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. b. Visualization for 5-Queens: Input: N = 5 Output: c. Visualization for 8-Queens: Input: N = 8 Output:

def print\_board(board):

for row in board:

print(" ".join(row))

print()

def solve\_n\_queens(n):

def backtrack(row):

if row == n:

results.append(["".join(board[i]) for i in range(n)])

return

for col in range(n):

if col in cols or row - col in diag1 or row + col in diag2:

continue

cols.add(col)

diag1.add(row - col)

diag2.add(row + col)

board[row][col] = 'Q'

backtrack(row + 1)

board[row][col] = '.'

cols.remove(col)

diag1.remove(row - col)

diag2.remove(row + col)

results = []

board = [["."] \* n for \_ in range(n)]

cols = set()

diag1 = set()

diag2 = set()

backtrack(0)

return results

# Example: Print solutions for N = 4, 5, 8

for n in [4, 5, 8]:

print(f"\nSolutions for {n}-Queens:")

solutions = solve\_n\_queens(n)

for sol in solutions:

print\_board([list(row) for row in 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] Explanation: Adapt the algorithm to place 8 queens on an 8×10 board, ensuring no two queens threaten each other. b. 5×5 Board with Obstacles: Input: N = 5, Obstacles at positions [(2, 2), (4, 4)] Output: Possible solution [1, 3, 5, 2, 4] Explanation: Modify the algorithm to avoid placing queens on obstacle positions, ensuring a valid solution that respects the constraints. c. 6×6 Board with Restricted Positions: Input: N = 6, Restricted positions at columns 2 and 4 for the first queen Output: Possible solution [1, 3, 5, 2, 4, 6] Explanation: Adjust the algorithm to handle restricted positions, ensuring the queens are placed without conflicts and within allowed columns.

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"]]

def solve\_sudoku(board):

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

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

for i in range(9):

if board[row][i] == num: return False # Check row

if board[i][col] == num: return False # Check column

if board[3 \* (row // 3) + i // 3][3 \* (col // 3) + i % 3] == num: return False # Check 3x3 sub-box

return True

def backtrack():

for row in range(9):

for col in range(9):

if board[row][col] == '.': # Empty cell

for num in map(str, range(1, 10)): # Digits 1-9

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

board[row][col] = num # Place num

if backtrack(): # Recursively try the next cell

return True

board[row][col] = '.' # Undo placement if it leads to no solution

return False # No valid num found, backtrack

return True # Solved

backtrack()

return board

# 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"]

]

# Solving the Sudoku

solved\_board = solve\_sudoku(board)

for row in solved\_board:

print(row)

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"]]

def solve\_sudoku(board):

# Function to check if placing num at board[row][col] is valid

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

# Check row and column

for i in range(9):

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

return False

# Check 3x3 sub-box

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

# Backtracking function to solve the board

def backtrack():

for row in range(9):

for col in range(9):

if board[row][col] == '.': # Empty cell

for num in map(str, range(1, 10)): # Digits '1' to '9'

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

board[row][col] = num # Place num

if backtrack(): # Recursive call

return True

board[row][col] = '.' # Undo if not valid

return False # No valid number found

return True # Puzzle solved

backtrack()

return board

# 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"]

]

# Solving the Sudoku

solved\_board = solve\_sudoku(board)

for row in solved\_board:

print(row)

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 Explanation: There are 5 ways to assign symbols to make the sum of nums be target 3. -1 + 1 + 1 + 1 + 1 = 3 +1 - 1 + 1 + 1 + 1 = 3 +1 + 1 - 1 + 1 + 1 = 3 +1 + 1 + 1 - 1 + 1 = 3 +1 + 1 + 1 + 1 - 1 = 3 Example 2: Input: nums = [1], target = 1 Output: 1

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

# Dictionary to memoize intermediate results

memo = {}

def backtrack(index, current\_sum):

# Check if this (index, current\_sum) has been computed before

if (index, current\_sum) in memo:

return memo[(index, current\_sum)]

# Base case: If all numbers are used, check if we reached the target

if index == len(nums):

return 1 if current\_sum == target else 0

# Try both adding and subtracting the current number

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

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

# Store the result in memo

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

return memo[(index, current\_sum)]

# Start the recursion from index 0 and a cumulative sum of 0

return backtrack(0, 0)

# Example 1

nums = [1, 1, 1, 1, 1]

target = 3

print("Number of ways:", find\_target\_sum\_ways(nums, target)) # Output: 5

# Example 2

nums = [1]

target = 1

print("Number of ways:", find\_target\_sum\_ways(nums, target)) # Output: 1

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 1: Input: arr = [3,1,2,4] Output: 17 Explanation: Subarrays are [3], [1], [2], [4], [3,1], [1,2], [2,4], [3,1,2], [1,2,4], [3,1,2,4]. Minimums are 3, 1, 2, 4, 1, 1, 2, 1, 1, 1. Sum is 17. Example 2: Input: arr = [11,81,94,43,3] Output: 444

