Remove duplicates from sorted list

```python
# Definition for singly-linked list.
class ListNode(object):
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

class Solution(object):
    def deleteDuplicates(self, head):
        """
        :type head: Optional[ListNode]
        :rtype: Optional[ListNode]
        """
        current = head

        # Traverse the list
        while current and current.next:
            # If the current value equals the next value, skip the duplicate
            if current.val == current.next.val:
                current.next = current.next.next
            else:
                # Move to the next node if no duplicate
                current = current.next

        return head
```

Find a Peck Element

```python
class Solution(object):
    def findPeakElement(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        left, right = 0, len(nums) - 1

        while left < right:
            mid = (left + right) // 2

            # Compare mid with its next element
            if nums[mid] > nums[mid + 1]:
                # Peak is in the left half (including mid)
                right = mid
            else:
                # Peak is in the right half
                left = mid + 1

        # Left and right converge to the peak index
        return left
```

Binary Tree: Inorder Traversal

```python
class Solution(object):
    def inorderTraversal(self, root):
        """
        :type root: Optional[TreeNode]
        :rtype: List[int]
        """
        def helper(node, result):
            if node:
                # Traverse left subtree
                helper(node.left, result)
                # Visit root
                result.append(node.val)
                # Traverse right subtree
                helper(node.right, result)

        result = []
        helper(root, result)
        return result
```

Valid Parentheses

```python
class Solution(object):
    def isValid(self, s):
        """
        :type s: str
        :rtype: bool
        """
        # Stack to keep track of opening brackets
        stack = []

        # Dictionary to match closing brackets with opening ones
        mapping = {")": "(", "}": "{", "]": "["}

        # Iterate through each character in the string
        for char in s:
            if char in mapping:  # If it's a closing bracket
                # Pop the top of the stack, if stack is empty return a mismatch
                top_element = stack.pop() if stack else '#'

                # If the top element of the stack does not match the expected
opening bracket
                if mapping[char] != top_element:
                    return False
            else:
                # If it's an opening bracket, push it to the stack
                stack.append(char)
        # If the stack is empty, all brackets matched correctly, else there's an
unmatched opening bracket
        return not stack
```

Merge Two Sorted Lists

```
# Definition for singly-linked list.
class ListNode(object):
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

class Solution(object):
    def mergeTwoLists(self, list1, list2):
        """
        :type list1: Optional[ListNode]
        :type list2: Optional[ListNode]
        :rtype: Optional[ListNode]
        """
        # Create a dummy node to act as the head of the new list
        dummy = ListNode(-1)
        current = dummy

        # Traverse both lists and merge
        while list1 and list2:
            if list1.val <= list2.val:
                current.next = list1
                list1 = list1.next
            else:
                current.next = list2
                list2 = list2.next
            current = current.next

        # Attach any remaining nodes
        current.next = list1 if list1 else list2

        # Return the merged list starting from the next of dummy
        return dummy.next
```

Find the Index of the First Occurrence in a String

```
class Solution(object):
    def strStr(self, haystack, needle):
        """
        :type haystack: str
        :type needle: str
        :rtype: int
        """
        # If the needle is an empty string, return 0
        if not needle:
            return 0

        # Iterate over the haystack to find the first occurrence of needle
        for i in range(len(haystack) - len(needle) + 1):
            if haystack[i:i + len(needle)] == needle:
                return i  # Return the index of the first occurrence

        # If no occurrence is found, return -1
        return -1
```

N-Queens

```
class Solution(object):
    def solveNQueens(self, n):
        """
        :type n: int
        :rtype: List[List[str]]
        """
        def is_safe(board, row, col):
            # Check if placing the queen at (row, col) is safe
            for i in range(row):
                # Check column and diagonals
                if board[i] == col or \
                   board[i] - i == col - row or \
                   board[i] + i == col + row:
                    return False
            return True

        def solve(board, row):
            # If all queens are placed, add the board configuration to results
            if row == n:
                result.append(['.' * i + 'Q' + '.' * (n - i - 1) for i in board])
                return

            for col in range(n):
                if is_safe(board, row, col):
                    board[row] = col  # Place queen
                    solve(board, row + 1)  # Recur for the next row
                    board[row] = -1  # Backtrack

        result = []
        solve([-1] * n, 0)  # Initialize the board and start solving from the
first row
        return result
```

Largest Number

```python
from functools import cmp_to_key

class Solution(object):
    def largestNumber(self, nums):
        """
        :type nums: List[int]
        :rtype: str
        """
        # Custom comparator
        def compare(x, y):
            if x + y > y + x:
                return -1
            elif x + y < y + x:
                return 1
            else:
                return 0

        # Convert numbers to strings for custom sorting
        nums = list(map(str, nums))
        # Sort using the custom comparator
        nums.sort(key=cmp_to_key(compare))
        # Concatenate sorted numbers
        result = ''.join(nums)
        # Handle the case where the result is all zeros
        return '0' if result[0] == '0' else result
```

Bitwise AND of Numbers Range

```python
class Solution(object):
    def rangeBitwiseAnd(self, left, right):
        # Shift left and right until they are equal
        shift = 0
        while left < right:
            left >>= 1
            right >>= 1
            shift += 1

        # Restore the common prefix
        return left << shift
```

