1. Implement modular addition and multiplication in GF(2^n).
Verify that these operations obey the properties of a finite field, such as closure, associativity, and distributivity.

Code -

```
#include <iostream>
#include <vector>

using namespace std;

// Define the field size (2^n)
const int n = 4;

// Define the irreducible polynomial (used for modulo operation)
const int irreduciblePoly = 0b10011; // x^4 + x + 1

// Function to perform modular addition in GF(2^n)
int gfAdd(int a, int b) {
    return a ^ b; // XOR operation
}

// Function to perform modular multiplication in GF(2^n)
int gfMultiply(int a, int b) {
    int result = 0;
    while (b > 0) {
        if (b & 1) // If the least significant bit of b is 1
            result ^= a; // Add a to the result using XOR
        a <<= 1; // Left-shift a
        if (a & (1 << n)) // If a has a term greater than x^n, reduce it using the irreducible polynomial
            a ^= irreduciblePoly;
        b >>= 1; // Right-shift b
    }
    return result;
}

int main() {
    cout << "Finite Field GF(2^" << n << ") Operations:" << endl;

    // Verify closure (addition)
    for (int i = 0; i < (1 << n); i++) {
        for (int j = 0; j < (1 << n); j++) {
            int result = gfAdd(i, j);
            if (result < 0 || result >= (1 << n)) {
                cout << "Addition does not satisfy closure property!" << endl;
                return 1;
            }
        }
    }

    // Verify closure (multiplication)
    for (int i = 0; i < (1 << n); i++) {
        for (int j = 0; j < (1 << n); j++) {
            int result = gfMultiply(i, j);
            if (result < 0 || result >= (1 << n)) {
                cout << "Multiplication does not satisfy closure property!" << endl;
                return 1;
            }
```

```
        }
    }

    // Verify associativity (addition)
    for (int i = 0; i < (1 << n); i++) {
        for (int j = 0; j < (1 << n); j++) {
            for (int k = 0; k < (1 << n); k++) {
                if (gfAdd(i, gfAdd(j, k)) != gfAdd(gfAdd(i, j), k)) {
                    cout << "Addition does not satisfy associativity property!" << endl;
                    return 1;
                }
            }
        }
    }

    // Verify associativity (multiplication)
    for (int i = 0; i < (1 << n); i++) {
        for (int j = 0; j < (1 << n); j++) {
            for (int k = 0; k < (1 << n); k++) {
                if (gfMultiply(i, gfMultiply(j, k)) != gfMultiply(gfMultiply(i, j), k)) {
                    cout << "Multiplication does not satisfy associativity property!" << endl;
                    return 1;
                }
            }
        }
    }

    // Verify distributivity
    for (int i = 0; i < (1 << n); i++) {
        for (int j = 0; j < (1 << n); j++) {
            for (int k = 0; k < (1 << n); k++) {
                if (gfMultiply(i, gfAdd(j, k)) != gfAdd(gfMultiply(i, j), gfMultiply(i, k))) {
                    cout << "Multiplication does not satisfy distributivity property!" << endl;
                    return 1;
                }
            }
        }
    }

    cout << "All properties of a finite field are satisfied." << endl;
    return 0;
}
```

2. Write a program to find primitive elements in GF(2^n). A primitive element is a generator of the field. Take n=3 and IRP as (x^3+x^2+1)

Code-

```
#include <iostream>
#include <vector>

using namespace std;
```

```
// Define the field size (2^n)
const int n = 3;

// Function to perform modular exponentiation in GF(2^n)
int gfPow(int base, int exponent, int prime) {
    int result = 1;
    while (exponent > 0) {
        if (exponent % 2 == 1) {
            result = (result * base) % prime;
        }
        base = (base * base) % prime;
        exponent /= 2;
    }
    return result;
}

// Function to find primitive elements in GF(2^n)
void findPrimitiveElements() {
    int prime = (1 << n) - 1; // Prime number for GF(2^n)
    vector<int> primitiveElements;

    for (int x = 2; x < prime; x++) {
        bool isPrimitive = true;

        for (int i = 1; i <= prime - 2; i++) {
            int power = gfPow(x, i, prime);
            if (power == 1) {
                isPrimitive = false;
                break;
            }
        }

        if (isPrimitive) {
            primitiveElements.push_back(x);
        }
    }

    cout << "Primitive elements in GF(2^" << n << "): ";
    for (int element : primitiveElements) {
        cout << element << " ";
    }
    cout << endl;
}

int main() {
    findPrimitiveElements();
    return 0;
}
```

3. represent elements of GF(2^n) using polynomial notation. For example, represent the element 1010101 as the polynomial x^6 + x^4 + x^2 + 1. Implement addition and multiplication of polynomials in GF(2^n).

