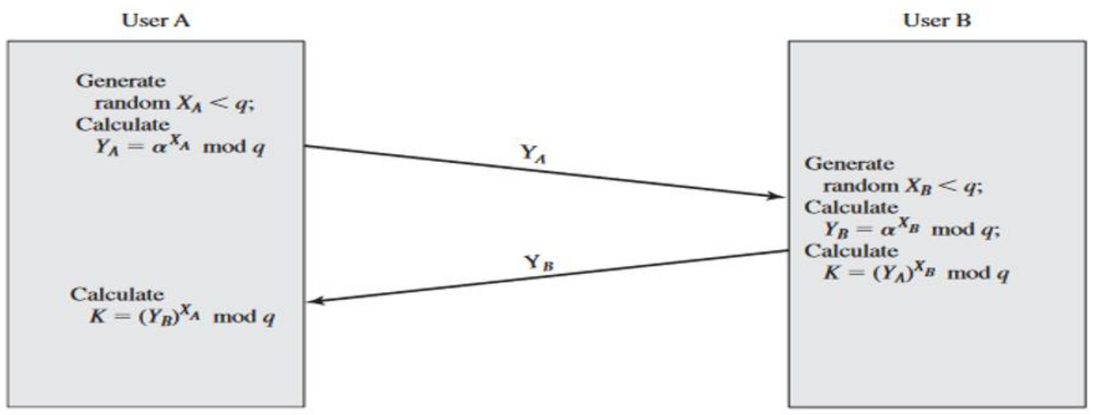
Experiment 9

Implement the Diffie-Hellman Key Exchange mechanism using HTML and JavaScript.

Aim: To implement the diffie-hellman key exchange algorithm

Description:

The Diffie-Hellman key exchange is a cryptographic protocol allowing two parties to securely generate a shared secret key over an insecure channel. Each party generates a public and private key, shares the public key, and uses both keys to compute a common secret. The security relies on the difficulty of solving the Discrete Logarithm Problem, making it hard for eavesdroppers to deduce the secret.

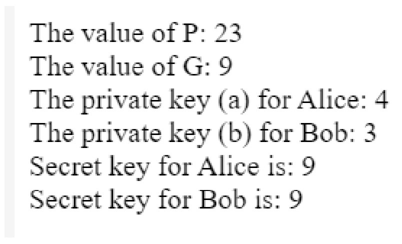

User A
User B
Generate random $X_A < q$; Calculate $Y_A = \alpha^{X_A} \bmod q$
Y_A
Generate random $X_B < q$; Calculate $Y_B = \alpha^{X_B} \bmod q$; Calculate $K = (Y_A)^{X_B} \bmod q$
Y_B
Calculate $K = (Y_B)^{X_A} \bmod q$

Source code:

```
<!DOCTYPE html>

<html>

<head>

  <title>Diffie-Hellman Key Exchange</title>

</head>

<body>

<script>

function power(a, b, p)

{

        if (b == 1)

                return a;

        else
```

```
            return((Math.pow(a, b)) % p);
    }


    var P, G, x, a, y, b, ka, kb;
    P = parseInt(prompt("Enter a prime number P:"));
    document.write("The value of P: " + P + "<br>");
    G = parseInt(prompt("Enter a primitive root G:"));
    document.write("The value of G: " + G + "<br>");
    a = parseInt(prompt("Enter the private key for Alice (a):"));
    document.write("The private key (a) for Alice: " + a + "<br>");
    x = power(G, a, P);
    b = parseInt(prompt("Enter the private key for Bob (b):"));
    document.write("The private key (b) for Bob: " + b + "<br>");
    y = power(G, b, P);
    ka = power(y, a, P);
    kb = power(x, b, P);
    document.write("Secret key for Alice is: " + ka + "<br>");
    document.write("Secret key for Bob is: " + kb + "<br>");
    </script>
    </body>
    </html>
```

Output :



```
The value of P: 23
The value of G: 9
The private key (a) for Alice: 4
The private key (b) for Bob: 3
Secret key for Alice is: 9
Secret key for Bob is: 9
```

Conclusion :

Diffie-Hellman enables secure key exchange over public channels by allowing two parties to establish a shared secret without directly transmitting it. Its security foundation on the Discrete

Logarithm Problem makes it highly resistant to interception. Widely used in secure communications, it provides a basis for modern encryption protocols like SSL/TLS.