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

# Dictionary to store the number of ways for each (index, current\_sum)

memo = {}

def backtrack(index, current\_sum):

# Check if we have already computed this subproblem

if (index, current\_sum) in memo:

return memo[(index, current\_sum)]

# Base case: all numbers are used

if index == len(nums):

return 1 if current\_sum == target else 0

# Choose to add or subtract the current number

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

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

# Save result in memo

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

return memo[(index, current\_sum)]

# Start recursion from index 0 and initial sum 0

return backtrack(0, 0)

# Example 1

nums = [1, 1, 1, 1, 1]

target = 3

print("Number of ways:", find\_target\_sum\_ways(nums, target)) # Output: 5

# Example 2

nums = [1]

target = 1

print("Number of ways:", find\_target\_sum\_ways(nums, target)) # Output: 1

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]] Explanation: 2 and 3 are candidates, and 2 + 2 + 3 = 7. Note that 2 can be used multiple times. 7 is a candidate, and 7 = 7. These are the only two combinations. Example 2: Input: candidates = [2,3,5], target = 8 Output: [[2,2,2,2],[2,3,3],[3,5]] 4. COMBINATION SUM 2:

def combination\_sum(candidates, target):

result = []

def backtrack(remaining, combo, start):

if remaining == 0:

result.append(list(combo)) # Found a valid combination

return

elif remaining < 0:

return # Exceeded the target, no need to proceed

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

combo.append(candidates[i]) # Choose the candidate

backtrack(remaining - candidates[i], combo, i) # Explore with the same candidate

combo.pop() # Undo choice

backtrack(target, [], 0)

return result

# Example 1

candidates = [2, 3, 6, 7]

target = 7

print("Combinations:", combination\_sum(candidates, target)) # Output: [[2, 2, 3], [7]]

# Example 2

candidates = [2, 3, 5]

target = 8

print("Combinations:", combination\_sum(candidates, target)) # Output: [[2, 2, 2, 2], [2, 3, 3], [3, 5]]

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] ] Example 2: Input: candidates = [2,5,2,1,2], target = 5 Output: [ [1,2,2], [5] ]

def combination\_sum2(candidates, target):

candidates.sort() # Sort to handle duplicates

result = []

def backtrack(remaining, combo, start):

if remaining == 0:

result.append(list(combo)) # Found a valid combination

return

elif remaining < 0:

return # Exceeded the target, no need to proceed

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

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

continue

combo.append(candidates[i]) # Choose the candidate

backtrack(remaining - candidates[i], combo, i + 1) # Move to next index

combo.pop() # Undo choice

backtrack(target, [], 0)

return result

# Example 1

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

target = 8

print("Combinations:", combination\_sum2(candidates, target)) # Output: [[1, 1, 6], [1, 2, 5], [1, 7], [2, 6]]

# Example 2

candidates = [2, 5, 2, 1, 2]

target = 5

print("Combinations:", combination\_sum2(candidates, target)) # Output: [[1, 2, 2], [5]]

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]] Example 2: Input: nums = [0,1] Output: [[0,1],[1,0]] Example 3: Input: nums = [1] Output: [[1]]

def permute(nums):

result = []

def backtrack(current\_permutation, used):

if len(current\_permutation) == len(nums):

result.append(list(current\_permutation)) # Found a full permutation

return

for i in range(len(nums)):

if used[i]:

continue # Skip if already used

used[i] = True

current\_permutation.append(nums[i])

backtrack(current\_permutation, used)

current\_permutation.pop() # Backtrack

used[i] = False

backtrack([], [False] \* len(nums))

return result

# Example 1

nums = [1, 2, 3]

print("Permutations:", permute(nums)) # Output: [[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]

# Example 2

nums = [0, 1]

print("Permutations:", permute(nums)) # Output: [[0,1],[1,0]]

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

def permute\_unique(nums):

nums.sort() # Sort to make it easier to handle duplicates

result = []

def backtrack(current\_permutation, used):

if len(current\_permutation) == len(nums):

result.append(list(current\_permutation)) # Found a full unique permutation

return

for i in range(len(nums)):

if used[i]:

continue # Skip if already used

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

continue # Skip duplicates

used[i] = True

current\_permutation.append(nums[i])

backtrack(current\_permutation, used)

current\_permutation.pop() # Backtrack

used[i] = False

backtrack([], [False] \* len(nums))

return result

# Example 1

nums = [1, 1, 2]

print("Unique Permutations:", permute\_unique(nums)) # Output: [[1,1,2], [1,2,1], [2,1,1]]

# Example 2

nums = [1, 2, 3]

print("Unique Permutations:", permute\_unique(nums)) # Output: [[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]