**Assignment – 4**

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

**A:**

*Pseudocode and Explanation –*

* The **get\_prime\_field**(int n) function generates a list of all prime numbers up to a given integer n. It uses the Sieve of Eratosthenes algorithm to efficiently identify primes by marking non-prime numbers in a boolean array. This function returns a vector containing all the prime numbers up to n.
* The **get\_extension\_field**(int n, vector<int> primes) function computes a list of numbers that are powers of the primes obtained from the get\_prime\_field function, provided these powers are less than n. For each prime, it generates numbers like squares, cubes, and higher powers, until these numbers exceed n. This function returns a vector of these power values, which represent the extension field based on the primes.
* In the **main**() function, the program prompts the user to enter a range value n. It then calls get\_prime\_field to retrieve all prime numbers up to n, and get\_extension\_field to get the extension field numbers derived from those primes. The prime numbers and extension field numbers are combined into a single vector, sorted, and printed.

*Code –*

#include<bits/stdc++.h>

using *namespace* std;

vector<*int*> get\_prime\_field(*int* *n*){

    vector<*bool*> prime(*n*+1, true);

    for(*int* i=2; i\*i <= *n*; i++){

        if(prime[i] == true){

            for(*int* j = i\*i; j<=*n*; j+=i){

                prime[j] = false;

            }

        }

    }

    vector<*int*> primes;

    // cout<<"Prime Field: ";

    for(*int* i=2;i<=*n*;i++){

        if(prime[i] == true){

            // cout<<i<<" ";

            primes.push\_back(i);

        }

    }

    return primes;

}

vector<*int*> get\_extension\_field(*int* *n*, vector<*int*> *primes*){

    vector<*int*> extension\_field;

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

*int* x = *primes*[i];

*int* p = x \* x;

        while(p < *n*) {

            extension\_field.push\_back(p);

            p \*= x;

        }

    }

    // cout<<"Extension Field: ";

    // for(int i = 0; i < extension\_field.size(); i++) {

    //     cout << extension\_field[i] << " ";

    // }

    return extension\_field;

}

*int* main()

{

*int* n;

    cout<<"Enter the range: "<<endl;

    cin>>n;

    vector<*int*> prime\_field = get\_prime\_field(n);

    cout<<endl;

    vector<*int*> extension\_field = get\_extension\_field(n, prime\_field);

    vector<*int*> field;

    for(*int* i=0;i<prime\_field.size();i++){

        field.push\_back(prime\_field[i]);

    }

    for(*int* i=0;i<extension\_field.size();i++){

        field.push\_back(extension\_field[i]);

    }

    sort(field.begin(),field.end());

    cout<<"Fields: "<<endl;

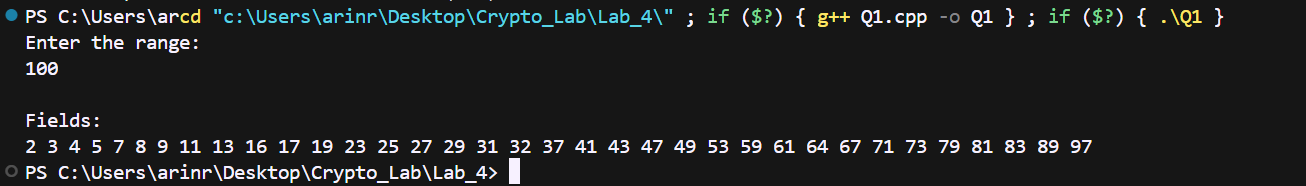
    for(*int* i=0;i<field.size();i++){

        cout<<field[i]<<" ";

    }

}

*Output –*

**

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

**A:**

*Pseudocode and Explanation –*

* The **get\_prime\_field**(int n) function identifies and returns all prime numbers up to a given integer n. It uses the Sieve of Eratosthenes algorithm to mark non-prime numbers in a boolean array. After processing, it prints out the prime numbers, which constitute the prime field, and stores them in a vector.
* The **get\_extension\_field**(int n, vector<int> primes) function calculates and returns the extension field, which consists of powers of the primes obtained from get\_prime\_field. For each prime number, it calculates successive powers (such as squares, cubes, etc.) until these values exceed the given limit n. The function then prints these values as the extension field.
* In the **main**() function, the program first calls get\_prime\_field(100) to generate and print all prime numbers up to 100. It then calls get\_extension\_field(100, prime\_field) to generate and print the extension field based on these primes. The results are displayed through the printed output, showing both the prime field and the extension field values up to 100.

