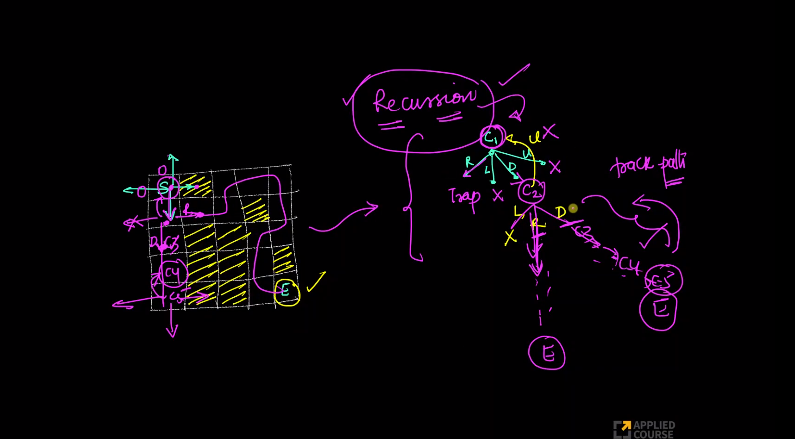


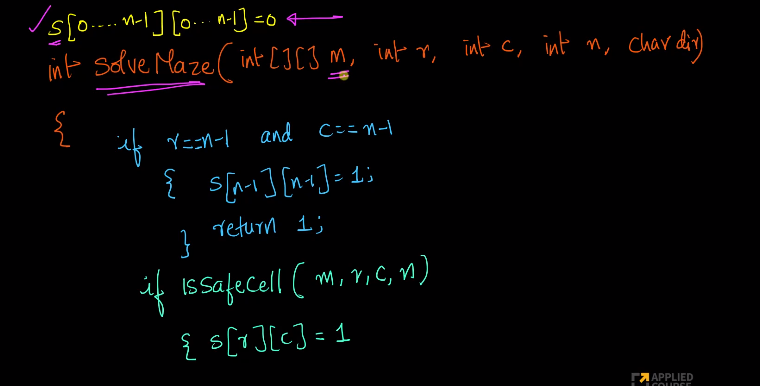
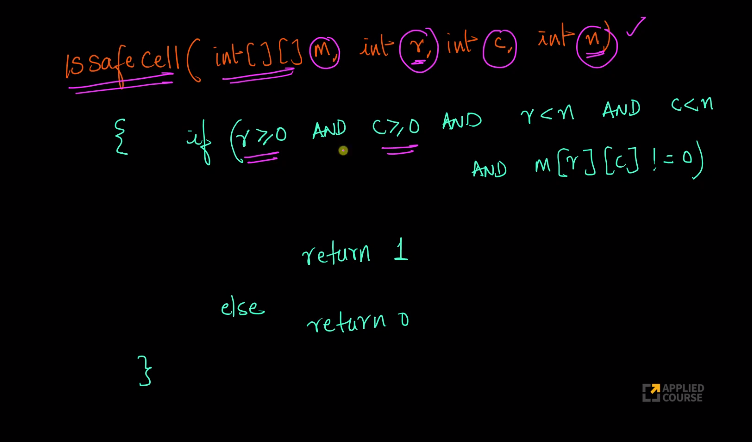
1. **class** Solution:
2. **def** solveNQueens(self, n):
4. puzzle = [["."\*n] **for** i **in** range(n)]
5. cols, diagonals, anti\_diagonals, result = set(), set(), set(), []
7. **def** place\_queen(row, puzzle):
9. **if**(row == n):
10. result.append(puzzle[:])
11. **return**
13. **for** col **in** range(n):
14. *# GIST, we can uniquely identify diagonals and antidiagonals by row-col and row+col*
15. **if**(col **in** cols **or** (row-col) **in** diagonals **or** (row + col) **in** anti\_diagonals):
16. **continue**
18. cols.add(col)
19. diagonals.add(row - col)
20. anti\_diagonals.add(row + col)
22. puzzle[row] = "." \* (col) + "Q" + "." \* (n-col-1)
24. place\_queen( row + 1, puzzle )
26. cols.remove(col)
27. diagonals.remove(row - col)
28. anti\_diagonals.remove(row + col)
30. puzzle[row] = "." \* n
32. place\_queen( 0, puzzle )
33. **return** result

