
Ceaser

```
public class Main {
    public static void main(String[] args) {
        String plaintext = "PrakharSinha22BCI0127";
        int shift = 3;
```

```java
        System.out.println("Caesar Cipher Demonstration");
        System.out.println("Original Message: " + plaintext);

        String encrypted = encrypt(plaintext, shift);
        System.out.println("Encrypted Message: " + encrypted);

        String decrypted = decrypt(encrypted, shift);
        System.out.println("Decrypted Message: " + decrypted);
    }

    public static String encrypt(String text, int shift) {
        StringBuilder result = new StringBuilder();

        for (char character : text.toUpperCase().toCharArray()) {
            if (Character.isLetter(character)) {
                int originalAlphabetPosition = character - 'A';
                int newAlphabetPosition = (originalAlphabetPosition + shift) % 26;
                char newCharacter = (char) ('A' + newAlphabetPosition);
                result.append(newCharacter);
            } else {
                result.append(character);
            }
        }

        return result.toString();
    }

    public static String decrypt(String text, int shift) {
        return encrypt(text, 26 - shift);  // Using the encrypt method with reverse shift
    }
}
```

Ceaser Shift4

```
public class Main {
    public static String encrypt(String message, int shift) {
        StringBuilder encrypted = new StringBuilder();
        for (int i = 0; i < message.length(); i++) {
            char ch = message.charAt(i);
            if (Character.isLetter(ch)) {
                ch = Character.toUpperCase(ch);
                ch = (char) ('A' + (ch - 'A' + shift) % 26);
            }
            encrypted.append(ch);
        }
        return encrypted.toString();
    }

    public static String decrypt(String encrypted, int shift) {
        return encrypt(encrypted, 26 - shift);
    }

    public static void main(String[] args) {
        String message = "CRYPTOGRAPHY AND NETWORK SECURITY";
        int shift = 4;

        String encrypted = encrypt(message, shift);
        System.out.println("Original Message: " + message);
        System.out.println("Encrypted Message: " + encrypted);

        String decrypted = decrypt(encrypted, shift);
        System.out.println("Decrypted Message: " + decrypted);

        System.out.println("\nValidation: " +
```

```
            (message.equals(decrypted) ? "Successful" : "Failed"));
    }
}
```

Playfare

```
public class PlayfairCipher {
    private char[][] matrix = new char[5][5];
    private static final String KEY = "MONARCHY";

    public static void main(String[] args) {
        PlayfairCipher cipher = new PlayfairCipher();
        String plaintext = "HELLO WORLD";

        System.out.println("Playfair Cipher Demonstration");
        System.out.println("Key: " + KEY);
        System.out.println("Original Message: " + plaintext);

        cipher.generateMatrix();
        cipher.displayMatrix();

        String encrypted = cipher.encrypt(plaintext);
        System.out.println("Encrypted Message: " + encrypted);

        String decrypted = cipher.decrypt(encrypted);
        System.out.println("Decrypted Message: " + decrypted.replace("X", ""));
    }

    private void generateMatrix() {
        String keyString = KEY + "ABCDEFGHIKLMNOPQRSTUVWXYZ";
        boolean[] used = new boolean[26];
        int row = 0, col = 0;
```

```
    for (char ch : keyString.toCharArray()) {
        if (ch == 'J') ch = 'I';
        int index = ch - 'A';

        if (!used[index]) {
            matrix[row][col] = ch;
            used[index] = true;
            col++;
            if (col == 5) {
                col = 0;
                row++;
            }
            if (row == 5) break;
        }
    }
}

private void displayMatrix() {
    System.out.println("\nPlayfair Matrix:");
    for (int i = 0; i < 5; i++) {
        for (int j = 0; j < 5; j++) {
            System.out.print(matrix[i][j] + " ");
        }
        System.out.println();
    }
    System.out.println();
}

private int[] findPosition(char ch) {
    if (ch == 'J') ch = 'I';
    for (int i = 0; i < 5; i++) {
```

```
        for (int j = 0; j < 5; j++) {
            if (matrix[i][j] == ch) {
                return new int[]{i, j};
            }
        }
    }
    return null;
}

public String encrypt(String text) {
    return processText(text, true);
}

public String decrypt(String text) {
    return processText(text, false);
}

private String processText(String text, boolean encrypt) {
    StringBuilder result = new StringBuilder();
    text = text.toUpperCase().replaceAll("[^A-Z]", "").replace("J", "I");

    for (int i = 0; i < text.length(); i += 2) {
        char first = text.charAt(i);
        char second;

        if (i + 1 < text.length()) {
            second = text.charAt(i + 1);
            if (first == second) {
                second = 'X';
                i--;
            }
```

```
        } else {
            second = 'X';
        }

        int[] pos1 = findPosition(first);
        int[] pos2 = findPosition(second);

        if (pos1[0] == pos2[0]) {
            result.append(matrix[pos1[0]][(pos1[1] + (encrypt ? 1 : 4)) % 5]);
            result.append(matrix[pos2[0]][(pos2[1] + (encrypt ? 1 : 4)) % 5]);
        } else if (pos1[1] == pos2[1]) {
            result.append(matrix[(pos1[0] + (encrypt ? 1 : 4)) % 5][pos1[1]]);
            result.append(matrix[(pos2[0] + (encrypt ? 1 : 4)) % 5][pos2[1]]);
        } else {
            result.append(matrix[pos1[0]][pos2[1]]);
            result.append(matrix[pos2[0]][pos1[1]]);
        }
    }

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

Hill