*Code-*

#include<bits/stdc++.h>

using *namespace* std;

vector<*int*> get\_prime\_field(*int* *n*){

    vector<*bool*> prime(*n*+1, true);

    for(*int* i=2; i\*i <= *n*; i++){

        if(prime[i] == true){

            for(*int* j = i\*i; j<=*n*; j+=i){

                prime[j] = false;

            }

        }

    }

    vector<*int*> primes;

    cout<<"Prime Field: ";

    for(*int* i=2;i<=*n*;i++){

        if(prime[i] == true){

            cout<<i<<" ";

            primes.push\_back(i);

        }

    }

    return primes;

}

vector<*int*> get\_extension\_field(*int* *n*, vector<*int*> *primes*){

    vector<*int*> extension\_field;

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

*int* x = *primes*[i];

*int* p = x \* x;

        while(p < *n*) {

            extension\_field.push\_back(p);

            p \*= x;

        }

    }

    cout<<"Extension Field: ";

    for(*int* i = 0; i < extension\_field.size(); i++) {

        cout << extension\_field[i] << " ";

    }

    return extension\_field;

}

*int* main()

{

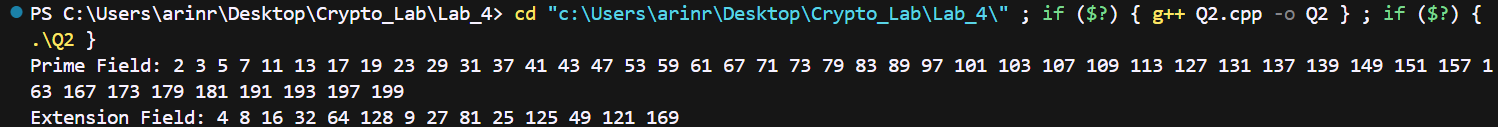
    vector<*int*> prime\_field = get\_prime\_field(200);

    cout<<endl;

    vector<*int*> extension\_field = get\_extension\_field(200, prime\_field);

}

*Output –*

**

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

**A:**

*Pseudocode and Explanation –*

* This code finds the smallest primitive root of a given integer n. It starts by asking the user to input n. The code then iterates through odd numbers from 1 to n-1. For each number i, it calculates the powers of i, takes them modulo n, and stores the results in a vector. After sorting the vector, the code checks for duplicates. If no duplicates are found, i is identified as the primitive root, and the program prints it and exits. The goal is to find the smallest number that generates all integers from 1 to n-1 when raised to successive powers.

*Code –*

#include <bits/stdc++.h>

using *namespace* std;

*int* main() {

*int* n;

    cout << "Enter an element: " << endl;

    cin >> n;

    for (*int* i = 1; i <= n - 1; i++) {

        if (i % 2 == 0) {

            continue;

        }

        vector<*int*> vec;

        for (*int* j = 0; j < n - 1; j++) {

*int* x = pow(i, j);

*int* y = x % n;

            vec.push\_back(y);

        }

        sort(vec.begin(), vec.end());

*bool* flag = false;

        for (*int* k = 1; k < vec.size(); k++) {

            if (vec[k - 1] == vec[k]) {

                flag = true;

                break;

            }

        }

        if (!flag) {

            cout << "Primitive Root for n = " << n << " is " << i << endl;

            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).**

**A:**

*Pseudocode and Explanation –*

* decToBinary(int n):
* Converts a decimal number n to its binary representation.
* The binary digits are stored in a vector in reverse order to facilitate easier processing.
* multiplyPolynomials(const vector<int>& poly1, const vector<int>& poly2):
* Multiplies two binary polynomials poly1 and poly2.
* The result is stored in a vector representing the coefficients of the resulting polynomial.
* main():
* Additive Inverses: Iterates through all pairs (i, j) in GF(16) and finds pairs where i ^ j == 0, indicating additive inverses.
* Binary Conversion and Padding: Converts each number from 0 to 15 into 4-bit binary using decToBinary. Pads binary numbers to ensure they are 4 bits long.
* Polynomial Multiplication: Multiplies the binary polynomials using multiplyPolynomials. Reduces the result modulo 2 to stay within GF(2).
* Polynomial Reduction: Checks if the polynomial results need to be reduced using an irreducible polynomial when the degree exceeds 3.
* Finding Multiplicative Inverses: Identifies the multiplicative inverse by checking the reduced polynomials that match the required degree. Prints the multiplicative inverses for each element in GF(16).