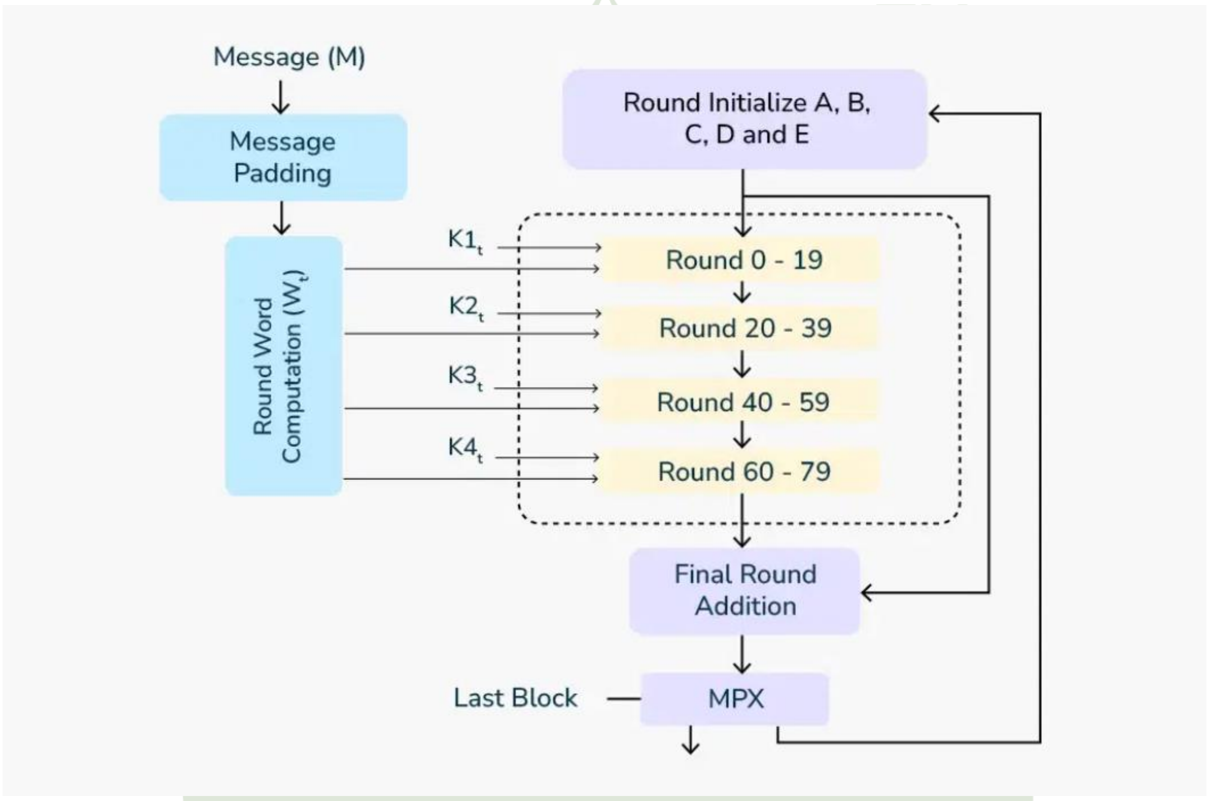
Experiment 10

Calculate the message digest of a text using the SHA-1 algorithm in JAVA.

Aim: To calculate the message digest of a given text using the SHA-1 algorithm

Description:

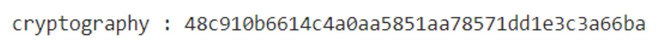
The SHA-1 (Secure Hash Algorithm 1) is a cryptographic hash function that produces a 160-bit hash value (or message digest) for an input message. Hash functions are used to ensure data integrity, as the hash will change with any alteration in the input text.



Source code:

```
package sha_1;

import java.math.BigInteger;

import java.security.MessageDigest;

import java.security.NoSuchAlgorithmException;

import java.util.*;

public class SHA_1

{

  public static String encryptThisString(String input)

    {
        try
         {
            MessageDigest md = MessageDigest.getInstance("SHA-1");
            byte[] messageDigest = md.digest(input.getBytes());
            BigInteger no = new BigInteger(1, messageDigest);
            String hashtext = no.toString(16);
            while (hashtext.length() < 32)
              {
                hashtext = "0" + hashtext;
              }
            return hashtext;
            }
        catch (NoSuchAlgorithmException e)
            {
              throw new RuntimeException(e);
            } }
         public static void main(String[] args) throws NoSuchAlgorithmException
         {
            String s1;
            s1="cryptography";
            System.out.println("\n" + s1 + " : " + encryptThisString(s1));
         }}
```

Output:



```
cryptography : 48c910b6614c4a0aa5851aa78571dd1e3c3a66ba
```

Conclusion :

SHA-1 is a widely used cryptographic hash function that generates a unique 160-bit hash value for a given input, ensuring data integrity by detecting even minor changes in input data. Although it has been largely phased out for security-sensitive applications due to vulnerability to collision attacks, SHA-1 remains a foundational hash function in cryptography histo

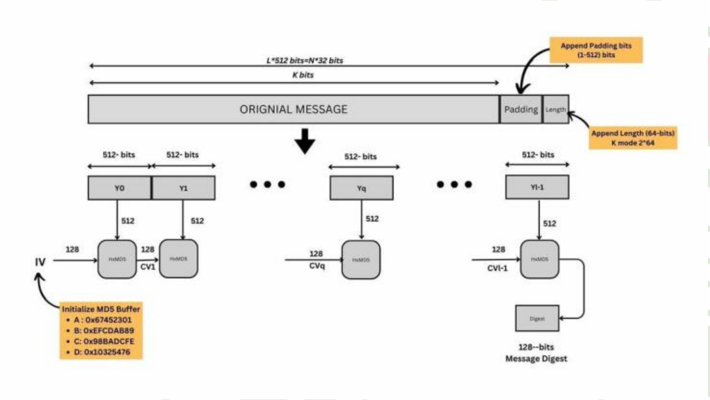
Experiment 11

Calculate the message digest of a text using the MD5 algorithm in JAVA.

Aim: To implement MD5 algorithm.

