

## Solution 4

```

import random

# Parameters for the problem
item_values = [12, 16, 22, 8] # Item values
item_weights = [4, 5, 7, 3] # Item weights
knapsack_capacity = 140 # Maximum capacity of the knapsack
total_items = len(item_values) # Count of items
max_item_count = 10 # Max count of each item

# Function for Greedy Approach
def calculate_greedy_solution():
    ratio_value_weight = [val/wt for val, wt in zip(item_values, item_weights)]
    sorted_items = sorted(range(total_items), key=lambda idx: ratio_value_weight[idx], reverse=True)

    knapsack_contents = [0] * total_items
    current_weight = 0

    for item in sorted_items:
        while current_weight + item_weights[item] <= knapsack_capacity and knapsack_contents[item] < max_item_count:
            knapsack_contents[item] += 1
            current_weight += item_weights[item]

    return knapsack_contents

# Function to Compute Total Value of Knapsack
def compute_knapsack_value(contents):
    return sum(value * quantity for value, quantity in zip(item_values, contents))

# Function for Metaheuristic - Iterative Optimization
def perform_metaheuristic(num_iterations):
    optimal_contents = calculate_greedy_solution()
    optimal_value = compute_knapsack_value(optimal_contents)
    encountered_solutions = set()
    encountered_solutions.add(str(optimal_contents))

    for _ in range(num_iterations):
        temp_contents = optimal_contents[:]
        random_item = random.randint(0, total_items - 1)
        temp_contents[random_item] = random.randint(0, max_item_count)

        if sum(wt * qty for wt, qty in zip(item_weights, temp_contents)) > knapsack_capacity:
            temp_contents = calculate_greedy_solution()


        temp_value = compute_knapsack_value(temp_contents)

        temp_contents_str = str(temp_contents)
        if temp_value > optimal_value and temp_contents_str not in encountered_solutions:
            optimal_contents = temp_contents
            optimal_value = temp_value
            encountered_solutions.add(temp_contents_str)

    return optimal_contents, optimal_value

# Execute the Metaheuristic Approach
num_iterations = 100
optimal_solution, highest_value = perform_metaheuristic(num_iterations)
print("Optimal Knapsack Contents:", optimal_solution)
print("Maximum Value Achieved:", highest_value)

```

 Optimal Knapsack Contents: [5, 10, 10, 0]  
Maximum Value Achieved: 440

## Solution 5

```

import numpy as np
import random
import math

# Knapsack configuration
item_values = np.array([12, 16, 22, 8]) # Individual item values
item_weights = np.array([4, 5, 7, 3]) # Individual item weights
knapsack_capacity = 140 # Maximum weight capacity of the knapsack
item_limit = 10 # Maximum number of each item
total_items = len(item_values) # Total number of distinct items

```

```

# Function to create an initial solution
def create_initial_solution():
    solution_vector = np.random.randint(0, item_limit + 1, size=total_items)
    while np.sum(solution_vector * item_weights) > knapsack_capacity:
        solution_vector = np.random.randint(0, item_limit + 1, size=total_items)
    return solution_vector

# Calculate the total value of items in the knapsack
def calculate_value(solution_vector):
    return np.dot(solution_vector, item_values)

# Generate a neighboring solution
def find_neighbor(solution_vector):
    neighbor_solution = solution_vector.copy()
    selected_item = random.randint(0, total_items - 1)
    modification = random.randint(-1, 1)
    neighbor_solution[selected_item] = max(0, min(item_limit, neighbor_solution[selected_item] + modification))

    # Ensure the neighbor solution is valid
    while np.sum(neighbor_solution * item_weights) > knapsack_capacity:
        neighbor_solution[selected_item] = max(0, neighbor_solution[selected_item] - 1)

    return neighbor_solution

# Calculate the probability of accepting a new solution
def calculate_acceptance(old_value, new_value, temp):
    if new_value > old_value:
        return 1.0
    return math.exp((new_value - old_value) / temp)

# The main Simulated Annealing algorithm
def perform_simulated_annealing(start_temp, end_temp, cooling_factor, iterations_limit):
    current_solution = create_initial_solution()
    current_value = calculate_value(current_solution)
    temp = start_temp

    optimal_solution = current_solution.copy()
    optimal_value = current_value

    for _ in range(iterations_limit):
        neighbor_solution = find_neighbor(current_solution)
        neighbor_value = calculate_value(neighbor_solution)

        # Decision to accept the new solution
        if calculate_acceptance(current_value, neighbor_value, temp) > random.random():
            current_solution = neighbor_solution
            current_value = neighbor_value

        # Update the best solution
        if neighbor_value > optimal_value:
            optimal_solution = neighbor_solution
            optimal_value = neighbor_value

        # Cooling down the temperature
        temp *= cooling_factor

        # Check for termination
        if temp < end_temp:
            break

    return optimal_solution, optimal_value

# Configuration for the algorithm
start_temp = 10000
end_temp = 1
cooling_factor = 0.99 # Rate of cooling
iterations_limit = 1000

# Execute the algorithm
optimal_solution, optimal_value = perform_simulated_annealing(start_temp, end_temp, cooling_factor, iterations_limit)
print("Optimal Solution:", optimal_solution)
print("Optimal Value:", optimal_value)

Optimal Solution: [10 5 9 4]
Optimal Value: 430

```

