Name: Karuna Bajirao Randive

Prn: 2020BTECS00024

Batch: B2

Cryptography & Network Security Lab

Title: RSA

Aim: To study and implement RSA Algorithm.

Theory:

Algorithm:

1. Select Primes: Choose two distinct large prime numbers, p and q.

2. Compute Modulus: Calculate n, where $n = p * q$. This becomes the modulus for the public and private keys.

3. Calculate Euler's Totient: Compute $\varphi(n)$ where $\varphi(n) = (p - 1)(q - 1)$.

4. Choose Public Key: Select a number e such that $1 < e < \varphi(n)$ and e is coprime to $\varphi(n)$.

5. Compute Private Key: Determine d as the modular multiplicative inverse of e modulo $\varphi(n)$ $(d * e) \bmod \varphi(n) = 1$.

Encryption:

6. Represent Message: Represent the message as an integer m, where $0 < m < n$.

7. Compute Cipher: Encrypt the message using the public key: \(c = m^e \mod n\).

Decryption:

8. Compute Message: Decrypt the ciphertext using the private key: \(m = c^d \mod n\).

Advantages:

- Enables secure transmission of data over insecure networks.
- Uses public and private keys for encryption and decryption, enhancing security.

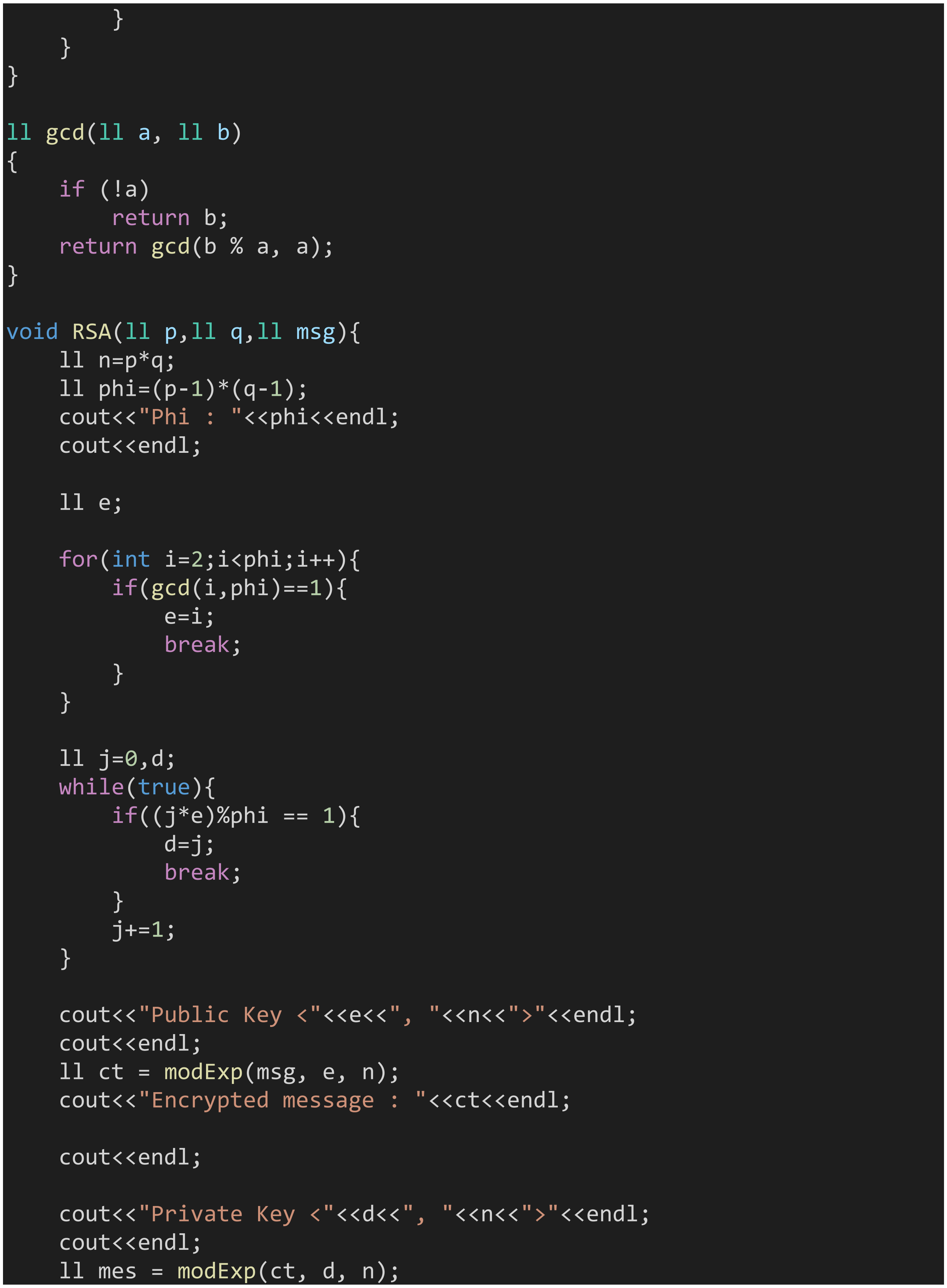
- Facilitates the creation and verification of digital signatures for data integrity and authenticity.

Disadvantages:

- Larger keys are needed for higher security, leading to increased computational load.
- RSA operations, particularly with larger keys, can be computationally intensive, affecting performance.
- Weaknesses in random number generation could compromise security.

❖ RSA Factorization

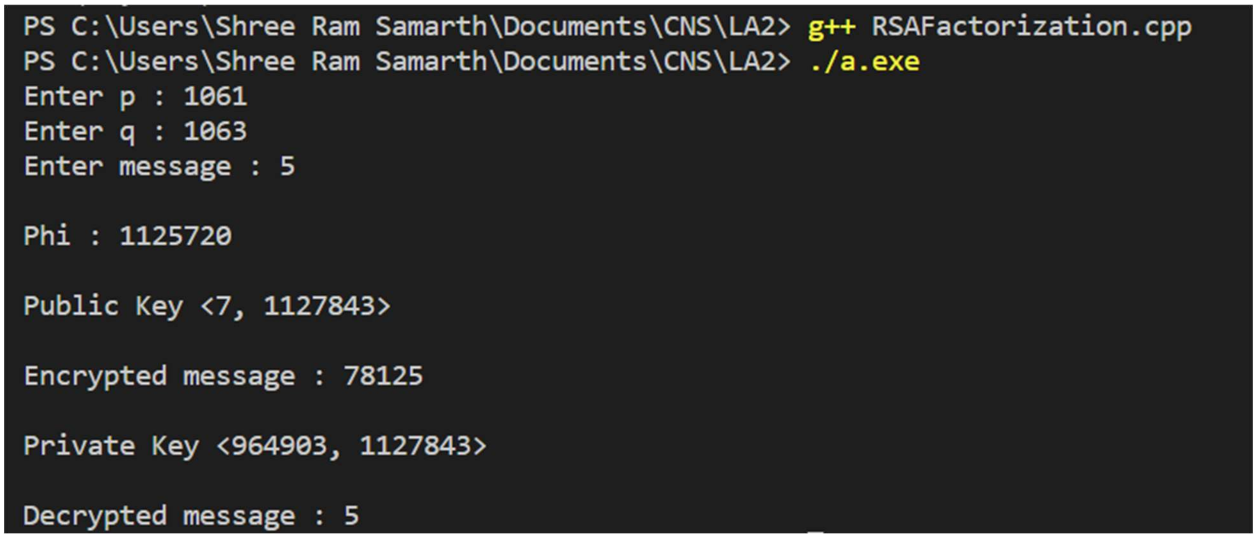
```cpp
#include <bits/stdc++.h>
using namespace std;
typedef unsigned long long int ll;

ll modExp(ll base, ll exponent, ll modulus) {
    if (modulus == 1) return 0; // In case of modulus being 1
    ll result = 1;
    base = base % modulus;
    while (exponent > 0) {
        if (exponent % 2 == 1)
            result = (result * base) % modulus;
        exponent = exponent >> 1;
        base = (base * base) % modulus;
    }
    return result;
}

ll prime_checker(ll p) {
    if (p < 1) {
        return -1;
    } else if (p > 1) {
        if (p == 2) {
            return 1;
        }

        for (int i = 2; i < p; i++) {
            if (p % i == 0) {
                return -1;
            }
            return 1;
```



```
        }
    }
}

ll gcd(ll a, ll b)
{
    if (!a)
        return b;
    return gcd(b % a, a);
}

void RSA(ll p,ll q,ll msg){
    ll n=p*q;
    ll phi=(p-1)*(q-1);
    cout<<"Phi : "<<phi<<endl;
    cout<<endl;

    ll e;

    for(int i=2;i<phi;i++){
        if(gcd(i,phi)==1){
            e=i;
            break;
        }
    }

    ll j=0,d;
    while(true){
        if((j*e)%phi == 1){
            d=j;
            break;
        }
        j+=1;
    }

    cout<<"Public Key <"<<e<<", "<<n<<">"<<endl;
    cout<<endl;
    ll ct = modExp(msg, e, n);
    cout<<"Encrypted message : "<<ct<<endl;

    cout<<endl;

    cout<<"Private Key <"<<d<<", "<<n<<">"<<endl;
    cout<<endl;
    ll mes = modExp(ct, d, n);
```

```cpp
    cout<<"Decrypted message : "<<mes<<endl;
}

int main(){
    ll p,q;
    while(1){
        cout<<"Enter p : ";
        cin>>p;
        if (prime_checker(p) == -1){
            cout<<"Number Is Not Prime, Please Enter Again!"<<endl;
            continue;
        }
        break;
    }

    while(1){
        cout<<"Enter q : ";
        cin>>q;
        if (prime_checker(q) == -1){
            cout<<"Number Is Not Prime, Please Enter Again!"<<endl;
            continue;
        }
        break;
    }

    ll msg;
    cout<<"Enter message : ";
    cin>>msg;

    cout<<endl;
    RSA(p,q,msg);

}
```

