DAY-10

1) Discuss the importance of visualizing the solutions of the N-Queens Problem to understand.the placement of queens better. Use a graphical representation to show how queens are,placed on the board for different values of N. Explain how visual tools can help in debugging,the algorithm and gaining insights into the problem's complexity. Provide examples of visual representations for N = 4, N = 5, and N = 8, showing different valid solutions.

a. Visualization for 4-Queens:Input: N = 4

CODE:

def print\_board(board):  
 for row in board:  
 print(" ".join(row))   
 print()  
def solve\_n\_queens(n):  
 board = [["." for \_ in range(n)] for \_ in range(n)]  
 results = []  
 solve(board, 0, results)  
 return results  
def solve(board, col, results):  
 if col >= len(board):  
 results.append(["".join(row) for row in board])  
 return  
 for i in range(len(board)):  
 if is\_safe(board, i, col):  
 board[i][col] = 'Q'  
 solve(board, col + 1, results)  
 board[i][col] = '.'  
def is\_safe(board, row, col):  
 for i in range(col):  
 if board[row][i] == 'Q':  
 return False

for i, j in zip(range(row, -1, -1), range(col, -1, -1)):  
 if board[i][j] == 'Q':  
 return False  
for i, j in zip(range(row, len(board)), range(col, -1, -1)):  
 if board[i][j] == 'Q':  
 return False  
 return True

n = 4  
solutions = solve\_n\_queens(n)  
for sol in solutions:  
 print\_board(sol)

2)Discuss the generalization of the N-Queens Problem to other board sizes and shapes, such as

rectangular boards or boards with obstacles. Explain how the algorithm can be adapted to

handle these variations and the additional constraints they introduce. Provide examples of

solving generalized N-Queens Problems for different board configurations, such as an 8×10

board, a 5×5 board with obstacles, and a 6×6 board with restricted positions.

a. 8×10 Board:

8 rows and 10 columns

Output: Possible solution [1, 3, 5, 7, 9, 2, 4, 6]

CODE:

def is\_safe(board, row, col, n\_rows, n\_cols):

for i in range(row):

if board[i] == col or abs(board[i] - col) == abs(i - row):

return False

return True

def solve\_n\_queens(board, row, n\_rows, n\_cols):

if row == n\_rows: # All queens are placed

return True

for col in range(n\_cols):

if is\_safe(board, row, col, n\_rows, n\_cols):

board[row] = col

if solve\_n\_queens(board, row + 1, n\_rows, n\_cols):

return True

board[row] = -1

return False

def n\_queens\_rectangular(n\_rows, n\_cols):

board = [-1] \* n\_rows # Initialize board

if solve\_n\_queens(board, 0, n\_rows, n\_cols):

return board

else:

return "No solution"

result = n\_queens\_rectangular(8, 10)

print(result)

OUTPUT:

[0,2,4,1,7,9,3,6]

3) Write a program to solve a Sudoku puzzle by filling the empty cells.A sudoku solution must

satisfy all of the following rules:Each of the digits 1-9 must occur exactly once in each

row.Each of the digits 1-9 must occur exactly once in each column.Each of the digits 1-9

must occur exactly once in each of the 9 3x3 sub-boxes of the grid.The '.' character

indicates empty cells.

Example 1:

Input: board =

[["5","3",".",".","7",".",".",".","."],

["6",".",".","1","9","5",".",".","."],

[".","9","8",".",".",".",".","6","."],

["8",".",".",".","6",".",".",".","3"],

["4",".",".","8",".","3",".",".","1"],

["7",".",".",".","2",".",".",".","6"],

[".","6",".",".",".",".","2","8","."],

[".",".",".","4","1","9",".",".","5"],

[".",".",".",".","8",".",".","7","9"]]

Output:

[["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","6","1","7","9"]]

CODE:

def is\_valid(board, row, col, num):

