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.

```python
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)
```

Output:

```
┌──(kali㉿kali)-[~]
└─$ vi lab8(1).py

┌──(kali㉿kali)-[~]
└─$ python3 lab8(1).py
Enter the first integer: 2
Enter the second integer: 3
The GCD of 2 and 3 is: 1
2 and 3 are co-prime.
These numbers are suitable for use in cryptographic key generation.
```

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.

```python
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}")
~
~
```

Output:

```
┌──(kali㉿kali)-[~/CYS/HYM]
└─$ vi lab8(2).py

┌──(kali㉿kali)-[~/CYS/HYM]
└─$ python3 lab8(2).py
Enter the number (n): 8
Enter the modulo (m): 19
The modular inverse of 8 modulo 19 is: 12
Verification: (8 * 12) mod 19 = 1
```

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.

```python
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')}")
```

Output:

```
┌──(kali㉿kali)-[~/CYS/HYM]
└─$ vi lab8(3).py

┌──(kali㉿kali)-[~/CYS/HYM]
└─$ python3 lab8(3).py
Generated 100-bit binary sequence:
1000010100100101010111101111101000100011110011101010100000010110110000100101100101000111111100010 11

Repeating subsequence found: 10
Length of repeating subsequence: 2

Number of 0s: 50
Number of 1s: 50
```

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.

```python
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")
-- INSERT --
```

Output:

```
┌──(kali㉿kali)-[~/CYS/HYM]
└─$ vi lab8(4).py

┌──(kali㉿kali)-[~/CYS/HYM]
└─$ python3 lab8(4).py

Testing sequence 1: 101011001010
──────────────────────────────────────────────────
Test 1 (Balance property): Passed
  Number of 1s: 6
  Number of 0s: 6
Test 2 (Run property): Failed
  Expected number of runs: 6
  Actual number of runs: 10
Test 3 (Run length property): Passed
  Run lengths:
    Length 1: 8 runs
    Length 2: 2 runs

Overall result: Failed all Golomb tests



Testing sequence 2: 111111000000
──────────────────────────────────────────────────
Test 1 (Balance property): Passed
  Number of 1s: 6
  Number of 0s: 6
Test 2 (Run property): Failed
  Expected number of runs: 6
  Actual number of runs: 2
Test 3 (Run length property): Failed
  Run lengths:
    Length 1: 0 runs
    Length 2: 0 runs
    Length 3: 0 runs
    Length 4: 0 runs
    Length 5: 0 runs
    Length 6: 2 runs

Overall result: Failed all Golomb tests
```