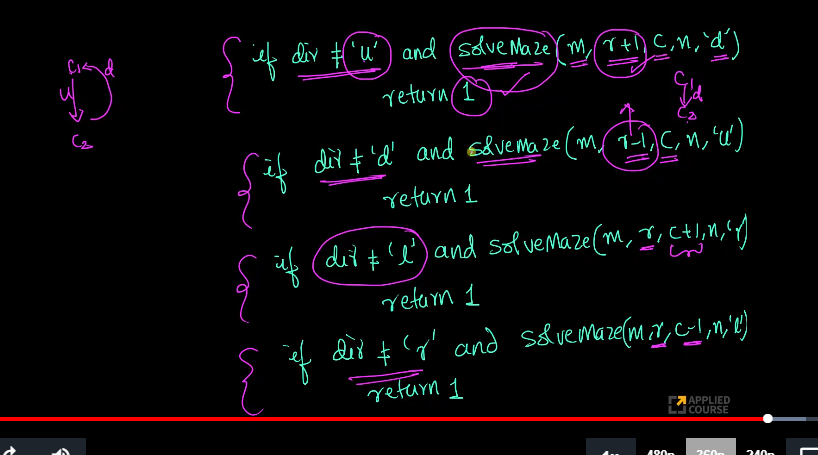
# Write a Program for Rat in a Maze











#Rat in Maze

#Backtracking

def is\_safe(maze, row, col, n) :

if row >= 0 and col >= 0 :

if row < n and col < n :

if maze[row][col] != 0 :

return True

else :

return False

def display(maze) :

for row in maze :

print(row)

def solveMaze(maze, row, col, n, direction, sol) :

if row == n-1 and col == n-1 :

sol[row][col] = 1

return True

if is\_safe(maze, row, col, n) :

maze[row][col] = 1

sol[row][col] = 1

#Going down

if direction != 'u' and solveMaze(maze, row+1, col, n, 'd', sol) :

return True

#Going up

if direction != 'd' and solveMaze(maze, row-1, col, n, 'u', sol) :

return True

#Going left

if direction != 'r' and solveMaze(maze, row, col-1, n, 'l', sol) :

return True

#Going right

if direction != 'l' and solveMaze(maze, row, col+1, n, 'r', sol) :

return True

#If all children of current node are invalid, then backtrack

maze[row][col] = 0

sol[row][col] = 0

return False

return False

if \_\_name\_\_ == '\_\_main\_\_' :

maze = [ [1, 0, 0, 0],

[1, 1, 0, 1],

[0, 1, 0, 0],

[1, 1, 1, 1] ]

n = len(maze)

sol = [[0 for i in range(n)] for j in range(n)]

if solveMaze(maze, 0, 0, n, 'r', sol) :

display(sol)

elif solveMaze(maze, 0, 0, n, 'd', sol) :

display(sol)

else :

print("No possible solution")

# Letter Combinations of a phone number: Problem Statement [Leetcode]

**Problem Statement:**

Given a string containing digits from 2-9 inclusive, return all possible letter combinations that the number could represent.

A mapping of digit to letters (just like on the telephone buttons) is given below. Note that 1 does not map to any letters.