# Check if 'num' is not in the current row, column, and 3x3 sub-box

for i in range(9):

if board[row][i] == num or board[i][col] == num or board[3 \* (row // 3) + i // 3][3 \* (col // 3) + i % 3] == num:

return False

return True

def solve\_sudoku(board):

for row in range(9):

for col in range(9):

if board[row][col] == '.':

for num in map(str, range(1, 10)):

if is\_valid(board, row, col, num):

board[row][col] = num

if solve\_sudoku(board):

return True

board[row][col] = '.'

return False

return True

board = [

["5","3",".",".","7",".",".",".","."],

["6",".",".","1","9","5",".",".","."],

[".","9","8",".",".",".",".","6","."],

["8",".",".",".","6",".",".",".","3"],

["4",".",".","8",".","3",".",".","1"],

["7",".",".",".","2",".",".",".","6"],

[".","6",".",".",".",".","2","8","."],

[".",".",".","4","1","9",".",".","5"],

[".",".",".",".","8",".",".","7","9"]

]

solve\_sudoku(board)

for row in board:

print(row)

OUTPUT:

[["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","6","1","7","9"]]

4) Write a program to solve a Sudoku puzzle by filling the empty cells.A sudoku solution must

satisfy all of the following rules:Each of the digits 1-9 must occur exactly once in each

row.Each of the digits 1-9 must occur exactly once in each column.Each of the digits 1-9

must occur exactly once in each of the 9 3x3 sub-boxes of the grid.The '.' character indicates

empty cells.

Example 1:

Input: board =

[["5","3",".",".","7",".",".",".","."],

["6",".",".","1","9","5",".",".","."],

[".","9","8",".",".",".",".","6","."],

["8",".",".",".","6",".",".",".","3"],

["4",".",".","8",".","3",".",".","1"],

["7",".",".",".","2",".",".",".","6"],

[".","6",".",".",".",".","2","8","."],

[".",".",".","4","1","9",".",".","5"],

[".",".",".",".","8",".",".","7","9"]]

Output:

[["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","6","1","7","9"]]

CODE:

def is\_valid(board, row, col, num):

for i in range(9):

if board[row][i] == num or board[i][col] == num or board[3 \* (row // 3) + i // 3][3 \* (col // 3) + i % 3] == num:

return False

return True

def solve\_sudoku(board):

for row in range(9):

for col in range(9):

if board[row][col] == '.':

for num in map(str, range(1, 10)):

if is\_valid(board, row, col, num):

board[row][col] = num

if solve\_sudoku(board):

return True

board[row][col] = '.'

return False

return True

board = [

["5","3",".",".","7",".",".",".","."],

["6",".",".","1","9","5",".",".","."],

[".","9","8",".",".",".",".","6","."],

["8",".",".",".","6",".",".",".","3"],

["4",".",".","8",".","3",".",".","1"],

["7",".",".",".","2",".",".",".","6"],

[".","6",".",".",".",".","2","8","."],

[".",".",".","4","1","9",".",".","5"],

[".",".",".",".","8",".",".","7","9"]

]

solve\_sudoku(board)

for row in board:

print(row)

OUTPUT:

[["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","6","1","7","9"]]

5) You are given an integer array nums and an integer target. You want to build an expression

out of nums by adding one of the symbols '+' and '-' before each integer in nums and then

concatenate all the integers.For example, if nums = [2, 1], you can add a '+' before 2 and a '-'

before 1 and concatenate them to build the expression "+2-1" Return the number of different

expressions that you can build, which evaluates to target.

Example 1:

Input: nums = [1,1,1,1,1], target = 3

Output: 5

CODE:

def find\_target\_sum\_ways(nums, target):

memo = {} # To store already computed states

def backtrack(index, current\_sum):

if index == len(nums):

return 1 if current\_sum == target else 0

if (index, current\_sum) in memo:

return memo[(index, current\_sum)]

add = backtrack(index + 1, current\_sum + nums[index])

subtract = backtrack(index + 1, current\_sum - nums[index])

memo[(index, current\_sum)] = add + subtract

return memo[(index, current\_sum)]

return backtrack(0, 0)

OUPUT:

5

6) Given an array of integers arr, find the sum of min(b), where b ranges over every

(contiguous) subarray of arr. Since the answer may be large, return the answer modulo 109 +