Description:

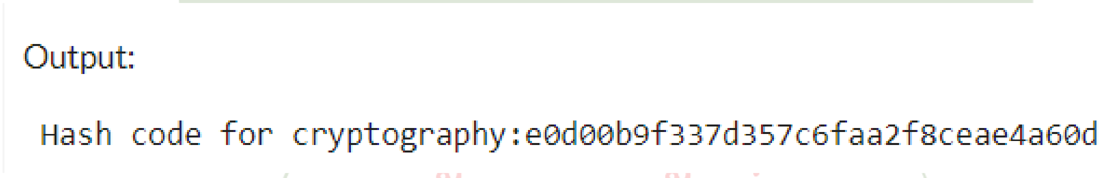
MD5 (Message Digest Algorithm 5) is a cryptographic hash function that produces a 128-bit hash value represented as a 32-character hexadecimal number. It is commonly used for checksums and data integrity verification, as even a small change in the input yields a completely different hash.



Source code :

```
import java.util.Date;

import java.math.BigInteger;

import java.security.MessageDigest;

import java.security.NoSuchAlgorithmException;

import java.util.Scanner;

public class MD5 {

    public static String getMd5(String input)

    {

        try {

            MessageDigest md = MessageDigest.getInstance("MD5");

            byte[] messageDigest = md.digest(input.getBytes());

            BigInteger no = new BigInteger(1, messageDigest);

            String hashtext = no.toString(16);

            while (hashtext.length() < 32) {
                hashtext = "0" + hashtext;


      }

            return hashtext;

         }

         catch (NoSuchAlgorithmException e) {

            throw new RuntimeException(e);

         }

   }

   public static void main(String args[]) throws NoSuchAlgorithmException

   {

     String s1;

     s1="cryptography";

      System.out.println(" Hash code for "+s1 + ":" + getMd5(s1));

   }

}
```

Output :



Output:

```
Hash code for cryptography:e0d00b9f337d357c6faa2f8ceae4a60d
```

Conclusion:

While MD5 was once popular for data integrity verification and digital signatures, it is now considered insecure for cryptographic use due to vulnerability to collisions. The algorithm’s susceptibility to attacks limits its effectiveness in security-critical applications, though it is still occasionally used in non-secure contexts where speed is prioritized over security.

Experiment 12

Implement Simple Columner Transposition technique and Advanced Columner Transposition technique

Aim: To implement the columnar transposition technique

Description:

The columnar transposition is a classical encryption method that rearranges the characters of a plaintext by writing them into a grid of columns and then reading them out in a different, predefined order based on a keyword.

- Choose a Keyword
- Write the Message in Rows
- Reorder Columns by Keyword
- Read Vertically – cipher text

The plaintext is written out in rows within a grid of fixed column width and then read vertically according to a defined key order. The order of columns is rearranged based on a keyword, creating the cipher text by reading the columns sequentially according to the key. This technique provides a level of security by scrambling the message structure, though it is susceptible to frequency analysis and thus best suited for simpler encryption needs.

Source code:

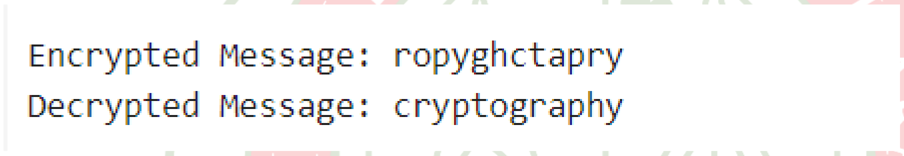
```
import java.util.*;

public class ColumnarTranspositionCipher {

   static final String key = "HACK";

   static Map<Character, Integer> keyMap = new HashMap<>();

   static void setPermutationOrder() {

      for (int i = 0; i < key.length(); i++) {

         keyMap.put(key.charAt(i), i);

      }

   }

   static String encryptMessage(String msg) {

      int row, col;

      StringBuilder cipher = new StringBuilder();

      col = key.length();
```

```
        row = (int) Math.ceil((double) msg.length() / col);
        char[][] matrix = new char[row][col];

    for (int i = 0, k = 0; i < row; i++) {
           for (int j = 0; j < col; ) {
              if (k < msg.length()) {
                 char ch = msg.charAt(k);
                 if (Character.isLetter(ch) || ch == ' ') {
                    matrix[i][j] = ch;
                    j++;
                 }
                 k++;
              } else {
                 matrix[i][j] = '_';
                 j++;
              }
           }
        }

        for (Map.Entry<Character, Integer> entry : keyMap.entrySet()) {
           int columnIndex = entry.getValue();
           for (int i = 0; i < row; i++) {
              if  (Character.isLetter(matrix[i][columnIndex])  ||  matrix[i][columnIndex]  ==  ' '  ||
    matrix[i][columnIndex] == '_') {
                 cipher.append(matrix[i][columnIndex]);
              }}}
        return cipher.toString();
      }
      static String decryptMessage(String cipher) {
```

```
    int col = key.length();
    int row = (int) Math.ceil((double) cipher.length() / col);
    char[][] cipherMat = new char[row][col];
    int k = 0;
    for (int j = 0; j < col; j++) {
        for (int i = 0; i < row; i++) {
            cipherMat[i][j] = cipher.charAt(k);
            k++;
        }
    }
    int index = 0;
    for (Map.Entry<Character, Integer> entry : keyMap.entrySet()) {
        entry.setValue(index++);
    }
    char[][] decCipher = new char[row][col];
    for (int l = 0; l < key.length(); l++) {
        int columnIndex = keyMap.get(key.charAt(l));
        for (int i = 0; i < row; i++) {
            decCipher[i][l] = cipherMat[i][columnIndex];
        }
    }
    StringBuilder msg = new StringBuilder();
    for (int i = 0; i < row; i++) {
        for (int j = 0; j < col; j++) {
            if (decCipher[i][j] != '_') {
                msg.append(decCipher[i][j]);
            }  } }
    return msg.toString();
}
```

```
    public static void main(String[] args) {
        String msg = "cryptography";
        setPermutationOrder();
        String cipher = encryptMessage(msg);

System.out.println("Encrypted Message: " + cipher);
        System.out.println("Decrypted Message: " + decryptMessage(cipher));
    }}
```

