Challenge Name: The Weight of Shadows

Category: Cryptography

Overview:

In this cryptography challenge, we work with a concept called the **super-increasing knapsack**, combined with modular arithmetic. The goal is to decrypt a series of numbers to reveal a hidden flag. This write-up breaks down the steps to solve it using simple concepts and Python code.

Step 1: Understanding the Super-Increasing Knapsack

A **super-increasing knapsack** is a sequence of numbers where each number is larger than the sum of all the previous numbers. This ensures that each number can be uniquely represented by a subset of the sequence.

In this challenge, we're given the first two weights as 1 and 2, and after the fourth weight, the pattern changes:

- For the first four weights:
  w[i] = sum of all previous weights + 1
  So, we get:
  w = [1, 2, 4, 8]
- For the next weights, each is double the previous one:
  w[i] = 2 * w[i-1]
  This gives us:
  w = [1, 2, 4, 8, 16, 32, 64, 128]

This is the full knapsack sequence we'll use for decryption.

Step 2: Modular Arithmetic and Decryption

The encrypted message uses **modular arithmetic** with the following elements:

- **Modular Base**: A prime number m between 250 and 260, which we set as m = 257.
- **Multiplier n**: A secret integer less than m. The inverse of n under modulo m (denoted n^-1) is key for decryption.

The decryption formula we'll use is:

decrypted_value = (encrypted_value * n^-1) % m

Step 3: Decrypting the Ciphertext

To decrypt the message, follow these steps:

1. **Find the Modular Inverse**: Use the Extended Euclidean Algorithm or brute force to compute the inverse of n (i.e., n^-1).
2. **Decrypt Each Value**: For each number in the encrypted message:
   - Compute the decrypted value using the formula: decrypted_value = (encrypted_value * n^-1) % m
   - Convert the decrypted value into binary by checking which weights from the knapsack are used.
   - Group the binary values into bytes and convert them to ASCII to reconstruct the plaintext.

Solution Code

Here's a simple Python implementation to solve the challenge:

```
def superincreasing_knapsack():
    knapsack = [1, 2]
    for i in range(2, 8):
        knapsack.append(sum(knapsack) + 1)
    return knapsack


def modular_inverse(a, m):
    for x in range(1, m):
        if (a * x) % m == 1:
            return x
    return None


def bits_to_string(bits):
```

```
    chars = []
    for b in range(0, len(bits), 8):
        byte = bits[b:b+8]
        char = chr(int(''.join(map(str, byte)), 2))
        chars.append(char)
    return ''.join(chars)


def decrypt_message(encrypted, knapsack, m):
    for n in range(1, m):
        n_inverse = modular_inverse(n, m)
        if n_inverse is None:
            continue
        decrypted_bits = []
        for number in encrypted:
            decrypted_value = (number * n_inverse) % m
            bits = []
            for weight in reversed(knapsack):
                if decrypted_value >= weight:
                    bits.append(1)
                    decrypted_value -= weight
                else:
                    bits.append(0)
            decrypted_bits.extend(reversed(bits))
        decrypted_string = bits_to_string(decrypted_bits)
        if "CTF{" in decrypted_string:
            return decrypted_string
    raise ValueError("No valid modular inverse found that produces the expected format.")
```

```
# Given data

encrypted = [490, 616, 920, 801, 340, 505, 456, 970, 634, 580, 652, 421, 1062, 520, 1062, 342, 690, 1062, 290, 614, 662, 1062, 882, 410, 662, 1062, 362, 662, 726, 520, 1062, 480, 360, 630, 362, 360, 234, 234, 1005]

m = 257


# Solve the challenge

knapsack = superincreasing_knapsack()

try:
  flag = decrypt_message(encrypted, knapsack, m)
  print("Decrypted flag:", flag)

except ValueError as e:
  print("Error:", e)
```

Step 4: The Final Output

After running the script, the decrypted flag is:

CTF{M4th_&_C5_ar3_7h3_b3sT_c0Mb0!!}

Key Insights:

- The **super-increasing knapsack** allows for unique binary representation, which simplifies the decryption process.
- **Modular inverses** are crucial in unlocking the encrypted message, showing the elegance of modular arithmetic in cryptography.
- This challenge not only tests cryptographic knowledge but also pushes critical thinking and problem-solving skills.