## 1. Longest Increasing Subsequence (LIS)

**Problem:**

Given an unsorted array of integers, find the length of the **longest increasing subsequence**. A subsequence is defined as a sequence derived by deleting some or no elements from the array without changing the order of the remaining elements.

**Example:**

```
arr = [10, 22, 9, 33, 21, 50, 41, 60, 80]
# The longest increasing subsequence is [10, 22, 33, 50, 60, 80], so the output should be 6.
```

**Hint:**

Use dynamic programming or binary search for optimization. The brute force solution has a time complexity of O(2^n), but you can optimize it to O(n^2) using dynamic programming, or even O(n log n) with binary search and a clever approach.

**Solution:**

```
# DP bottom up approach

class Solution:

    def lengthOfLIS(self, nums: List[int]) -> int:

        Lis = [1] * len(nums) # longest increasing sequence dp memoization array


        for i in range(len(nums) - 1, -1, -1):

            for j in range(i + 1, len(nums)):

                if nums[i] < nums[j]: # strictly increasing

                    Lis[i] = max(Lis[i], 1 + Lis[j])

        return max(Lis) # return the max value in Lis
```

## 2. Sudoku Solver

**Problem:**

Write a function that takes a partially filled 9x9 Sudoku board and fills in the missing values to solve it. If the board is unsolvable, return `False`.

**Example:**

```
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]
]

# The function should fill in the board and return True if solved.
```

**Hint:**

This problem can be solved using **backtracking**. Try to fill each empty space with numbers from 1 to 9, and recursively check if the number placement is valid. If the board is valid, continue; if not, backtrack and try a different number.

**Solution:**

```
from typing import List


class SudokuSolver:
```

```
def solveSudoku(self, board: List[List[int]]) -> bool:
    """
    Solves the Sudoku puzzle using backtracking.
    return: True if the Sudoku is solved successfully, False otherwise.
    """
    empty = self.findEmpty(board)
    if not empty:
        return True  # Sudoku solved
    row, col = empty

    for num in range(1, 10):
        if self.isValid(board, num, (row, col)):
            board[row][col] = num  # Tentatively place num

            if self.solveSudoku(board):
                return True  # If successful, propagate True

            board[row][col] = 0  # Backtrack if not successful

    return False  # Trigger backtracking

def findEmpty(self, board: List[List[int]]) -> tuple:
    """
    Finds the next empty cell in the Sudoku board.
```

```
        return: Tuple of (row, col) if an empty cell is found, None otherwise.
        """
        for i in range(9):
            for j in range(9):
                if board[i][j] == 0:
                    return (i, j)  # Row, Column
        return None

    def isValid(self, board: List[List[int]], num: int, pos: tuple) -> bool:
        """
        Checks whether it's valid to place a number in a given position.
        return: True if valid, False otherwise.
        """
        row, col = pos

        # Check row
        if any(board[row][i] == num for i in range(9) if i != col):
            return False

        # Check column
        if any(board[i][col] == num for i in range(9) if i != row):
            return False

        # Check 3x3 subgrid
```

```
box_x = col // 3
box_y = row // 3

for i in range(box_y*3, box_y*3 + 3):
    for j in range(box_x*3, box_x*3 + 3):
        if board[i][j] == num and (i, j) != pos:
            return False

return True
```

## 3. N-Queens Problem

**Problem:**

Solve the **N-Queens problem**, which asks you to place **N queens on an NxN chessboard** such that no two queens threaten each other. A queen can attack another queen if they are on the same row, column, or diagonal.

**Example:**

For `N = 4`, one valid solution would be:

```
[ [0, 1, 2, 3],
  [1, 3, 0, 2],
  [2, 0, 3, 1],
  [3, 2, 1, 0] ]
```

**Hint:**

You can solve this problem using **backtracking**. Keep track of columns, main diagonals, and anti-diagonals where queens have already been placed to avoid conflicts.

**Solution:**

```
from typing import List


class Solution:

    def solveNQueens(self, n: int) -> List[int]:

        def solve(row: int, board: List[int]) -> bool:

            if row == n:

                return True

            for col in range(n):

                if self.is_safe(row, col, board):

                    board[row] = col
```

```
            if solve(row + 1, board):
                return True
            board[row] = -1
        return False

    board = [-1] * n
    if solve(0, board):
        return board
    return None

def is_safe(self, row: int, col: int, board: List[int]) -> bool:
    for prev_row in range(row):
        if board[prev_row] == col or \
           abs(board[prev_row] - col) == abs(prev_row - row):
            return False
    return True
```

## 4. Word Ladder

**Problem:**

Given two words (beginWord and endWord), and a dictionary of words, find the shortest transformation sequence from `beginWord` to `endWord`, such that:

- Only one letter can be changed at a time.
- Each transformed word must exist in the dictionary.

For example:

```
beginWord = "hit"

endWord = "cog"

wordList = ["hot", "dot", "dog", "lot", "log", "cog"]

# The shortest transformation sequence is: "hit" -> "hot" -> "dot" ->
"dog" -> "cog"
```

**Hint:**

Use **breadth-first search (BFS)** to explore the word transformation graph. Each word is a node, and each letter change forms an edge. Keep track of the visited nodes to avoid cycles and redundant work.

