21CSE1003 | ASHISH SINGH

Q1. Write a program to list all Zn which is a field under addition and multiplication in the range of 2 to 100.

1. Initialize Parameters:

- Define the function is_prime(num) which checks if a number num is a prime number.
- Define the function prime_powers(a, b) which finds all prime numbers and their positive powers between a and b.
- Set the range values x and y (e.g., x = 2, y = 100).

2. Check for Prime Numbers:

- **Function is_prime(num)**:
    1. If num is less than or equal to 1, return False.
    2. Iterate through all integers i from 2 to the square root of num (rounded up) plus 1.
    3. If num is divisible by any i in this range, return False (indicating num is not prime).
    4. If no divisors are found, return True (indicating num is prime).

3. Find Primes and Powers of Primes:

- **Function prime_powers(a, b)**:
    1. Initialize two empty sets, primes, and powers_of_primes.
    2. Loop through each number num from a to b (inclusive).
    3. For each num, check if it is prime using is_prime(num):
        - If num is prime, add it to the primes set.
        - Calculate the powers of the prime by multiplying it by itself iteratively, as long as the power is within the range [a, b].
        - Add each calculated power to the powers_of_primes set.
    4. Return the union of the primes set and the powers_of_primes set.

4. Execute the Main Program:

- Call the function prime_powers(x, y) with the specified range (x = 2, y = 100).
- Store the result in variable p.

5. Output the Result:

- Print the set p, which contains all prime numbers and their positive powers within the specified range [x, y].

LAB_4\all_Zn_fields.py

```python

# Ashish Singh
# 21CSE1003

# Q1. Write a program to list all Zn which is a field under addition and multiplication
# in the range of 2 to 100.

def is_prime(num):
    if num <= 1:
        return False
    for i in range(2, int(num**0.5) + 1):
        if num % i == 0:
            return False
    return True

def prime_powers(a, b):
    primes = set()
    powers_of_primes = set()
    for num in range(a, b + 1):
        if is_prime(num):
            primes.add(num)
            power = num
            while power <= b:
                powers_of_primes.add(power)
                power *= num
    return primes | powers_of_primes

x, y = 2, 100

p = prime_powers(x, y)

print(f"List of Zn which is field: {p}.")

```

Q2. Write a program to find the list of prime field and extension field in the range of 2 to 200.

1. Initialize Parameters:

- Define the function is_prime(num) which checks if a number num is a prime number.
- Define the function prime_powers(a, b) which finds all prime numbers and their positive powers between a and b.
- Set the range values x and y (e.g., x = 2, y = 100).

2. Check for Prime Numbers:

- **Function is_prime(num)**:
    1. If num is less than or equal to 1, return False.
    2. Iterate through all integers i from 2 to the square root of num (rounded up) plus 1.
    3. If num is divisible by any i in this range, return False (indicating num is not prime).
    4. If no divisors are found, return True (indicating num is prime).

3. Find Primes and Powers of Primes:

- **Function prime_powers(a, b)**:
    1. Initialize two empty sets, primes and powers_of_primes.
    2. Loop through each number num from a to b (inclusive).
    3. For each num, check if it is prime using is_prime(num):
        - If num is prime, add it to the primes set.
        - Calculate the powers of the prime by multiplying it by itself iteratively, as long as the power is within the range [a, b].
        - Add each calculated power to the powers_of_primes set.
    4. Calculate the extension_fields by subtracting the primes set from the powers_of_primes set (i.e., powers_of_primes - primes).
    5. Return the primes set and the extension_fields set.

4. Execute the Main Program:

- Call the function prime_powers(x, y) with the specified range (x = 2, y = 100).
- Store the results in variables p and q, where:
    - p represents the prime fields.
    - q represents the extension fields (powers of primes but not primes themselves).

5. Output the Results:

- Print the p set, which contains all prime numbers within the specified range [x, y].
- Print the q set, which contains all numbers that are powers of primes but not primes themselves within the specified range [x, y].

