## Name: **Ketan Kunkalikar**

## Roll: **21CSE1016**

## Lab date: **4th September 2024**

## Lab instructor: **Dr. Meenakshi Panda**

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

#include <bits/stdc++.h>

using namespace std;

bool isGroup(vector<int> zn, int n)

{

    // closure

    for (int i = 0; i < zn.size(); i++)

    {

        for (int j = 0; j < zn.size(); j++)

        {

            if ((zn[i] + zn[j]) % n >= n || (zn[i] + zn[j]) % n < 0)

            {

                return false;

            }

        }

    }

    // associativity

    for (int i = 0; i < zn.size(); i++)

    {

        for (int j = 0; j < zn.size(); j++)

        {

            for (int k = 0; k < zn.size(); k++)

            {

                if (((zn[i] + zn[j]) % n + zn[k]) % n != (zn[i] + (zn[j] + zn[k]) % n) % n)

                {

                    return false;

                }

            }

        }

    }

    // identity element

    bool hasIdentity = false;

    for (int i = 0; i < zn.size(); i++)

    {

        bool isIdentity = true;

        for (int j = 0; j < zn.size(); j++)

        {

            if ((zn[i] + zn[j]) % n != zn[j])

            {

                isIdentity = false;

                break;

            }

        }

        if (isIdentity)

        {

            hasIdentity = true;

            break;

        }

    }

    if (!hasIdentity)

    {

        return false;

    }

    // inverse elements

    for (int i = 0; i < zn.size(); i++)

    {

        bool hasInverse = false;

        for (int j = 0; j < zn.size(); j++)

        {

            if ((zn[i] + zn[j]) % n == 0)

            {

                hasInverse = true;

                break;

            }

        }

        if (!hasInverse)

        {

            return false;

        }

    }

    return true;

}

bool isAbelianGroup(vector<int> zn, int n)

{

    // Check if it's a group first

    if (!isGroup(zn, n))

    {

        return false;

    }

    // Check for commutativity

    for (int i = 0; i < zn.size(); i++)

    {

        for (int j = 0; j < zn.size(); j++)

        {

            if ((zn[i] + zn[j]) % n != (zn[j] + zn[i]) % n)

            {

                return false;

            }

        }

    }

    return true;

}

bool isCyclicGroup(vector<int> zn, int n){

    for (int i = 0; i < zn.size(); i++)

    {

        vector<bool> generated(n, false);

        int element = zn[i];

        for (int j = 0; j < n; j++)

        {

            int result = (element \* j) % n;

            if (find(zn.begin(), zn.end(), result) != zn.end())

            {

                generated[result] = true;

            }

        }

        if (all\_of(generated.begin(), generated.end(), [](bool v) { return v; }))

        {

            return true;  // Found a generator, so it's cyclic

        }

    }

    return false;  // No generator found, not cyclic

}

bool isRing(vector<int> zn, int n) {

    // Check if it's an abelian group under addition

    if (!isAbelianGroup(zn, n)) {

        return false;

    }

    // Check closure under multiplication

    for (int i = 0; i < zn.size(); i++) {

        for (int j = 0; j < zn.size(); j++) {

            int product = (zn[i] \* zn[j]) % n;

            if (find(zn.begin(), zn.end(), product) == zn.end()) {

                return false;

            }

        }

    }

    // Check associativity under multiplication

    for (int i = 0; i < zn.size(); i++) {

        for (int j = 0; j < zn.size(); j++) {

            for (int k = 0; k < zn.size(); k++) {

                if (((zn[i] \* zn[j]) % n \* zn[k]) % n != (zn[i] \* (zn[j] \* zn[k]) % n) % n) {

                    return false;

                }

            }

        }

    }

    // Check distributivity

    for (int i = 0; i < zn.size(); i++) {

        for (int j = 0; j < zn.size(); j++) {

            for (int k = 0; k < zn.size(); k++) {

                if ((zn[i] \* (zn[j] + zn[k]) % n) % n != ((zn[i] \* zn[j]) % n + (zn[i] \* zn[k]) % n) % n) {

                    return false;

                }

            }

        }

    }

    return true;

}

bool isCommutativeRing(vector<int> zn, int n) {

    // First, check if it's a ring

    if (!isRing(zn, n)) {

        return false;

    }

    // Check commutativity under multiplication

    // (We don't need to check for addition as isAbelianGroup already ensures that)

    for (int i = 0; i < zn.size(); i++) {

        for (int j = 0; j < zn.size(); j++) {

            if ((zn[i] \* zn[j]) % n != (zn[j] \* zn[i]) % n) {

                return false;