*Code –*

#include<bits/stdc++.h>

using *namespace* std;

vector<*int*> decToBinary(*int* *n*)

{

    // array to store binary number

*int* binaryNum[32];

    // counter for binary array

*int* i = 0;

    while (*n* > 0) {

        // storing remainder in binary array

        binaryNum[i] = *n* % 2;

*n* = *n* / 2;

        i++;

    }

    // printing binary array in reverse order

    vector<*int*> vec;

    for (*int* j = i - 1; j >= 0; j--){

        vec.push\_back(binaryNum[j]);

    }

    return vec;

}

vector<*int*> multiplyPolynomials(*const* vector<*int*>*&* *poly1*, *const* vector<*int*>*&* *poly2*) {

*int* size1 = *poly1*.size();

*int* size2 = *poly2*.size();

    vector<*int*> result(size1 + size2 - 1, 0);

    for (*int* i = 0; i < size1; ++i) {

        for (*int* j = 0; j < size2; ++j) {

            result[i + j] += *poly1*[i] \* *poly2*[j];

        }

    }

    return result;

}

*int* main()

{

*int* n = 16;

    for(*int* i=0;i<16;i++){

        for(*int* j=0;j<16;j++){

*unsigned* *int* a = i^j;

            if(a == 0){

                cout<<"Additive Inverse of "<<i<<" is : "<<j<<endl;

            }

        }

    }

    vector<vector<*int*>> ans;

    for(*int* i=0;i<16;i++){

        for(*int* j=0;j<16;j++){

            vector<*int*> first = decToBinary(i);

            vector<*int*> second = decToBinary(j);

            for(*int* k=first.size();k<4;k++){

                first.insert(first.begin(),0);

            }

            for(*int* k=second.size();k<4;k++){

                second.insert(second.begin(),0);

            }

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

            //     cout<<first[k]<<" ";

            // }

            // cout<<"     ";

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

            //     cout<<second[k]<<" ";

            // }

            vector<*int*> result = multiplyPolynomials(first, second);

            // cout<<"     ";

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

                if(result[k]%2 == 0){

                    result[k] = 0;

                }else{

                    result[k] = 1;

                }

                // cout<<result[k]<<" ";

            }

            ans.push\_back(result);

            // cout<<endl;

        }

    }

    for(*int* i=0;i<ans.size();i++){

        vector<*int*> irre\_poly;

        if(ans[i][0] == 1){ // x^6

                irre\_poly.push\_back(1);

                irre\_poly.push\_back(0);

                irre\_poly.push\_back(0);

                irre\_poly.push\_back(1);

                irre\_poly.push\_back(1);

                irre\_poly.push\_back(0);

                irre\_poly.push\_back(0);

                for(*int* k = 0;k<7;k++){

                    if((ans[i][k] + irre\_poly[k])%2 == 0){

                        ans[i][k] = 0;

                    }else{

                        ans[i][k] = 1;

                    }

                }

        }

        if(ans[i][1] == 1){ // x^5

                irre\_poly.clear();

                irre\_poly.push\_back(0);

                irre\_poly.push\_back(1);

                irre\_poly.push\_back(0);

                irre\_poly.push\_back(0);

                irre\_poly.push\_back(1);

                irre\_poly.push\_back(1);

                irre\_poly.push\_back(0);

                for(*int* k = 0;k<7;k++){

                    if((ans[i][k] + irre\_poly[k])%2 == 0){

                        ans[i][k] = 0;

                    }else{

                        ans[i][k] = 1;

                    }

                }

        }

        if(ans[i][2] == 1){ // x^4

                irre\_poly.clear();

                irre\_poly.push\_back(0);

                irre\_poly.push\_back(0);

                irre\_poly.push\_back(1);

                irre\_poly.push\_back(0);

                irre\_poly.push\_back(0);

                irre\_poly.push\_back(1);

                irre\_poly.push\_back(1);

                for(*int* k = 0;k<7;k++){

                    if((ans[i][k] + irre\_poly[k])%2 == 0){

                        ans[i][k] = 0;

                    }else{

                        ans[i][k] = 1;

