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 –

```
PS C:\Users\arcd "c:\Users\arinr\Desktop\Crypto_Lab\Lab_4\" ; if ($?) { g++ Q1.cpp -o Q1 } ; if ($?) { .\Q1 }
Enter the range:
100

Fields:
2 3 4 5 7 8 9 11 13 16 17 19 23 25 27 29 31 32 37 41 43 47 49 53 59 61 64 67 71 73 79 81 83 89 97
PS C:\Users\arinr\Desktop\Crypto_Lab\Lab_4>
```

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 –

```
PS C:\Users\arinr\Desktop\Crypto_Lab\Lab_4> cd "c:\Users\arinr\Desktop\Crypto_Lab\Lab_4\" ; if ($?) { g++ Q2.cpp -o Q2 } ; if ($?) {
.\Q2 }
Prime Field: 2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97 101 103 107 109 113 127 131 137 139 149 151 157 1
63 167 173 179 181 191 193 197 199
Extension Field: 4 8 16 32 64 128 9 27 81 25 125 49 121 169
```

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;
```

```cpp
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 –

```
Enter an element:
5
Primitive Root for n = 5 is 3
```

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 –

```
PS C:\Users\arinr\Desktop\Crypto_Lab\Lab_4>
cd "c:\Users\arinr\Desktop\Crypto_Lab\Lab_4\
" ; if ($?) { g++ Q4.cpp -o Q4 } ; if ($?) {
 .\Q4 }
Additive Inverse of 0 is : 0

Additive Inverse of 1 is : 1

Additive Inverse of 2 is : 2

Additive Inverse of 3 is : 3

Additive Inverse of 4 is : 4

Additive Inverse of 5 is : 5

Additive Inverse of 6 is : 6

Additive Inverse of 7 is : 7

Additive Inverse of 8 is : 8
Multiplicative Inverse of 1 is: 1
Multiplicative Inverse of 2 is: 9
Multiplicative Inverse of 3 is: 14
Multiplicative Inverse of 4 is: 13
Multiplicative Inverse of 5 is: 11
Multiplicative Inverse of 6 is: 7
Multiplicative Inverse of 7 is: 6
Multiplicative Inverse of 8 is: 15
Multiplicative Inverse of 9 is: 2
Multiplicative Inverse of 10 is: 12
Multiplicative Inverse of 11 is: 5
Multiplicative Inverse of 12 is: 10
Multiplicative Inverse of 13 is: 4
Multiplicative Inverse of 14 is: 3
Multiplicative Inverse of 15 is: 8
```

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 –

```
PS C:\Users\arinr\Desktop\Crypto_Lab\Lab_4> cd "c:\Users\arinr\Desktop\Crypto_Lab\Lab_4\" ; if ($?) { g++ Q5.cpp -o Q5 } ; if ($?) {
.\Q5 }
The multiplicative inverse of 95 in GF(128) is 78.
PS C:\Users\arinr\Desktop\Crypto_Lab\Lab_4>
```