LAB_4\prime_extension_field.py

```python
# Q2. Write a program to find the list of prime field and extension field
# in the range of 2 to 200.

def is_prime(num):
    if num <= 1:
        return False
    for i in range(2, int(num**0.5) + 1):
        if num % i == 0:
            return False
    return True

def prime_powers(a, b):
    primes = set()
    powers_of_primes = set()
    for num in range(a, b + 1):
        if is_prime(num):
            primes.add(num)
            power = num
            while power <= b:
                powers_of_primes.add(power)
                power *= num
    return primes, powers_of_primes - primes # extension fields is power of primes but not actual primes

x, y = 2, 100

p, q = prime_powers(x, y)

print(f"Prime fields: {p}.")
print(f"Extension fields: {q}")

```

Q3. Write a program to find the primitive root of GF(n) where n is a prime number of powers 1.

1. Input:

- Prompt the user to enter a prime number n.
- Store the input value in the variable n.

2. Initialize:

- Create a list s containing all integers from 1 to n-1.
- Initialize a flag primitive_root_found and set it to False. This flag will be used to track whether a primitive root is found.

3. Loop Through Potential Primitive Roots:

- Start a loop with a variable j ranging from 1 to n-1.

4. Skip Even Numbers:

- Within the loop, check if j is even (i.e., j % 2 == 0).
- If j is even, use continue to skip to the next iteration of the loop.

5. Check If j is a Primitive Root:

- If j is odd, create a copy of list s and store it in a.
- Initialize a counter m to 0.
- While the value (j ** m) % n exists in the list a and the length of a is greater than or equal to 0, perform the following:
    - Remove the value (j ** m) % n from a.
    - Increment m by 1.

6. Determine Primitive Root:

- After the while loop, check if the length of list a is 0:
    - If a is empty, it indicates that j is a primitive root of GF(n).
    - Print that j is the primitive root and set primitive_root_found to True.
    - Break out of the loop since a primitive root has been found.

7. Handle Case When No Primitive Root is Found:

- After the loop, check if primitive_root_found is still False:
    - If no primitive root was found, print "Primitive root does not exist".

LAB_4\primitive_root.py

```python

# Q3. Write a program to find the primitive root of GF(n) where n is
# a prime number of powers 1.

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

s = [i for i in range(1, n)]

for j in range(1, n):
    a = s.copy()
    if j % 2 == 0:
        continue
    else:
        m = 0
        while (j ** m) % n in a and len(a) >= 0:
            a.remove((j ** m) % n)
            m += 1
    if len(a) == 0:
        print(f"Primitive root of GF({n}) is: {j}")
        break
    else:
        print("Primitive root does not exist")

```

Q4. Perform addition and multiplication operations on GF(16) and find the additive and multiplicative inverse of each element present in GF(16).

1. Initialize Parameters:

- Define an irreducible polynomial for GF(16) as a binary value 0b10011, which corresponds to x4+x+1x^4 + x + 1x4+x+1.

2. Define Functions:

2.1 Addition in GF(16):

- Function add_GF16(a, b):
    - Perform addition using XOR (bitwise exclusive OR) operation.
    - Return the result of a ^ b.

2.2 Multiplication in GF(16):

- Function multiply_GF16(a, b):
    - Initialize the result to 0.
    - While b is greater than 0:
        - If the least significant bit of b (i.e., b & 1) is 1:
            - XOR result with a (i.e., result ^= a).
        - Shift a to the left by 1 (i.e., a <<= 1).
        - If a has more than 4 bits (i.e., a & 0b10000), reduce a by XORing it with the irreducible polynomial (i.e., a ^= irreducible_poly).
        - Shift b to the right by 1 (i.e., b >>= 1).
    - Return result.

2.3 Additive Inverse in GF(16):

- Function additive_inverse_GF16(a):
    - In GF(16), the additive inverse is the number itself. Return a.

2.4 Multiplicative Inverse in GF(16):

- Function multiplicative_inverse_GF16(a):
    - If a is 0, raise an error indicating no inverse exists.
    - For each integer i from 1 to 15:
        - Compute the product of a and i using multiply_GF16(a, i).
        - If the result is 1, i is the multiplicative inverse of a. Return i.
    - If no multiplicative inverse is found, raise an error indicating that no inverse exists for a.

3. Generate GF(16) Elements:

- Create a list GF16_elements containing all integers from 0 to 15.

4. Compute and Print Results:

4.1 Addition Table:

- Print a table of addition for all pairs (i, j) in GF16_elements using add_GF16(i, j).

4.2 Multiplication Table:

- Print a table of multiplication for all pairs (i, j) in GF16_elements using multiply_GF16(i, j).

