Discrete Math Problem Solution

```
# Problem 1: Relations R1 and R2 on Set A = {1,2,3,4}

def find_relation_pairs(set_A):
    # R1: {(a,b) | a divides b}
    R1 = [(a, b) for a in set_A for b in set_A if b % a == 0]

    # R2: {(a,b) | a ≤ b}
    R2 = [(a, b) for a in set_A for b in set_A if a <= b]

    print("R1 (a divides b):", R1)
    print("R2 (a ≤ b):", R2)

# Test the function
A = [1, 2, 3, 4]
find_relation_pairs(A)

# Problem 2: Relation from Set A to Set B with specific conditions

def find_relation_matrix(A, B, a1, a2, a3, b1, b2):
    # Create the relation R
    R = []
    for a in A:
        for b in B:
            if a > b:
                R.append((a, b))
```

```
    # Create the relation matrix
    relation_matrix = [[0 for _ in range(len(B))] for _ in range(len(A))]

    for a, b in R:
        row_index = A.index(a)
        col_index = B.index(b)
        relation_matrix[row_index][col_index] = 1

    print("Relation R:", R)
    print("Relation Matrix:")
    for row in relation_matrix:
        print(row)

# Test the function with given conditions
A = [1, 2, 3]
B = [1, 2]
a1, a2, a3 = 1, 2, 3
b1, b2 = 1, 2

find_relation_matrix(A, B, a1, a2, a3, b1, b2)

# Problem 3: Graph Coloring by Welch-Powell's Algorithm

def welch_powell_coloring(graph):
    # Sort vertices by degree in descending order
    vertices = sorted(graph.keys(), key=lambda x: len(graph[x]), reverse=True)
```

```
# Initialize colors
colors = {}
color_count = 0

while vertices:
    # Assign a new color
    color_count += 1
    colored_vertices = []

    # Color the first uncolored vertex
    first_vertex = vertices[0]
    colors[first_vertex] = color_count
    colored_vertices.append(first_vertex)

    # Try to color other uncolored vertices
    for vertex in vertices[1:]:
        # Check if this vertex can be colored with the current color
        if all(colors.get(adj, 0) != color_count for adj in graph[vertex]):
            colors[vertex] = color_count
            colored_vertices.append(vertex)

    # Remove colored vertices
    for v in colored_vertices:
        vertices.remove(v)

return colors
```

```
# Example graph representation (adjacency list)
graph = {
    'A': ['B', 'C'],
    'B': ['A', 'C', 'D'],
    'C': ['A', 'B', 'D', 'E'],
    'D': ['B', 'C', 'E', 'F'],
    'E': ['C', 'D'],
    'F': ['D']
}

# Test the algorithm
color_assignment = welch_powell_coloring(graph)
print("Graph Coloring:")
for vertex, color in color_assignment.items():
    print(f"Vertex {vertex}: Color {color}")

# Problem 4: Shortest Path by Warshall's Algorithm

def warshalls_algorithm(adjacency_matrix):
    # Get the number of vertices
    n = len(adjacency_matrix)

    # Create a copy of the adjacency matrix
    dist = [row[:] for row in adjacency_matrix]

    # Warshall's algorithm
    for k in range(n):
```

```
        for i in range(n):
            for j in range(n):
                # If k is an intermediate vertex on the shortest path from i to j
                dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])

    return dist

# Example adjacency matrix (use a large value for no direct connection)
INF = float('inf')
graph = [
    [0, 5, INF, 10],
    [INF, 0, 3, INF],
    [INF, INF, 0, 1],
    [INF, INF, INF, 0]
]

# Test the algorithm
shortest_paths = warshalls_algorithm(graph)
print("Shortest Path Matrix:")
for row in shortest_paths:
    print(row)
# Problem 5: Matrix Operations for Relations M(R1∪R2) and M(R1∩R2)

import numpy as np

def matrix_union_intersection(MR1, MR2):
    # Convert to numpy arrays for easier matrix operations
```

```
    MR1 = np.array(MR1)