Output:



```
Encrypted Message: ropyghctapry
Decrypted Message: cryptography
```

Conclusion:

The columnar transposition cipher is an effective method for rearranging plaintext to create a more secure cipher text through simple reordering. While it offers basic encryption and is easy to implement, its vulnerability to frequency analysis limits its use to low-security applications. This technique is often combined with other ciphers to enhance overall encryption strength.

Experiment 13

Implement Euclidean Algorithm and Advanced Euclidean Algorithm

Aim: To implement the advanced Euclidean algorithm.

Description:

The Euclidean algorithm is a way to find the greatest common divisor of two positive integers. GCD of two numbers is the largest number that divides both of them. A simple way to find GCD is to factorize both numbers and multiply common prime factors.

The algorithm is based on the principles that :

If we subtract a smaller number from a larger one (we reduce a larger number), GCD doesn’t change. So if we keep subtracting repeatedly the larger of two, we end up with GCD.

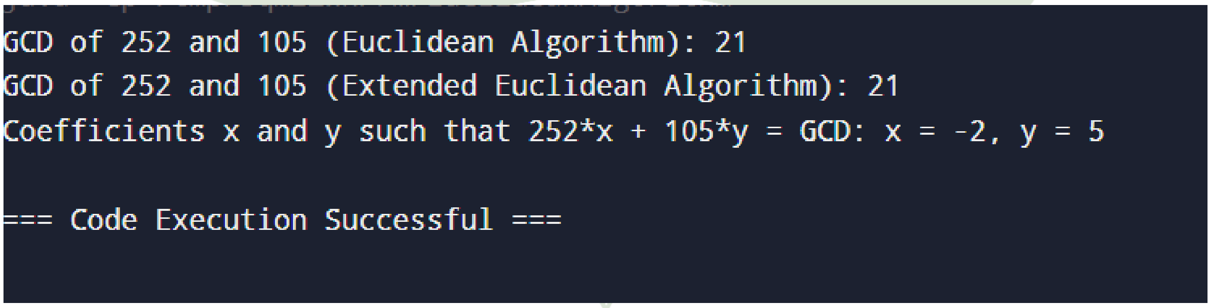
Now instead of subtraction, if we divide the larger number, the algorithm stops when we find the remainder 0.

The extended Euclidean algorithm updates the results of gcd(a, b) using the results calculated by the recursive call gcd(b%a, a).

Source Code:

```
public class EuclideanAlgorithm {

    public static int gcd(int a, int b) {

        while (b != 0) {

            int temp = b;

            b = a % b;

            a = temp;

        }

        return a;

    }

    public static int[] extendedGcd(int a, int b) {

        if (b == 0) {

            return new int[] { a, 1, 0 };

        }

        int[] result = extendedGcd(b, a % b);
        int gcd = result[0];

        int x = result[2];

        int y = result[1] - (a / b) * result[2];


        return new int[] { gcd, x, y };

    }


    public static void main(String[] args) {

        int a = 252, b = 105;

        int gcdResult = gcd(a, b);

        System.out.println("GCD of " + a + " and " + b + " (Euclidean Algorithm): " + gcdResult);

        int[] extendedResult = extendedGcd(a, b);

        System.out.println("GCD of " + a + " and " + b + " (Extended Euclidean Algorithm): " +
extendedResult[0]);

        System.out.println("Coefficients x and y such that " + a + "*x + " + b + "*y = GCD: x =
" + extendedResult[1] + ", y = " + extendedResult[2]);

    }

}
```

Output:



```
GCD of 252 and 105 (Euclidean Algorithm): 21
GCD of 252 and 105 (Extended Euclidean Algorithm): 21
Coefficients x and y such that 252*x + 105*y = GCD: x = -2, y = 5

=== Code Execution Successful ===
```

Conclusion:

This Java program demonstrates the use of the Euclidean Algorithm to compute the GCD of two integers. Additionally, it implements the Extended Euclidean Algorithm to find integer coefficients x and y such that the linear combination of the inputs results in their GCD. These algorithms are essential in cryptography and number theory for solving problems involving divisibility and modular arithmetic.

