Course: B.Tech. IV Year/ VII Sem
Cryptography and Network Security Lab (BCSE0071)

1. Alice and Bob wish to share private messages using a shift cipher algorithm to ensure confidentiality. Both can work as a sender and receiver so individual functions should be implemented for encryption, decryption and brute force with the following conditions:

- **Plaintext should be in lowercase. (Do not accept any number or special symbol)**
- **Ciphertext should be in uppercase. (Do not accept any number or special symbol)**
- **Brute force attack. (Find the key value)**

CODE:-

```
public class ShiftCipher {
 public static void main(String[] args) {
    String plaintext = "mayan";
    int key = 3;
    // Encryption
    String ciphertext = encrypt(plaintext, key);
    System.out.println("Encrypted: " + ciphertext);
    // Decryption
    String decryptedText = decrypt(ciphertext, key);
    System.out.println("Decrypted: " + decryptedText);
    // Brute Force Attack
    bruteForceAttack(ciphertext);


 // Encryption function
 public static String encrypt(String plaintext, int key) {
    StringBuilder ciphertext = new StringBuilder();
    for (int i = 0; i < plaintext.length(); i++) {
       char currentChar = plaintext.charAt(i);
       if (Character.isLowerCase(currentChar)) {
          char encryptedChar = (char) ((currentChar - 'a' + key) % 26 + 'A');
          ciphertext.append(encryptedChar);
       } else {
          // Ignore non-lowercase characters
        ciphertext.append(currentChar);
       }
    }
    return ciphertext.toString();
 }

 // Decryption function
 public static String decrypt(String ciphertext, int key) {
    StringBuilder decryptedText = new StringBuilder();
    for (int i = 0; i < ciphertext.length(); i++) {
       char currentChar = ciphertext.charAt(i);
       if (Character.isUpperCase(currentChar)) {
          char decryptedChar = (char) ((currentChar - 'A' - key + 26) % 26 + 'a');
          decryptedText.append(decryptedChar);
       } else {
          // Ignore non-uppercase characters
          decryptedText.append(currentChar);
       }
```

```
        }
        return decryptedText.toString();
    }


    // Brute Force Attack function
    public static void bruteForceAttack(String ciphertext) {
        System.out.println("Brute Force Attack:");
        for (int key = 1; key <= 25; key++) {
            String decryptedText = decrypt(ciphertext, key);
            System.out.println("Key " + key + ": " + decryptedText);
        }
    }
}
```

OUTPUT:

Encrypted: HADPSOHSODL
Decrypted: mayank
Brute Force Attack:
Key 0: hadpsohsodlq
Key 1: gzcorngrnckpv
Key 2: fybnqmfqmbj
Key 3: exampleplaint
Key 4: dwzlokdokzhm
Key 5: cvyknjcnjyglrcvr
Key 6: buxjmibmixfkqb
Key 7: atwilhalhwejpa
Key 8: zsvhkgzkgvdioz
Key 9: yrugjfyjfuchny
Key 10: xqtfiexietbg
Key 11: wpsehdwhds
Key 12: vordgcvgcrze
Key 13: unqcfbufbqyd
Key 14: tmpbeateapx
Key 15: sloadzsdzow
Key 16: rknzcyrcynv

2. Alice and Bob wish to share private messages using a multiplicative cipher algorithm to ensure confidentiality. The key value taken by both parties should be coprime with modulo 26. Both can work as a sender and receiver so individual functions should be implemented for encryption, decryption and brute force with the following conditions must be satisfied:

- **Plaintext should be in lowercase. (Do not accept any number or special symbol)**
- **Ciphertext should be uppercase. (Do not accept any number or special symbol)**
- **Brute force attack. (Find the key value**