            }

        }

    }

    return true;

}

bool isIntegralDomain(vector<int> zn, int n) {

    // First, check if it's a commutative ring

    if (!isCommutativeRing(zn, n)) {

        return false;

    }

    // Check for multiplicative identity

    bool hasMultiplicativeIdentity = false;

    int identity = -1;

    for (int i = 0; i < zn.size(); i++) {

        bool isIdentity = true;

        for (int j = 0; j < zn.size(); j++) {

            if ((zn[i] \* zn[j]) % n != zn[j]) {

                isIdentity = false;

                break;

            }

        }

        if (isIdentity) {

            hasMultiplicativeIdentity = true;

            identity = zn[i];

            break;

        }

    }

    if (!hasMultiplicativeIdentity) {

        return false;

    }

    // Check for zero divisors

    for (int i = 0; i < zn.size(); i++) {

        if (zn[i] == 0) continue; // Skip zero

        for (int j = 0; j < zn.size(); j++) {

            if (zn[j] == 0) continue; // Skip zero

            if ((zn[i] \* zn[j]) % n == 0) {

                return false; // Found a zero divisor

            }

        }

    }

    return true;

}

bool isField(vector<int> zn, int n) {

    // First, check if it's an integral domain

    if (!isIntegralDomain(zn, n)) {

        return false;

    }

    // Find the multiplicative identity

    int identity = -1;

    for (int i = 0; i < zn.size(); i++) {

        bool isIdentity = true;

        for (int j = 0; j < zn.size(); j++) {

            if ((zn[i] \* zn[j]) % n != zn[j]) {

                isIdentity = false;

                break;

            }

        }

        if (isIdentity) {

            identity = zn[i];

            break;

        }

    }

    // Check for multiplicative inverse for every non-zero element

    for (int i = 0; i < zn.size(); i++) {

        if (zn[i] == 0) continue; // Skip zero

        bool hasInverse = false;

        for (int j = 0; j < zn.size(); j++) {

            if ((zn[i] \* zn[j]) % n == identity) {

                hasInverse = true;

                break;

            }

        }

        if (!hasInverse) {

            return false; // Found an element without a multiplicative inverse

        }

    }

    return true;

}

int main()

{

    for (int n = 2; n <= 100; n++) {

        vector<int> zn;

        for (int i = 0; i < n; i++) {

            zn.push\_back(i);

        }

        if (isField(zn, n)) {

            if (n == 2) {

                cout << "The following values of Zn are fields:" << endl;

                cout << "Z" << n;

            } else {

                cout << ", Z" << n;

            }

        }

        if (n == 100) {

            cout << endl;

        }

    }

}

## **Output:**

## 

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

#include <bits/stdc++.h>

using namespace std;

bool isPrime(int num) {

    if (num <= 1) return false;

    for (int i = 2; i <= sqrt(num); i++) {

        if (num % i == 0) return false;

    }

    return true;

}

int main() {

    vector<int> primeFields;

    vector<int> extensionFields;

    for (int i = 2; i <= 200; i++) {

        if (isPrime(i)) {

            primeFields.push\_back(i);

        } else {

            // Check if it's a power of a prime

            for (int j = 2; j <= sqrt(i); j++) {

                if (isPrime(j)) {

                    int power = j;

                    while (power <= i) {

                        if (power == i) {

                            extensionFields.push\_back(i);

                            break;

                        }

                        power \*= j;

                    }

                    if (power == i) break;

                }

            }

        }

    }

    cout << "Prime Fields: ";

    for (int prime : primeFields) {

        cout << prime << " ";

    }

    cout << "\nExtension Fields: ";

    for (int ext : extensionFields) {

        cout << ext << " ";

    }

    return 0;

}

## **Output:**

## 

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

#include <bits/stdc++.h>

using namespace std;

int main()

{

    cout << "enter prime number n : ";

    int n;

    cin >> n;

    for(int i = 3; i < n; i+=2){

        for(int j = 0; j < n; j++){

            int result = 1;

            vector<bool> generated(n, false);

            for (int k = 1; k < n; k++) {

                result = (result \* i) % n;

                if (!generated[result]) {

                    generated[result] = true;

                    cout << i << " ^ " << k << " = " << result << endl;

                    j++;

                }

            }

            if (j == n - 1) {

                cout << i << " is a primitive root of GF(" << n << ")" << endl;

                return 0;