Code-

```
#include <iostream>
#include <vector>
```

```
#include <bitset>

using namespace std;

// Define the field size (2^n)
const int n = 8;

// Define the irreducible polynomial (used for modulo operation)
const int irreduciblePoly = 0b100011011; // x^8 + x^4 + x^3 + x + 1

// Function to add two polynomials in GF(2^n)
int addPolynomials(int poly1, int poly2) {
    return poly1 ^ poly2;
}

// Function to multiply two polynomials in GF(2^n)
int multiplyPolynomials(int poly1, int poly2) {
    int result = 0;
    for (int i = 0; i < n; i++) {
        if ((poly1 >> i) & 1) {
            result ^= (poly2 << i);
        }
    }
    // Perform modulo operation using the irreducible polynomial
    while (result >= (1 << n)) {
        int shift = 0;
        while (((result >> shift) & 1) == 0) {
            shift++;
        }
        result ^= (irreduciblePoly << shift);
    }
    return result;
}

// Function to convert a binary number to polynomial notation
string binaryToPolynomial(int binary) {
    string polynomial;
    for (int i = n - 1; i >= 0; i--) {
        if ((binary >> i) & 1) {
            if (i == 0) {
                polynomial += "1";
            } else if (i == 1) {
                polynomial += "x + ";
            } else {
                polynomial += "x^" + to_string(i) + " + ";
            }
        }
    }
    return polynomial;
}

int main() {
    int poly1 = 0b1010101; // x^6 + x^4 + x^2 + 1
    int poly2 = 0b1101;    // x^3 + x^2 + 1

    cout << "Polynomial 1: " << binaryToPolynomial(poly1) << endl;
    cout << "Polynomial 2: " << binaryToPolynomial(poly2) << endl;
```

```
    int sum = addPolynomials(poly1, poly2);
    cout << "Polynomial 1 + Polynomial 2: " << binaryToPolynomial(sum) << endl;

    int product = multiplyPolynomials(poly1, poly2);
    cout << "Polynomial 1 * Polynomial 2: " << binaryToPolynomial(product) << endl;

    return 0;
}
```

4. Write a program to calculate the multiplicative inverses of elements in GF(2^n) with respect to an given IRP. Let the IRP is (x^8 + x^4 + x^3 + x + 1)

Code-

```
#include <iostream>
#include <vector>

using namespace std;

// Define the field size (2^n)
const int n = 8;

// Define the irreducible polynomial (used for modulo operation)
const int irreduciblePoly = 0b100011011; // x^8 + x^4 + x^3 + x + 1

// Function to perform modular multiplication in GF(2^n)
int gfMultiply(int a, int b) {
    int result = 0;
    for (int i = 0; i < n; i++) {
        if (b & 1) {
            result ^= a;
        }
        bool highestBitSet = (a >> (n - 1)) & 1;
        a <<= 1;
        if (highestBitSet) {
            a ^= irreduciblePoly;
        }
        b >>= 1;
    }
    return result;
}

// Function to calculate the multiplicative inverse using Extended Euclidean Algorithm
int multiplicativeInverse(int element) {
    int t0 = 0, t1 = 1;
    int q, temp, quotient;

    int a = element;
    int b = irreduciblePoly;

    while (a > 1) {
        quotient = a / b;
        temp = b;
        b = a % b;
        a = temp;
```

```
        temp = t0;
        t0 = t1 - quotient * t0;
        t1 = temp;
    }

    if (t1 < 0) {
        t1 += irreduciblePoly;
    }

    return t1;
}

int main() {
    int IRP = irreduciblePoly;
    cout << "Irreducible Polynomial: " << bitset<n>(IRP) << endl;

    for (int element = 1; element < (1 << n); element++) {
        int inverse = multiplicativeInverse(element);
        cout << "Element: " << bitset<n>(element) << " - Inverse: " << bitset<n>(inverse) << endl;
    }

    return 0;
}
```