```
public class HillCipher {
    private static final int[][] KEY = {  // Hardcoded key matrix (3x3)
        {17, 17, 5},
        {21, 18, 21},
        {2, 2, 19}
    };
    private static final int SIZE = 3;
```

```java
// Inverse of the key matrix
private static final int[][] INVERSE_KEY = {
      {4, 9, 15},
      {15, 17, 6},
      {24, 0, 17}
};

public static void main(String[] args) {
   String plaintext = "HELLO WORLD";  // Hardcoded input

   System.out.println("Hill Cipher Demonstration");
   System.out.println("Original Message: " + plaintext);

   displayKey();
   String encrypted = encrypt(plaintext);
   System.out.println("Encrypted Message: " + encrypted);

   String decrypted = decrypt(encrypted);
   System.out.println("Decrypted Message: " + decrypted);
}

private static void displayKey() {
   System.out.println("\nKey Matrix:");
   for (int i = 0; i < SIZE; i++) {
      for (int j = 0; j < SIZE; j++) {
         System.out.printf("%4d", KEY[i][j]);
      }
      System.out.println();
   }
   System.out.println();
}
```

```
public static String encrypt(String text) {
    StringBuilder result = new StringBuilder();
    text = text.toUpperCase().replaceAll("[^A-Z]", "");

    // Pad the text with 'X' if necessary
    while (text.length() % SIZE != 0) {
        text += 'X';
    }

    // Process text in blocks of size 3
    for (int i = 0; i < text.length(); i += SIZE) {
        int[] vector = new int[SIZE];

        // Convert block of characters to numbers
        for (int j = 0; j < SIZE; j++) {
            vector[j] = text.charAt(i + j) - 'A';
        }

        // Multiply key matrix with vector
        for (int j = 0; j < SIZE; j++) {
            int sum = 0;
            for (int k = 0; k < SIZE; k++) {
                sum += KEY[j][k] * vector[k];
            }
            // Convert back to letter and append
            result.append((char) ((sum % 26) + 'A'));
        }
    }

    return result.toString();
```

```java
}

public static String decrypt(String text) {
    StringBuilder result = new StringBuilder();
    text = text.toUpperCase().replaceAll("[^A-Z]", "");

    // Process text in blocks of size 3
    for (int i = 0; i < text.length(); i += SIZE) {
        int[] vector = new int[SIZE];

        // Convert block of characters to numbers
        for (int j = 0; j < SIZE; j++) {
            vector[j] = text.charAt(i + j) - 'A';
        }

        // Multiply inverse key matrix with vector
        for (int j = 0; j < SIZE; j++) {
            int sum = 0;
            for (int k = 0; k < SIZE; k++) {
                sum += INVERSE_KEY[j][k] * vector[k];
            }
            // Convert back to letter and append
            result.append((char) ((sum % 26) + 'A'));
        }
    }