    MR2 = np.array(MR2)

    # Matrix Union
    M_union = np.logical_or(MR1, MR2).astype(int)

    # Matrix Intersection
    M_intersection = np.logical_and(MR1, MR2).astype(int)

    print("M(R1 ∪ R2):")
    print(M_union)
    print("\nM(R1 ∩ R2):")
    print(M_intersection)

# Given matrices
MR1 = [
    [1, 0, 1],
    [0, 1, 0],
    [0, 0, 1]
]

MR2 = [
    [1, 0, 1],
    [0, 1, 0],
    [1, 0, 1]
]
```

```
# Test the function
matrix_union_intersection(MR1, MR2)

# Problem 6: Newton-Gregory Forward Interpolation for Population

def newton_gregory_forward(x, years, populations, target_year):
    # Number of data points
    n = len(years)

    # Calculate forward difference table
    diff_table = [[0 for _ in range(n)] for _ in range(n)]

    # First column is the original populations
    for i in range(n):
        diff_table[i][0] = populations[i]

    # Calculate forward differences
    for j in range(1, n):
        for i in range(n - j):
            diff_table[i][j] = diff_table[i+1][j-1] - diff_table[i][j-1]

    # Calculate u and interpolated value
    h = years[1] - years[0]  # uniform interval
    u = (target_year - years[0]) / h

    # Calculate interpolated population
    population = diff_table[0][0]
```

```
    u_term = 1
    factorial = 1

    for j in range(1, n):
        u_term *= (u - j + 1)
        factorial *= j
        population += (u_term / factorial) * diff_table[0][j]

    return population

# Given data
years = [1911, 1921, 1931, 1941, 1951, 1961]
populations = [12, 15, 20, 27, 39, 52]

# Interpolate for 1946
target_year = 1946
interpolated_population = newton_gregory_forward(len(years), years, populations,
target_year)
print(f"Interpolated population in {target_year}: {interpolated_population:.2f}")

# Problem 7: Newton-Gregory Backward Interpolation

def newton_gregory_backward(x, x_values, fx_values, target_x):
    # Number of data points
    n = len(x_values)

    # Calculate backward difference table
```

```
diff_table = [[0 for _ in range(n)] for _ in range(n)]

# First column is the original function values
for i in range(n):
    diff_table[i][0] = fx_values[i]

# Calculate backward differences
for j in range(1, n):
    for i in range(n - j):
        diff_table[i][j] = diff_table[i+1][j-1] - diff_table[i][j-1]

# Calculate u and interpolated value
h = x_values[1] - x_values[0]  # uniform interval
u = (target_x - x_values[-1]) / h

# Calculate interpolated value
value = diff_table[-1][0]
u_term = 1
factorial = 1

for j in range(1, n):
    u_term *= (u + j - 1)
    factorial *= j
    value += (u_term / factorial) * diff_table[-1][j]

return value
```

```
# Given data
x_values = [1, 2, 3, 4, 5, 6, 7, 8]
fx_values = [1, 8, 27, 64, 125, 216, 343, 512]
target_x = 5.5

# Calculate interpolated value
interpolated_value = newton_gregory_backward(len(x_values), x_values, fx_values, target_x)
print(f"Interpolated value at x = {target_x}: {interpolated_value:.2f}")

# Problem 8: Newton's Divided Difference Interpolation Formula

def newton_divided_difference(x_values, fx_values, target_x):
    # Number of data points
    n = len(x_values)

    # Create divided difference table
    divided_diff = [[0 for _ in range(n)] for _ in range(n)]

    # First column is the original function values
    for i in range(n):
        divided_diff[i][0] = fx_values[i]

    # Calculate divided differences
    for j in range(1, n):
        for i in range(n - j):
            divided_diff[i][j] = (divided_diff[i+1][j-1] - divided_diff[i][j-1]) / (x_values[i+j] - x_values[i])
```

```
    # Interpolation calculation
    result = divided_diff[0][0]