**Solution:**

```
from typing import List

from collections import deque


class Solution:

    def ladderLength(self, beginWord: str, endWord: str, wordList: List[str]) -> List[str]:

        """

        Finds the shortest transformation sequence from beginWord to endWord.

        """
```

```
# Convert wordList to a set for efficient lookup
word_set = set(wordList)
if endWord not in word_set:
    return []  # Early exit if endWord is not in wordList

# Initialize BFS queue with the beginWord and its path
queue = deque([[beginWord]])
visited = set([beginWord])  # To keep track of visited words

while queue:
    # Keep track of words visited at the current level
    current_level_visited = set()
    level_size = len(queue)

    for _ in range(level_size):
        current_path = queue.popleft()
        current_word = current_path[-1]

        # Generate all possible one-letter transformations
        for i in range(len(current_word)):
            for c in 'abcdefghijklmnopqrstuvwxyz':
                if c == current_word[i]:
                    continue  # Skip same character
                next_word = current_word[:i] + c + current_word[i+1:]
```

```
            if next_word == endWord:
                return current_path + [endWord]  # Found the shortest path

            if next_word in word_set and next_word not in visited:
                current_level_visited.add(next_word)
                queue.append(current_path + [next_word])

    # Mark words visited at this level to prevent revisiting
    visited.update(current_level_visited)

return []  # No transformation sequence found
```

## 5. Find the Median of Two Sorted Arrays

**Problem:**

Given two sorted arrays `nums1` and `nums2`, find the **median** of the two sorted arrays. The overall run time complexity should be O(log(min(n, m))) where n and m are the lengths of the arrays.

**Example:**

```
nums1 = [1, 3]

nums2 = [2]

# The median is 2.0
```

**Hint:**

This is a classic **binary search** problem. Instead of merging the arrays (which takes O(n+m) time), use binary search to partition the arrays in such a way that the left part and the right part of the merged array are balanced.

**Solution:**

```
from typing import List


class Solution:

    def findMedianSortedArrays(self, nums1: List[int], nums2: List[int]) -> float:

        # Ensure nums1 is the smaller array to minimize binary search iterations

        if len(nums1) > len(nums2):

            nums1, nums2 = nums2, nums1


        x = len(nums1)

        y = len(nums2)
```

```
low = 0
high = x

while low <= high:
    partitionX = (low + high) // 2
    partitionY = (x + y + 1) // 2 - partitionX

    # Handle edge cases where partition is at the beginning or end
    maxLeftX = float('-inf') if partitionX == 0 else nums1[partitionX - 1]
    minRightX = float('inf') if partitionX == x else nums1[partitionX]

    maxLeftY = float('-inf') if partitionY == 0 else nums2[partitionY - 1]
    minRightY = float('inf') if partitionY == y else nums2[partitionY]

    # Check if we have found the correct partition
    if maxLeftX <= minRightY and maxLeftY <= minRightX:
        # If total length is even
        if (x + y) % 2 == 0:
            return (max(maxLeftX, maxLeftY) + min(minRightX, minRightY)) / 2.0
        else:
            return max(maxLeftX, maxLeftY)
    elif maxLeftX > minRightY:
        # Move towards left in nums1
        high = partitionX - 1
```

```
    else:
        # Move towards right in nums1
        low = partitionX + 1

# If the arrays are not sorted or inputs are invalid
raise ValueError("Input arrays are not sorted or invalid.")
```

## 6. Graph Cycle Detection (Directed Graph)

**Problem:**

Given a directed graph, determine if there is a **cycle** in the graph. If there is a cycle, return `True`; otherwise, return `False`.

**Example:**

```
graph = {

    0: [1],

    1: [2],

    2: [3],

    3: [0]

}

# This graph contains a cycle, so the result should be True.
```

**Hint:**

This is a classic **graph traversal** problem. Use **Depth First Search (DFS)** with a **recursion stack** (or a `visited` list) to track the state of each node. If you revisit a node that is already in the recursion stack, you have detected a cycle.

**Solution:**

```
from typing import Dict, List, Set

class GraphCycleDetector:
    def __init__(self, graph: Dict[int, List[int]]):
        """
        Initializes the GraphCycleDetector with a directed graph.
        """
        self.graph = graph
        self.visited: Set[int] = set()
```

```
    self.recStack: Set[int] = set()

def hasCycle(self) -> bool:
    """
    Determines if the directed graph contains a cycle.

    return: True if there is at least one cycle in the graph, False otherwise.
    """
    for node in self.graph:
        if node not in self.visited:
            if self._dfs(node):
                return True
    return False

def _dfs(self, node: int) -> bool:
    """
    Performs DFS traversal to detect a cycle starting from the given node.
    return: True if a cycle is detected, False otherwise.
    """
    # Mark the current node as visited and add it to the recursion stack
    self.visited.add(node)
    self.recStack.add(node)

    # Recur for all the vertices adjacent to this vertex
    for neighbour in self.graph.get(node, []):
        if neighbour not in self.visited:
            if self._dfs(neighbour):
                return True
        elif neighbour in self.recStack:
            # If the neighbour is in recursion stack, a cycle is detected
            return True

    # The node needs to be removed from the recursion stack before function ends
    self.recStack.remove(node)
```

```
        return False

    def addEdge(self, from_node: int, to_node: int):
        """
        Adds a directed edge to the graph.
        """
        if from_node in self.graph:
            self.graph[from_node].append(to_node)
        else:
            self.graph[from_node] = [to_node]

    def removeEdge(self, from_node: int, to_node: int):
        """
        Removes a directed edge from the graph.
        """
        if from_node in self.graph:
            if to_node in self.graph[from_node]:
                self.graph[from_node].remove(to_node)
```