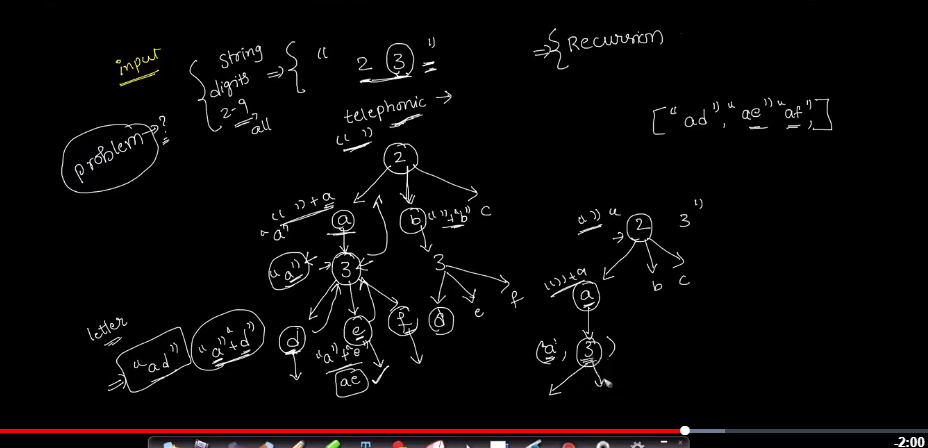
**Example:**

**Input:** "23"

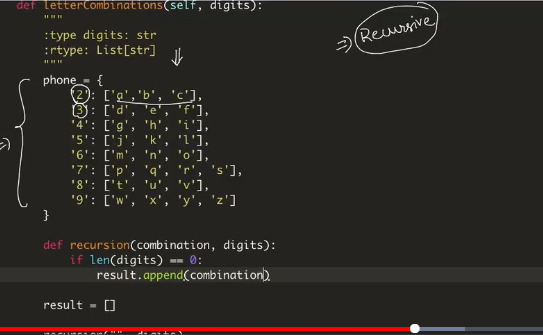
**Output:** ["ad", "ae", "af", "bd", "be", "bf", "cd", "ce", "cf"].

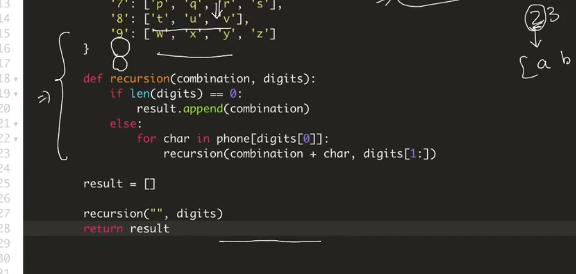
**Note:**

Although the above answer is in lexicographical order, your answer could be in any order you want.









# Permutations: Problem Statement [Leetcode]

**Problem Statement:**

Given a collection of **distinct** integers, return all possible permutations.

**Example:**

**Input:** [1,2,3]

**Output:**

[

[1,2,3],

[1,3,2],

[2,1,3],

[2,3,1],

[3,1,2],

[3,2,1]

]

**Level: Mediium**

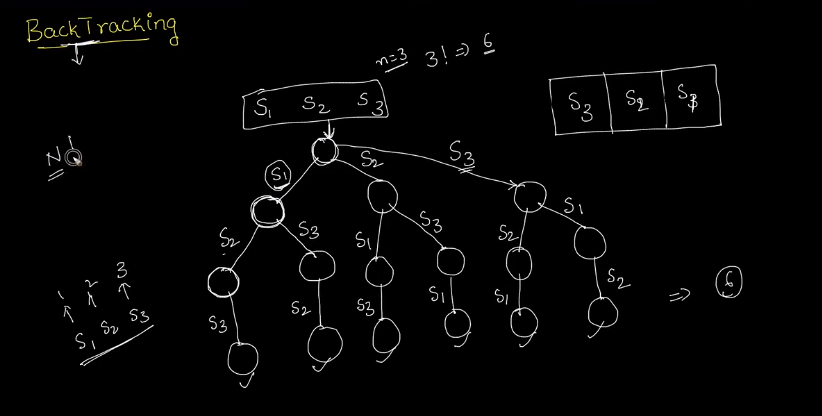
**Problem Practice link:** <https://leetcode.com/problems/permutations/>