                    }

                }

        }

    }

    // for(int i=0;i<ans.size();i++){

    //     for(int j=0;j<ans[i].size();j++){

    //         cout<<ans[i][j];

    //     }

    //     cout<<endl;

    // }

    vector<*int*> inverses;

    for(*int* i=16;i<ans.size();i++){

        if((ans[i][6] == 1) && (ans[i][5] == 0) && (ans[i][4] == 0) && (ans[i][3] == 0)){

            inverses.push\_back(i%16);

        }

    }

    cout<<inverses.size()<<endl;

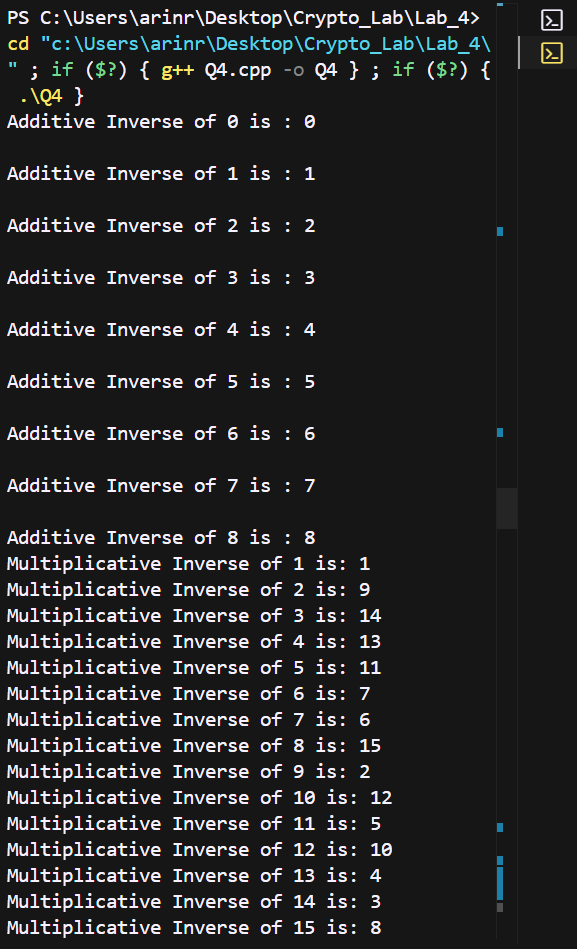
    for(*int* i=0;i<inverses.size();i++){

        cout<<"Multiplicative Inverse of "<<i+1<<" is: "<<inverses[i]<<endl;

    }

}

*Output –*

**

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

**A:**

*Pseudocode and Explanation –*

* **Multiply\_GF128(int a, int b)**: Multiplies two elements in GF(128) using a specific polynomial for reduction.
* **Multiplicative\_inverse\_GF128(int a)**: Finds the multiplicative inverse of an element in GF(128), or returns an error if not found.
* **Main()**: Computes and prints the multiplicative inverse of 95 in GF(128).

*Code –*

#include <iostream>

using *namespace* std;

*const* *int* irreducible\_poly = 0b10000011;

*int* multiply\_GF128(*int* *a*, *int* *b*) {

*int* 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;

}

*int* multiplicative\_inverse\_GF128(*int* *a*) {

    if (*a* == 0) {

        cout << "0 has no multiplicative inverse in GF(128)." << endl;

        return -1;

    }

    for (*int* i = 1; i < 128; ++i) {

        if (multiply\_GF128(*a*, i) == 1) {

            return i;

        }

    }

    cout << "No multiplicative inverse found for " << *a* << " in GF(128)." << endl;

    return -1;

}

*int* main() {

*int* a = 95;

*int* inverse = multiplicative\_inverse\_GF128(a);

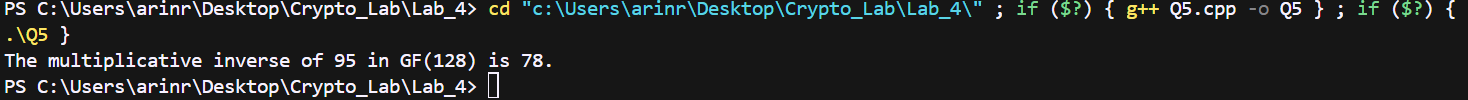
    if (inverse != -1) {

        cout << "The multiplicative inverse of " << a << " in GF(128) is " << inverse << "." << endl;

    }

    return 0;

}

*Output – *