    // Remove 'X' padding from the decrypted message
    while (result.length() > 0 && result.charAt(result.length() - 1) == 'X') {
        result.setLength(result.length() - 1);
    }
```

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

Hill 2x2

```
public class Main {
  public static void main(String[] args) {
    String message = "HELP";

    int[][] keyMatrix = {
      {2, 3},
      {1, 4}
    };

    int[] messageVector = convertToNumbers(message);

    System.out.println("Step 1: Converting message to numbers");
    System.out.println("H = 7, E = 4, L = 11, P = 15");

    System.out.println("\nStep 2: Matrix multiplication and modulus operations");
    for (int i = 0; i < messageVector.length; i += 2) {
      int[] pair = {messageVector[i], messageVector[i + 1]};

      int[] result = matrixMultiply(keyMatrix, pair);

      result[0] = result[0] % 26;
      result[1] = result[1] % 26;

      System.out.printf("For pair %c%c (%d,%d):\n",
        message.charAt(i), message.charAt(i + 1),
        pair[0], pair[1]);
```

```
            System.out.println("Matrix multiplication:");
            System.out.printf("[2 3] [%d] = [%d] = [%d] mod 26 = [%c]\n",
                pair[0], (2 * pair[0] + 3 * pair[1]), result[0],
                (char)(result[0] + 'A'));
            System.out.printf("[1 4] [%d] = [%d] = [%d] mod 26 = [%c]\n\n",
                pair[1], (1 * pair[0] + 4 * pair[1]), result[1],
                (char)(result[1] + 'A'));
        }

        System.out.println("Final encrypted message:");
        encryptMessage(message, keyMatrix);
    }

    private static int[] convertToNumbers(String message) {
        int[] numbers = new int[message.length()];
        for (int i = 0; i < message.length(); i++) {
            numbers[i] = message.charAt(i) - 'A';
        }
        return numbers;
    }

    private static int[] matrixMultiply(int[][] keyMatrix, int[] pair) {
        int[] result = new int[2];
        result[0] = keyMatrix[0][0] * pair[0] + keyMatrix[0][1] * pair[1];
        result[1] = keyMatrix[1][0] * pair[0] + keyMatrix[1][1] * pair[1];
        return result;
    }

    private static void encryptMessage(String message, int[][] keyMatrix) {
        int[] messageVector = convertToNumbers(message);
        StringBuilder encrypted = new StringBuilder();
```

```
        for (int i = 0; i < messageVector.length; i += 2) {
            int[] pair = {messageVector[i], messageVector[i + 1]};
            int[] result = matrixMultiply(keyMatrix, pair);

            result[0] = result[0] % 26;
            result[1] = result[1] % 26;

            encrypted.append((char)(result[0] + 'A'));
            encrypted.append((char)(result[1] + 'A'));
        }

        System.out.println(encrypted.toString());
    }
}
```

Vigenere

```
public class VigenereCipher {
    private static final String KEY = "CIPHER"; // Hardcoded key

    public static void main(String[] args) {
        String plaintext = "HELLO WORLD";  // Hardcoded input

        System.out.println("Vigenere Cipher Demonstration");
        System.out.println("Key: " + KEY);
        System.out.println("Original Message: " + plaintext);

        String encrypted = encrypt(plaintext);
        System.out.println("Encrypted Message: " + encrypted);

        String decrypted = decrypt(encrypted);
        System.out.println("Decrypted Message: " + decrypted);
```