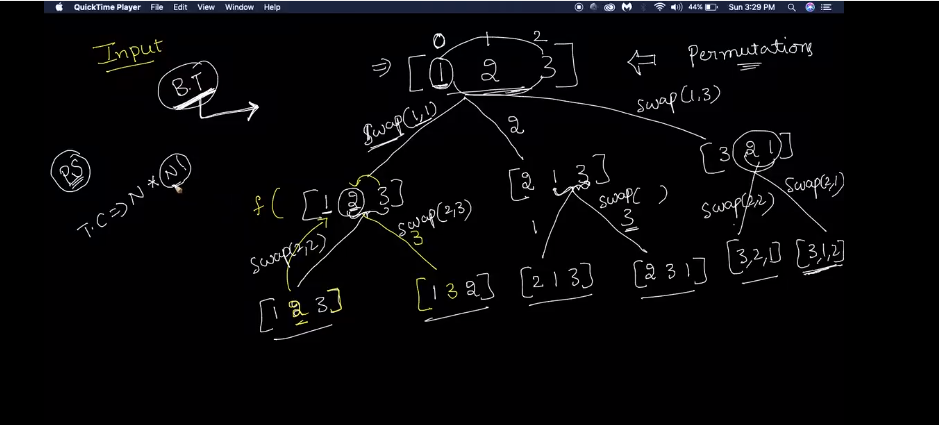
**Similar Problems:**

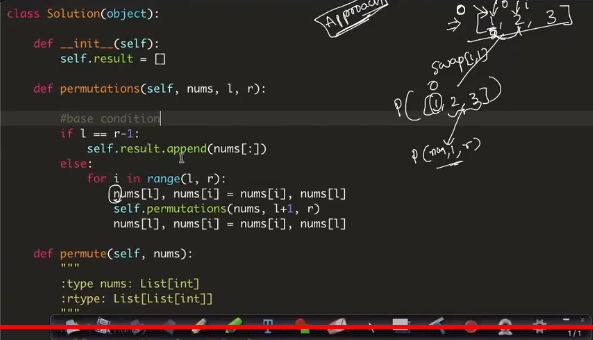
1. Next Permutation: <https://leetcode.com/problems/next-permutation/>

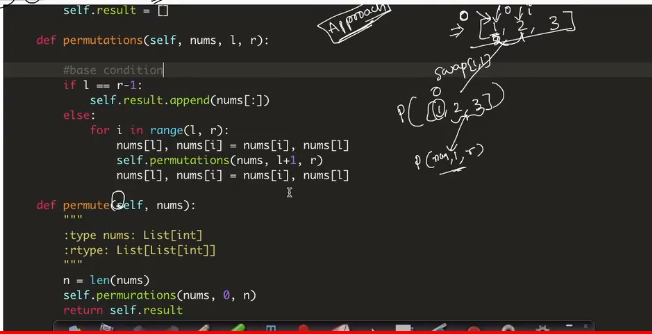
2. Permutation Sequence: <https://leetcode.com/problems/permutation-sequence/>

