DAA 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.

```
import matplotlib.pyplot as plt

import numpy as np

def plot_n_queens_solution(solution, N):

    board = np.zeros((N, N))

    for i in range(N):

        board[i, solution[i]] = 1

    plt.imshow(board, cmap='binary')

    plt.xticks([])  # Hide axis labels

    plt.yticks([])

    for i in range(N):

        plt.text(solution[i], i, '♛', ha='center', va='center', fontsize=20)

    plt.show()

solutions = {

    4: [1, 3, 0, 2],

    5: [0, 2, 4, 1, 3],

    8: [0, 4, 7, 5, 2, 6, 1, 3]

}

for N, solution in solutions.items():

    plot_n_queens_solution(solution, N)
```

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.

```
import numpy as np

N = 5

rows, cols = 5, 5

obstacles = [(2, 2), (4, 3)]

restricted_positions = [(1, 0), (3, 4)]

board = np.zeros((rows, cols))

for r, c in obstacles:

    board[r][c] = -1

queens = []

for r in range(rows):

    for c in range(cols):

        if (r, c) not in obstacles and (r, c) not in restricted_positions:

            safe = True

            for qr, qc in queens:

                if qr == r or qc == c or abs(qr - r) == abs(qc - c):

                    safe = False

                    break

            if safe:

                queens.append((r, c))

                board[r][c] = 1

                break

for row in board:

    print(" ".join('Q' if cell == 1 else 'X' if cell == -1 else '.' for cell in row))
```

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.

```
def is_safe(board, row, col, num):
    for x in range(9):
        if board[row][x] == num:
            return False
    for x in range(9):
        if board[x][col] == num:
            return False
    start_row, start_col = 3 * (row // 3), 3 * (col // 3)
    for i in range(3):
        for j in range(3):
            if board[i + start_row][j + start_col] == num:
                return False
    return True
def solve_sudoku(board):
    for row in range(9):
        for col in range(9):
            if board[row][col] == 0:  # Find an empty cell
                for num in range(1, 10):  # Try all numbers from 1 to 9
                    if is_safe(board, row, col, num):
                        board[row][col] = num  # Tentatively place num
                        if solve_sudoku(board):  # Continue with the next empty cell
                            return True
                        board[row][col] = 0  # Backtrack if no number works
                return False
    return True
def print_board(board):
    for row in board:
        print(" ".join(str(num) for num in row))
```

```
board = [
    [5, 3, 0, 0, 7, 0, 0, 0, 0],
    [6, 0, 0, 1, 9, 5, 0, 0, 0],
    [0, 9, 8, 0, 0, 0, 0, 6, 0],
    [8, 0, 0, 0, 6, 0, 0, 0, 3],
    [4, 0, 0, 8, 0, 3, 0, 0, 1],
    [7, 0, 0, 0, 2, 0, 0, 0, 6],
    [0, 6, 0, 0, 0, 0, 2, 8, 0],
    [0, 0, 0, 4, 1, 9, 0, 0, 5],
    [0, 0, 0, 0, 8, 0, 0, 7, 9]
]
if solve_sudoku(board):
    print_board(board)
else:
    print("No solution exists")
```

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

```
def solveSudoku(board):
    def isValid(r, c, num):
        return all(num not in (board[i][c], board[r][j], board[r//3*3+j//3][c//3*3+j%3]) for i in range(9) for j in range(9))
    def backtrack():
        for r in range(9):
            for c in range(9):
                if board[r][c] == '.':
                    for num in '123456789':
                        if isValid(r, c, num):
                            board[r][c] = num
                            if backtrack(): return True
                            board[r][c] = '.'
```

```
            return False
        return True
    backtrack()
```

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.

```
def find_target_sum_ways(nums, target):
    total_sum = sum(nums)
    if target > total_sum or (total_sum + target) % 2 != 0:
        return 0
    target_sum = (total_sum + target) // 2
    dp = [0] * (target_sum + 1)
    dp[0] = 1
    for num in nums:
        for j in range(target_sum, num - 1, -1):
            dp[j] += dp[j - num]
    return dp[target_sum]
nums = [1, 1, 1, 1, 1]
target = 3
print(find_target_sum_ways(nums, target))
```

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.

```
def sumSubarrayMins(arr):
    MOD = 10**9 + 7
    n = len(arr)
    stack = []
    result = 0
    prev_sum = [0] * n
    for i in range(n):
        count = 1
        while stack and stack[-1][0] >= arr[i]:
```

```
            val, cnt = stack.pop()
            count += cnt
        prev_sum[i] = (arr[i] * count + (prev_sum[stack[-1][1]] if stack else 0)) % MOD
        result = (result + prev_sum[i]) % MOD
        stack.append((arr[i], count))
    return result
arr = [3, 1, 2, 4]
print(sumSubarrayMins(arr))
```

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.

```
def combinationSum(candidates, target):
    def backtrack(start, target, path):
        if target == 0:
            result.append(path)
            return
        if target < 0:
            return
        for i in range(start, len(candidates)):
            backtrack(i, target - candidates[i], path + [candidates[i]])
    result = []
    backtrack(0, target, [])
    return result
candidates = [2, 3, 6, 7]
target = 7
print(combinationSum(candidates, target))
```

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.

```
def combinationSum2(candidates, target):
    def backtrack(start, target, path):
        if target == 0:
            result.append(path)
            return
        if target < 0:
            return
        for i in range(start, len(candidates)):
            # Skip duplicates
            if i > start and candidates[i] == candidates[i - 1]:
                continue
            backtrack(i + 1, target - candidates[i], path + [candidates[i]])
    candidates.sort()  # Sort to handle duplicates
    result = []
    backtrack(0, target, [])
    return result
candidates = [10, 1, 2, 7, 6, 1, 5]
target = 8
print(combinationSum2(candidates, target))
```

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

```
def permute(nums):
    def backtrack(start):
        if start == len(nums):
            result.append(nums[:])
            return
        for i in range(start, len(nums)):
            nums[start], nums[i] = nums[i], nums[start]  # Swap
```

```
            backtrack(start + 1)
            nums[start], nums[i] = nums[i], nums[start]  # Swap back
    result = []
    backtrack(0)
    return result
nums = [1, 2, 3]
print(permute(nums))
```

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

```
def permuteUnique(nums):
    def backtrack(path, visited):
        if len(path) == len(nums):
            result.append(path[:])
            return
        for i in range(len(nums)):
            if visited[i] or (i > 0 and nums[i] == nums[i - 1] and not visited[i - 1]):
                continue
            visited[i] = True
            path.append(nums[i])
            backtrack(path, visited)
            path.pop()
            visited[i] = False
    nums.sort()
    result = []
    backtrack([], [False] * len(nums))
    return result
nums = [1, 1, 2]
print(permuteUnique(nums))
```