Experiment 14

Familiarize the cryptographic tools (opencv)

Aim: To explore the cryptographic tools

Description:

- **PyCryptodome**: PyCryptodome is an easy-to-use and versatile Python library for implementing cryptographic functions. It is built as a fork of the now-legacy PyCrypto library and offers a variety of cryptographic algorithms for both high-level and low-level operations. Some of its key features include symmetric encryption algorithms such as AES (Advanced Encryption Standard), DES (Data Encryption Standard), and 3DES, along with asymmetric encryption like RSA and DSA (Digital Signature Algorithm). PyCryptodome also supports hashing algorithms (SHA-256, MD5, etc.), HMAC (Hash-based Message Authentication Code), digital signatures, and key generation functions. This library is particularly useful for developers who need a comprehensive toolset for secure data transmission, encryption of files, or protection of sensitive data in storage. It is widely used for secure email, file encryption, and building custom cryptographic protocols.

- **Cryptography**: The Cryptography library is one of the most widely adopted cryptographic libraries in Python. It provides both high-level cryptographic primitives, such as encryption and signing, and low-level cryptographic operations for fine-tuned control. The library supports a range of encryption algorithms, including AES (for symmetric encryption), RSA and ECC (for asymmetric encryption), and hashing algorithms like SHA-1, SHA-256, and SHA-512. Cryptography also includes key derivation functions (e.g., PBKDF2, bcrypt, and scrypt) for securely deriving encryption keys from passwords. Additionally, it has built-in support for key management and cryptographic protocols like SSL/TLS, making it a go-to choice for developers working on secure web applications, VPNs, or encrypted communication systems. The library ensures high security by following best practices and continuously auditing for vulnerabilities.

- **hashlib**: hashlib is a built-in Python library for generating cryptographic hashes. It is lightweight and simple to use, making it ideal for quick tasks like checking the integrity of data or generating unique identifiers. The library includes support for widely-used hashing algorithms such as MD5 (though not recommended for security-sensitive use cases), SHA-1, SHA-256, SHA-512, and others. Hashes are commonly used for ensuring data integrity, as in verifying file downloads or comparing large datasets. hashlib can also be used for securely hashing passwords by combining the hash with salt to defend against rainbow table attacks. It's part of the Python standard library, which means it doesn't require any additional installation. Despite its simplicity, hashlib is a crucial tool for any developer dealing with data security.

- **PyNaCl**: PyNaCl is a Python wrapper around the NaCl (Networking and Cryptography Library) created by the creators of the Libsodium cryptographic library. NaCl provides a set of cryptographic primitives designed to be easy to use and resistant to common cryptographic attacks. PyNaCl is particularly known for its simplicity and security by default. It includes high-level primitives for public-key encryption (Box), secret-key encryption (SecretBox), and digital signatures (SigningKey), all of which use modern cryptographic algorithms such as Curve25519 for key exchange and XSalsa20 for symmetric encryption. It also supports hashing with SHA-512 and key derivation. PyNaCl's focus on providing secure-by-default operations makes it an excellent choice for developers looking to implement end-to-end encryption or secure communication protocols without dealing with the complexities and pitfalls of low-level cryptographic algorithms.

- **M2Crypto**: M2Crypto is a Python wrapper around OpenSSL that enables developers to access OpenSSL's powerful cryptographic functionalities. M2Crypto supports a wide range of cryptographic algorithms, including symmetric encryption (AES, DES), asymmetric encryption (RSA, DSA, ECC), and secure hashing functions. One of its key features is its ability to handle SSL/TLS communications, making it particularly useful for implementing HTTPS and other secure protocols. It also offers support for certificate handling, including reading, writing, and parsing X.509 certificates, making it ideal for SSL/TLS certificate management and validation. M2Crypto can be used in secure email systems, client-server communication, and even in cryptographic applications that require integration with other security protocols. While it's a robust tool, developers should be aware that M2Crypto's API is not as user-friendly as some of the other modern libraries.

- **PyOpenSSL**: PyOpenSSL is another Python interface to OpenSSL that focuses on making OpenSSL's functionality accessible to Python developers. Like M2Crypto, PyOpenSSL supports a wide range of cryptographic operations, including symmetric encryption, asymmetric encryption, and message digest algorithms (e.g., SHA-256). It excels at handling SSL/TLS protocols and is commonly used in building secure server-client communications. With PyOpenSSL, you can create secure HTTPS connections, handle client certificates, and validate SSL certificates. Additionally, it includes tools for generating private/public keys, creating digital signatures, and implementing public-key infrastructure (PKI) systems. Developers often use PyOpenSSL for web applications, secure APIs, or systems requiring encrypted communications. PyOpenSSL is also commonly used to interact with TLS/SSL certificates and cryptographic modules in web servers and other network applications.

- **Fernet (from cryptography)**: Fernet is a high-level symmetric encryption system provided by the Cryptography library. It uses AES encryption (with a 128-bit key) and HMAC for authentication, making it an ideal solution for encrypting and decrypting small pieces of data securely. Fernet is designed to be simple to use—developers can encrypt data with a single key and later decrypt it with the same key, ensuring confidentiality and integrity. It's particularly useful for storing sensitive information like passwords or API tokens in databases or for encrypting messages in transit. Fernet also automatically handles things like padding and key management, making it an attractive option for developers who want secure encryption without needing to deal with low-level details.