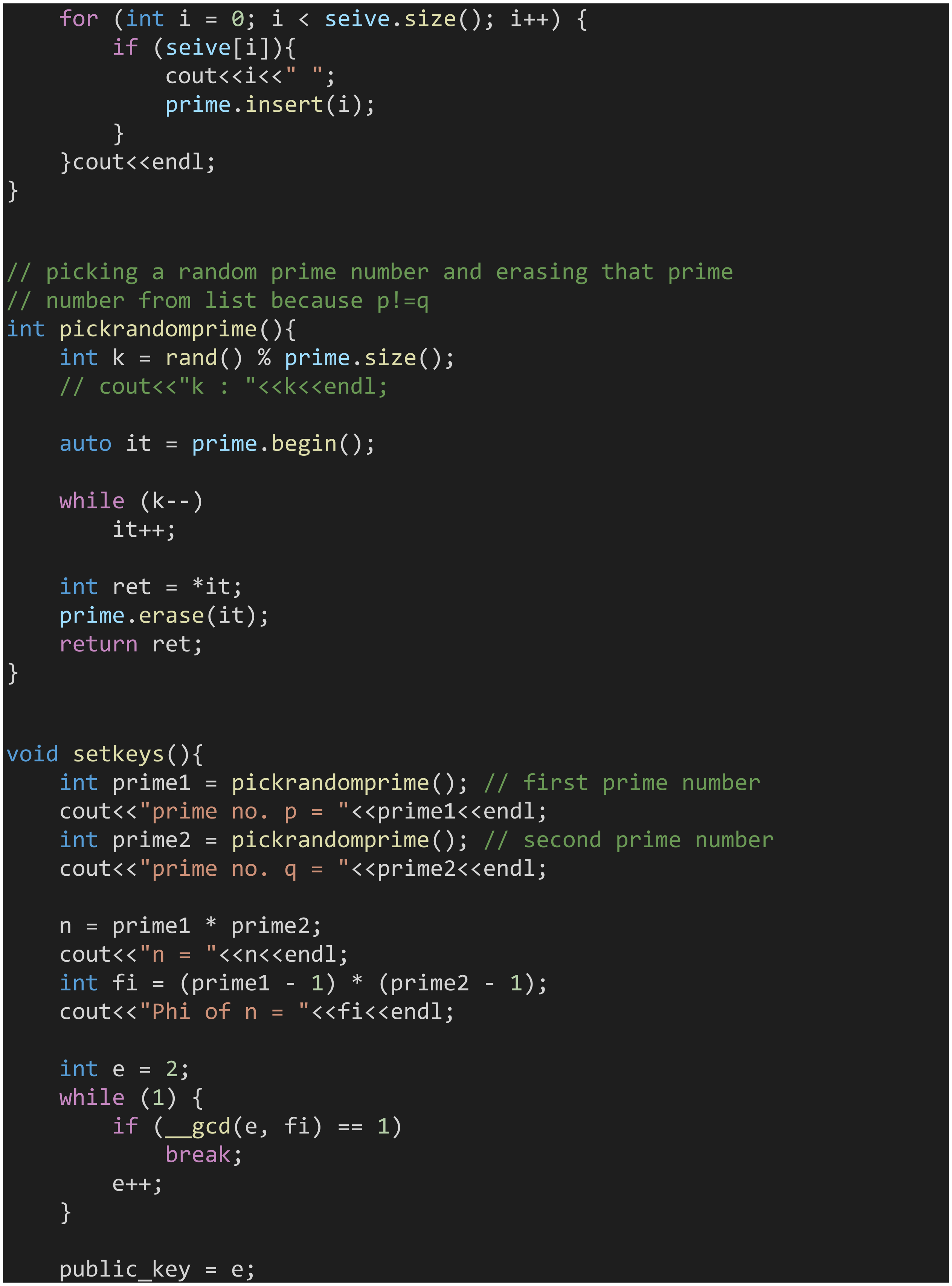
Output:



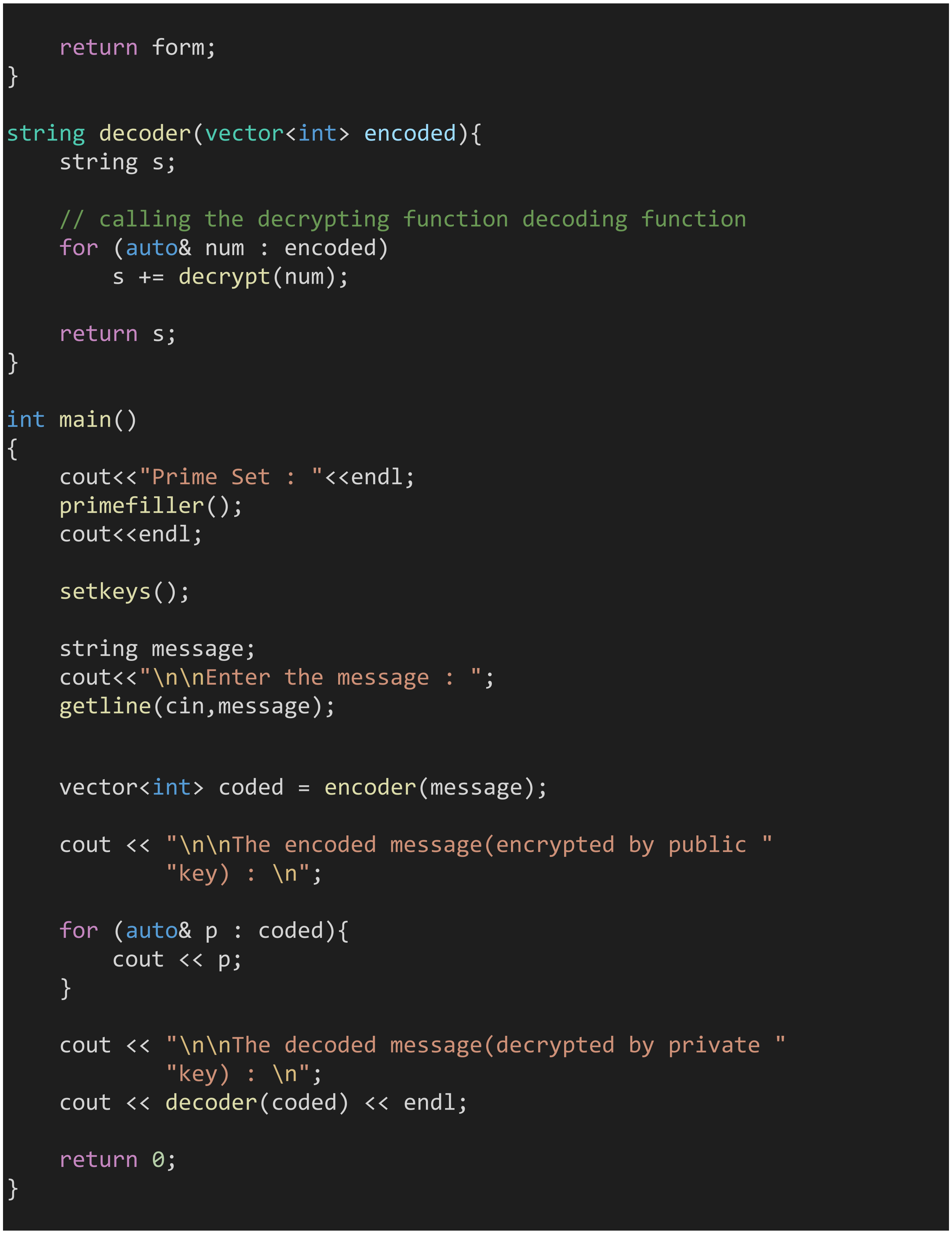
```
PS C:\Users\Shree Ram Samarth\Documents\CNS\LA2> g++ RSAFactorization.cpp
PS C:\Users\Shree Ram Samarth\Documents\CNS\LA2> ./a.exe
Enter p : 1061
Enter q : 1063
Enter message : 5

Phi : 1125720

Public Key <7, 1127843>

Encrypted message : 78125

Private Key <964903, 1127843>

Decrypted message : 5
```

❖ **RSA Text**

```cpp
#include <bits/stdc++.h>
using namespace std;
set<int> prime; // a set will be the collection of prime numbers,
        // where we can select random primes p and q
int public_key;
int private_key;
int n;

// we will run the function only once to fill the set of
// prime numbers
void primefiller(){
    // method used to fill the primes set is seive of
    // eratosthenes(a method to collect prime numbers)
    vector<bool> seive(250, true);

    seive[0] = false;
    seive[1] = false;

    for (int i = 2; i < 250; i++) {
        for (int j = i * 2; j < 250; j += i) {
            seive[j] = false;
        }
    }
    // filling the prime numbers
```

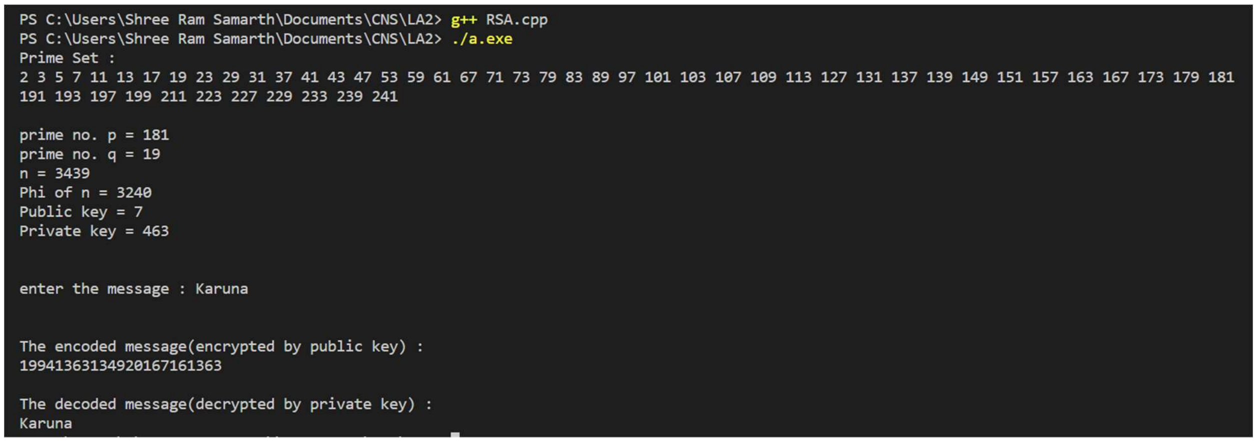


```cpp
    for (int i = 0; i < seive.size(); i++) {
        if (seive[i]){
            cout<<i<<" ";
            prime.insert(i);
        }
    }cout<<endl;
}


// picking a random prime number and erasing that prime
// number from list because p!=q
int pickrandomprime(){
    int k = rand() % prime.size();
    // cout<<"k : "<<k<<endl;

    auto it = prime.begin();

    while (k--)
        it++;

    int ret = *it;
    prime.erase(it);
    return ret;
}


void setkeys(){
    int prime1 = pickrandomprime(); // first prime number
    cout<<"prime no. p = "<<prime1<<endl;
    int prime2 = pickrandomprime(); // second prime number
    cout<<"prime no. q = "<<prime2<<endl;

    n = prime1 * prime2;
    cout<<"n = "<<n<<endl;
    int fi = (prime1 - 1) * (prime2 - 1);
    cout<<"Phi of n = "<<fi<<endl;

    int e = 2;
    while (1) {
        if (__gcd(e, fi) == 1)
            break;
        e++;
    }

    public_key = e;
    cout<<"Public key = "<<public_key<<endl;

    // d = (k*Φ(n) + 1) / e for some integer k
    int d = 2;
    while (1) {
        if ((d * e) % fi == 1)
            break;
        d++;
    }

    private_key = d;
    cout<<"Private key = "<<private_key<<endl;
}

// to encrypt the given number
long long int encrypt(double message){
    int e = public_key;
    long long int encrpyted_text = 1;
    while (e--) {
        encrpyted_text *= message;
        encrpyted_text %= n;
    }
    return encrpyted_text;
}

// to decrypt the given number
long long int decrypt(int encrpyted_text){
    int d = private_key;
    long long int decrypted = 1;
    while (d--) {
        decrypted *= encrpyted_text;
        decrypted %= n;
    }
    return decrypted;
}

// first converting each character to its ASCII value and
// then encoding it then decoding the number to get the
// ASCII and converting it to character
vector<int> encoder(string message){
    vector<int> form;

    // calling the encrypting function in encoding function
    for (auto& letter : message)
        form.push_back(encrypt((int)letter));
```



```cpp
    return form;
}

string decoder(vector<int> encoded){
    string s;

    // calling the decrypting function decoding function
    for (auto& num : encoded)
        s += decrypt(num);

    return s;
}

int main()
{
    cout<<"Prime Set : "<<endl;
    primefiller();
    cout<<endl;

    setkeys();

    string message;
    cout<<"\n\nEnter the message : ";
    getline(cin,message);


    vector<int> coded = encoder(message);

    cout << "\n\nThe encoded message(encrypted by public "
            "key) : \n";

    for (auto& p : coded){
        cout << p;
    }

    cout << "\n\nThe decoded message(decrypted by private "
            "key) : \n";
    cout << decoder(coded) << endl;

    return 0;
}
```

Output:



```
PS C:\Users\Shree Ram Samarth\Documents\CNS\LA2> g++ RSA.cpp
PS C:\Users\Shree Ram Samarth\Documents\CNS\LA2> ./a.exe
Prime Set :
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97 101 103 107 109 113 127 131 137 139 149 151 157 163 167 173 179 181
191 193 197 199 211 223 227 229 233 239 241

prime no. p = 181
prime no. q = 19
n = 3439
Phi of n = 3240
Public key = 7
Private key = 463


enter the message : Karuna


The encoded message(encrypted by public key) :
19941363134920167161363

The decoded message(decrypted by private key) :
Karuna
```

Name: Karuna Bajirao Randive.