CODE:-

```
public class MultiplicativeCipher {
    public static void main(String[] args) {
        String plaintext = "hello";
        int key = 9;
        // Check if the key is coprime with 26
        if (isCoprime(key, 26)) {
            // Encryption
            String ciphertext = encrypt(plaintext, key);
            System.out.println("Encrypted: " + ciphertext);
            // Decryption
            String decryptedText = decrypt(ciphertext, key);
            System.out.println("Decrypted: " + decryptedText)
            // Brute Force Attack
            bruteForceAttack(ciphertext);
        } else {
            System.out.println("Key is not coprime with 26. Choose a different key.");
        }
    }
    // Encryption function
    public static String encrypt(String plaintext, int key) {
        StringBuilder ciphertext = new StringBuilder()
        for (int i = 0; i < plaintext.length(); i++) {
            char currentChar = plaintext.charAt(i);
            if (Character.isLowerCase(currentChar)) {
                char encryptedChar = (char) (((currentChar - 'a') * key) % 26 + 'A');
                ciphertext.append(encryptedChar);
            } else {
                // Ignore non-lowercase characters
                ciphertext.append(currentChar);
            }
        }
        return ciphertext.toString();
    }
    // Decryption function
    public static String decrypt(String ciphertext, int key) {
        int modInverse = findModularInverse(key, 26);
        StringBuilder decryptedText = new StringBuilder();
        for (int i = 0; i < ciphertext.length(); i++) {
            char currentChar = ciphertext.charAt(i);
            if (Character.isUpperCase(currentChar)) {
                char decryptedChar = (char) (((currentChar - 'A') * modInverse) % 26 + 'a');
                decryptedText.append(decryptedChar);
```

```
            } else {
                // Ignore non-uppercase characters
                decryptedText.append(currentChar);
            }
        }
        return decryptedText.toString();
    }
    // Brute Force Attack function
    public static void bruteForceAttack(String ciphertext) {
        System.out.println("Brute Force Attack:");
        for (int key = 1; key <= 25; key++) {
            if (isCoprime(key, 26)) {
                String decryptedText = decrypt(ciphertext, key);
                System.out.println("Key " + key + ": " + decryptedText);
            }
        }
    }
    // Check if two numbers are coprime
    public static boolean isCoprime(int a, int b) {
        return findGCD(a, b) == 1;
    }
    // Find the greatest common divisor using Euclid's algorithm
    public static int findGCD(int a, int b) {
        while (b != 0) {
            int temp = b;
            b = a % b;
            a = temp;
        }
        return a;
    }
    // Find the modular inverse using extended Euclidean algorithm
    public static int findModularInverse(int a, int m) {
        for (int x = 1; x < m; x++) {
            if ((a * x) % m == 1) {
                return x;
            }
        }
        return -1; // Modular inverse does not exist
    }
}
```

OUTPUT :-

```
Encrypted: XCZZU
 Decrypted: hello
Key: 1 - Decrypted: kpmmh
 Key: 2 - Decrypted: qloot
Key: 3 - Decrypted: mfeel
Key: 4 - Decrypted: qloot
Key: 5 - Decrypted: cdssr
Key: 6 - Decrypted: qloot
 Key: 7 - Decrypted: uryyb
Key: 8 - Decrypted: qloot
Key: 9 - Decrypted: etkkv
Key: 10 - Decrypted: qloot
Key: 11 - Decrypted: izuud
```

3. To enhance the security Amit and Anil agreed to use the affine cipher algorithm with two keys. So the first key must be coprime with modulo 26 and the second key varies from 1 to 26. Both can work as a sender and receiver so individual functions should be implemented for encryption, decryption and brute force with the following conditions must be satisfied:

- **Plaintext should be in lowercase. (Do not accept any number or special symbol)**
- **Ciphertext should be uppercase. (Do not accept any number or special symbol)**
- **Brute force attack. (Find the key value**

CODE:-

```
public class AffineCipher {
    public static void main(String[] args) {
        String plaintext = "hello";
        int key1 = 5; // First key (coprime with 26)
        int key2 = 8; // Second key (1 to 26)
        // Check if the first key is coprime with 26
        if (isCoprime(key1, 26)) {
            // Encryption
            String ciphertext = encrypt(plaintext, key1, key2);
            System.out.println("Encrypted: " + ciphertext);
            // Decryption
            String decryptedText = decrypt(ciphertext, key1, key2);
            System.out.println("Decrypted: " + decryptedText);
            // Brute Force Attack
            bruteForceAttack(ciphertext);
        } else {
            System.out.println("First key is not coprime with 26. Choose a different key.");
        }
    }
    // Encryption function
    public static String encrypt(String plaintext, int key1, int key2) {
        StringBuilder ciphertext = new StringBuilder();
     for (int i = 0; i < plaintext.length(); i++) {
            char currentChar = plaintext.charAt(i);
            if (Character.isLowerCase(currentChar)) {
                char encryptedChar = (char) (((currentChar - 'a') * key1 + key2) % 26 + 'A');
                ciphertext.append(encryptedChar);
            } else {
                // Ignore non-lowercase characters
                ciphertext.append(currentChar);
            }
        }
        return ciphertext.toString();
    }
    // Decryption function
    public static String decrypt(String ciphertext, int key1, int key2) {
        int modInverse = findModularInverse(key1, 26);
        StringBuilder decryptedText = new StringBuilder();
        for (int i = 0; i < ciphertext.length(); i++) {
            char currentChar = ciphertext.charAt(i);
            if (Character.isUpperCase(currentChar)) {
                char decryptedChar = (char) ((modInverse * (currentChar - 'A' - key2 + 26)) % 26 + 'a');
```

```
            decryptedText.append(decryptedChar);
        } else {
            // Ignore non-uppercase characters
            decryptedText.append(currentChar);
        }
    }
    return decryptedText.toString();
}
// Brute Force Attack function
public static void bruteForceAttack(String ciphertext) {
    System.out.println("Brute Force Attack:");
    for (int key1 = 1; key1 <= 25; key1++) {
        if (isCoprime(key1, 26)) {
            for (int key2 = 1; key2 <= 26; key2++) {
            String decryptedText = decrypt(ciphertext, key1, key2);
                System.out.println("Key1: " + key1 + ", Key2: " + key2 + ": " + decryptedText);
            }
        }
    }

// Check if two numbers are coprime
public static boolean isCoprime(int a, int b) {
    return findGCD(a, b) == 1;
}
// Find the greatest common divisor using Euclid's algorithm
public static int findGCD(int a, int b) {
    while (b != 0) {
        int temp = b;
        b = a % b;
        a = temp;
    }
    return a;
}
// Find the modular inverse using extended Euclidean algorithm
public static int findModularInverse(int a, int m) {
    for (int x = 1; x < m; x++) {
        if ((a * x) % m == 1) {
            return x;
        }
    }
    return -1; // Modular inverse does not exist
}
}
```

```
OUTPUT:-
Encrypted: DLBBY
Decrypted: hello
Brute Force Attack:
Key1: 1, Key2: 1: pbjjm
Key1: 1, Key2: 2: oaiil
Key1: 1, Key2: 3: nhhhk
Key1: 5, Key2: 22: hello
Key1: 5, Key2: 23: gdkkn
Key1: 5, Key2: 24: fcjjm
```