```
}

public static String encrypt(String text) {
    StringBuilder result = new StringBuilder();
    text = text.toUpperCase();

    for (int i = 0, j = 0; i < text.length(); i++) {
        char c = text.charAt(i);
        if (Character.isLetter(c)) {
            // Apply Vigenère formula: Ci = (Pi + Ki) mod 26
            char encryptedChar = (char) (((c - 'A' + (KEY.charAt(j) - 'A')) % 26) + 'A');
            result.append(encryptedChar);
            j = (j + 1) % KEY.length();
        } else {
            result.append(c);
        }
    }
    return result.toString();
}

public static String decrypt(String text) {
    StringBuilder result = new StringBuilder();
    text = text.toUpperCase();

    for (int i = 0, j = 0; i < text.length(); i++) {
        char c = text.charAt(i);
        if (Character.isLetter(c)) {
            // Apply Vigenère formula: Pi = (Ci - Ki + 26) mod 26
            char decryptedChar = (char) (((c - KEY.charAt(j) + 26) % 26) + 'A');
            result.append(decryptedChar);
            j = (j + 1) % KEY.length();
```

```
            } else {
                result.append(c);
            }
        }
        return result.toString();
    }
}
```

DES

```
# Hexadecimal to binary conversion
def hex2bin(s):
    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': "1110", 'F': "1111"
    }
    binary = ""
    for char in s:
        binary += mp[char]
    return binary

# Binary to hexadecimal conversion
def bin2hex(s):
    mp = {
        "0000": '0', "0001": '1', "0010": '2', "0011": '3',
        "0100": '4', "0101": '5', "0110": '6', "0111": '7',
        "1000": '8', "1001": '9', "1010": 'A', "1011": 'B',
        "1100": 'C', "1101": 'D', "1110": 'E', "1111": 'F'
    }
    hex_result = ""
    for i in range(0, len(s), 4):
```

```
        hex_result += mp[s[i:i+4]]
    return hex_result

# Binary to decimal conversion
def bin2dec(binary):
    decimal = 0
    i = 0
    while binary != 0:
        decimal += (binary % 10) * (2 ** i)
        binary //= 10
        i += 1
    return decimal

# Decimal to binary conversion
def dec2bin(num):
    binary = bin(num).replace("0b", "")
    while len(binary) % 4 != 0:
        binary = '0' + binary
    return binary

# Permute function to rearrange the bits
def permute(k, arr, n):
    permutation = ""
    for i in range(n):
        permutation += k[arr[i] - 1]
    return permutation

# Left shift bits by n positions
def shift_left(k, nth_shifts):
    for _ in range(nth_shifts):
        k = k[1:] + k[0]
```

```
    return k

# XOR operation on two binary strings
def xor(a, b):
    return ''.join('0' if a[i] == b[i] else '1' for i in range(len(a)))

# Initial Permutation Table
initial_perm = [
    58, 50, 42, 34, 26, 18, 10, 2,
    60, 52, 44, 36, 28, 20, 12, 4,
    62, 54, 46, 38, 30, 22, 14, 6,
    64, 56, 48, 40, 32, 24, 16, 8,
    57, 49, 41, 33, 25, 17,  9, 1,
    59, 51, 43, 35, 27, 19, 11, 3,
    61, 53, 45, 37, 29, 21, 13, 5,
    63, 55, 47, 39, 31, 23, 15, 7
]

# Expansion D-box Table
exp_d = [
    32, 1, 2, 3, 4, 5, 4, 5,
     6, 7, 8, 9, 8, 9, 10, 11,
    12, 13, 12, 13, 14, 15, 16, 17,
    16, 17, 18, 19, 20, 21, 20, 21,
    22, 23, 24, 25, 24, 25, 26, 27,
    28, 29, 28, 29, 30, 31, 32, 1
]

# Straight Permutation Table
per = [
    16,  7, 20, 21, 29, 12, 28, 17,
```

```
    1, 15, 23, 26,  5, 18, 31, 10,
    2,  8, 24, 14, 32, 27,  3,  9,
   19, 13, 30,  6, 22, 11,  4, 25
]

# S-box Table
sbox = [
   [[14, 4, 13, 1, 2, 15, 11, 8, 3, 10, 6, 12, 5, 9, 0, 7],
    [0, 15, 7, 4, 14, 2, 13, 1, 10, 6, 12, 11, 9, 5, 3, 8],
    [4, 1, 14, 8, 13, 6, 2, 11, 15, 12, 9, 7, 3, 10, 5, 0],
    [15, 12, 8, 2, 4, 9, 1, 7, 5, 11, 3, 14, 10, 0, 6, 13]],