Prn: 2020BTECS00024

Batch: B2

Cryptography & Network Security Lab

Title: Diffie-Hellman Algorithm.

Aim: To study and implement Diffie-Hellman Algorithm.

Theory:

The Diffie-Hellman key exchange is a method used to securely establish a shared secret key between two parties over an insecure communication channel. Here is an outline of the Diffie-Hellman algorithm:

1. Setup:
2. Key Exchange:
    a. Public Value Calculation:
       Alice computes her public value (A): $A=g^a \bmod p$
       Bob computes his public value (B): $B=g^b \bmod p$
       Alice and Bob exchange their calculated public values (A and B).
    b. Shared Secret Key Generation:
       Alice uses Bob's public value and her private key to compute the shared secret: $s=B^a \bmod p$
       Bob uses Alice's public value and his private key to compute the same shared secret: $s=A^b \bmod p$
3. Result: Both Alice and Bob now possess a shared secret key (s) that is identical.
4. Usage of Shared Key: The shared secret key (s) can be utilized for further secure communication, such as symmetric encryption or decryption.

Advantages:

- Facilitates secure exchange of cryptographic keys over an insecure communication channel.
- Uses public and private keys, ensuring a shared secret key without directly transmitting the private keys.
- Prevents eavesdroppers from deriving the shared secret key, as it's computationally difficult to deduce from exchanged public values.

Disadvantages:

- Vulnerable to potential man-in-the-middle attacks where an intruder intercepts and alters the exchanged public keys.
- Diffie-Hellman only provides a method for secure key exchange, lacking built-in authentication mechanisms to verify the identity of the parties involved.
- If a private key is compromised, it can compromise the secrecy of the shared key in subsequent communications.

Code:

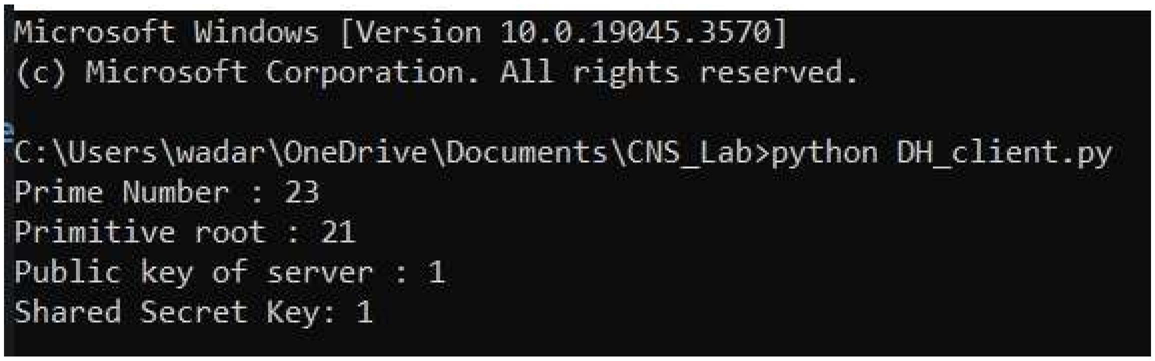
Client:

```python
import socket
import random

q = 23

def mod_pow(base, exponent, modulus):
    if modulus == 1:
        return 0
    result = 1
    base = base % modulus
    while exponent > 0:
        if exponent % 2 == 1:
            result = (result * base) % modulus
        base = (base * base) % modulus
        exponent = exponent // 2
    return result

def is_primitive_root(g, p):
    values = set()
    for i in range(1, p):
        values.add(pow(g, i, p))
        if len(values) == p - 1:
            return True
    return False

for candidate in range(2, q):
    if is_primitive_root(candidate, q):
        a=candidate

client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client.connect(("192.168.164.1", 8888))

server_public_key = int(client.recv(1024).decode())

private_key = random.randint(1, q - 1)
public_key = mod_pow(a, private_key, q)

client.send(str(public_key).encode())

shared_secret = mod_pow(server_public_key, private_key, q)

print(f"Prime Number : {q}")
print(f"Primitive root : {a}")
print(f"Public key of server : {server_public_key}")
print(f"Shared Secret Key: {shared_secret}")

client.close()
```

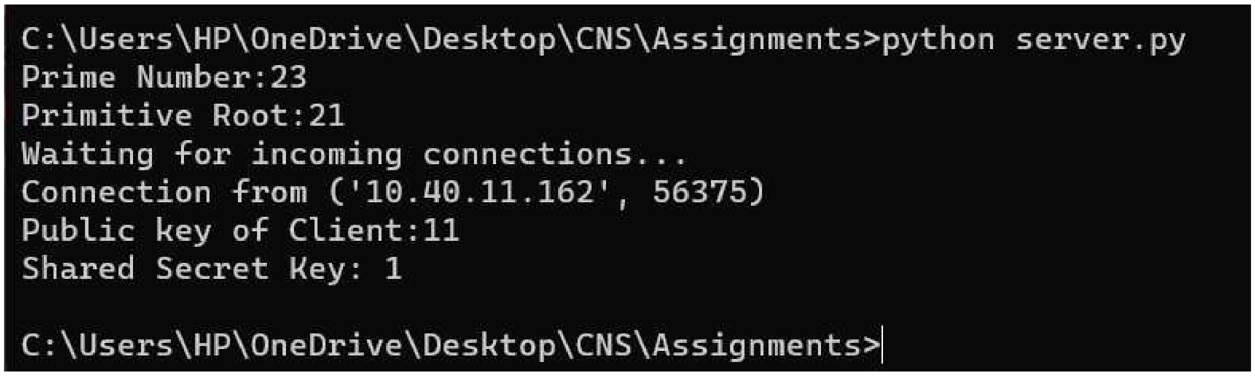
Server:

```python
import socket
import random

q = 23


def mod_pow(base, exponent, modulus):
    if modulus == 1:
        return 0
    result = 1
    base = base % modulus
    while exponent > 0:
        if exponent % 2 == 1:
            result = (result * base) % modulus
        base = (base * base) % modulus
        exponent = exponent // 2
    return result
def is_primitive_root(g, p):
    # Check if g is a primitive root modulo p
    values = set()
    for i in range(1, p):
        values.add(pow(g, i, p))
        if len(values) == p - 1:
            return True
    return False
for candidate in range(2, q):
    if is_primitive_root(candidate, q):
        a=candidate
print(f"Prime Number:{q}")
print(f"Primitive Root:{a}")
server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server.bind(("192.168.164.1", 8888))
server.listen(1)

print("Waiting for incoming connections...")
client, addr = server.accept()
print(f"Connection from {addr}")

private_key = random.randint(1, q - 1)
public_key = mod_pow(a, private_key, q)

client.send(str(public_key).encode())
client_public_key = int(client.recv(1024).decode())
print(f"Public key of Client:{client_public_key}")
shared_secret = mod_pow(client_public_key, private_key, q)

print(f"Shared Secret Key: {shared_secret}")

client.close()
server.close()
```

Output:



```
Microsoft Windows [Version 10.0.19045.3570]
(c) Microsoft Corporation. All rights reserved.

C:\Users\wadar\OneDrive\Documents\CNS_Lab>python DH_client.py
Prime Number : 23
Primitive root : 21
Public key of server : 1
Shared Secret Key: 1
```



```
C:\Users\HP\OneDrive\Desktop\CNS\Assignments>python server.py
Prime Number:23
Primitive Root:21
Waiting for incoming connections...
Connection from ('10.40.11.162', 56375)
Public key of Client:11
Shared Secret Key: 1

C:\Users\HP\OneDrive\Desktop\CNS\Assignments>
```

Name: Karuna Bajirao Randive

Prn: 2020BTECS00024

Batch: B2

Cryptography & Network Security Lab

Title: Chinese Remainder Theorem.

Aim: To study and implement Chinese Remainder Theorem.

Theory:

The Chinese Remainder Theorem (CRT) is a mathematical technique used in number theory and cryptography. It's primarily used in modular arithmetic to speed up certain calculations. Here are the algorithm steps, advantages, and disadvantages of the CRT:

Algorithm Steps:

- Suppose you have a large integer 'N' that you want to work with and perform calculations modulo 'N.' You can break this down into simpler calculations by using the CRT.
- First, you need to find the prime factorization of 'N.' You express 'N' as a product of its prime factors: N = p1^e1 * p2^e2 * ... * pk^ek, where p1, p2, ..., pk are distinct prime numbers, and e1, e2, ..., ek are their respective exponents.
- For a given number 'x,' you calculate its remainders when divided by each of the prime factors. This means computing x mod p1, x mod p2, ..., x mod pk.
- Inverse Modulus: You then calculate the modular inverse of each prime factor, which is the number 'y' such that (p1^e1 * p2^e2 * ... * pk^ek) * y ≡ 1 (mod pi) for each prime factor pi.
- Using the remainders and the modular inverses, you can calculate the value of 'x' modulo 'N' as follows:

  x mod N = (x mod p1) * (p1^e1 * y1) + (x mod p2) * (p2^e2 * y2) + ... + (x mod pk) * (pk^ek * yk).

Advantages:

1. CRT allows you to perform modular arithmetic operations more efficiently by breaking them down into smaller computations.

2. You can perform the CRT calculations for different primes in parallel, which can significantly speed up the process.
3. The CRT can reduce the space needed to store large numbers in some applications.

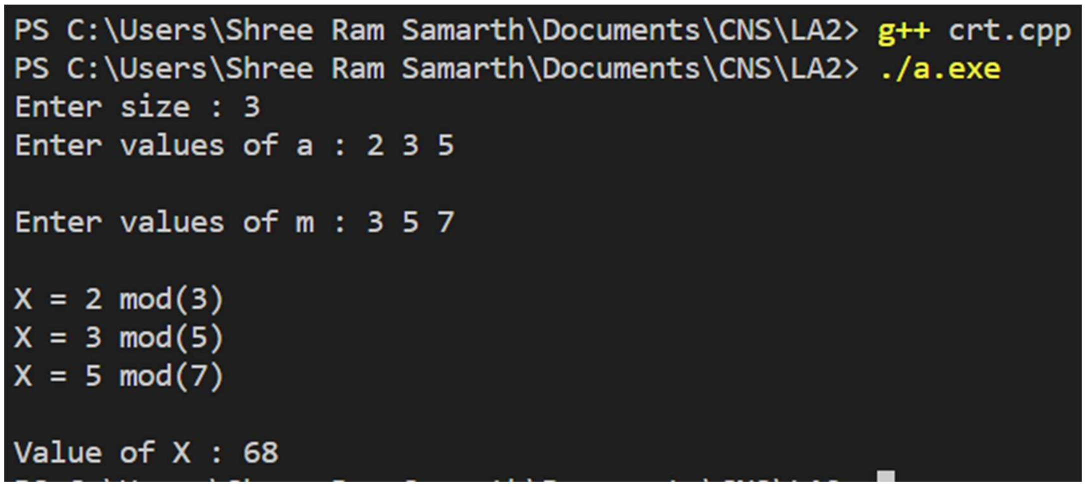
Disadvantages:

1. The CRT algorithm can be complex and requires knowledge of prime factorization and modular inverses.
2. CRT is applicable when you can factor 'N' into its prime factors. In cases where 'N' is not factorizable, CRT cannot be used.
3. The CRT can introduce errors if not implemented carefully due to the risk of overflow or precision issues in intermediate calculations.
4. In cryptography, the CRT can potentially be vulnerable to attacks if not implemented properly, leading to information leaks.