4. Rustom wants to send a confidential message “Meet me after toga party” to Kelvin. So both are ready to share the initial key and other keys are generated automatically then the individual function should be implemented for encryption, decryption and brute force with the following conditions must be satisfied:

- **Plaintext should be in lowercase. (Do not accept any number or special symbol)**
- **Ciphertext should be uppercase. (Do not accept any number or special symbol)**
- **Brute force attack. (Find the key value**

CODE:-

```
public class VigenereCipher {
    public static void main(String[] args) {
        String plaintext = "meet me after toga party";
        String keyword = "KEY"; // Shared initial key
        // Encryption
        String ciphertext = encrypt(plaintext, keyword);
        System.out.println("Encrypted: " + ciphertext);
        // Decryption
        String decryptedText = decrypt(ciphertext, keyword);
        System.out.println("Decrypted: " + decryptedText);
        // Brute Force Attack
        bruteForceAttack(ciphertext);
    }
    // Encryption function
    public static String encrypt(String plaintext, String keyword) {
        StringBuilder ciphertext = new StringBuilder();
        int keywordIndex = 0;
        for (int i = 0; i < plaintext.length(); i++) {
            char currentChar = plaintext.charAt(i);
            if (Character.isLowerCase(currentChar)) {
                char encryptedChar = (char) ((currentChar - 'a' + keyword.charAt(keywordIndex) - 'A') %
26 + 'A');
                ciphertext.append(encryptedChar);
                // Move to the next letter in the keyword
                keywordIndex = (keywordIndex + 1) % keyword.length();
            } else {
                // Ignore non-lowercase characters
                ciphertext.append(currentChar);
            }
        }
        return ciphertext.toString();
    }
    // Decryption function
    public static String decrypt(String ciphertext, String keyword) {
        StringBuilder decryptedText = new StringBuilder();
        int keywordIndex = 0;
        for (int i = 0; i < ciphertext.length(); i++) {
            char currentChar = ciphertext.charAt(i);
            if (Character.isUpperCase(currentChar)) {
                char decryptedChar = (char) ((currentChar - 'A' - (keyword.charAt(keywordIndex) - 'A') +
26) % 26 + 'a');
                decryptedText.append(decryptedChar);
```

```
            // Move to the next letter in the keyword
            keywordIndex = (keywordIndex + 1) % keyword.length();
        } else {
            // Ignore non-uppercase characters
            decryptedText.append(currentChar);
        }
    }
 return decryptedText.toString();
  }
  // Brute Force Attack function
  public static void bruteForceAttack(String ciphertext) {
    System.out.println("Brute Force Attack:");
    for (int i = 0; i < 26; i++) {
        StringBuilder candidateKeyword = new StringBuilder();
        for (int j = 0; j < ciphertext.length(); j++) {
            char currentChar = ciphertext.charAt(j);
            if (Character.isUpperCase(currentChar)) {
                char decryptedChar = (char) ((currentChar - 'A' - i + 26) % 26 + 'a');
                candidateKeyword.append((char) ('A' + decryptedChar - ciphertext.charAt(j)));
            } else {
                // Ignore non-uppercase characters
                candidateKeyword.append(currentChar);
            }
        }
        System.out.println("Key: " + candidateKeyword.toString() + ", Decrypted: " +
decrypt(ciphertext, candidateKeyword.toString()));
    }
  }
}
```

```
OUTPUT:-
Encrypted: RNVQRIALQHFCFTZLC
Decrypted: meetmeaftertogaparty
Brute Force Attack:
Key: KEY, Decrypted: meetmeaftertogaparty
Key: JEX, Decrypted: sifblblrcflfzxcm
Key: OFT, Decrypted: bhlwkwkoaxaehihe
```

5. The project investigates a cipher that is somewhat more complicated than the simple substitution cipher. In the Playfair cipher, there is not a single translation of each letter of the alphabet; that is, you don't just decide that every B will be turned into an F. Instead, pairs of letters are translated into other pairs of letters. Here is how it works. To start, pick a keyword that does not contain any letter more than once. For example, I'll pick the word keyword. Now write the letters of that word in the first squares of a five-by-five matrix. Then finish filling up the remaining squares of the matrix with the remaining letters of the alphabet, in alphabetical order. Since there are 26 letters and only 25 squares, we assign I and J to the same square. So implement Playfair Cipher to encrypt & decrypt the given message where the key matrix can be formed by using a given keyword.