CODE:

def sum\_subarray\_mins(arr):

MOD = 10\*\*9 + 7

n = len(arr)

prev\_smaller = [-1] \* n

next\_smaller = [n] \* n

stack = []

for i in range(n):

while stack and arr[stack[-1]] >= arr[i]:

stack.pop()

if stack:

prev\_smaller[i] = stack[-1]

stack.append(i)

stack = []

for i in range(n):

while stack and arr[stack[-1]] > arr[i]:

index = stack.pop()

next\_smaller[index] = i

stack.append(i)

result = 0

for i in range(n):

left = i - prev\_smaller[i]

right = next\_smaller[i] - i

result = (result + arr[i] \* left \* right) % MOD

return result

OUTPUT:

17

7) Given an array of distinct integers candidates and a target integer target, return a list of all

unique combinations of candidates where the chosen numbers sum to target. You may return

the combinations in any order.The same number may be chosen from candidates an unlimited

number of times. Two combinations are unique if the frequency of at least one of the chosen

numbers is different.The test cases are generated such that the number of unique

combinations that sum up to target is less than 150 combinations for the given input.

Example 1:

Input: candidates = [2,3,6,7], target = 7

Output: [[2,2,3],[7]]

CODE:

def combinationSum(candidates, target):

result = []

def backtrack(remaining, combination, start):

if remaining == 0:

result.append(list(combination))

return

elif remaining < 0:

return

for i in range(start, len(candidates)):

combination.append(candidates[i])

backtrack(remaining - candidates[i], combination, i) # Can reuse the same element

combination.pop()

backtrack(target, [], 0)

return result

OUTPUT:

[[2, 2, 3], [7]]

8) Given a collection of candidate numbers (candidates) and a target number (target), find all

unique combinations in candidates where the candidate numbers sum to target. Each number

in candidates may only be used once in the combination. The solution set must not contain

duplicate combinations.

Example 1:

Input: candidates = [10,1,2,7,6,1,5], target = 8

Output:

[

[1,1,6],

[1,2,5],

[1,7],

[2,6]

CODE:

def combinationSum2(candidates, target):

result = []

candidates.sort() # Sort to handle duplicates

def backtrack(remaining, combination, start):

if remaining == 0:

result.append(list(combination))

return

elif remaining < 0:

return

for i in range(start, len(candidates)):

if i > start and candidates[i] == candidates[i - 1]:

continue

combination.append(candidates[i]

backtrack(remaining - candidates[i], combination, i + 1

combination.pop()

backtrack(target, [], 0)

return result

OUTPUT:

[

[1, 1, 6],

[1, 2, 5],

[1, 7],

[2, 6]

]

9) Given an array nums of distinct integers, return all the possible permutations. You can return

the answer in any order.

Example 1:

Input: nums = [1,2,3]

Output: [[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]

CODE:

def permute(nums):

def backtrack(start):

if start == len(nums):

# All numbers are used, add the current permutation to the result

result.append(nums[:])

return

for i in range(start, len(nums)):

nums[start], nums[i] = nums[i], nums[start]

backtrack(start + 1)

nums[start], nums[i] = nums[i], nums[start]

result = []

backtrack(0) # Start with the first index

return result

nums = [1, 2, 3]

print(permute(nums))

OUPUT:

[[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]

10) 10. Given a collection of numbers, nums, that might contain duplicates, return all possible unique

permutations in any order.

Example 1:

Input: nums = [1,1,2]

Output:

[[1,1,2],

[1,2,1],

[2,1,1]]

CODE:

def permuteUnique(nums):

def backtrack(start):

if start == len(nums)

result.append(nums[:])

return

for i in range(start, len(nums)):

if i > start and nums[i] == nums[i - 1]:

continue

nums[start], nums[i] = nums[i], nums[start] # Swap

backtrack(start + 1) # Recurse

nums[start], nums[i] = nums[i], nums[start]

nums.sort()

result = []

backtrack(0)

return result

nums = [1, 1, 2]

print(permuteUnique(nums))

OUTPUT:

[[1, 1, 2], [1, 2, 1], [2, 1, 1]]