4.3 Additive Inverses:

- For each element in GF16_elements, print its additive inverse using additive_inverse_GF16(i).

4.4 Multiplicative Inverses:

- For each non-zero element in GF16_elements, print its multiplicative inverse using multiplicative_inverse_GF16(i).

LAB_4\operation_on_GF(n).py

```python
# Q4. Perform addition and multiplication operation on GF(16) and finds additive
# and multiplicative inverse of each element present in GF(16).
# for GF(16)

irreducible_poly = 0b10011

def add_GF16(a, b):
    return a ^ b

def multiply_GF16(a, b):
    result = 0
    while b > 0:
        if b & 1:
            result ^= a
        a <<= 1
        if a & 0b10000:  # If degree is greater than or equal to 4
            a ^= irreducible_poly
        b >>= 1
    return result

def additive_inverse_GF16(a): return a

def multiplicative_inverse_GF16(a):
    if a == 0:
        raise ValueError("0 has no multiplicative inverse in GF(16).")
    for i in range(1, 16):
        if multiply_GF16(a, i) == 1:
            return i
    raise ValueError(f"No multiplicative inverse found for {a} in GF(16).")

GF16_elements = list(range(16))

print("Addition table for GF(16):")
for i in GF16_elements:
    for j in GF16_elements: print(f"{i} + {j} = {add_GF16(i, j)}")

print("\nMultiplication table for GF(16):")
for i in GF16_elements:
    for j in GF16_elements: print(f"{i} * {j} = {multiply_GF16(i, j)}")

print("\nAdditive inverses in GF(16):")
for i in GF16_elements: print(f"Additive inverse of {i} is {additive_inverse_GF16(i)}")

print("\nMultiplicative inverses in GF(16):")
for i in GF16_elements:
    if i != 0:
        print(f"Multiplicative inverse of {i} is {multiplicative_inverse_GF16(i)}")

```

Q5. Find the multiplicative inverse of 95 in GF(128).

1. Initialize Parameters:

- Define an irreducible polynomial for GF(128) as a binary value 0b10000011, which corresponds to x7+x2+1x^7 + x^2 + 1x7+x2+1.

2. Define Functions:

2.1 Multiplication in GF(128):

- Function multiply_GF128(a, b):
  - Initialize result to 0.
  - While b is greater than 0:
    - If the least significant bit of b (i.e., b & 1) is 1:
      - XOR result with a (i.e., result ^= a).
    - Shift a to the left by 1 (i.e., a <<= 1).
    - If a has more than 7 bits (i.e., a & 0b10000000), reduce a by XORing it with the irreducible polynomial (i.e., a ^= irreducible_poly).
    - Shift b to the right by 1 (i.e., b >>= 1).
  - Return result.

2.2 Multiplicative Inverse in GF(128):

- Function multiplicative_inverse_GF128(a):
  - If a is 0, raise an error indicating no inverse exists.
  - For each integer i from 1 to 127:
    - Compute the product of a and i using multiply_GF128(a, i).
    - If the result is 1, i is the multiplicative inverse of a. Return i.
  - If no multiplicative inverse is found, raise an error indicating that no inverse exists for a.

3. Compute and Print Multiplicative Inverse:

- Set a to 95.
- Call multiplicative_inverse_GF128(a) to find the inverse and store it in the inverse.
- Print the result indicating the multiplicative inverse of an in GF(128).

LAB_4\multiplicative_inverse_in_GF.py

```python

# Q5. Find multiplicative inverse of 95 in GF(128).

irreducible_poly = 0b10000011

def multiply_GF128(a, b):
    result = 0
    while b > 0:
        if b & 1:
            result ^= a
        a <<= 1
        if a & 0b10000000:  # If degree is greater than or equal to 7
            a ^= irreducible_poly
        b >>= 1
    return result

def multiplicative_inverse_GF128(a):
    if a == 0:
        raise ValueError("0 has no multiplicative inverse in GF(128).")
    for i in range(1, 128):
        if multiply_GF128(a, i) == 1:
            return i
    raise ValueError(f"No multiplicative inverse found for {a} in GF(128).")

a = 95
inverse = multiplicative_inverse_GF128(a)

print(f"The multiplicative inverse of {a} in GF(128) is {inverse}.")

```