Code:

```
#include <bits/stdc++.h>
using namespace std;

bool check(int m[],int n){
    for (int i=0;i<n;i++){
        for (int j=i+1;j<n;j++)
        {
            if (__gcd(m[i],m[j]) != 1) {
                return true;
            }
        }
    }
    return false;
}

int CRT(int a[],int m[],int n){
    int x[n];
    int q,r,r1,r2,t,t1,t2;

    int M=1;
    for(int i=0;i<n;i++){
        M*=m[i];
    }

    for(int i=0;i<n;i++){
        r1=m[i];
        r2=M/m[i];
        t1=0,t2=1;

        while(r2>0){
            q=r1/r2;
            r=r1-q*r2;
            r1=r2;
            r2=r;

            t=t1-q*t2;
            t1=t2;
            t2=t;
        }

        if(r1==1){
            x[i]=t1;
        }
        if(x[i]<0){
            x[i]+=m[i];
        }
    }

    int res=0;
    for(int i=0;i<n;i++){
        res+=(a[i]*M*x[i])/m[i];
    }
    return res%M;
}

int main(){
    cout<<"Enter size : ";
    int n;
    cin>>n;

    int a[n],m[n];
    cout<<"Enter values of a : ";
    for(int i=0;i<n;i++){
        cin>>a[i];
    }cout<<endl;

    cout<<"Enter values of m : ";
    for(int i=0;i<n;i++){
        cin>>m[i];
    }cout<<endl;

    for(int i=0;i<n;i++){
        cout<<"X = "<<a[i]<<" mod("<<m[i]<<")"<<endl;
    }cout<<endl;

    if(!check(m,n)){
        cout<<"Value of X : "<<CRT(a,m,n)<<endl;
    }else{
        cout<<"Values of m are not co-prime.Thus, Solution does not
exist!!!"<<endl;
    }
}
```

Output:



```
PS C:\Users\Shree Ram Samarth\Documents\CNS\LA2> g++ crt.cpp
PS C:\Users\Shree Ram Samarth\Documents\CNS\LA2> ./a.exe
Enter size : 3
Enter values of a : 2 3 5

Enter values of m : 3 5 7

X = 2 mod(3)
X = 3 mod(5)
X = 5 mod(7)

Value of X : 68
```

Name: Karuna Bajirao Randive

Prn: 2020BTECS00024

Batch: B2

Cryptography & Network Security Lab

Title: Euclidean Theorem

Aim: To study and implement Euclidean Theorem.

Theory:

The Euclidean Algorithm is a fundamental mathematical algorithm used to find the greatest common divisor (GCD) of two integers. It has various applications in number theory, cryptography, and computer science. Here are the algorithm steps, advantages, and disadvantages of the Euclidean Algorithm:

Algorithm Steps:

- Start with two integers, 'a' and 'b,' where 'a' is greater than or equal to 'b.'
- Divide 'a' by 'b' to obtain a quotient 'q' and a remainder 'r,' such that:

$$a = b * q + r$$

- Set 'a' to 'b' and 'b' to 'r,' then repeat the division and remainder calculation until 'b' becomes zero.
- The algorithm terminates when 'b' becomes zero. The GCD is then the value of 'a' at this point.

Advantages:

1. The Euclidean Algorithm is highly efficient for finding the GCD of two integers, and its time complexity is proportional to the number of bits in the input values.
2. It can be applied to both positive and negative integers and is not limited to just two values; it can find the GCD of multiple integers.
3. Mathematical Simplicity: The algorithm is based on simple mathematical operations (division and remainder), making it easy to understand and implement.
4. Basis for Other Algorithms: The Euclidean Algorithm serves as the basis for various other algorithms, including the Extended Euclidean Algorithm used in modular arithmetic and modular inverses.

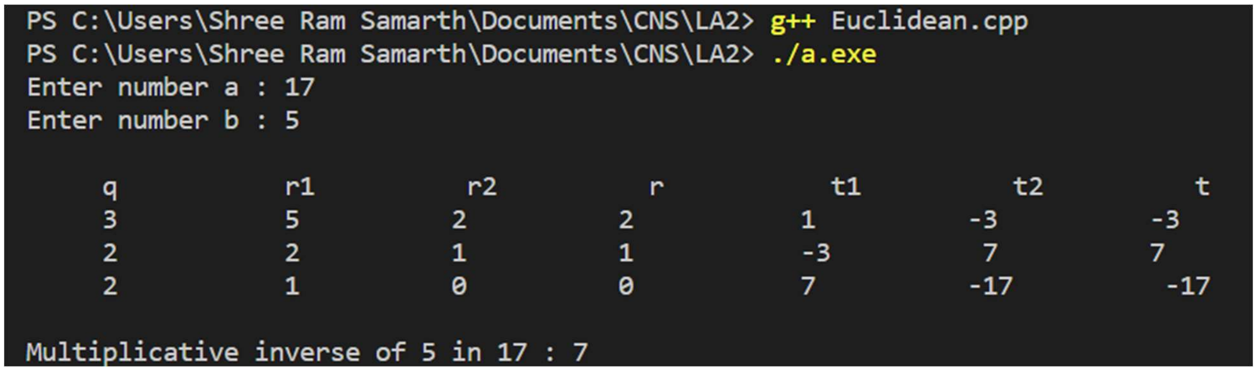
Disadvantages:

1. The algorithm works well for integers, but it may not handle very large numbers efficiently due to limitations in computational resources and time.
2. The Euclidean Algorithm is designed for integers and cannot be directly applied to non-integer data types.
3. If implemented recursively, the algorithm may lead to a stack overflow for extremely large input values. To avoid this, an iterative version is often preferred for practical applications.
4. The Euclidean Algorithm does not directly factor numbers or provide information about prime factorization, which may be needed in some applications.

Code:

```cpp
#include <bits/stdc++.h>
using namespace std;

int EuclideanAlgo(int a,int b){
    int s1=1,s2=0,s,t1=0,t2=1,t;
    int r1=a,r2=b,q,r;

    cout<<"     q           r1          r2          r           t1           t2
       t       "<<endl;
    while(r2>0){
        q=r1/r2;

        r=r1-q*r2;
        r1=r2;
        r2=r;

        // s=s1-q*s2;
        // s1=s2;
        // s2=s;

        t=t1-q*t2;
        t1=t2;
        t2=t;
        cout<<"     "<<q<<"           "<<r1<<"           "<<r2<<"           "<<r<<"
          "<<t1<<"          "<<t2<<"          "<<t<<endl;
    }
    cout<<endl;
    int res=-1;
    if(r1==1){
        res=t1;
    }
    return res;
}

int main(){
    int a,b;
    cout<<"Enter number a : ";
    cin>>a;

    cout<<"Enter number b : ";
    cin>>b;

    cout<<endl;

    int ans=INT_MAX;
    if(b>a){
        ans=EuclideanAlgo(b,a);
    }else{
        ans=EuclideanAlgo(a,b);
    }

    if(ans!=-1){
        cout<<"Multiplicative inverse of "<<b<<" in "<<a<<" : "<<ans;
        if(ans<0){
            if(b>a){
                ans+=b;
            }else{
                ans+=a;
            }
            cout<<" and "<<ans<<endl;
        }
    }else{
        cout<<"Multiplicative inverse is not present."<<endl;
    }
    cout<<endl;
}
```

Output:



```
PS C:\Users\Shree Ram Samarth\Documents\CNS\LA2> g++ Euclidean.cpp
PS C:\Users\Shree Ram Samarth\Documents\CNS\LA2> ./a.exe
Enter number a : 17
Enter number b : 5

     q           r1          r2          r           t1          t2          t
     3           5          2          2          1          -3          -3
     2           2          1          1          -3          7          7
     2           1          0          0          7          -17          -17

Multiplicative inverse of 5 in 17 : 7
```

Name: Karuna Bajirao Randive

Prn: 2020BTECS00024

Batch: B2

Cryptography & Network Security Lab

Title: Extended Euclidean Theorem.

Aim: To study and implement Extended Euclidean Theorem.

Theory:

The Extended Euclidean Algorithm is an extension of the Euclidean Algorithm that not only calculates the greatest common divisor (GCD) of two integers but also finds the Bézout coefficients, which are used to compute the modular inverse of an integer. Here are the algorithm steps, advantages, and disadvantages of the Extended Euclidean Algorithm:

Algorithm Steps:

- Start with two integers 'a' and 'b,' where 'a' is greater than or equal to 'b.'
- Initialize two sets of variables: 'x0,' 'x1,' 'y0,' and 'y1.' Set 'x0' and 'y1' to 1, and 'x1' and 'y0' to 0.
- Divide 'a' by 'b' to obtain a quotient 'q' and a remainder 'r,' such that:

$$a = b * q + r$$

- Update Variables: Update 'a' to 'b' and 'b' to 'r.' Then update the sets of variables as follows:
    - 'x0' becomes 'x1.'
    - 'x1' becomes 'x0 - q * x1.'
    - 'y0' becomes 'y1.'
    - 'y1' becomes 'y0 - q * y1.'
- Repeat steps 3 and 4 until 'b' becomes zero.
- At this point, 'a' is the GCD of the original 'a' and 'b,'. x0 and y0 (also called as Bézout coefficients)can be used to find the modular inverse of 'a' modulo 'b' if 'a' and 'b' are coprime (GCD = 1).