Integer to English Words

```python
class Solution:
    def numberToWords(self, num):
        if num == 0:
            return "Zero"

        # Mappings for numbers
        below_20 = [
            "One", "Two", "Three", "Four", "Five", "Six", "Seven", "Eight", 
"Nine", "Ten",
            "Eleven", "Twelve", "Thirteen", "Fourteen", "Fifteen", "Sixteen", 
"Seventeen",
            "Eighteen", "Nineteen"
        ]
        tens = [
            "Twenty", "Thirty", "Forty", "Fifty", "Sixty", "Seventy", "Eighty", 
"Ninety"
        ]
        thousands = ["", "Thousand", "Million", "Billion"]

        # Helper function to process numbers below 1000
        def helper(n):
            if n == 0:
                return ""
            elif n < 20:
                return below_20[n - 1] + " "
            elif n < 100:
                return tens[n // 10 - 2] + " " + helper(n % 10)
            else:
                return below_20[n // 100 - 1] + " Hundred " + helper(n % 100)

        # Main conversion logic
        result = ""
        for i, suffix in enumerate(thousands):
            if num % 1000 != 0:
                result = helper(num % 1000) + suffix + " " + result
            num //= 1000

        return result.strip()
```

Magic Squares In Grid

```python
class Solution:
    def numMagicSquaresInside(self, grid):
        def isMagicSquare(r, c):
            # Collect all elements in the 3x3 subgrid
            nums = [grid[r+i][c+j] for i in range(3) for j in range(3)]
            # Check if all elements are distinct and within [1, 9]
            if sorted(nums) != [1, 2, 3, 4, 5, 6, 7, 8, 9]:
                return False

            # Calculate sums of rows, columns, and diagonals
            rows = [sum(grid[r+i][c:c+3]) for i in range(3)]
            cols = [sum(grid[r+i][c+j] for i in range(3)) for j in range(3)]
            diag1 = sum(grid[r+i][c+i] for i in range(3))
            diag2 = sum(grid[r+i][c+2-i] for i in range(3))

            # All sums must be equal to 15
            return rows == cols == [15, 15, 15] and diag1 == diag2 == 15

        # Grid dimensions
        rows, cols = len(grid), len(grid[0])
        count = 0

        # Check all possible 3x3 subgrids
        for r in range(rows - 2):
            for c in range(cols - 2):
                if isMagicSquare(r, c):
                    count += 1

        return count
```

Spiral Matrix III

```
class Solution:
    def spiralMatrixIII(self, rows, cols, rStart, cStart):
        directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]  # Right, Down, Left, Up
        result = []
        total_cells = rows * cols
        steps = 0
        direction_index = 0
        current_row, current_col = rStart, cStart

        while len(result) < total_cells:
            if direction_index % 2 == 0:  # Increase steps every two turns
                steps += 1

            for _ in range(steps):
                # Add valid positions to the result
                if 0 <= current_row < rows and 0 <= current_col < cols:
                    result.append([current_row, current_col])

                # Move in the current direction
                current_row += directions[direction_index][0]
                current_col += directions[direction_index][1]

            # Change direction (rotate clockwise)
            direction_index = (direction_index + 1) % 4

        return result
```

Number of Ways to Split Array

```python
class Solution:
    def waysToSplitArray(self, nums):
        total_sum = sum(nums)
        left_sum = 0
        valid_splits_count = 0

        for i in range(len(nums) - 1):  # Only go up to n-1 to ensure right side
isn't empty
            left_sum += nums[i]
            right_sum = total_sum - left_sum

            if left_sum >= right_sum:
                valid_splits_count += 1

        return valid_splits_count
```

Count Number of Possible Root Nodes

```python
from collections import defaultdict

class Solution(object):
    def rootCount(self, edges, guesses, k):
        """
        :type edges: List[List[int]]
        :type guesses: List[List[int]]
        :type k: int
        :rtype: int
        """
        # Create adjacency list for the tree
        tree = defaultdict(list)
        for u, v in edges:
            tree[u].append(v)
            tree[v].append(u)

        # Convert guesses to a set for quick lookup
        guess_set = set((u, v) for u, v in guesses)

        # Function to calculate initial correct guesses with root at 0
        def dfs(node, parent):
            correct = 0
            for neighbor in tree[node]:
                if neighbor == parent:
                    continue
                if (node, neighbor) in guess_set:
                    correct += 1
                correct += dfs(neighbor, node)
            return correct

        # Initial correct guesses with root at 0
        initial_correct = dfs(0, -1)
```

```python
# Re-rooting function to calculate correct guesses for all roots
def reroot(node, parent, current_correct):
    if current_correct >= k:
        self.result += 1
    for neighbor in tree[node]:
        if neighbor == parent:
            continue
        # Adjust correct guesses for re-rooting
        new_correct = current_correct
        if (node, neighbor) in guess_set:
            new_correct -= 1
        if (neighbor, node) in guess_set:
            new_correct += 1
        reroot(neighbor, node, new_correct)

# Start re-rooting from 0
self.result = 0
reroot(0, -1, initial_correct)

return self.result
```