**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**

**Output:**

**Explanation: Each 'Q' represents a queen, and '.' represents an empty space.**

**Program:**

def is\_safe(board, row, col, N):

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, N, 1), range(col, -1, -1)):

if board[i][j] == 'Q':

return False

return True

def solve\_nqueens\_util(board, col, N):

if col >= N:

print\_board(board, N)

print()

return True

res = False

for i in range(N):

if is\_safe(board, i, col, N):

board[i][col] = 'Q'

res = solve\_nqueens\_util(board, col + 1, N) or res

board[i][col] = '.'

return res

def solve\_nqueens(N):

board = [['.' for \_ in range(N)] for \_ in range(N)]

solve\_nqueens\_util(board, 0, N)

def print\_board(board, N):

for row in board:

print(" ".join(row))

print("Solutions for 4-Queens:")

solve\_nqueens(4)

**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]**

**Explanation: Adapt the algorithm to place 8 queens on an 8×10 board, ensuring no two queens threaten each other.**

**Program:**

class NQueensSolver:

def \_\_init\_\_(self, rows, cols, obstacles=None, restricted\_positions=None):

self.rows = rows

self.cols = cols

self.obstacles = set(obstacles) if obstacles else set()

self.restricted\_positions = set(restricted\_positions) if restricted\_positions else set()

self.solutions = []

def is\_safe(self, board, row, col):

if (row, col) in self.obstacles or (row, col) in self.restricted\_positions:

return False

for r, c in enumerate(board[:row]):

if c == col or abs(c - col) == abs(r - row):

return False

return True

def solve\_n\_queens\_util(self, board, row):

if row == self.rows:

self.solutions.append(board[:])

return

for col in range(self.cols):

if self.is\_safe(board, row, col):

board[row] = col

self.solve\_n\_queens\_util(board, row + 1)

board[row] = -1

def solve(self):

board = [-1] \* self.rows

self.solve\_n\_queens\_util(board, 0)

return self.solutions

def format\_solution(solution):

return [[(row, col) for row, col in enumerate(sol)] for sol in solution]

solver\_8x10 = NQueensSolver(rows=8, cols=10)

solutions\_8x10 = solver\_8x10.solve()

print(f"8x10 Board Solutions: {format\_solution(solutions\_8x10)}")

solver\_5x5\_with\_obstacles = NQueensSolver(rows=5, cols=5, obstacles=[(2, 2), (3, 3)])

solutions\_5x5\_with\_obstacles = solver\_5x5\_with\_obstacles.solve()

print(f"5x5 Board with Obstacles Solutions: {format\_solution(solutions\_5x5\_with\_obstacles)}")

solver\_6x6\_restricted = NQueensSolver(rows=6, cols=6, restricted\_positions=[(0, 1), (1, 3), (2, 5)])

solutions\_6x6\_restricted = solver\_6x6\_restricted.solve()

print(f"6x6 Board with Restricted Positions Solutions: {format\_solution(solutions\_6x6\_restricted)}")

**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 indicates empty cells.**

**Example :**

**Input: board = [["5","3",".",".","7",".",".",".","."], ["6",".",".","1","9","5",".",".","."], grid.The '.' character [".","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"]]**

**Program:**

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

for i in range(9):

if board[row][i] == num:

return False

for i in range(9):

if board[i][col] == num:

return False

start\_row = 3 \* (row // 3)

start\_col = 3 \* (col // 3)

for i in range(3):

for j in range(3):

if board[start\_row + i][start\_col + j] == num:

return False

return True

def solve\_sudoku(board):

empty\_pos = find\_empty\_position(board)

if not empty\_pos:

return True

row, col = empty\_pos

for num in '123456789':

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

board[row][col] = num

if solve\_sudoku(board):

return True

board[row][col] = '.'

return False

def find\_empty\_position(board):

for i in range(9):

for j in range(9):

if board[i][j] == '.':

return (i, j)

return None

def print\_board(board):

for row in board:

print(" ".join(row))

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"]

]

print("Original Sudoku Board:")

print\_board(board)

if solve\_sudoku(board):

print("\nSolved Sudoku Board:")

print\_board(board)

else:

print("\nNo solution exists for the given Sudoku board.")

**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 :**

**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"]]**

**Program:**

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

for i in range(9):

if board[row][i] == num:

return False

for i in range(9):

if board[i][col] == num:

return False

start\_row = 3 \* (row // 3)

start\_col = 3 \* (col // 3)

for i in range(3):

for j in range(3):

if board[start\_row + i][start\_col + j] == num:

return False

return True

def solve\_sudoku(board):

empty\_pos = find\_empty\_position(board)

if not empty\_pos:

return True

row, col = empty\_pos

for num in '123456789':

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

board[row][col] = num

if solve\_sudoku(board):

return True

board[row][col] = '.'

return False

def find\_empty\_position(board):

for i in range(9):

for j in range(9):

if board[i][j] == '.':

return (i, j)

return None

def print\_board(board):

for row in board:

print(" ".join(row))

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"]

]

print("Original Sudoku Board:")

print\_board(board)

if solve\_sudoku(board):

print("\nSolved Sudoku Board:")

print\_board(board)

else:

print("\nNo solution exists for the given Sudoku board.")