Advantages:

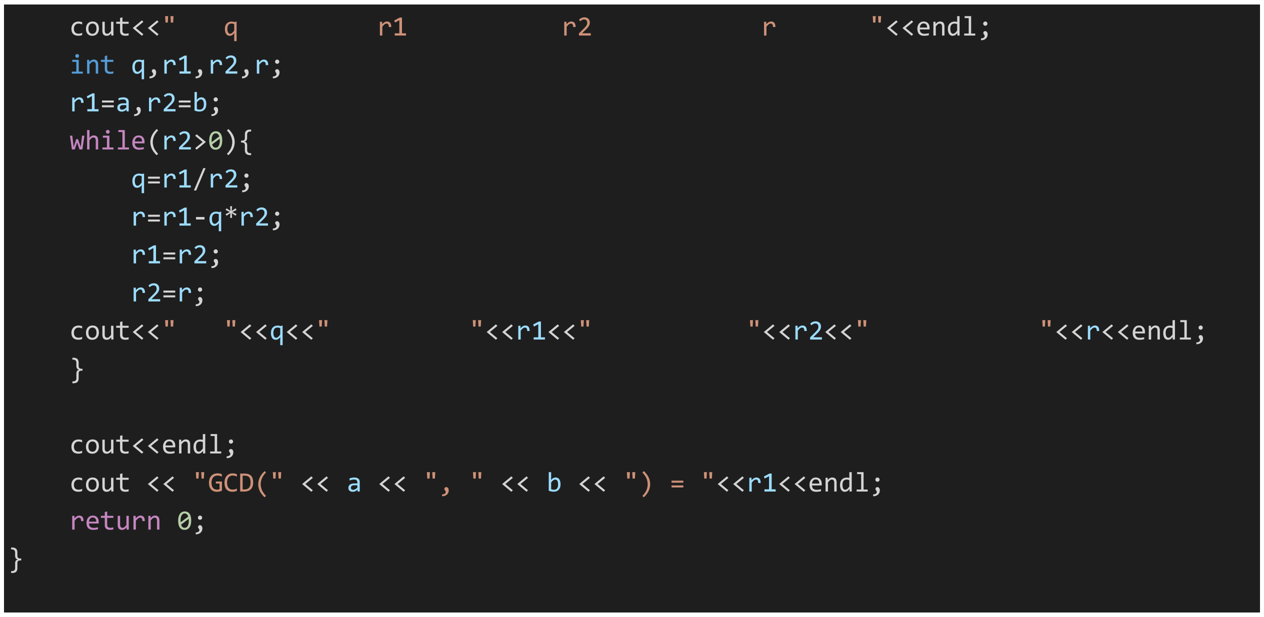
1. The Extended Euclidean Algorithm not only finds the GCD of two integers but also computes the Bézout coefficients, which are useful for solving linear Diophantine equations and calculating modular inverses.
2. It is an efficient algorithm for finding both the GCD and the Bézout coefficients. Its time complexity is proportional to the number of bits in the input values.
3. The algorithm can handle both positive and negative integers, making it suitable for various applications in number theory and cryptography.
4. The Bézout coefficients obtained from the Extended Euclidean Algorithm are used in modular arithmetic to find the modular inverse of an integer.

Disadvantages:

1. Like the Euclidean Algorithm, the Extended Euclidean Algorithm may not handle very large numbers efficiently due to computational resource limitations.
2. It is designed for integers and may not be directly applied to non-integer data types.
3. While the algorithm itself is conceptually straightforward, the implementation can become complex, especially in coding the updates to the variables, leading to potential programming errors.
4. The algorithm is primarily used for finding modular inverses and solving linear Diophantine equations. It may not be the best choice for other mathematical operations.

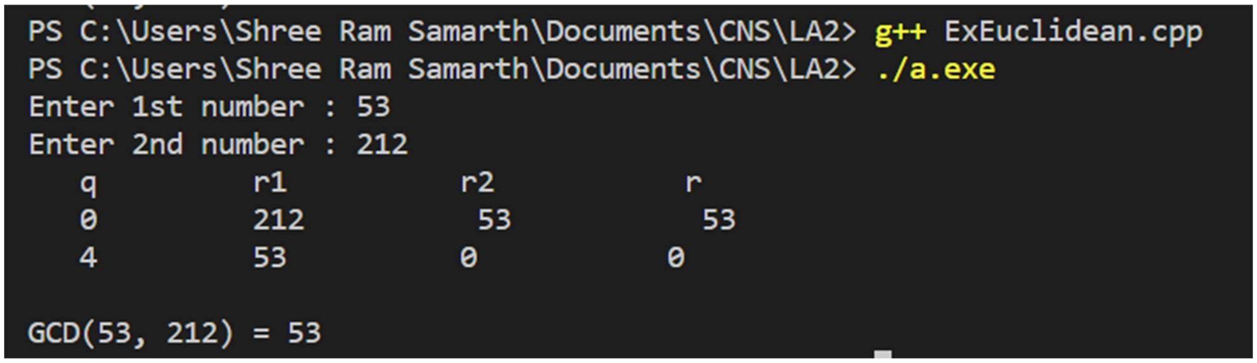
Code:

```
#include <bits/stdc++.h>
using namespace std;

int main(){
    int a,b;
    cout<<"Enter 1st number : ";
    cin>>a;

    cout<<"Enter 2nd number : ";
    cin>>b;
```



```
    cout<<"   q         r1          r2           r      "<<endl;
    int q,r1,r2,r;
    r1=a,r2=b;
    while(r2>0){
        q=r1/r2;
        r=r1-q*r2;
        r1=r2;
        r2=r;
    cout<<"   "<<q<<"         "<<r1<<"          "<<r2<<"           "<<r<<endl;
    }

    cout<<endl;
    cout << "GCD(" << a << ", " << b << ") = "<<r1<<endl;
    return 0;
}
```

Output:



```
PS C:\Users\Shree Ram Samarth\Documents\CNS\LA2> g++ ExEuclidean.cpp
PS C:\Users\Shree Ram Samarth\Documents\CNS\LA2> ./a.exe
Enter 1st number : 53
Enter 2nd number : 212
   q         r1          r2           r
   0         212          53           53
   4         53          0           0

GCD(53, 212) = 53
```

Name: Karuna Bajirao Randive

Prn: 2020BTECS00024

Batch: B2

Cryptography & Network Security Lab

Title: Advanced Encryption Standard.

Aim: To study and implement Advanced Encryption Standard.

Theory:

The Advanced Encryption Standard (AES) is a widely used symmetric encryption algorithm that provides strong data security. Here are the algorithm steps, advantages, and disadvantages of AES:

Algorithm Steps:

- The AES algorithm begins with a key expansion phase, where the original encryption key is expanded into a set of round keys. These round keys are used in the subsequent encryption rounds.
- **Initial Round (AddRoundKey):** The plaintext data is divided into blocks. In the initial round, the round key is added to the plaintext using a bitwise XOR operation.
- AES operates in a fixed number of rounds, which depends on the key size. For AES-128, there are 10 rounds; for AES-192, there are 12 rounds; and for AES-256, there are 14 rounds. Each round consists of the following operations:
    - **SubBytes:** Substitutes each byte in the block with a corresponding byte from the S-Box, a fixed substitution table.
    - **ShiftRows:** Shifts the rows of the block to the left by different offsets.
    - **MixColumns:** Mixes the columns of the block using a mathematical transformation.
    - **AddRoundKey:** Adds the round key to the block using a bitwise XOR operation.
- In the final round, the SubBytes, ShiftRows, and MixColumns operations are performed, followed by adding the final round key.
- The result after the final round is the ciphertext.

Advantages:

1. AES is widely considered to be highly secure and is used by governments, organizations, and individuals for protecting sensitive information.
2. AES is a fast and efficient encryption algorithm, making it suitable for a wide range of applications, including secure communication and data storage.
3. AES has been standardized by the National Institute of Standards and Technology (NIST) and is widely adopted and accepted in the security community.
4. AES supports multiple key lengths (128, 192, and 256 bits), allowing users to choose the level of security they need.
5. AES has withstood extensive cryptanalysis and has not been found to have any significant vulnerabilities.

Disadvantages:

1. AES is a symmetric encryption algorithm, which means that both the sender and receiver need to possess the same key. Secure key distribution can be a challenge, especially over insecure channels.
2. While AES is highly secure, it may not be the best choice for all encryption scenarios. For example, it may not be suitable for public-key encryption or digital signatures, which require asymmetric cryptography.
3. While AES is generally efficient, it can introduce some performance overhead, particularly for very resource-constrained devices or when using large key sizes.
4. Like many cryptographic algorithms, AES can be vulnerable to side-channel attacks (e.g., timing or power analysis attacks) if not properly implemented and protected.

Code:

```
#include<bits/stdc++.h>
using namespace std;

string hex2bin(string s){
    map<char,string> mp={{'0',"0000"},{'1',"0001"},{'2',"0010"},{'3',"0011"},
                         {'4',"0100"},{'5',"0101"},{'6',"0110"},{'7',"0111"},{'8'
,"1000"},{'9',"1001"},{'A',"1010"},{'B',"1011"},{'C',"1100"},{'D',"1101"},{'E',"1
110"},{'F',"1111"}};
    string bin="";
    for(int i=0;i<s.length();i++){
        bin+=mp[s[i]];
    }
    return bin;
}

void AddroundKey(string cipher_key[4][4],string key[4],int k){
    string ans="";
    for(int i=0;i<4;i++){
        string s1=hex2bin(cipher_key[k][i]);
        string s2=hex2bin(key[i]);

        int x1=stoi(s1);
        int x2=stoi(s2);

        int x=x1^x2;
        ans+=to_string(x);
    }

}

void g(string g_w[4]){
    for(int i=0;i<4;i++){
        g_w[i]=g_w[3][(i+1)%4];
    }
}

void KeyExpansion(string cipher_key[4][4],string words[10][4]){
    string g_w3[4];
    for(int i=0;i<4;i++){
        g_w3[i]=cipher_key[3][i];
    }
    g(g_w3);
    AddroundKey(cipher_key,g_w3,3);


}

int main(){
    string plain_text[4][4]={{"8E", "9F", "F1", "C6"},
                             {"4D", "DC", "E1", "C7"},
                                {"A1", "58", "D1", "C8"},
                                {"BC", "9D", "C1", "C9"}};

    string cipher_key[4][4]={{"24", "75", "A2", "B3"},
                                {"34", "75", "56", "88"},
                                {"31", "E2", "12", "00"},
                                {"13", "AA", "54", "87"}};

    string s_box[16][16]={
        {"63", "7C", "77", "7B", "F2", "6B", "6F", "C5", "30", "01", "67", "2B",
"FE", "D7", "AB", "76"},
        {"CA", "82", "C9", "7D", "FA", "59", "47", "F0", "AD", "D4", "A2", "AF",
"9C", "A4", "72", "C0"},
        {"B7", "FD", "93", "26", "36", "3F", "F7", "CC", "34", "A5", "E5", "F1",
"71", "D8", "31", "15"},
        {"04", "C7", "23", "C3", "18", "96", "05", "9A", "07", "12", "80", "E2",
"EB", "27", "B2", "75"},
        {"09", "83", "2C", "1A", "1B", "6E", "5A", "A0", "52", "3B", "D6", "B3",
"29", "E3", "2F", "84"},
        {"53", "D1", "00", "ED", "20", "FC", "B1", "5B", "6A", "CB", "BE", "39",
"4A", "4C", "58", "CF"},
        {"D0", "EF", "AA", "FB", "43", "4D", "33", "85", "45", "F9", "02", "7F",
"50", "3C", "9F", "A8"},
        {"51", "A3", "40", "8F", "92", "9D", "38", "F5", "BC", "B6", "DA", "21",
"10", "FF", "F3", "D2"},
        {"CD", "0C", "13", "EC", "5F", "97", "44", "17", "C4", "A7", "7E", "3D",
"64", "5D", "19", "73"},
        {"60", "81", "4F", "DC", "22", "2A", "90", "88", "46", "EE", "B8", "14",
"DE", "5E", "0B", "DB"},
        {"E0", "32", "3A", "0A", "49", "06", "24", "5C", "C2", "D3", "AC", "62",
"91", "95", "E4", "79"},
        {"E7", "C8", "37", "6D", "8D", "D5", "4E", "A9", "6C", "56", "F4", "EA",
"65", "7A", "AE", "08"},
        {"BA", "78", "25", "2E", "1C", "A6", "B4", "C6", "E8", "DD", "74", "1F",
"4B", "BD", "8B", "8A"},
        {"70", "3E", "B5", "66", "48", "03", "F6", "0E", "61", "35", "57", "B9",
"86", "C1", "1D", "9E"},
        {"E1", "F8", "98", "11", "69", "D9", "8E", "94", "9B", "1E", "87", "E9",
"CE", "55", "28", "DF"},
        {"8C", "A1", "89", "0D", "BF", "E6", "42", "68", "41", "99", "2D", "0F",
"B0", "54", "BB", "16"}};

    string RCon[10][4] ={{"01", "00", "00", "00"},
                         {"02", "00", "00", "00"},
                         {"03", "00", "00", "00"},
                        {"08", "00", "00", "00"},
                        {"10", "00", "00", "00"},
                        {"20", "00", "00", "00"},
                        {"40", "00", "00", "00"},
                        {"80", "00", "00", "00"},
                        {"1B", "00", "00", "00"},
                        {"36", "00", "00", "00"}};

    string words[10][4];
    KeyExpansion(cipher_key,words);


}
```