CODE:-

```
import java.util.Scanner;
public class PlayfairCipher
 {
   private char[][] keyMatrix;
   public PlayfairCipher(String key)
  {
      keyMatrix = generateKeyMatrix(key);
   }



   public String encrypt(String plaintext)
 {
      StringBuilder ciphertext = new StringBuilder();
      for (int i = 0; i < plaintext.length(); i += 2) {
         char first = plaintext.charAt(i);
         char second = (i + 1 < plaintext.length()) ? plaintext.charAt(i + 1) : 'X';
         int[] firstPos = findPosition(first);
         int[] secondPos = findPosition(second)
         ciphertext.append(getEncryptedPair(firstPos, secondPos));
      }
      return ciphertext.toString();
   }


   Public String decrypt(String ciphertext) {
      StringBuilder plaintext = new StringBuilder();
      for (int i = 0; i < ciphertext.length(); i += 2) {
         char first = ciphertext.charAt(i);
         char second = ciphertext.charAt(i + 1);
    int[] firstPos = findPosition(first);
         int[] secondPos = findPosition(second);
         plaintext.append(getDecryptedPair(firstPos, secondPos));
      }
      return plaintext.toString();
   }


   private char[][] generateKeyMatrix(String key) {
      char[][] matrix = new char[5][5];
```

```
        boolean[] usedLetters = new boolean[26];
        int row = 0, col = 0
        for (char letter : key.toCharArray()) {
            if (!usedLetters[letter - 'A']) {
                matrix[row][col] = letter;
                usedLetters[letter - 'A'] = true;
                col = (col + 1) % 5;
                if (col == 0) row++;
            }
        }
        for (char letter = 'A'; letter <= 'Z'; letter++)
       {
            if (letter != 'J' && !usedLetters[letter - 'A'])
        {
                matrix[row][col] = letter;
                col = (col + 1) % 5;
                if (col == 0) row++
        }
        return matrix;
    }


    private int[] findPosition(char letter) {
        int[] position = new int[2];
        for (int i = 0; i < keyMatrix.length; i++) {
            for (int j = 0; j < keyMatrix[i].length; j++) {
                if (keyMatrix[i][j] == letter) {
                    position[0] = i;
                    position[1] = j;
                  return position;
                }
            }
        }
        return position;
    }



    private String getEncryptedPair(int[] firstPos, int[] secondPos)
{
        if (firstPos[0] == secondPos[0]) {
            firstPos[1] = (firstPos[1] + 1) % 5;
            secondPos[1] = (secondPos[1] + 1) % 5;
        } else if (firstPos[1] == secondPos[1]) {
            firstPos[0] = (firstPos[0] + 1) % 5;
            secondPos[0] = (secondPos[0] + 1) % 5;
        } else {
            int temp = firstPos[1];
            firstPos[1] = secondPos[1];
            secondPos[1] = temp;
        }

        return "" + keyMatrix[firstPos[0]][firstPos[1]] + keyMatrix[secondPos[0]][secondPos[1]];
    }
```

```
    private String getDecryptedPair(int[] firstPos, int[] secondPos) {
        if (firstPos[0] == secondPos[0]) {
            firstPos[1] = (firstPos[1] - 1 + 5) % 5;
            secondPos[1] = (secondPos[1] - 1 + 5) % 5;
        } else if (firstPos[1] == secondPos[1]) {
            firstPos[0] = (firstPos[0] - 1 + 5) % 5;
            secondPos[0] = (secondPos[0] - 1 + 5) % 5;
        } else {
            int temp = firstPos[1];
            firstPos[1] = secondPos[1];
            secondPos[1] = temp;
        }

        return "" + keyMatrix[firstPos[0]][firstPos[1]] + keyMatrix[secondPos[0]][secondPos[1]];
    }


    public static void main(String[] args)
{
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter the keyword: ");
        String keyword = scanner.nextLine().toUpperCase().replaceAll("[^A-Z]", "");
        PlayfairCipher playfairCipher = new PlayfairCipher(keyword);
        System.out.print("Enter the plaintext: ");
        String plaintext = scanner.nextLine().toUpperCase().replaceAll("[^A-Z]", "");
        String ciphertext = playfairCipher.encrypt(plaintext);
        System.out.println("Encrypted text: " + ciphertext);
        String decryptedText = playfairCipher.decrypt(ciphertext);
        System.out.println("Decrypted text: " + decryptedText);
        scanner.close();
    }
}
```

```
OUTPUT
Enter the keyword: PLAYFAIR
Enter the plaintext: HELLO
Encrypted text: EBFNX
Decrypted text: HELLO
```

6. Write a program to implement Hill Cipher to encrypt & decrypt the given message by using a given key matrix. Show the values for the key and its corresponding key inverse values.