3. Combinations: <https://leetcode.com/problems/combinations/>









# Word Search: Problem Statement [leetcode]

**Problem Statement:**

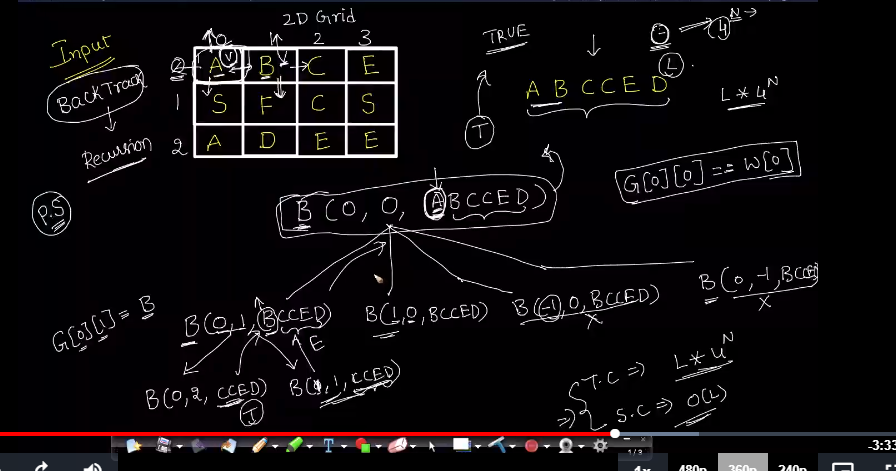
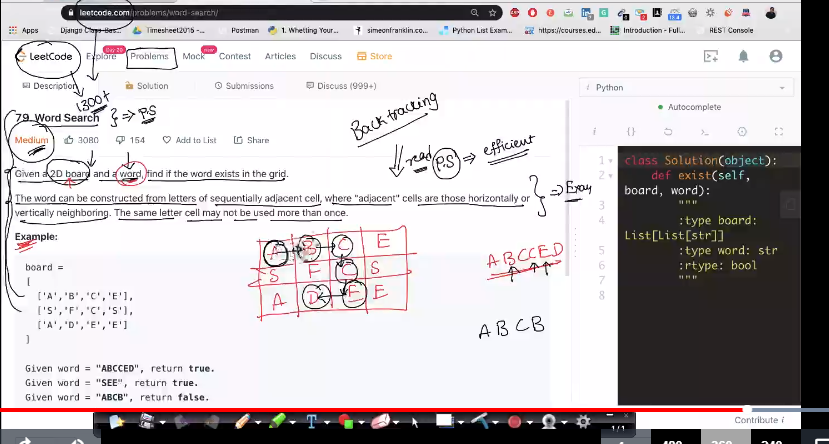
Given a 2D board and a word, find if the word exists in the grid.  
  
The word can be constructed from letters of sequentially adjacent cell, where "adjacent" cells are those horizontally or vertically neighboring. The same letter cell may not be used more than once.  
  
**Example:**  
  
board = [ ['A','B','C','E'], ['S','F','C','S'], ['A','D','E','E'] ]  
  
Given word = "**ABCCED**", return **true**.  
  
Given word = "**SEE**", return **true**.  
  
Given word = "**ABCB**", return **false**.  
  
**Constraints:**

* board and word consists only of lowercase and uppercase English letters.

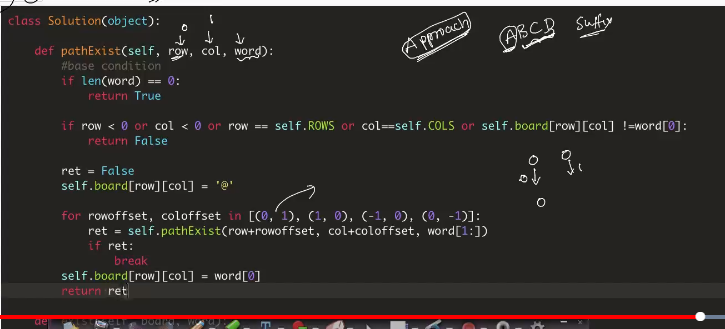
* 1 <= board.length <= 200

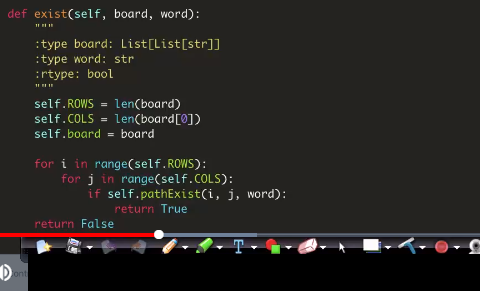
* 1 <= board[i].length <= 200

* 1 <= word.length <= 10^3









# Generate Parenthesis: Problem Statement [Leetcode]

**Problem Statement:**

Given *n* pairs of parentheses, write a function to generate all combinations of well-formed parentheses.  
  
For example, given *n* = 3, a solution set is:  
  
[  
  
"((()))",  
  
"(()())",  
  
"(())()",  
  
"()(())",  
  
"()()()" ]  
  
**Level: Medium  
  
Problem Practice Link:**<https://leetcode.com/problems/generate-parentheses/>  
  
**Similar Problems:**  
  
Valid Parenthesis: <https://leetcode.com/problems/valid-parentheses/>