Output:

```
from block_cipher import BlockCipher, BlockCipherWrapper
from block_cipher import MODE_ECB, MODE_CBC, MODE_CFB, MODE_OFB, MODE_CTR

def print_str_into_AES_matrix(str):
    characters = ' '.join([str[i:i+2] for i in range(0, len(str), 2)]).split()
    matrix = [[characters[i] for i in range(j, j + 4)] for j in range(0,
len(characters), 4)]

    for col in range(4):
        for row in range(4):
            print(matrix[row][col], end=" ")
        print()

__all__ = [
    'new', 'block_size', 'key_size',
    'MODE_ECB', 'MODE_CBC', 'MODE_CFB', 'MODE_OFB', 'MODE_CTR'
]


SBOX = (
    0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5,
    0x30, 0x01, 0x67, 0x2b, 0xfe, 0xd7, 0xab, 0x76,
    0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0,
    0xad, 0xd4, 0xa2, 0xaf, 0x9c, 0xa4, 0x72, 0xc0,
    0xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc,
    0x34, 0xa5, 0xe5, 0xf1, 0x71, 0xd8, 0x31, 0x15,
    0x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05, 0x9a,
    0x07, 0x12, 0x80, 0xe2, 0xeb, 0x27, 0xb2, 0x75,
    0x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0,
    0x52, 0x3b, 0xd6, 0xb3, 0x29, 0xe3, 0x2f, 0x84,
    0x53, 0xd1, 0x00, 0xed, 0x20, 0xfc, 0xb1, 0x5b,
    0x6a, 0xcb, 0xbe, 0x39, 0x4a, 0x4c, 0x58, 0xcf,
    0xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85,
    0x45, 0xf9, 0x02, 0x7f, 0x50, 0x3c, 0x9f, 0xa8,
    0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5,
    0xbc, 0xb6, 0xda, 0x21, 0x10, 0xff, 0xf3, 0xd2,
    0xcd, 0x0c, 0x13, 0xec, 0x5f, 0x97, 0x44, 0x17,
    0xc4, 0xa7, 0x7e, 0x3d, 0x64, 0x5d, 0x19, 0x73,
    0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88,
    0x46, 0xee, 0xb8, 0x14, 0xde, 0x5e, 0x0b, 0xdb,
    0xe0, 0x32, 0x3a, 0x0a, 0x49, 0x06, 0x24, 0x5c,
    0xc2, 0xd3, 0xac, 0x62, 0x91, 0x95, 0xe4, 0x79,
    0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5, 0x4e, 0xa9,
    0x6c, 0x56, 0xf4, 0xea, 0x65, 0x7a, 0xae, 0x08,
    0xba, 0x78, 0x25, 0x2e, 0x1c, 0xa6, 0xb4, 0xc6,
    0xe8, 0xdd, 0x74, 0x1f, 0x4b, 0xbd, 0x8b, 0x8a,
    0x70, 0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e,
    0x61, 0x35, 0x57, 0xb9, 0x86, 0xc1, 0x1d, 0x9e,
    0xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0x8e, 0x94,
    0x9b, 0x1e, 0x87, 0xe9, 0xce, 0x55, 0x28, 0xdf,
    0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68,
    0x41, 0x99, 0x2d, 0x0f, 0xb0, 0x54, 0xbb, 0x16,
)
INV_SBOX = (
    0x52, 0x09, 0x6a, 0xd5, 0x30, 0x36, 0xa5, 0x38,
    0xbf, 0x40, 0xa3, 0x9e, 0x81, 0xf3, 0xd7, 0xfb,
    0x7c, 0xe3, 0x39, 0x82, 0x9b, 0x2f, 0xff, 0x87,
    0x34, 0x8e, 0x43, 0x44, 0xc4, 0xde, 0xe9, 0xcb,
    0x54, 0x7b, 0x94, 0x32, 0xa6, 0xc2, 0x23, 0x3d,
    0xee, 0x4c, 0x95, 0x0b, 0x42, 0xfa, 0xc3, 0x4e,
    0x08, 0x2e, 0xa1, 0x66, 0x28, 0xd9, 0x24, 0xb2,
    0x76, 0x5b, 0xa2, 0x49, 0x6d, 0x8b, 0xd1, 0x25,
    0x72, 0xf8, 0xf6, 0x64, 0x86, 0x68, 0x98, 0x16,
    0xd4, 0xa4, 0x5c, 0xcc, 0x5d, 0x65, 0xb6, 0x92,
    0x6c, 0x70, 0x48, 0x50, 0xfd, 0xed, 0xb9, 0xda,
    0x5e, 0x15, 0x46, 0x57, 0xa7, 0x8d, 0x9d, 0x84,
    0x90, 0xd8, 0xab, 0x00, 0x8c, 0xbc, 0xd3, 0x0a,
    0xf7, 0xe4, 0x58, 0x05, 0xb8, 0xb3, 0x45, 0x06,
    0xd0, 0x2c, 0x1e, 0x8f, 0xca, 0x3f, 0x0f, 0x02,
    0xc1, 0xaf, 0xbd, 0x03, 0x01, 0x13, 0x8a, 0x6b,
    0x3a, 0x91, 0x11, 0x41, 0x4f, 0x67, 0xdc, 0xea,
    0x97, 0xf2, 0xcf, 0xce, 0xf0, 0xb4, 0xe6, 0x73,
    0x96, 0xac, 0x74, 0x22, 0xe7, 0xad, 0x35, 0x85,
    0xe2, 0xf9, 0x37, 0xe8, 0x1c, 0x75, 0xdf, 0x6e,
```

```python
    0x47, 0xf1, 0x1a, 0x71, 0x1d, 0x29, 0xc5, 0x89,
    0x6f, 0xb7, 0x62, 0x0e, 0xaa, 0x18, 0xbe, 0x1b,
    0xfc, 0x56, 0x3e, 0x4b, 0xc6, 0xd2, 0x79, 0x20,
    0x9a, 0xdb, 0xc0, 0xfe, 0x78, 0xcd, 0x5a, 0xf4,
    0x1f, 0xdd, 0xa8, 0x33, 0x88, 0x07, 0xc7, 0x31,
    0xb1, 0x12, 0x10, 0x59, 0x27, 0x80, 0xec, 0x5f,
    0x60, 0x51, 0x7f, 0xa9, 0x19, 0xb5, 0x4a, 0x0d,
    0x2d, 0xe5, 0x7a, 0x9f, 0x93, 0xc9, 0x9c, 0xef,
    0xa0, 0xe0, 0x3b, 0x4d, 0xae, 0x2a, 0xf5, 0xb0,
    0xc8, 0xeb, 0xbb, 0x3c, 0x83, 0x53, 0x99, 0x61,
    0x17, 0x2b, 0x04, 0x7e, 0xba, 0x77, 0xd6, 0x26,
    0xe1, 0x69, 0x14, 0x63, 0x55, 0x21, 0x0c, 0x7d,
)

round_constants = (0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36)

block_size = 16
key_size = None


def new(key, mode, IV=None, **kwargs) -> BlockCipherWrapper:

    if mode in (MODE_CBC, MODE_CFB, MODE_OFB) and IV is None:
        raise ValueError("This mode requires an IV")

    cipher = BlockCipherWrapper()
    cipher.block_size = block_size
    cipher.IV = IV
    cipher.mode = mode
    cipher.cipher = AES(key)

    if mode == MODE_CFB:
        cipher.segment_size = kwargs.get('segment_size', block_size * 8)
    elif mode == MODE_CTR:
        counter = kwargs.get('counter')
        if counter is None:
            raise ValueError("CTR mode requires a callable counter object")
        cipher.counter = counter

    return cipher


class AES(BlockCipher):

    def __init__(self, key: bytes):
        self.key = key
        self.Nk = len(self.key) // 4  # words per key
        if self.Nk not in (4, 6, 8):
            raise ValueError("Invalid key size")
        self.Nr = self.Nk + 6
        self.Nb = 4  # words per block
        self.state: list[list[int]] = []
        # raise NotImplementedError
        # key schedule
        self.w: list[list[int]] = []
        for i in range(self.Nk):
            self.w.append(list(key[4*i:4*i+4]))
        for i in range(self.Nk, self.Nb*(self.Nr+1)):
            tmp: list[int] = self.w[i-1]
            q, r = divmod(i, self.Nk)
            if not r:
                tmp = self.sub_word(self.rot_word(tmp))
                tmp[0] ^= round_constants[q-1]
            elif self.Nk > 6 and r == 4:
                tmp = self.sub_word(tmp)
            self.w.append(
                [a ^ b for a, b in zip(self.w[i-self.Nk], tmp)]
            )

    def encrypt_block(self, block: bytes) -> bytes:

        self.set_state(block)

        self.add_round_key(0)
        print("\nInitial:")
        print_str_into_AES_matrix(self.get_state().hex())

        for r in range(1, self.Nr):
            # print(f"\nRound {r}:")

            self.sub_bytes()
            # print("After SubBytes:")
            # print_str_into_AES_matrix(self.get_state().hex())

            self.shift_rows()
            # print("After ShiftRows:")
            # print_str_into_AES_matrix(self.get_state().hex())

            self.mix_columns()
```

```python
            # print("After MixColumns:")
            # print_str_into_AES_matrix(self.get_state().hex())

            self.add_round_key(r)
            # print("After AddRoundKey:")
            # print_str_into_AES_matrix(self.get_state().hex())

        print(f"\nFinal Round {r+1}:")
        self.sub_bytes()
        print("After SubBytes:")
        print_str_into_AES_matrix(self.get_state().hex())

        self.shift_rows()
        print("After ShiftRows:")
        print_str_into_AES_matrix(self.get_state().hex())

        self.add_round_key(self.Nr)
        print("After AddRoundKey:")
        print_str_into_AES_matrix(self.get_state().hex())

        return self.get_state()

    def decrypt_block(self, block: bytes) -> bytes:

        self.set_state(block)
        print("\nInitial:")
        print_str_into_AES_matrix(self.get_state().hex())


        self.add_round_key(self.Nr)
        for r in range(self.Nr-1, 0, -1):
            # print(f"\nRound {r}:")

            self.inv_shift_rows()
            # print("After Inverse ShiftRows:")
            # print_str_into_AES_matrix(self.get_state().hex())

            self.inv_sub_bytes()
            # print("After Inverse SubBytes:")
            # print_str_into_AES_matrix(self.get_state().hex())

            self.add_round_key(r)
            # print("After AddRoundKey:")
            # print_str_into_AES_matrix(self.get_state().hex())
            self.inv_mix_columns()
            # print("After Inverse MixColumns:")
            # print_str_into_AES_matrix(self.get_state().hex())

        print(f"\nFinal Round {r}:")
        self.inv_shift_rows()
        print("After Inverse ShiftRows:")
        print_str_into_AES_matrix(self.get_state().hex())

        self.inv_sub_bytes()
        print("After Inverse SubBytes:")
        print_str_into_AES_matrix(self.get_state().hex())

        self.add_round_key(0)
        print("After AddRoundKey:")
        print_str_into_AES_matrix(self.get_state().hex())

        return self.get_state()

    @staticmethod
    def rot_word(word: list[int]):
        # for key schedule
        return word[1:] + word[:1]

    @staticmethod
    def sub_word(word: list[int]):
        # for key schedule
        return [SBOX[b] for b in word]

    def set_state(self, block: bytes):

        self.state = [
            list(block[i:i+4])
            for i in range(0, 16, 4)
        ]

    def get_state(self) -> bytes:

        return b''.join(
            bytes(col)
            for col in self.state
        )

    def add_round_key(self, r: int):
        round_key = self.w[r*self.Nb:(r+1)*self.Nb]
        for col, word in zip(self.state, round_key):
            for row_index in range(4):
                col[row_index] ^= word[row_index]

    def mix_columns(self):

        for i, word in enumerate(self.state):
            new_word = []
            for j in range(4):
                # element wise cl mul with constants 2, 3, 1, 1
                value = (word[0] << 1)
                value ^= (word[1] << 1) ^ word[1]
                value ^= word[2] ^ word[3]
                # polynomial reduction in constant time
                value ^= 0x11b & -(value >> 8)
                new_word.append(value)
                # rotate word in order to match the matrix multiplication
                word = self.rot_word(word)
            self.state[i] = new_word

    def inv_mix_columns(self):

        for i, word in enumerate(self.state):
            new_word = []
            for j in range(4):
                # element wise cl mul with constants 0xe, 0xb, 0xd, 0x9
                value = (word[0] << 3) ^ (word[0] << 2) ^ (word[0] << 1)
                value ^= (word[1] << 3) ^ (word[1] << 1) ^ word[1]
                value ^= (word[2] << 3) ^ (word[2] << 2) ^ word[2]
                value ^= (word[3] << 3) ^ word[3]
                # polynomial reduction in constant time
                value ^= (0x11b << 2) & -(value >> 10)
                value ^= (0x11b << 1) & -(value >> 9)
                value ^= 0x11b & -(value >> 8)
                new_word.append(value)
                # rotate word in order to match the matrix multiplication
                word = self.rot_word(word)
            self.state[i] = new_word

    def shift_rows(self):

        for row_index in range(4):
            row = [
                col[row_index] for col in self.state
```