   [[15, 1, 8, 14, 6, 11, 3, 4, 9, 7, 2, 13, 12, 0, 5, 10],
    [3, 13, 4, 7, 15, 2, 8, 14, 12, 0, 1, 10, 6, 9, 11, 5],
    [0, 14, 7, 11, 10, 4, 13, 1, 5, 8, 12, 6, 9, 3, 2, 15],
    [13, 8, 10, 1, 3, 15, 4, 2, 11, 6, 7, 12, 0, 5, 14, 9]],

   [[10, 0, 9, 14, 6, 3, 15, 5, 1, 13, 12, 7, 11, 4, 2, 8],
    [13, 7, 0, 9, 3, 4, 6, 10, 2, 8, 5, 14, 12, 11, 15, 1],
    [13, 6, 4, 9, 8, 15, 3, 0, 11, 1, 2, 12, 5, 10, 14, 7],
    [1, 10, 13, 0, 6, 9, 8, 7, 4, 15, 14, 3, 11, 5, 2, 12]],

   [[7, 13, 14, 3, 0, 6, 9, 10, 1, 2, 8, 5, 11, 12, 4, 15],
    [13, 8, 11, 5, 6, 15, 0, 3, 4, 7, 2, 12, 1, 10, 14, 9],
    [10, 6, 9, 0, 12, 11, 7, 13, 15, 1, 3, 14, 5, 2, 8, 4],
    [3, 15, 0, 6, 10, 1, 13, 8, 9, 4, 5, 11, 12, 7, 2, 14]],

   [[2, 12, 4, 1, 7, 10, 11, 6, 8, 5, 3, 15, 13, 0, 14, 9],
    [14, 11, 2, 12, 4, 7, 13, 1, 5, 0, 15, 10, 3, 9, 8, 6],
    [4, 2, 1, 11, 10, 13, 7, 8, 15, 9, 12, 5, 6, 3, 0, 14],
    [11, 8, 12, 7, 1, 14, 2, 13, 6, 15, 0, 9, 10, 4, 5, 3]],
```

```
    [[12, 1, 10, 15, 9, 2, 6, 8, 0, 13, 3, 4, 14, 7, 5, 11],
     [10, 15, 4, 2, 7, 12, 9, 5, 6, 1, 13, 14, 0, 11, 3, 8],
     [9, 14, 15, 5, 2, 8, 12, 3, 7, 0, 4, 10, 1, 13, 11, 6],
     [4, 3, 2, 12, 9, 5, 15, 10, 11, 14, 1, 7, 6, 0, 8, 13]],

    [[4, 11, 2, 14, 15, 0, 8, 13, 3, 12, 9, 7, 5, 10, 6, 1],
     [13, 0, 11, 7, 4, 9, 1, 10, 14, 3, 5, 12, 2, 15, 8, 6],
     [1, 4, 11, 13, 12, 3, 7, 14, 10, 15, 6, 8, 0, 5, 9, 2],
     [6, 11, 13, 8, 1, 4, 10, 7, 9, 5, 0, 15, 14, 2, 3, 12]],

    [[13, 2, 8, 4, 6, 15, 11, 1, 10, 9, 3, 14, 5, 0, 12, 7],
     [1, 15, 13, 8, 10, 3, 7, 4, 12, 5, 6, 11, 0, 14, 9, 2],
     [7, 11, 4, 1, 9, 12, 14, 2, 0, 6, 10, 13, 15, 3, 5, 8],
     [2, 1, 14, 7, 4, 10, 8, 13, 15, 12, 9, 0, 3, 5, 6, 11]]
]

# Final Permutation Table
final_perm = [
    40, 8, 48, 16, 56, 24, 64, 32,
    39, 7, 47, 15, 55, 23, 63, 31,
    38, 6, 46, 14, 54, 22, 62, 30,
    37, 5, 45, 13, 53, 21, 61, 29,
    36, 4, 44, 12, 52, 20, 60, 28,
    35, 3, 43, 11, 51, 19, 59, 27,
    34, 2, 42, 10, 50, 18, 58, 26,
    33, 1, 41, 9, 49, 17, 57, 25
]
def encrypt(pt, rkb, rk):
    pt = hex2bin(pt)  # Initial Permutation
    pt = permute(pt, initial_perm, 64)
```