- **PyCrypto**: PyCrypto is one of the older Python libraries for cryptographic operations and remains widely used in legacy projects, even though it is no longer actively maintained. It supports a broad set of cryptographic algorithms, including AES, DES, RSA, DSA, and elliptic curve cryptography (ECC). PyCrypto also includes hashing algorithms like SHA and HMAC. It was one of the first libraries to provide an easy-to-use interface for encryption and decryption operations in Python, which led to its widespread adoption. Despite no longer being maintained, it is still a good option for projects where cryptographic compatibility is needed with older systems, though PyCryptodome is recommended for new projects due to better security and continued support.

- **Passlib**: Passlib is a specialized Python library for securely hashing and verifying passwords. It provides support for a wide range of password hashing algorithms, including bcrypt, Argon2, PBKDF2, and SHA-1. Passlib includes built-in features to handle password strength verification, help with upgrading password hashes (when newer, more secure algorithms become available), and automatically generate salt. One of its key features is its ability to store and verify passwords securely by using key derivation functions (KDFs) to slow down brute-force attacks. Passlib is an excellent choice for developers building authentication systems, such as login forms or user management systems, where password security is a priority.

Result analysis:

Each of these tools provides specialized cryptographic operations and is designed to address different use cases, from secure communications and data encryption to password hashing and key management. Depending on your needs (e.g., secure file storage, communication, or integrity verification), you can choose the tool that best fits your application’s security requirements.

BEYOND SYLLABUS

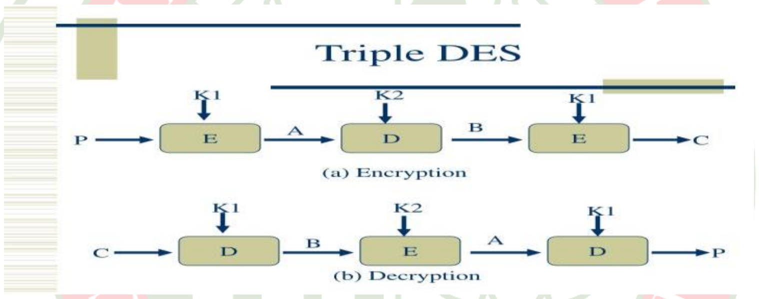
1. Implement the triple DES encryption algorithm

Aim: To implement the triple DES algorithm

Description:

Triple DES (3DES) is an encryption algorithm that extends the Data Encryption Standard (DES) by applying it three times in succession with either two or three different keys. It was designed to enhance the security of DES, which is now considered weak due to its short key length of 56 bits. Triple DES is a symmetric-key block cipher, meaning it uses the same key for both encryption and decryption.

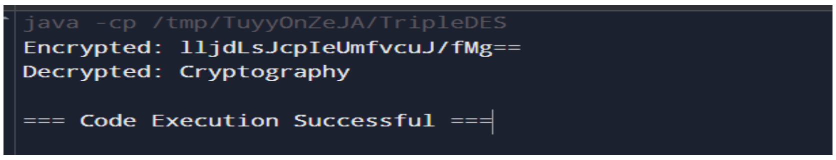
3-Key Triple DES: Uses three independent keys (Key 1, Key 2, and Key 3), resulting in a total key length of 168 bits.



Source code:

```
import javax.crypto.Cipher;

import javax.crypto.KeyGenerator;

import javax.crypto.SecretKey;

import javax.crypto.spec.SecretKeySpec;

import javax.crypto.Cipher;

import java.util.Base64;

public class TripleDES {

  public static String encrypt(String plaintext, String key) throws Exception {

    SecretKeySpec keySpec = new SecretKeySpec(key.getBytes(), "DESede");

    Cipher cipher = Cipher.getInstance("DESede/ECB/PKCS5Padding");

    cipher.init(Cipher.ENCRYPT_MODE, keySpec);

    byte[] encrypted = cipher.doFinal(plaintext.getBytes());

    return Base64.getEncoder().encodeToString(encrypted);
  }
  public static String decrypt(String ciphertext, String key) throws Exception {
    SecretKeySpec keySpec = new SecretKeySpec(key.getBytes(), "DESede");
    Cipher cipher = Cipher.getInstance("DESede/ECB/PKCS5Padding");
    cipher.init(Cipher.DECRYPT_MODE, keySpec);
    byte[] decoded = Base64.getDecoder().decode(ciphertext);
    byte[] decrypted = cipher.doFinal(decoded);
    return new String(decrypted);
  }
  public static void main(String[] args) {
    try {
        String key = "123456789012345678901234";
        String plaintext = "Cryptography";
        String encryptedText = encrypt(plaintext, key);
        System.out.println("Encrypted: " + encryptedText);
        String decryptedText = decrypt(encryptedText, key);
        System.out.println("Decrypted: " + decryptedText);
    } catch (Exception e) {
        e.printStackTrace();
    }
  }
}
```

Output:



```
java -cp /tmp/TuyyOnZeJA/TripleDES
Encrypted: lljdLsJcpIeUmfvcuJ/fMg==
Decrypted: Cryptography

=== Code Execution Successful ===
```

Conclusion:

Triple DES provides enhanced security compared to DES by applying encryption three times, making it resistant to brute-force attacks. This is a very robust encryption algorithm.

2. Implement the following transposition ciphers

a) Rail – fence

Aim: To implement the rail-fence transposition cipher technique.