CODE:-

```
import java.util.ArrayList;
import java.util.Scanner;
public class HillCipherExample
   private static int[][] getKeyMatrix() {
      Scanner scanner = new Scanner(System.in);
      System.out.println("Enter key matrix:");
      String key = scanner.nextLine().replaceAll("[^a-zA-Z]", "").toUpperCase();
      int size = (int) Math.sqrt(key.length());
      if (key.length() != size * size) {
         throw new RuntimeException("Cannot form a square matrix");
      }
      int[][] keyMatrix = new int[size][size];
      int index = 0;
      for (int i = 0; i < size; i++) {
         for (int j = 0; j < size; j++) {
            keyMatrix[i][j] = key.charAt(index++) - 'A';
         }
      }

      return keyMatrix;
   }
   private static int calculateDeterminant(int[][] matrix) {
      return matrix[0][0] * matrix[1][1] - matrix[0][1] * matrix[1][0];
   }
   private static int[][] calculateInverseMatrix(int[][] keyMatrix) {
      int determinant = calculateDeterminant(keyMatrix) % 26;
      int inverseDeterminant = 0
      for (int i = 1; i < 26; i++) {
         if ((determinant * i) % 26 == 1) {
            inverseDeterminant = i;
            break;
         }
      }

      int[][] inverseMatrix = new int[2][2];
      inverseMatrix[0][0] = keyMatrix[1][1] * inverseDeterminant % 26;
      inverseMatrix[0][1] = (26 - keyMatrix[0][1]) * inverseDeterminant % 26;
      inverseMatrix[1][0] = (26 - keyMatrix[1][0]) * inverseDeterminant % 26;
      inverseMatrix[1][1] = keyMatrix[0][0] * inverseDeterminant % 26;
      return inverseMatrix;
   }


   private static ArrayList<Integer> convertToNumbers(String text) {
      ArrayList<Integer> numbers = new ArrayList<>();
      text = text.replaceAll("[^a-zA-Z]", "").toUpperCase();
      for (char c : text.toCharArray()) {
         numbers.add(c - 'A');
      }
```

```
        return numbers;
    }




    private static void printResult(String label, int adder, ArrayList<Integer> numbers) {
        System.out.print(label);
        for (int i = 0; i < numbers.size(); i++) {
            System.out.print(Character.toChars(numbers.get(i) + adder + 'A'));
            if (i + 1 < numbers.size()) {
                System.out.print("-");
            }
        }
        System.out.println();
    }



    private static void performEncryptionOrDecryption(boolean isEncryption, String text, boolean
alphaZero) {
        int adder = alphaZero ? 0 : 1;
        int[][] keyMatrix = getKeyMatrix();
        int[][] inverseMatrix = calculateInverseMatrix(keyMatrix);
        ArrayList<Integer> numbers = convertToNumbers(text);
        ArrayList<Integer> result = new ArrayList<>();
        for (int i = 0; i < numbers.size(); i += 2) {
            int x = (keyMatrix[0][0] * numbers.get(i) + keyMatrix[0][1] * numbers.get(i + 1)) % 26;
            int y = (keyMatrix[1][0] * numbers.get(i) + keyMatrix[1][1] * numbers.get(i + 1)) % 26;
            result.add(alphaZero ? x : (x == 0 ? 26 : x));
            result.add(alphaZero ? y : (y == 0 ? 26 : y));
        }

        printResult((isEncryption ? "Encoded" : "Decoded") + " phrase: ", adder, result);
    }
    public static void encrypt(String text, boolean alphaZero) {
        performEncryptionOrDecryption(true, text, alphaZero);
    }
    public static void decrypt(String text, boolean alphaZero) {

performEncryptionOrDecryption(false, text, alphaZero);
    }



    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.println("Hill Cipher Implementation (2x2)");
        System.out.println("-------------------------");
        System.out.println("1. Encrypt text (A=0, B=1, ..., Z=25)");
        System.out.println("2. Decrypt text (A=0, B=1, ..., Z=25)");
        System.out.println("3. Encrypt text (A=1, B=2, ..., Z=26)");
        System.out.println("4. Decrypt text (A=1, B=2, ..., Z=26)");
        System.out.println();
        System.out.println("Type any other character to exit");
```

```
        System.out.println();
        System.out.print("Select your choice: ");
        String opt = scanner.nextLine();
        switch (opt)



{
            case "1":
               System.out.print("Enter phrase to encrypt: ");
               String phraseToEncrypt = scanner.nextLine();
               encrypt(phraseToEncrypt, true);
               break;

            case "2":
               System.out.print("Enter phrase to decrypt: ");
               String phraseToDecrypt = scanner.nextLine();
               decrypt(phraseToDecrypt, true);
               break;

            case "3":
               System.out.print("Enter phrase to encrypt: ");
               String phraseToEncryptAlphaOne = scanner.nextLine();
               encrypt(phraseToEncryptAlphaOne, false);
               break;

            case "4":
               System.out.print
```

```
OUTPUT:
Hill Cipher Implementation (2x2)
------------------------
1. Encrypt text (A=0, B=1, ..., Z=25)
2. Decrypt text (A=0, B=1, ..., Z=25)
3. Encrypt text (A=1, B=2, ..., Z=26)
4. Decrypt text (A=1, B=2, ..., Z=26
Type any other character to exit
Select your choice: 1
Enter a phrase to encrypt: HELLO
Encoded phrase: DDAYN-KW
```