```python
print("After initial permutation", bin2hex(pt))

# Splitting
left = pt[0:32]
right = pt[32:64]

for i in range(0, 16):
    # Expansion D-box: Expanding the 32 bits data into 48 bits
    right_expanded = permute(right, exp_d, 48)

    # XOR RoundKey[i] and right_expanded
    xor_x = xor(right_expanded, rkb[i])

    # S-boxes: substituting the value from s-box table by calculating row and column
    sbox_str = ""
    for j in range(0, 8):
        row = bin2dec(int(xor_x[j * 6] + xor_x[j * 6 + 5]))
        col = bin2dec(int(
            xor_x[j * 6 + 1] + xor_x[j * 6 + 2] +
            xor_x[j * 6 + 3] + xor_x[j * 6 + 4]
        ))
        val = sbox[j][row][col]
        sbox_str = sbox_str + dec2bin(val)

    # Straight D-box: After substituting, rearranging the bits
    sbox_str = permute(sbox_str, per, 32)

    # XOR left and sbox_str
    result = xor(left, sbox_str)
    left = result
```

```
        # Swapper
        if i != 15:
            left, right = right, left
        print("Round ", i + 1, " ", bin2hex(left),
              " ", bin2hex(right), " ", rk[i])

    # Combination
    combine = left + right

    # Final permutation: final rearranging of bits to get cipher text
    cipher_text = permute(combine, final_perm, 64)
    return cipher_text

pt = "123456ABCD132536"
key = "AABB09182736CCDD"
keyy = "AABB09182736CCDD"
pt1= hex2bin(pt)
# Key generation
# --hex to binary
key = hex2bin(key)

# --parity bit drop table
keyp = [57, 49, 41, 33, 25, 17, 9,
        1, 58, 50, 42, 34, 26, 18,
        10, 2, 59, 51, 43, 35, 27,
        19, 11, 3, 60, 52, 44, 36,
        63, 55, 47, 39, 31, 23, 15,
        7, 62, 54, 46, 38, 30, 22,
        14, 6, 61, 53, 45, 37, 29,
        21, 13, 5, 28, 20, 12, 4]
```

```
# Getting 56-bit key from 64-bit using the parity bits
key = permute(key, keyp, 56)

# Number of bit shifts
shift_table = [1, 1, 2, 2,
               2, 2, 2, 2,
               1, 2, 2, 2,
               2, 2, 2, 1]

# Key-Compression Table: Compression of key from 56 bits to 48 bits
key_comp = [14, 17, 11, 24, 1, 5,
            3, 28, 15, 6, 21, 10,
            23, 19, 12, 4, 26, 8,
            16, 7, 27, 20, 13, 2,
            41, 52, 31, 37, 47, 55,
            30, 40, 51, 45, 33, 48,
            44, 49, 39, 56, 34, 53,
            46, 42, 50, 36, 29, 32]

# Splitting
left = key[0:28]  # rkb for RoundKeys in binary
right = key[28:56]  # rk for RoundKeys in hexadecimal

rkb = []
rk = []

# Generating 16 keys
for i in range(16):
  # Shifting
  left = shift_left(left, shift_table[i])
  right = shift_left(right, shift_table[i])
```

```
    # Combining
    combine_str = left + right

    # Key compression
    round_key = permute(combine_str, key_comp, 48)

    rkb.append(round_key)
    rk.append(bin2hex(round_key))

print("\nCode Submitted by Prakhar Sinha 22BCI0127")
print(f"Plaintext: {pt}")
print(f"Key: {keyy}")
print("\n--- Encryption Process ---")
print(f"Initial Plaintext: {pt}")

cipher_text = bin2hex(encrypt(pt, rkb, rk))
print("Cipher Text : ", cipher_text)

print("\n--- Decryption Process ---")
rkb_rev = rkb[::-1]
rk_rev = rk[::-1]
print(f"Initial Ciphertext: {cipher_text}")
text = bin2hex(encrypt(cipher_text, rkb_rev, rk_rev))
print("Plain Text : ", text)
```

DES 64 bit key size and 64 bit block size

Consider a sender and receiver who need to exchange data confidentiality using symmetric encryption. Write a program that implements DES encryption and decryption using a 64 bit key size and 64 bit block size