Description:

The Rail Fence Cipher is a form of transposition cipher in which the plaintext is written in a zigzag pattern (like a fence) across multiple "rails" (rows), and then read off row by row to create the ciphertext.

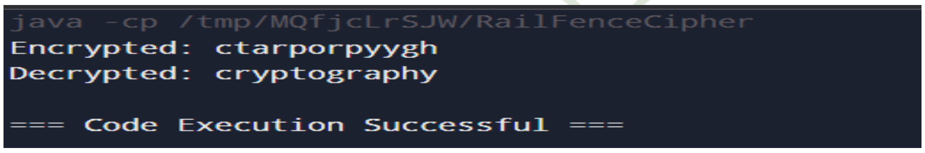
The rail-fence algorithm works as :

- Choose the number of rails: This is the number of rows (or levels) used in the zigzag pattern. For example, if you choose 3 rails, you will write the message across 3 rows.
- Write the message in a zigzag pattern: You write the message diagonally across the rails, starting from the top, moving down to the bottom, and then back up to the top, and so on.
- Read the message row by row: Once the message is written in the zigzag pattern, you read the characters row by row to get the ciphertext.

Source code:

```
public class RailFenceCipher {

    public static String encrypt(String plaintext, int rails) {

        char[][] fence = new char[rails][plaintext.length()];

        for (char[] row : fence) {

            for (int i = 0; i < row.length; i++) {

                row[i] = ' ';  } }

        int rail = 0;

        boolean down = true;

        for (int i = 0; i < plaintext.length(); i++) {

            fence[rail][i] = plaintext.charAt(i);

            if (rail == 0) down = true;

            if (rail == rails - 1) down = false;

            rail = down ? rail + 1 : rail - 1;

        }
        StringBuilder encrypted = new StringBuilder();
        for (int r = 0; r < rails; r++) {
            for (int c = 0; c < plaintext.length(); c++) {
                if (fence[r][c] != ' ') encrypted.append(fence[r][c]);
            }
        }
        return encrypted.toString();
    }
    public static String decrypt(String ciphertext, int rails) {
        char[][] fence = new char[rails][ciphertext.length()];
        for (char[] row : fence) {
            for (int i = 0; i < row.length; i++) {
                row[i] = ' ';
            }
        }
        int rail = 0;
        boolean down = true;
        for (int i = 0; i < ciphertext.length(); i++) {
            fence[rail][i] = '*';
            if (rail == 0) down = true;
            if (rail == rails - 1) down = false;
            rail = down ? rail + 1 : rail - 1;
        }
        int index = 0;
        for (int r = 0; r < rails; r++) {
            for (int c = 0; c < ciphertext.length(); c++) {
                if (fence[r][c] == '*' && index < ciphertext.length()) {
                    fence[r][c] = ciphertext.charAt(index++);
            }  } }
        StringBuilder decrypted = new StringBuilder();
        rail = 0;
        down = true;
        for (int i = 0; i < ciphertext.length(); i++) {
            decrypted.append(fence[rail][i]);
            if (rail == 0) down = true;
            if (rail == rails - 1) down = false;
            rail = down ? rail + 1 : rail - 1;
        }
        return decrypted.toString();
    }
    public static void main(String[] args) {
        String plaintext = "cryptography";
        int rails = 3;
        String encryptedText = encrypt(plaintext, rails);
        System.out.println("Encrypted: " + encryptedText);
        String decryptedText = decrypt(encryptedText, rails);
        System.out.println("Decrypted: " + decryptedText);
    }
}
```

Output:



```
java -cp /tmp/MQfjcLrSJW/RailFenceCipher
Encrypted: ctarporpyygh
Decrypted: cryptography

=== Code Execution Successful ===
```

Conclusion:

The Rail Fence cipher works by rearranging the characters in a zigzag pattern, providing a basic transposition cipher technique.

b) Vernam cipher

Aim: To implement the vernam cipher transposition technique

Description:

The Vernam Cipher is a type of symmetric-key cipher and is essentially a one-time pad (OTP) where each character of the plaintext is XORed with a key of the same length. The key is a random string of the same length as the plaintext and used only once. This cipher is theoretically unbreakable if the key is truly random, as long as it is used only once and kept secret.

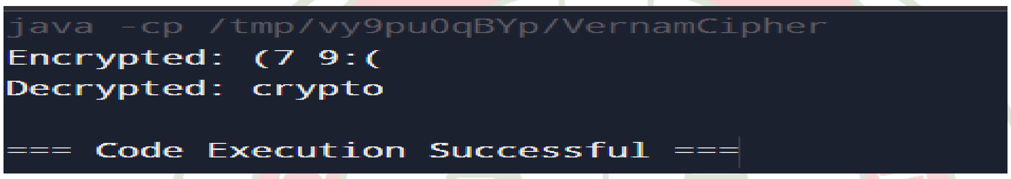
The Vernam Cipher Works as follows:

1. Key Generation: A key is generated that is as long as the plaintext message and is completely random. This key is shared securely between the sender and the receiver.
2. Encryption: Each character of the plaintext is XORed with the corresponding character in the key. In binary terms, the plaintext and key are converted to binary, and each bit of the plaintext is XORed with the corresponding bit of the key.
3. Decryption: To decrypt the message, the ciphertext is XORed again with the same key, which restores the original plaintext.

