**LAB 8**

**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(a, b):

while b != 0:

a, b = b, a % b

return a

def main():

try:

# Take input from the user

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

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

# Calculate GCD

result\_gcd = gcd(num1, num2)

# Check if they are co-prime

if result\_gcd == 1:

print(f"The GCD of {num1} and {num2} is {result\_gcd}.")

print("These numbers are co-prime and can be used in cryptographic key generation.")

else:

print(f"The GCD of {num1} and {num2} is {result\_gcd}.")

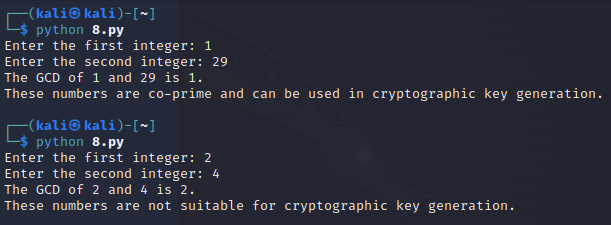
print("These numbers are not suitable for cryptographic key generation.")

except ValueError:

print("Please enter valid integers.")

if \_\_name\_\_ == "\_\_main\_\_":

main()

****

**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\_gcd(a, b):

if a == 0:

return b, 0, 1

gcd, x1, y1 = extended\_gcd(b % a, a)

x = y1 - (b // a) \* x1

y = x1

return gcd, x, y

def modular\_inverse(n, m):

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

if gcd != 1:

return None # Inverse doesn't exist

else:

return x % m # Ensure the result is positive

def main():

try:

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

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

inv = modular\_inverse(n, m)

if inv is None:

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

else:

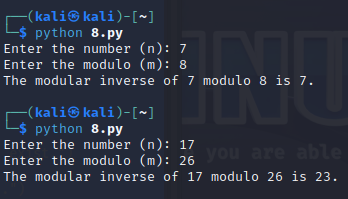
print(f"The modular inverse of {n} modulo {m} is {inv}.")

except ValueError:

print("Please enter valid integers.")

if \_\_name\_\_ == "\_\_main\_\_":

main()



**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):

"""Generates a random binary string of specified length."""

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

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

"""Checks if there are any repeating subsequences in the binary string."""

n = len(binary\_string)

seen\_subsequences = set()

# Check for all possible lengths of subsequences

for length in range(1, n // 2 + 1):

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

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

if subseq in seen\_subsequences:

return True # Found a repeating subsequence

seen\_subsequences.add(subseq)

return False # No repeating subsequence found

def main():

# Generate random binary number of length 100

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

print(f"Generated binary sequence: {binary\_sequence}")

# Check for repeating subsequences

if has\_repeating\_subsequence(binary\_sequence):

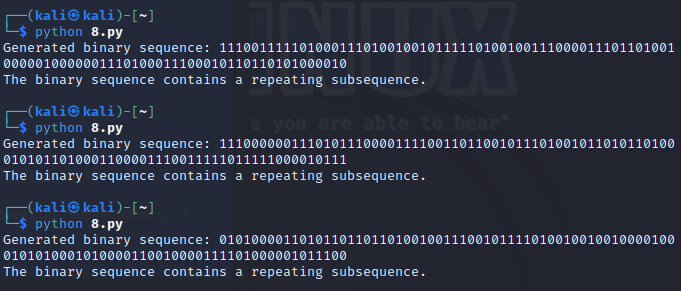
print("The binary sequence contains a repeating subsequence.")

else:

print("The binary sequence does not contain any repeating subsequences.")

if \_\_name\_\_ == "\_\_main\_\_":

main()



**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 golomb\_tests(binary\_sequence):

# Test 1: Count of 1's vs 0's

count\_1s = binary\_sequence.count('1')

count\_0s = binary\_sequence.count('0')

# Test 2: Check for repeating subsequences of length 2 or more

def has\_repeating\_subsequence(seq):

seen = set()

n = len(seq)

for length in range(2, n // 2 + 1): # Check for lengths from 2 to n//2

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

subseq = seq[i:i + length]

if subseq in seen:

return True # Found a repeating subsequence

seen.add(subseq)

return False

has\_repeats = has\_repeating\_subsequence(binary\_sequence)

# Test 3: Check for direct patterns (like alternating bits)

is\_patterned = '01' in binary\_sequence or '10' in binary\_sequence

# Print results of tests

print(f"Testing sequence: {binary\_sequence}")

print(f"Count of 1's: {count\_1s}, Count of 0's: {count\_0s}")

print(f"Has repeating subsequences: {has\_repeats}")

print(f"Has a direct pattern: {is\_patterned}")

# Determine if the sequence passes the Golomb tests

if count\_1s != count\_0s and not has\_repeats and not is\_patterned:

print("The sequence passes the Golomb tests.")

else:

print("The sequence does not pass the Golomb tests.")

def main():

sequences = [

"101011001010",

"111111000000",

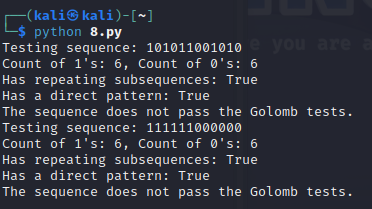
]

for seq in sequences:

golomb\_tests(seq)

if \_\_name\_\_ == "\_\_main\_\_":

main()



**================================================================================================**

**"Always do your best. What you plant now, you will harvest later." —Og Mandino**

**================================================================================================**