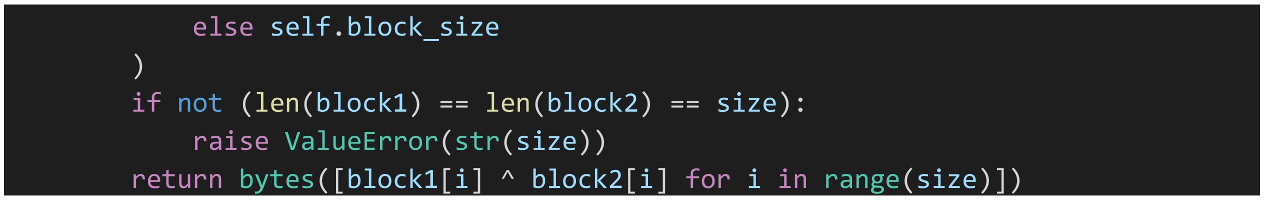
```python
            ]
            row = row[row_index:] + row[:row_index]
            for col_index in range(4):
                self.state[col_index][row_index] = row[col_index]

    def inv_shift_rows(self):

        for row_index in range(4):
            row = [
                col[row_index] for col in self.state
            ]
            row = row[-row_index:] + row[:-row_index]
            for col_index in range(4):
                self.state[col_index][row_index] = row[col_index]

    def sub_bytes(self):

        for col in self.state:
            for row_index in range(4):
                col[row_index] = SBOX[col[row_index]]

    def inv_sub_bytes(self):

        for col in self.state:
            for row_index in range(4):
                col[row_index] = INV_SBOX[col[row_index]]

    def print_state(self):
        # debug function
        for row_index in range(4):
            print(' '.join(f'{col[row_index]:02x}' for col in self.state))
        print()

# Main Code
ch = int(input("What do you want to perform?\n1. Encryption\n2. Decryption\n"))

if(ch == 1):
    msg = str(input("Enter the message to be encrypted(16 characters only):\n"))
    if(len(msg) != 16):
        print("Invalid Message size!")
        exit()

    key = str(input("Enter the key for encryption(16 or 24 or 32 
characters):\n"))
    key_length = len(key)
    if (key_length!=16 and key_length!=24 and key_length!=32):
        print("Invalid Key size!")
        exit()

    # mode = int(input("Choose the Mode of Operation:\n1. ECB\n2. CBC\n3. CFB\n4.
OFB\n5. CTR\n"))
    mode = 1

    iv = None
    if mode == 1:
        AES_MODE = MODE_ECB
    elif mode == 2:
        AES_MODE = MODE_CBC
        iv = str(input("Enter the Initialization Vector(IV) [16 characters]:\n"))
        if(len(iv) != 16):
            print("Invalid IV size!")
            exit()
    elif mode == 3:
        AES_MODE = MODE_CFB
        iv = str(input("Enter the Initialization Vector(IV) [16 characters]:\n"))
        if(len(iv) != 16):
            print("Invalid IV size!")
            exit()
    elif mode == 4:
        AES_MODE = MODE_OFB
        iv = str(input("Enter the Initialization Vector(IV) [16 characters]:\n"))
        if(len(iv) != 16):
            print("Invalid IV size!")
            exit()
    elif mode == 5:
        AES_MODE = MODE_CTR
    else:
        print("Invalid choice !!")
        exit()

    key = bytes.fromhex(key.encode('utf-8').hex())
    plain_text = bytes.fromhex(msg.encode('utf-8').hex())
    if iv is not None:
        iv = bytes.fromhex(iv.encode('utf-8').hex())

    cipher = new(key, AES_MODE, IV=iv)
    cipher_text = cipher.encrypt(plain_text)

    print(f"\nCiphertext is: {cipher_text.hex()}")
```

```python
elif(ch == 2):
    c_txt = str(input("Enter the ciphertext to be decrypted(16 characters) [in 
hex format]:\n"))
    if(len(c_txt) != 32):
        print("Invalid Cipher text size!")
        exit()

    key = str(input("Enter the key for decryption(16 or 24 or 32 
characters):\n"))
    key = bytes.fromhex(key.encode('utf-8').hex())
    key_length = len(key)
    if (key_length!=16 and key_length!=24 and key_length!=32):
        print("Invalid Key size!")
        exit()

    # mode = int(input("Choose the Mode of Operation used:\n1. ECB\n2. CBC\n3. 
CFB\n4. OFB\n5. CTR\n"))
    mode = 1

    iv = None
    if mode == 1:
        AES_MODE = MODE_ECB
    elif mode == 2:
        AES_MODE = MODE_CBC
        iv = str(input("Enter the Initialization Vector(IV) [16 characters]:\n"))
        if(len(iv) != 16):
            print("Invalid IV size!")
            exit()
    elif mode == 3:
        AES_MODE = MODE_CFB
        iv = str(input("Enter the Initialization Vector(IV) [16 characters]:\n"))
        if(len(iv) != 16):
            print("Invalid IV size!")
            exit()
    elif mode == 4:
        AES_MODE = MODE_OFB
        iv = str(input("Enter the Initialization Vector(IV) [16 characters]:\n"))
        if(len(iv) != 16):
            print("Invalid IV size!")
            exit()
    elif mode == 5:
        AES_MODE = MODE_CTR
    else:
        print("Invalid choice !!")
        exit()
    if iv is not None:
        iv = bytes.fromhex(iv.encode('utf-8').hex())

    cipher = new(key, AES_MODE, IV=iv)
    dec_bytes = cipher.decrypt(bytes.fromhex(c_txt))
    dec_txt = dec_bytes.decode('utf-8')

    print(f"\nDecrypted message is: {dec_txt}")

else:
    print("Invalid input!")
```

block cipher :