7. ElGamal cryptosystem can be defined as the cryptography algorithm that uses the public and private key concepts to secure communication between two systems. It can be considered the asymmetric algorithm where the encryption and decryption happen by using public and private keys. To encrypt the message, the public key is used by the client, while the message could be decrypted using the private key on the server end. This is considered an efficient algorithm to perform encryption and decryption as the keys are extremely tough to predict. The sole purpose of introducing the message transaction's signature is to protect it against MITM, which this algorithm could very effectively achieve. Write a program to implement Elgamal Cryptosystem to generate the pair of keys and then show the encryption & decryption of a given message.

CODE:-

```
import java.math.BigInteger;
import java.security.SecureRandom;
public class ElGamalEncryption {
   public static void main(String[] args) {
      KeyPair keyPair = generateKeyPair();
      BigInteger[] cipherText = encrypt("Hello", keyPair.getPublicKey());
            String decryptedMessage = decrypt(cipherText, keyPair.getPrivateKey())
      System.out.println("Public Key (p, g, y): " + keyPair.getPublicKey());
      System.out.println("Private Key (x): " + keyPair.getPrivateKey());
      System.out.println("Encrypted Message: " + cipherText[0] + ", " + cipherText[1]);
      System.out.println("Decrypted Message: " + decryptedMessage);
   }
   private static KeyPair generateKeyPair() {
      // Step 1: Select a large prime number p
      BigInteger p = new BigInteger("23");
      // Step 2: Select a primitive root g
      BigInteger g = new BigInteger("5");
      // Step 3: Choose private key x randomly in the range [1, p-2]
      BigInteger x = new BigInteger("6");
      // Step 4: Compute public key y = g^x mod p
      BigInteger y = g.modPow(x, p);
      return new KeyPair(new PublicKey(p, g, y), x);
   }
   private static BigInteger[] encrypt(String message, PublicKey publicKey) {
      BigInteger p = publicKey.getP();
      BigInteger g = publicKey.getG();
      BigInteger y = publicKey.getY();
      // Choose a random number k in the range [1, p-2]
      BigInteger k = new BigInteger(p.bitLength() - 2, new SecureRandom()).add(BigInteger.ONE);
      // Compute c1 = g^k mod p
      BigInteger c1 = g.modPow(k, p);
      // Compute c2 = (y^k * message) mod p
      BigInteger c2 = y.modPow(k, p).multiply(new BigInteger(message.getBytes())).mod(p);
      return new BigInteger[]{c1, c2};
   }
   private static String decrypt(BigInteger[] cipherText, BigInteger privateKey) {
      BigInteger p = privateKey;
      BigInteger c1 = cipherText[0];
      BigInteger c2 = cipherText[1];

      // Compute s = c1^x mod p
      BigInteger s = c1.modPow(privateKey, p);
```

```
        // Compute s_inv = s^(-1) mod p
        BigInteger sInv = s.modInverse(p);
        // Compute decrypted message = (c2 * s_inv) mod p
        BigInteger decryptedMessage = c2.multiply(sInv).mod(p)
        return new String(decryptedMessage.toByteArray());
    }
    static class KeyPair {
        private final PublicKey publicKey;
        private final BigInteger privateKey;
        public KeyPair(PublicKey publicKey, BigInteger privateKey) {
            this.publicKey = publicKey;
            this.privateKey = privateKey;
        }
        public PublicKey getPublicKey() {
            return publicKey;
        }
        public BigInteger getPrivateKey() {
            return privateKey;
        }
    static class PublicKey {
        private final BigInteger p;
        private final BigInteger g;
        private final BigInteger y
        public PublicKey(BigInteger p, BigInteger g, BigInteger y) {
            this.p = p;
            this.g = g;
            this.y = y;
        }

        public BigInteger getP() {
            return p;
        }
        public BigInteger getG() {
            return g;
        }
        public BigInteger getY() {
            return y;
        }
        @Override
        public String toString() {
            return "(" + p + ", " + g + ", " + y + ")";
        }
    }
}
```

```
OUTPUT:
Public Key (p, g, y): (23, 5, 8)
Private Key (x): 6
Encrypted Message: 10, 19
Decrypted Message: Hello
```

8. User A and B want to communicate with each other by shared key so both parties decided that using Asymmetric key cryptography to generate a shared key and exchange with the help of Diffie-Hellman key exchange Algorithm. Perform exchange encryption & decryption using key exchange algorithm.

