```
def is_valid_sudoku(board):
    def is_valid_unit(unit):
        nums = [num for num in unit if num != '.']  # Ignore empty cells ('.')
        return len(nums) == len(set(nums))  # Check for duplicates

    # Check Rows
    for row in board:
        if not is_valid_unit(row):
            return False

    # Check Columns
    for col in zip(*board):  # Transposes rows to columns
        if not is_valid_unit(col):
            return False

    # Check 3×3 Subgrids
    for i in range(0, 9, 3):  # Step through rows in increments of 3
        for j in range(0, 9, 3):  # Step through columns in increments of 3
            subgrid = [board[x][y] for x in range(i, i + 3) for y in range(j, j + 3)]
            if not is_valid_unit(subgrid):
                return False

    return True

# Example Sudoku Board
sudoku_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"]
]

print("Is the Sudoku board valid?", is_valid_sudoku(sudoku_board))



from collections import defaultdict


def word_frequency_alt(text):
    text = text.lower()
    text = re.sub(r'[^\w\s]', '', text)
    words = text.split()

    freq = defaultdict(int)
    for word in words:
        freq[word] += 1

    return dict(freq)

print(word_frequency_alt(text))
```

```
def knapsack(weights, values, capacity):
    n = len(weights)
    dp = [0] * (capacity + 1)  # 1D DP array

    for i in range(n):  # Iterate through items
        for w in range(capacity, weights[i] - 1, -1):  # Iterate backwards
            dp[w] = max(dp[w], values[i] + dp[w - weights[i]])

    return dp[capacity]

# Example usage
weights = [2, 3, 4, 5]
values = [3, 4, 5, 6]
capacity = 5
print("Maximum value in knapsack:", knapsack(weights, values, capacity))
```

Merge Intervals

Objective: Merge overlapping intervals in a list of intervals.

Input: A list of intervals where each interval is represented as a pair of integers [start,end][start, end][start,end].

Output: A list of merged intervals.

Hint: Sort the intervals by start time and merge if the start of the current interval is less than or equal to the end of the previous one.

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

    if n % 2 == 0:  # Even length: Average of middle elements
        return (merged[mid - 1] + merged[mid]) / 2
    else:  # Odd length: Middle element
        return merged[mid]

# Example usage
nums1 = [1, 3]
nums2 = [2]
print("Median:", find_median_sorted_arrays(nums1, nums2))  # Output: 2.0
```

```
def find_median_sorted_arrays(nums1, nums2):
    if len(nums1) > len(nums2):  # Ensure nums1 is smaller
        nums1, nums2 = nums2, nums1

    x, y = len(nums1), len(nums2)
    low, high = 0, x

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

        # Edge cases: If partition is at start 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]

        if maxLeftX <= minRightY and maxLeftY <= minRightX:
            # Found correct partition
            if (x + y) % 2 == 0:
                return (max(maxLeftX, maxLeftY) + min(minRightX, minRightY)) / 2
            else:
                return max(maxLeftX, maxLeftY)
        elif maxLeftX > minRightY:
            high = partitionX - 1  # Move left
        else:
            low = partitionX + 1  # Move right

# Example usage
nums1 = [1, 3]
nums2 = [2]
print("Median:", find_median_sorted_arrays(nums1, nums2))  # Output: 2.0
```

```
def max_subarray_sum(nums):
    if not nums:
        return 0  # Edge case: empty list

    current_sum = max_sum = nums[0]

    for i in range(1, len(nums)):
        current_sum = max(nums[i], current_sum + nums[i])  # Extend or restart subarray
        max_sum = max(max_sum, current_sum)  # Update global max

    return max_sum

# Example usage
nums = [-2, 1, -3, 4, -1, 2, 1, -5, 4]
print("Maximum Subarray Sum:", max_subarray_sum(nums))  # Output: 6
```

```
from collections import deque

def word_ladder_length(start, end, word_list):
    word_set = set(word_list)  # Convert list to set for O(1) lookups
    if end not in word_set:
        return 0  # No possible transformation

    queue = deque([(start, 1)])  # (current word, transformation steps)
    while queue:
        word, steps = queue.popleft()

        if word == end:
            return steps  # Found shortest path

        # Generate all possible one-letter transformations
        for i in range(len(word)):
            for c in 'abcdefghijklmnopqrstuvwxyz':
                new_word = word[:i] + c + word[i+1:]  # Change one letter

                if new_word in word_set:
                    queue.append((new_word, steps + 1))
                    word_set.remove(new_word)  # Avoid revisiting

    return 0  # No transformation found

# Example Usage
start_word = "hit"
end_word = "cog"
word_list = ["hot", "dot", "dog", "lot", "log", "cog"]

print("Shortest Transformation Length:", word_ladder_length(start_word, end_word, word_list))  # Output: 5
```