**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 :**

**Input: nums = [1,1,1,1,1]**

**target = 3**

**Output: 5**

**Program:**

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

total\_sum = sum(nums)

if total\_sum < abs(target) or (total\_sum + target) % 2 != 0:

return 0

s = (total\_sum + target) // 2

dp = [0] \* (s + 1)

dp[0] = 1

for num in nums:

for j in range(s, num - 1, -1):

dp[j] += dp[j - num]

return dp[s]

nums1 = [1, 1, 1, 1, 1]

target1 = 3

print(f"Number of ways to reach the target {target1}:", find\_target\_sum\_ways(nums1, target1))

**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 + 7.**

**Example :**

**Input: arr = [3,1,2,4]**

**Output: 17**

**Program:**

def sum\_of\_subarray\_mins(arr):

MOD = 10\*\*9 + 7

n = len(arr)

prev\_less = [0] \* n

next\_less = [0] \* n

stack = []

for i in range(n):

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

stack.pop()

prev\_less[i] = stack[-1] if stack else -1

stack.append(i)

stack.clear()

for i in range(n - 1, -1, -1):

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

stack.pop()

next\_less[i] = stack[-1] if stack else n

stack.append(i)

total\_sum = 0

for i in range(n):

left\_count = i - prev\_less[i]

right\_count = next\_less[i] - i

total\_sum += arr[i] \* left\_count \* right\_count

total\_sum %= MOD

return total\_sum

arr = [3, 1, 2, 4]

output = sum\_of\_subarray\_mins(arr)

print(f"Output: {output}")

**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 :**

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

**target = 7**

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

**Program:**

def combination\_sum(candidates, target):

def backtrack(start, current\_combination, current\_sum):

if current\_sum == target:

results.append(list(current\_combination))

return

elif current\_sum > target:

return

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

current\_combination.append(candidates[i])

backtrack(i, current\_combination, current\_sum + candidates[i]) to the next index since we can reuse the same element

current\_combination.pop() # Backtrack to try the next candidate

results = []

backtrack(0, [], 0) # Start backtracking

return results

candidates = [2, 3, 6, 7]

target = 7

output = combination\_sum(candidates, target)

print(f"Output: {output}")

**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 :**

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

**target = 8**

**Output: [ [1,1,6], [1,2,5], [1,7], [2,6] ]**

**Program:**

def combination\_sum2(candidates, target):

def backtrack(start, current\_combination, current\_sum):

if current\_sum == target:

results.append(list(current\_combination))

return

elif current\_sum > target:

return

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

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

continue

current\_combination.append(candidates[i]) # Choose the candidate

backtrack(i + 1, current\_combination, current\_sum + candidates[i]) # Move to the next index

current\_combination.pop() # Backtrack to try the next candidate

candidates.sort() # Sort the candidates to facilitate skipping duplicates

results = []

backtrack(0, [], 0) # Start backtracking

return results

candidates = [10, 1, 2, 7, 6, 1, 5]

target = 8

output = combination\_sum2(candidates, target)

print(f"Output: {output}")

**9. Given an array nums of distinct integers, return all the possible permutations. You can return the answer in any order.**

**Example : Input: nums = [1,2,3]**

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

**Program:**

def permute(nums):

def backtrack(start):

if start == len(nums):

results.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]

results = []

backtrack(0)

return results

nums = [1, 2, 3]

output = permute(nums)

print(f"Output: {output}")nuvvu chetene

**10. Given a collection of numbers, nums, that might contain duplicates, return all possible unique permutations in any order.**

**Example :**

**Input: nums = [1,1,2]**

**Output: [[1,1,2], [1,2,1], [2,1,1]]**

**Program:**

def permute\_unique(nums):

def backtrack(path, used):

if len(path) == len(nums):

results.append(path[:])

return

for i in range(len(nums)):

if used[i]:

continue

if i > 0 and nums[i] == nums[i - 1] and not used[i - 1]:

continue

used[i] = True

path.append(nums[i])

backtrack(path, used)

used[i] = False

path.pop()

nums.sort()

results = []

used = [False] \* len(nums)

backtrack([], used)

return results

nums = [1, 1, 2]

output = permute\_unique(nums)

print(f"Output: {output}")