    product_term = 1

    for j in range(1, n):
        product_term *= (target_x - x_values[j-1])
        result += divided_diff[0][j] * product_term

    return result

# Given data
x_values = [4, 5, 7, 10, 11, 13]
fx_values = [48, 100, 294, 900, 1210, 2028]
target_x = 15

# Calculate interpolated value
interpolated_value = newton_divided_difference(x_values, fx_values, target_x)
print(f"Interpolated value at x = {target_x}: {interpolated_value:.2f}")

# Problem 9: Lagrange's Interpolation Formula for Unequal Intervals

def lagrange_interpolation(x_values, y_values, target_x):
    # Number of data points
    n = len(x_values)

    # Initialize interpolated value
```

```python
    interpolated_y = 0

    # Lagrange interpolation formula
    for i in range(n):
        # Calculate Lagrange basis polynomial
        basis_poly = 1
        for j in range(n):
            if i != j:
                basis_poly *= (target_x - x_values[j]) / (x_values[i] - x_values[j])

        # Multiply basis polynomial with corresponding y value
        interpolated_y += y_values[i] * basis_poly

    return interpolated_y

# Given data
x_values = [5, 6, 9, 11]
y_values = [12, 13, 14, 16]
target_x = 10

# Calculate interpolated value
interpolated_y = lagrange_interpolation(x_values, y_values, target_x)
print(f"Interpolated y value at x = {target_x}: {interpolated_y:.2f}")

# Problem 10: Bisection Method to Find Real Root

def bisection_method(f, a, b, tolerance=1e-6, max_iterations=100):
```

```
# Check if root is bracketed
if f(a) * f(b) >= 0:
    print("Bisection method fails: No sign change between a and b")
    return None

# Iterations
for iteration in range(max_iterations):
    # Calculate midpoint
    c = (a + b) / 2
    fc = f(c)

    # Print iteration details
    print(f"Iteration {iteration + 1}: a = {a}, b = {b}, c = {c}, f(c) = {fc}")

    # Check if midpoint is the root
    if abs(fc) < tolerance:
        print(f"Root found: {c}")
        return c

    # Update interval
    if f(a) * fc < 0:
        b = c
    else:
        a = c

# If max iterations reached
print("Maximum iterations reached")
```

```
    return (a + b) / 2

# Define the function x^2 - 4x - 10 = 0
def f(x):
    return x**2 - 4*x - 10

# Solve for root between -2 and -1.5
root = bisection_method(f, -2, -1.5)
if root is not None:
    print(f"Approximate root: {root}")
    print(f"Function value at root: {f(root)}")

# Problem 11: False Position Method to Find Root

def false_position_method(f, a, b, tolerance=1e-6, max_iterations=100):
    # Check if root is bracketed
    if f(a) * f(b) >= 0:
        print("False Position method fails: No sign change between a and b")
        return None

    # Iterations
    for iteration in range(max_iterations):
        # Calculate intersection point with x-axis using False Position formula
        c = (a * f(b) - b * f(a)) / (f(b) - f(a))
        fc = f(c)

        # Print iteration details
```

```
        print(f"Iteration {iteration + 1}: a = {a}, b = {b}, c = {c}, f(c) = {fc}")

        # Check if root is found
        if abs(fc) < tolerance:
            print(f"Root found: {c}")
            return c

        # Update interval
        if f(a) * fc < 0:
            b = c
        else:
            a = c

    # If max iterations reached
    print("Maximum iterations reached")
    return (a + b) / 2

# Define the function x^2 - x - 2 = 0
def f(x):
    return x**2 - x - 2

# Solve for root between 1 and 3
root = false_position_method(f, 1, 3)
if root is not None:
    print(f"Approximate root: {root}")
    print(f"Function value at root: {f(root)}")
```