TASK 6

39. Sudoku Validator

Checks if the given 9x9 Sudoku board is valid.

Parameters:

board (list of list): A 9x9 matrix representing the Sudoku board.

Returns:

bool: True if the board is valid, otherwise False.

Code:

```
def is_valid_sudoku(board):
    def is_valid_group(group):
        """Helper function to check if a row, column, or box has duplicates (excluding '.')."""
        nums = [num for num in group if num != '.']
        return len(nums) == len(set(nums))

    # Check rows
    for row in board:
        if not is_valid_group(row):
            return False

    # Check columns
```

```
    for col in zip(*board):  # Transpose to check columns
        if not is_valid_group(col):
            return False

    # Check 3x3 sub-grids
    for i in range(0, 9, 3):
        for j in range(0, 9, 3):
            block = [board[x][y] for x in range(i, i+3) for y in range(j, j+3)]
            if not is_valid_group(block):
                return False

    return True
sudoku_board = []
print("Enter the Sudoku board row by row (use '.' for empty cells):")
for _ in range(9):
    sudoku_board.append(list(input().strip().split()))

# Validate Sudoku
print("Valid Sudoku?" , is_valid_sudoku(sudoku_board))
```

40. Word Frequency in Text

Counts the frequency of words in a given text.

Parameters: text (str): Input string.

Returns: dict: Dictionary with words as keys and their frequencies as values.

Code:

```
from collections import Counter


def word_frequency(text):
    words = text.lower().split()
    return dict(Counter(words))


# Interactive Input
text = input("Enter a text: ")
print("Word Frequency:", word_frequency(text))
```

41. Knapsack Problem

```
    Solves the 0/1 Knapsack problem using dynamic programming.

    Parameters:
    weights (list): List of item weights.
    values (list): List of item values.
    capacity (int): Maximum weight capacity of the knapsack.

    Returns:
    int: Maximum value that can be obtained.
```

Code:

```
def knapsack(weights, values, capacity):
    n = len(weights)
    dp = [[0] * (capacity + 1) for _ in range(n + 1)]

    # Fill the DP table
    for i in range(1, n + 1):
        for w in range(capacity + 1):
            if weights[i-1] <= w:
                dp[i][w] = max(dp[i-1][w], dp[i-1][w-weights[i-1]] + values[i-1])
            else:
                dp[i][w] = dp[i-1][w]

    return dp[n][capacity]
n = int(input("Enter number of items: "))
weights = list(map(int, input("Enter weights: ").split()))
values = list(map(int, input("Enter values: ").split()))
capacity = int(input("Enter knapsack capacity: "))

print("Maximum value in Knapsack:", knapsack(weights, values, capacity))
```

42. Merge Intervals

Merges overlapping intervals.

Parameters:

intervals (list of lists): List of intervals [start, end].

Returns:

list: Merged list of intervals.

Code:

```
def merge_intervals(intervals):
    if not intervals:
        return []

    # Sort intervals based on the start value
    intervals.sort()

    merged = [intervals[0]]

    for start, end in intervals[1:]:
        last_end = merged[-1][1]

        if start <= last_end:  # Overlapping case
            merged[-1][1] = max(last_end, end)
        else:
            merged.append([start, end])

    return merged
```

```
n = int(input("Enter number of intervals: "))
intervals = [list(map(int, input("Enter interval (start end): ").split())) for _ in range(n)]

print("Merged Intervals:", merge_intervals(intervals))
```

43. Find the Median of Two Sorted Arrays

```
    Finds the median of two sorted arrays.

    Parameters:
    nums1 (list): First sorted list.
    nums2 (list): Second sorted list.

    Returns:
    float: Median of the merged sorted list.
```

Code:

```
def find_median_sorted_arrays(nums1, nums2):
    merged = sorted(nums1 + nums2)
    n = len(merged)
    mid = n // 2

    if n % 2 == 0:
        return (merged[mid - 1] + merged[mid]) / 2
    return merged[mid]
```

```
# Interactive Input
nums1 = list(map(int, input("Enter first sorted list: ").split()))
nums2 = list(map(int, input("Enter second sorted list: ").split()))

print("Median:", find_median_sorted_arrays(nums1, nums2))
```

44. Maximal Rectangle in Binary Matrix

Finds the largest rectangle of 1's in a binary matrix.

Parameters:

matrix (list of lists): 2D binary matrix.

Returns:

int: Maximum area of the rectangle formed by 1's.

Code:

```
def maximal_rectangle(matrix):
    if not matrix:
        return 0

    max_area = 0
    height = [0] * len(matrix[0])

    for row in matrix:
        for i in range(len(row)):
```

```
            height[i] = height[i] + 1 if row[i] == '1' else 0

        stack, area = [], 0
        for i, h in enumerate(height + [0]):
            while stack and height[stack[-1]] > h:
                h_top = height[stack.pop()]
                width = i if not stack else i - stack[-1] - 1
                area = max(area, h_top * width)
            stack.append(i)

        max_area = max(max_area, area)

    return max_area
rows = int(input("Enter number of rows: "))
matrix = [list(input("Enter row (0s & 1s): ")) for _ in range(rows)]

print("Maximal Rectangle Area:", maximal_rectangle(matrix))
```

45. Largest Sum Contiguous Subarray

Finds the largest sum of a contiguous subarray using Kadane's Algorithm.

Parameters:

arr (list): List of integers.

Returns:

int: Maximum sum.

Code:

```
def max_subarray_sum(arr):
    max_sum = arr[0]
    current_sum = arr[0]

    for num in arr[1:]:
        current_sum = max(num, current_sum + num)
        max_sum = max(max_sum, current_sum)

    return max_sum
arr = list(map(int, input("Enter array elements: ").split()))
print("Maximum Subarray Sum:", max_subarray_sum(arr))
```