CODE:-

```
import java.security.*;
import java.security.spec.X509EncodedKeySpec;
import java.util.Base64;
import javax.crypto.KeyAgreement;
public class DiffieHellman {

   public static void main(String[] args) throws Exception {
      KeyPairGenerator keyPairGenerator = KeyPairGenerator.getInstance("DH");
      keyPairGenerator.initialize(2048)

      // Alice

      KeyPair keyPairAlice = keyPairGenerator.generateKeyPair();
      PublicKey publicKeyAlice = keyPairAlice.getPublic();
      String publicKeyStrAlice =
Base64.getEncoder().encodeToString(publicKeyAlice.getEncoded());
     PublicKey receivedPublicKeyAlice = KeyFactory.getInstance("DH")
     .generatePublic(newX509EncodedKeySpec(Base64.getDecoder().decode(publicKeyStrAlice)));

      // Bob

      KeyPair keyPairBob = keyPairGenerator.generateKeyPair();
      PublicKey publicKeyBob = keyPairBob.getPublic();
      String publicKeyStrBob = Base64.getEncoder().encodeToString(publicKeyBob.getEncoded());
      PublicKey receivedPublicKeyBob = KeyFactory.getInstance("DH")
            .generatePublic(new
X509EncodedKeySpec(Base64.getDecoder().decode(publicKeyStrBob)));

      // Key agreement

      KeyAgreement keyAgreementAlice = KeyAgreement.getInstance("DH");
      keyAgreementAlice.init(keyPairAlice.getPrivate());
      keyAgreementAlice.doPhase(receivedPublicKeyBob, true);
      byte[] sharedSecretAlice = keyAgreementAlice.generateSecret();

      KeyAgreement keyAgreementBob = KeyAgreement.getInstance("DH");
      keyAgreementBob.init(keyPairBob.getPrivate());
      keyAgreementBob.doPhase(receivedPublicKeyAlice, true);
      byte[] sharedSecretBob = keyAgreementBob.generateSecret();

      System.out.println("Shared Secret Alice: " +
Base64.getEncoder().encodeToString(sharedSecretAlice));

      System.out.println("Shared Secret Bob: " +
Base64.getEncoder().encodeToString(sharedSecretBob));

   }
}
```

9. Alice is of one of the employee of company XYZ. He wants to ensure that whatever data he is sending to Bob should be checked for accuracy. Implement the RSA digital signature for the message "This is an example" for showing the Digital Signature in such a scenario.

CODE:-

```
import java.security.*;
public class RSADigitalSignature
{
   public static void main(String[] args) throws Exception

 {
      KeyPair keyPair = generateKeyPair();
      String message = "This is an mayank";
      byte[] signature = sign(message, keyPair.getPrivate());
      boolean isVerified = verify(message, signature, keyPair.getPublic());
      System.out.println("Is the signature verified? " + isVerified);
   }

   private static KeyPair generateKeyPair() throws Exception
{
      return KeyPairGenerator.getInstance("RSA").initialize(2048).generateKeyPair();
   }

   private static byte[] sign(String message, PrivateKey privateKey) throws Exception
 {
      Signature signature = Signature.getInstance("SHA256withRSA");
      signature.initSign(privateKey);
      signature.update(message.getBytes());
      return signature.sign();
   }

   private static boolean verify(String message, byte[] signature, PublicKey publicKey) throws
Exception
 {
      Signature verifier = Signature.getInstance("SHA256withRSA");
      verifier.initVerify(publicKey);
      verifier.update(message.getBytes());
      return verifier.verify(signature);
   }
}
```

10. Alice is of one of the employee of company XYZ. He wants to ensure that whatever data he is sending to Bob should be checked for accuracy. Implement the Elgamal digital signature for the message "Hello how are you" for showing the Digital Signature in such a scenario.

CODE:-

```
import java.security.*;
public class ElGamalDigitalSignature
{
   public static void main(String[] args) throws Exception
  {
      KeyPair keyPair = generateKeyPair();
      PrivateKey privateKey = keyPair.getPrivate();
      PublicKey publicKey = keyPair.getPublic();
      String message = "Hello, how are you";
      byte[] signature = sign(message, privateKey);
      boolean isVerified = verify(message, signature, publicKey);
      System.out.println("Message: " + message);
      System.out.println("Signature: " + bytesToHex(signature));
      System.out.println("Signature Verified: " + isVerified);
   }
   private static KeyPair generateKeyPair() throws NoSuchAlgorithmException {
      return KeyPairGenerator.getInstance("ElGamal").initialize(2048).generateKeyPair();
   }
   private static byte[] sign(String message, PrivateKey privateKey)
         throws NoSuchAlgorithmException, InvalidKeyException, SignatureException {
      Signature signature = Signature.getInstance("SHA256withElGamal");
      signature.initSign(privateKey);
      signature.update(message.getBytes());
      return signature.sign();
   }

   private static boolean verify(String message, byte[] signature, PublicKey publicKey)
         throws NoSuchAlgorithmException, InvalidKeyException, SignatureException
  {
      Signature verifier = Signature.getInstance("SHA256withElGamal");
      verifier.initVerify(publicKey);
      verifier.update(message.getBytes());
      return verifier.verify(signature);
  }
   private static String bytesToHex(byte[] bytes) {
      StringBuilder hexString = new StringBuilder();
      for (byte b : bytes) {
         hexString.append(String.format("%02X", b));
      }
      return hexString.toString();
   }
}
```

OUTPUT: -
Message: Hello, how are you
Signature: [hexadecimal]
Signature Verified: true

11. Alice uses a Bob's public key for sending a confidential message. Alice select a two large prime numbers to generate a private and public key so that eve could not break the ciphertext. So as a developer implement this Algorithm to generate a pair of keys and each message should be encrypted by different key pairs .