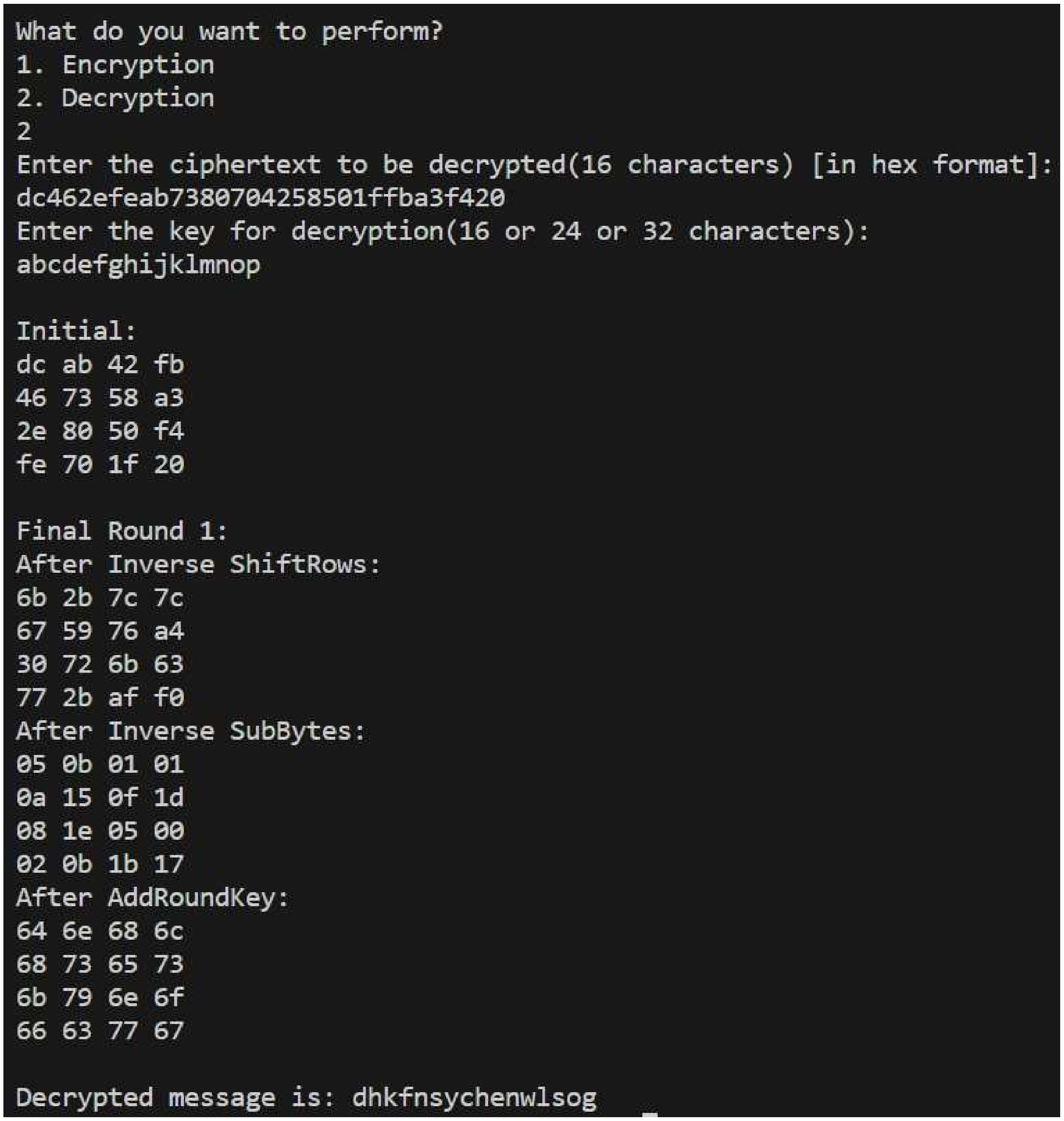
```
MODE_ECB = 1
MODE_CBC = 2
MODE_CFB = 3
MODE_PGP = 4  # optional
MODE_OFB = 5
MODE_CTR = 6


class BlockCipher:

    def encrypt_block(self, block: bytes) -> bytes:

        raise NotImplementedError

    def decrypt_block(self, block: bytes) -> bytes:

        raise NotImplementedError


class Counter:

    def __init__(self, nonce: bytes, block_size: int, byte_order='big'):

        self.nonce = nonce
        self.counter_size = block_size - len(nonce)
        self.byte_order = byte_order
        self.counter = 0

    def __call__(self) -> bytes:
```

```python
        out = self.nonce + self.counter.to_bytes(
            self.counter_size, self.byte_order
        )
        self.counter += 1
        return out


class BlockCipherWrapper:

    def __init__(self):
        """initiate instance attributes."""
        # PEP 272 required attributes
        self.block_size: int = NotImplemented  # measured in bytes
        self.IV: bytes = NotImplemented  # initialization vector
        # other attributes
        self.mode: int = NotImplemented
        self.cipher: BlockCipher = NotImplemented
        self.counter: Counter = NotImplemented
        self.segment_size: int = NotImplemented

    def encrypt(self, byte_string: bytes) -> bytes:

        if self.mode == MODE_CFB and len(byte_string) * 8 % self.segment_size:
            raise ValueError("Message length doesn't match segment size")
        if self.mode == MODE_CFB and self.segment_size & 7:
            raise NotImplementedError
        if self.mode != MODE_CFB and len(byte_string) % self.block_size:
            raise ValueError("Message length doesn't match block size")

        blocks = [
            byte_string[i:i+self.block_size]
            for i in range(0, len(byte_string), self.block_size)
        ]

        if self.mode == MODE_ECB:
            return b''.join([
                self.cipher.encrypt_block(block) for block in blocks
            ])
        elif self.mode == MODE_CBC:
            cipher_blocks = [self.IV]
            for block in blocks:
                cipher_blocks.append(
                    self.cipher.encrypt_block(
                        self.xor(block, cipher_blocks[-1])
                    )
                )
            return b''.join(cipher_blocks[1:])
        elif self.mode == MODE_CFB:
            s = self.segment_size >> 3
            cipher = b''
            current_input = self.IV
            while byte_string:
                cipher += self.xor(
                    byte_string[:s],
                    self.cipher.encrypt_block(current_input)[:s]
                )
                byte_string = byte_string[s:]
                current_input = current_input[s:] + cipher[-s:]
            return cipher
        elif self.mode == MODE_PGP:
            raise NotImplementedError
        elif self.mode == MODE_OFB:
            last_output = self.IV
            cipher_blocks = [self.IV]
            for block in blocks:
                last_output = self.cipher.encrypt_block(last_output)
                cipher_blocks.append(self.xor(block, last_output))
            return b''.join(cipher_blocks[1:])
        elif self.mode == MODE_CTR:
            cipher_blocks = []
            for block in blocks:
                ctr = self.counter()
                if len(ctr) != self.block_size:
                    raise ValueError("Counter has the wrong size")
                cipher_blocks.append(
                    self.xor(self.cipher.encrypt_block(ctr), block)
                )
            return b''.join(cipher_blocks)
        else:
            raise NotImplementedError("This mode is not supported")

    def decrypt(self, byte_string: bytes) -> bytes:

        if self.mode == MODE_CFB and len(byte_string) * 8 % self.segment_size:
            raise ValueError("Message length doesn't match segment size")
        if self.mode == MODE_CFB and self.segment_size & 7:
            raise NotImplementedError
        if self.mode != MODE_CFB and len(byte_string) % self.block_size:
            raise ValueError("Message length doesn't match block size")
        # split up into blocks
        blocks = [
            byte_string[i:i+self.block_size]
            for i in range(0, len(byte_string), self.block_size)
        ]

        if self.mode == MODE_ECB:
            return b''.join([
                self.cipher.decrypt_block(block)
                for block in blocks
            ])
        elif self.mode == MODE_CBC:
            plain_blocks = []
            blocks.insert(0, self.IV)
            for i in range(1, len(blocks)):
                plain_blocks.append(self.xor(
                    self.cipher.decrypt_block(blocks[i]), blocks[i-1]
                ))
            return b''.join(plain_blocks)
        elif self.mode == MODE_CFB:
            s = self.segment_size >> 3
            plain = b''
            current_input = self.IV
            while byte_string:
                plain += self.xor(
                    byte_string[:s],
                    self.cipher.encrypt_block(current_input)[:s]
                )
                current_input = current_input[s:] + byte_string[:s]
                byte_string = byte_string[s:]
            return plain
        elif self.mode == MODE_PGP:
            raise NotImplementedError("PGP mode is not supported")
        elif self.mode == MODE_OFB:
            return self.encrypt(byte_string)
        elif self.mode == MODE_CTR:
            return self.encrypt(byte_string)
        else:
            raise ValueError("Unknown mode")

    def xor(self, block1, block2):

        size = (
            self.segment_size >> 3
            if self.mode == MODE_CFB
```

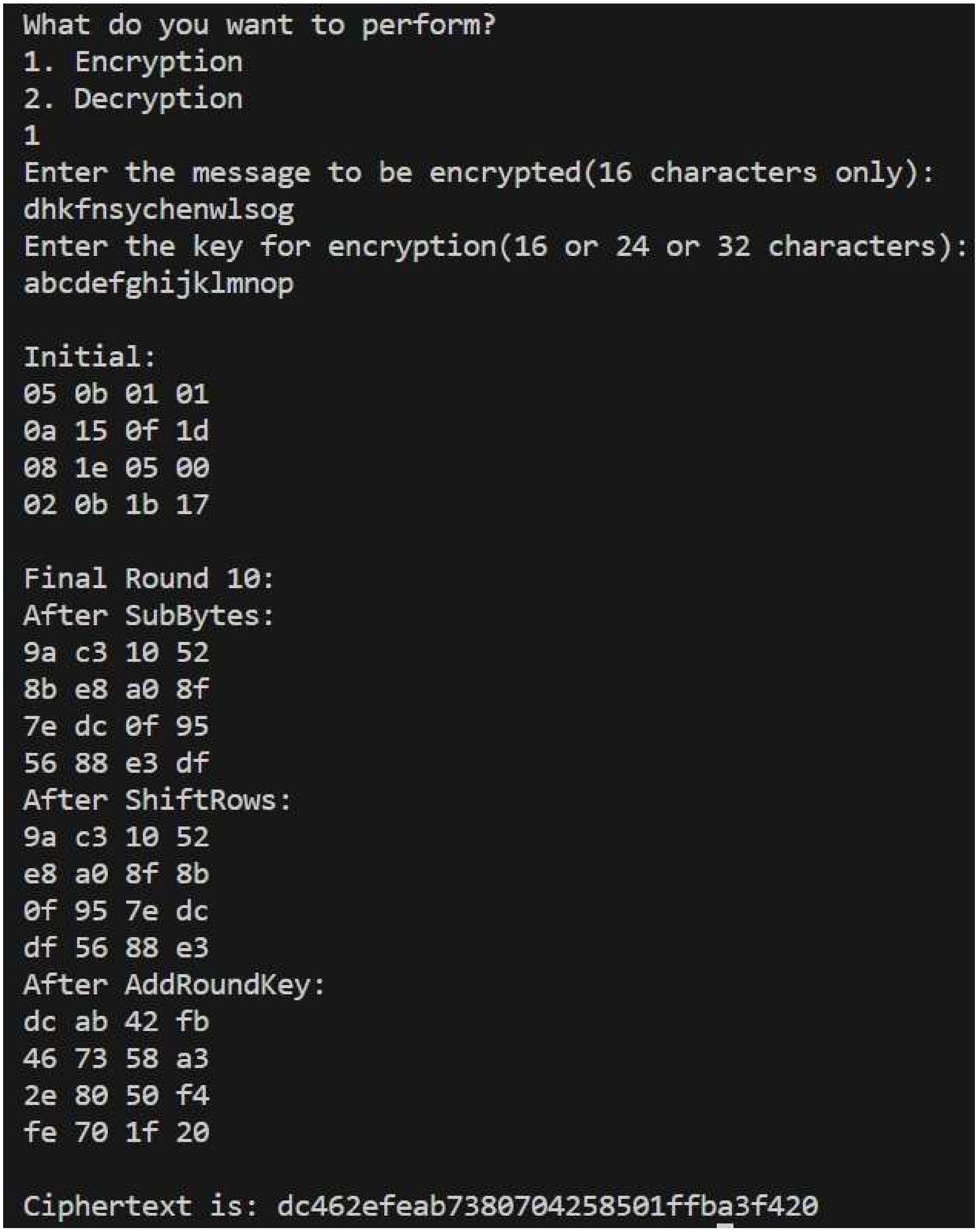


```
            else self.block_size
        )
        if not (len(block1) == len(block2) == size):
            raise ValueError(str(size))
        return bytes([block1[i] ^ block2[i] for i in range(size)])
```

Output:



```
What do you want to perform?
1. Encryption
2. Decryption
2
Enter the ciphertext to be decrypted(16 characters) [in hex format]:
dc462efeab7380704258501ffba3f420
Enter the key for decryption(16 or 24 or 32 characters):
abcdefghijklmnop

Initial:
dc ab 42 fb
46 73 58 a3
2e 80 50 f4
fe 70 1f 20

Final Round 1:
After Inverse ShiftRows:
6b 2b 7c 7c
67 59 76 a4
30 72 6b 63
77 2b af f0
After Inverse SubBytes:
05 0b 01 01
0a 15 0f 1d
08 1e 05 00
02 0b 1b 17
After AddRoundKey:
64 6e 68 6c
68 73 65 73
6b 79 6e 6f
66 63 77 67

Decrypted message is: dhkfnsychenwlsog
```



```
What do you want to perform?
1. Encryption
2. Decryption
1
Enter the message to be encrypted(16 characters only):
dhkfnsychenwlsog
Enter the key for encryption(16 or 24 or 32 characters):
abcdefghijklmnop

Initial:
05 0b 01 01
0a 15 0f 1d
08 1e 05 00
02 0b 1b 17

Final Round 10:
After SubBytes:
9a c3 10 52
8b e8 a0 8f
7e dc 0f 95
56 88 e3 df
After ShiftRows:
9a c3 10 52
e8 a0 8f 8b
0f 95 7e dc
df 56 88 e3
After AddRoundKey:
dc ab 42 fb
46 73 58 a3
2e 80 50 f4
fe 70 1f 20

Ciphertext is: dc462efeab7380704258501ffba3f420
```