**1. Write a Python script that takes two integers as input and calculates their GCD using the Euclidean algorithm.**

**Based on the result, determine whether these numbers are co-prime.**

**If they are co-prime, print a message indicating that they can be used in cryptographic key generation; otherwise, print a message that they are not suitable.**

def gcd\_euclidean(a, b):

"""Calculate the Greatest Common Divisor of a and b using the Euclidean algorithm."""

while b:

a, b = b, a % b

return a

def are\_coprime(a, b):

"""Check if two numbers are co-prime."""

return gcd\_euclidean(a, b) == 1

def check\_cryptographic\_suitability(a, b):

"""Check if two numbers are suitable for cryptographic key generation."""

gcd = gcd\_euclidean(a, b)

coprime = gcd == 1

print(f"The GCD of {a} and {b} is: {gcd}")

if coprime:

print(f"{a} and {b} are co-prime.")

print("These numbers are suitable for use in cryptographic key generation.")

else:

print(f"{a} and {b} are not co-prime.")

print("These numbers are not suitable for use in cryptographic key generation.")

# Get input from the user

num1 = int(input("Enter the first integer: "))

num2 = int(input("Enter the second integer: "))

# Check suitability for cryptographic key generation

check\_cryptographic\_suitability(num1, num2)

**2. Write a python script to take two integer values (number (n) and modulo (m)) from the user and find the modular inverse using extended Euclidean algorithm.**

def extended\_euclidean(a, b):

"""

Extended Euclidean Algorithm

Returns (gcd, x, y) such that a \* x + b \* y = gcd

"""

if a == 0:

return b, 0, 1

else:

gcd, x, y = extended\_euclidean(b % a, a)

return gcd, y - (b // a) \* x, x

def modular\_inverse(n, m):

"""

Calculates the modular inverse of n modulo m using Extended Euclidean Algorithm

Returns the inverse if it exists, or None if it doesn't

"""

gcd, x, \_ = extended\_euclidean(n, m)

if gcd != 1:

return None # Modular inverse doesn't exist

else:

return x % m

# Get input from the user

n = int(input("Enter the number (n): "))

m = int(input("Enter the modulo (m): "))

# Calculate the modular inverse

inverse = modular\_inverse(n, m)

# Print the result

if inverse is None:

print(f"The modular inverse of {n} modulo {m} does not exist.")

else:

print(f"The modular inverse of {n} modulo {m} is: {inverse}")

# Verification

print(f"Verification: ({n} \* {inverse}) mod {m} = {(n \* inverse) % m}")

**3. Write a Python script that generates a random binary number of length 100. The output should be a string of 100 binary digits (0s and 1s).**

**After generating the binary sequence, implement a function to check whether any subsequence of digits repeats itself within the sequence.**

import random

def generate\_random\_binary(length):

"""Generate a random binary sequence of specified length."""

return ''.join(random.choice('01') for \_ in range(length))

def find\_repeating\_subsequence(binary\_string):

"""

Check if any subsequence repeats within the binary string.

Returns a tuple (bool, str) where bool indicates if a repeat was found,

and str is the repeating subsequence (or empty string if none found).

"""

n = len(binary\_string)

for length in range(2, n // 2 + 1): # Check subsequences up to half the string length

for i in range(n - length + 1):

subsequence = binary\_string[i:i+length]

if binary\_string.count(subsequence) > 1:

return True, subsequence

return False, ""

# Generate a random 100-bit binary number

binary\_sequence = generate\_random\_binary(100)

print("Generated 100-bit binary sequence:")

print(binary\_sequence)

# Check for repeating subsequences

has\_repeat, repeating\_sequence = find\_repeating\_subsequence(binary\_sequence)

if has\_repeat:

print(f"\nRepeating subsequence found: {repeating\_sequence}")

print(f"Length of repeating subsequence: {len(repeating\_sequence)}")

else:

print("\nNo repeating subsequences found.")

# Additional analysis

print(f"\nNumber of 0s: {binary\_sequence.count('0')}")

print(f"Number of 1s: {binary\_sequence.count('1')}")

**4. Write a Python script that performs the Golomb test to the numbers provided below.**

**101011001010**

**111111000000**

**The script should**

**- Perform and print the results of the three Golomb tests on the sequence.**

**- Print a message indicating whether the sequence passes the Golomb tests or not.**

def count\_runs(sequence):

"""Count the number of runs of 0s and 1s in the sequence."""

runs = []

current\_run = 1

for i in range(1, len(sequence)):

if sequence[i] == sequence[i-1]:

current\_run += 1

else:

runs.append(current\_run)

current\_run = 1

runs.append(current\_run)

return runs

def golomb\_test(sequence):

n = len(sequence)

ones = sequence.count('1')

zeros = sequence.count('0')

runs = count\_runs(sequence)

# Test 1: Balance property

balance = abs(ones - zeros) <= 1

print(f"Test 1 (Balance property): {'Passed' if balance else 'Failed'}")

print(f" Number of 1s: {ones}")

print(f" Number of 0s: {zeros}")

# Test 2: Run property

expected\_runs = (n + 1) // 2

actual\_runs = len(runs)

run\_property = abs(actual\_runs - expected\_runs) <= 2

print(f"Test 2 (Run property): {'Passed' if run\_property else 'Failed'}")

print(f" Expected number of runs: {expected\_runs}")

print(f" Actual number of runs: {actual\_runs}")

# Test 3: Run length property

run\_lengths = {i: runs.count(i) for i in range(1, max(runs) + 1)}

run\_length\_property = True

for k in range(1, len(run\_lengths)):

if k+1 in run\_lengths:

if run\_lengths[k] < run\_lengths[k+1]:

run\_length\_property = False

break

print(f"Test 3 (Run length property): {'Passed' if run\_length\_property else 'Failed'}")

print(" Run lengths:")

for length, count in run\_lengths.items():

print(f" Length {length}: {count} runs")

# Overall result

passed\_all = balance and run\_property and run\_length\_property

print(f"\nOverall result: {'Passed' if passed\_all else 'Failed'} all Golomb tests")

return passed\_all

# Test sequences

sequences = [

"101011001010",

"111111000000"

]

for i, seq in enumerate(sequences, 1):

print(f"\nTesting sequence {i}: {seq}")

print("-" \* 50)

golomb\_test(seq)

print("\n")