CODE : –

```
import java.math.BigInteger;
import java.util.Random;
public class RSA
 {
   public static void main(String[] args)
  {
      KeyPair bobKeys = generateKeyPair();
      String message = "Hello Bob!";
      BigInteger ciphertext = encrypt(message, bobKeys.publicKey);
      String decryptedMessage = decrypt(ciphertext, bobKeys.privateKey);

      System.out.println("Original Message: " + message);
      System.out.println("Encrypted Message: " + ciphertext);
      System.out.println("Decrypted Message: " + decryptedMessage);
   }
   static KeyPair generateKeyPair() {
      BigInteger p = generateLargePrime();
      BigInteger q = generateLargePrime();
      BigInteger n = p.multiply(q);
      BigInteger phi = (p.subtract(BigInteger.ONE)).multiply(q.subtract(BigInteger.ONE));
      BigInteger e = BigInteger.valueOf(65537);
      BigInteger d = e.modInverse(phi);
      return new KeyPair(new PublicKey(n, e), new PrivateKey(n, d));
   }
   static BigInteger encrypt(String message, PublicKey publicKey)
    {
      return new BigInteger(message.getBytes()).modPow(publicKey.e, publicKey.n);
    }
   static String decrypt(BigInteger ciphertext, PrivateKey privateKey)
    {
      return new String(ciphertext.modPow(privateKey.d, privateKey.n).toByteArray());
    }
   static BigInteger generateLargePrime()
   {
      return BigInteger.probablePrime(512, new Random());
   }
}
class KeyPair {
   PublicKey publicKey;
   PrivateKey privateKey;

   KeyPair(PublicKey publicKey, PrivateKey privateKey)
{
      this.publicKey = publicKey;
      this.privateKey = privateKey;
   }
}
```

```
class PublicKey {
   BigInteger n;
   BigInteger e;

   PublicKey(BigInteger n, BigInteger e)
 {
      this.n = n;
      this.e = e;
   }
}
class PrivateKey
 {
   BigInteger n;
   BigInteger d;

   PrivateKey(BigInteger n, BigInteger d)

 {
      this.n = n;
      this.d = d;
   }
}
```

OUTPUT:

Original Message: Hello Bob!

Encrypted:-
-
435836996936984575460943309816787166212736493274116935864551628844524409106801891893300454171354840377158237600802564152537440510312727327171717346848900671430199 50

Decrypted Message: -Hello Bob!

12. Write a program to implement Rabin Miller Primality Test to check whether given number is prime or composite.

CODE:-

```
import java.math.BigInteger;
import java.util.Random;
public class RabinMillerPrimalityTest
{
   Public static void main(String[] args)
  {
      BigInteger number = new BigInteger("1125899839733759");
      int iterations = 5; // Number of iterations for the Rabin-Miller test
      boolean isPrime = isPrime(number, iterations);
      if (isPrime) {
         System.out.println(number + " is likely prime.");
      } else {
         System.out.println(number + " is composite.");
      }
  }
   static boolean isPrime(BigInteger n, int iterations)
 {
      // Handle small numbers
      if (n.compareTo(BigInteger.valueOf(2)) == 0) {
         return true;
      }
      if (n.compareTo(BigInteger.valueOf(2)) < 0 ||
n.mod(BigInteger.valueOf(2)).equals(BigInteger.ZERO)) {
         return false;
      }

      BigInteger d = n.subtract(BigInteger.ONE);
      int r = 0;
      while (d.mod(BigInteger.valueOf(2)).equals(BigInteger.ZERO)) {
         d = d.divide(BigInteger.valueOf(2));
         r++;
      }
```

```
        // Rabin-Miller test
        Random rand = new Random();
        for (int i = 0; i < iterations; i++) {
            BigInteger a = new BigInteger(n.bitLength(), rand);
            a = a.mod(n.subtract(BigInteger.valueOf(2))).add(BigInteger.valueOf(2)); // Ensure 2 <= a <
 n-2
            BigInteger x = a.modPow(d, n);
            if (x.equals(BigInteger.ONE) || x.equals(n.subtract(BigInteger.ONE))) {
                continue;
            }
            for (int j = 0; j < r - 1; j++) {
                x = x.modPow(BigInteger.valueOf(2), n);
                if (x.equals(BigInteger.ONE)) {
                    return false; // Composite
                }
                if (x.equals(n.subtract(BigInteger.ONE))) {
                    break;
                }
            }
            if (!x.equals(n.subtract(BigInteger.ONE))) {
                return false; // Composite
            }
        }
        return true; // Likely prime
    }
}
```

OUTPUT:- 1125899839733759 is likely prime.