            }

        }

    }

    return 0;

}

## **Output:**

## 

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

#include <iostream>

#include <vector>

#include <iomanip>

using namespace std;

// Define the irreducible polynomial

const uint8\_t irr\_poly = 0b10011; // x^4 + x + 1

// Function to add two elements in GF(16)

uint8\_t gf\_add(uint8\_t a, uint8\_t b)

{

    return a ^ b;

}

// Function to multiply two elements in GF(16)

uint8\_t gf\_mul(uint8\_t a, uint8\_t b)

{

    uint8\_t p = 0;

    while (b)

    {

        if (b & 1)

            p ^= a;

        a <<= 1;

        if (a & 0b10000)

            a ^= irr\_poly;

        b >>= 1;

    }

    return p;

}

// Function to find multiplicative inverse in GF(16)

uint8\_t gf\_inv(uint8\_t a)

{

    for (uint8\_t i = 1; i < 16; i++)

    {

        if (gf\_mul(a, i) == 1)

            return i;

    }

    return 0; // 0 has no multiplicative inverse

}

// Function to print a 4-bit binary representation

void print\_binary(uint8\_t n)

{

    for (int i = 3; i >= 0; i--)

    {

        cout << ((n >> i) & 1);

    }

}

int main()

{

    // Create addition and multiplication tables

    vector<vector<uint8\_t>> add\_table(16, vector<uint8\_t>(16));

    vector<vector<uint8\_t>> mul\_table(16, vector<uint8\_t>(16));

    for (uint8\_t i = 0; i < 16; i++)

    {

        for (uint8\_t j = 0; j < 16; j++)

        {

            add\_table[i][j] = gf\_add(i, j);

            mul\_table[i][j] = gf\_mul(i, j);

        }

    }

    // Print addition table

    cout << "Addition Table:" << endl;

    for (const auto &row : add\_table)

    {

        for (uint8\_t val : row)

        {

            cout << setw(2) << static\_cast<int>(val) << " ";

        }

        cout << endl;

    }

    // Print multiplication table

    cout << "\nMultiplication Table:" << endl;

    for (const auto &row : mul\_table)

    {

        for (uint8\_t val : row)

        {

            cout << setw(2) << static\_cast<int>(val) << " ";

        }

        cout << endl;

    }

    // Print additive and multiplicative inverses

    cout << "\nAdditive and Multiplicative Inverses:" << endl;

    for (uint8\_t i = 0; i < 16; i++)

    {

        uint8\_t add\_inv = gf\_add(i, i); // Additive inverse is the element itself

        uint8\_t mul\_inv = gf\_inv(i);

        cout << "Element: ";

        print\_binary(i);

        cout << ", Additive Inverse: ";

        print\_binary(add\_inv);

        cout << ", Multiplicative Inverse: ";

        if (mul\_inv != 0 || i == 1)

        {

            print\_binary(mul\_inv);

        }

        else

        {

            cout << "N/A";

        }

        cout << endl;

    }

    return 0;

}

## **Output:**

## 

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

#include <bits/stdc++.h>

using namespace std;

const uint8\_t FIELD\_SIZE = 128;

const uint8\_t IRREDUCIBLE\_POLY = 0b10000011;  // x^7 + x + 1

// Function to multiply two elements in GF(128)

uint8\_t gf\_mul(uint8\_t a, uint8\_t b) {

    uint8\_t result = 0;

    while (b) {

        if (b & 1)

            result ^= a;

        a <<= 1;

        if (a & 0b10000000)

            a ^= IRREDUCIBLE\_POLY;

        b >>= 1;

    }

    return result;

}

// Function to find multiplicative inverse in GF(128)

uint8\_t gf\_inv(uint8\_t a) {

    for (uint8\_t i = 1; i < FIELD\_SIZE; i++) {

        if (gf\_mul(a, i) == 1)

            return i;

    }

    return 0;  // This should never happen for non-zero input in GF(128)

}

int main() {

    uint8\_t element = 95;

    uint8\_t inverse = gf\_inv(element);

    cout << "Element: " << bitset<7>(element) << " (decimal " << (int)element << ")" << endl;

    cout << "Multiplicative Inverse: " << bitset<7>(inverse) << " (decimal " << (int)inverse << ")" << endl;

    // Verify the result

    uint8\_t product = gf\_mul(element, inverse);

    cout << "Product of element and its inverse: " << bitset<7>(product) << " (decimal " << (int)product << ")" << endl;

    return 0;

}

## **Output:**

## 