Source code:

```
public class VernamCipher {

   public static String encrypt(String plaintext, String key) {

      StringBuilder encrypted = new StringBuilder();

      for (int i = 0; i < plaintext.length(); i++) {

         encrypted.append((char) (plaintext.charAt(i) ^ key.charAt(i)));

      }

      return encrypted.toString();

   }

   public static String decrypt(String ciphertext, String key) {

      StringBuilder decrypted = new StringBuilder();

      for (int i = 0; i < ciphertext.length(); i++) {

         decrypted.append((char) (ciphertext.charAt(i) ^ key.charAt(i)));

      }

      return decrypted.toString();

   }

   public static void main(String[] args) {
      String plaintext = "crypto";

      String key = "KEYING";

      String encryptedText = encrypt(plaintext, key);

      System.out.println("Encrypted: " + encryptedText);

      String decryptedText = decrypt(encryptedText, key);

      System.out.println("Decrypted: " + decryptedText);

   }

}
```

Output:



```
java -cp /tmp/vy9pu0qBYp/VernamCipher
Encrypted: (7 9:(
Decrypted: crypto

=== Code Execution Successful ===
```

Conclusion:

The Vernam cipher is a symmetric cipher that uses the XOR operation, providing strong security when used with a random key of the same length as the plaintext.

c) Vigenere cipher

Aim: To implement vigenere cipher transposition technique.

Description:

The Vigenère Cipher is a method of encrypting alphabetic text by using a series of different Caesar ciphers based on the letters of a keyword. Unlike the simple Caesar cipher, where the shift is constant, the Vigenère cipher uses a keyword to vary the shift for each letter of the plaintext.

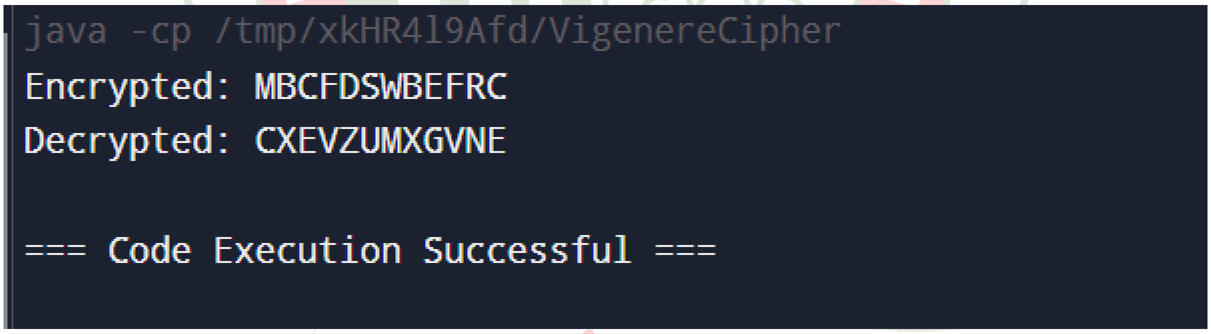
the Vigenère Cipher Works as:

1. Plaintext: The message you want to encrypt.
2. Keyword: A keyword (or key) is repeated to match the length of the plaintext. The keyword is used to determine the shift for each letter in the plaintext.
3. Encryption Process: Each letter of the plaintext is shifted along the alphabet according to the corresponding letter of the keyword. The position of each letter in the alphabet is used to determine the shift (A = 0, B = 1, C = 2, etc.).
4. Decryption Process: To decrypt, the ciphertext is shifted back by the same number of positions using the same keyword.

Source code:

```
public class VigenereCipher {

  public static String encrypt(String plaintext, String key) {

    StringBuilder encrypted = new StringBuilder();

    key = key.repeat(plaintext.length() / key.length() + 1).substring(0, plaintext.length());

    for (int i = 0; i < plaintext.length(); i++) {

      char p = plaintext.charAt(i);

      char k = key.charAt(i);

      encrypted.append((char) ((p + k - 2 * 'A') % 26 + 'A'));

    }

    return encrypted.toString();

  }

  public static String decrypt(String ciphertext, String key) {

    StringBuilder decrypted = new StringBuilder();

    key = key.repeat(ciphertext.length() / key.length() + 1).substring(0, ciphertext.length());

    for (int i = 0; i < ciphertext.length(); i++) {
      char c = ciphertext.charAt(i);
      char k = key.charAt(i);
      decrypted.append((char) ((c - k + 26) % 26 + 'A'));
    }
    return decrypted.toString();}
  public static void main(String[] args) {
    String plaintext = "Cryptography";
    String key = "KEY";
    String encryptedText = encrypt(plaintext, key);
    System.out.println("Encrypted: " + encryptedText);
    String decryptedText = decrypt(encryptedText, key);
    System.out.println("Decrypted: " + decryptedText);
  }}
```

Output:



```
java -cp /tmp/xkHR4l9Afd/VigenereCipher
Encrypted: MBCFDSWBEFRC
Decrypted: CXEVZUMXGVNE

=== Code Execution Successful ===
```

Conclusion:

The Vigenère cipher uses a keyword to apply multiple Caesar ciphers, making it more resistant to frequency analysis than simpler ciphers.