```
# Hexadecimal to binary conversion
```

```
def hex2bin(s):
    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': "1110", 'F': "1111"
    }
    binary = ""
    for char in s:
        binary += mp[char]
    return binary

# Binary to hexadecimal conversion
def bin2hex(s):
    mp = {
        "0000": '0', "0001": '1', "0010": '2', "0011": '3',
        "0100": '4', "0101": '5', "0110": '6', "0111": '7',
        "1000": '8', "1001": '9', "1010": 'A', "1011": 'B',
        "1100": 'C', "1101": 'D', "1110": 'E', "1111": 'F'
    }
    hex_result = ""
    for i in range(0, len(s), 4):
        hex_result += mp[s[i:i+4]]
    return hex_result

# Binary to decimal conversion
def bin2dec(binary):
    decimal = 0
    i = 0
    while binary != 0:
        decimal += (binary % 10) * (2 ** i)
```

```
        binary //= 10
        i += 1
    return decimal

# Decimal to binary conversion
def dec2bin(num):
    binary = bin(num).replace("0b", "")
    while len(binary) % 4 != 0:
        binary = '0' + binary
    return binary

# Permute function to rearrange the bits
def permute(k, arr, n):
    permutation = ""
    for i in range(n):
        permutation += k[arr[i] - 1]
    return permutation

# Left shift bits by n positions
def shift_left(k, nth_shifts):
    for _ in range(nth_shifts):
        k = k[1:] + k[0]
    return k

# XOR operation on two binary strings
def xor(a, b):
    return ''.join('0' if a[i] == b[i] else '1' for i in range(len(a)))

# Initial Permutation Table
initial_perm = [
    58, 50, 42, 34, 26, 18, 10, 2,
```

```
    60, 52, 44, 36, 28, 20, 12, 4,
    62, 54, 46, 38, 30, 22, 14, 6,
    64, 56, 48, 40, 32, 24, 16, 8,
    57, 49, 41, 33, 25, 17,  9, 1,
    59, 51, 43, 35, 27, 19, 11, 3,
    61, 53, 45, 37, 29, 21, 13, 5,
    63, 55, 47, 39, 31, 23, 15, 7
]

# Expansion D-box Table
exp_d = [
    32, 1, 2, 3, 4, 5, 4, 5,
     6, 7, 8, 9, 8, 9, 10, 11,
    12, 13, 12, 13, 14, 15, 16, 17,
    16, 17, 18, 19, 20, 21, 20, 21,
    22, 23, 24, 25, 24, 25, 26, 27,
    28, 29, 28, 29, 30, 31, 32, 1
]

# Straight Permutation Table
per = [
    16,  7, 20, 21, 29, 12, 28, 17,
     1, 15, 23, 26,  5, 18, 31, 10,
     2,  8, 24, 14, 32, 27,  3,  9,
    19, 13, 30,  6, 22, 11,  4, 25
]

# S-box Table
sbox = [
    [[14, 4, 13, 1, 2, 15, 11, 8, 3, 10, 6, 12, 5, 9, 0, 7],
     [0, 15, 7, 4, 14, 2, 13, 1, 10, 6, 12, 11, 9, 5, 3, 8],
```

[4, 1, 14, 8, 13, 6, 2, 11, 15, 12, 9, 7, 3, 10, 5, 0],

[15, 12, 8, 2, 4, 9, 1, 7, 5, 11, 3, 14, 10, 0, 6, 13]],

[[15, 1, 8, 14, 6, 11, 3, 4, 9, 7, 2, 13, 12, 0, 5, 10],

[3, 13, 4, 7, 15, 2, 8, 14, 12, 0, 1, 10, 6, 9, 11, 5],

[0, 14, 7, 11, 10, 4, 13, 1, 5, 8, 12, 6, 9, 3, 2, 15],

[13, 8, 10, 1, 3, 15, 4, 2, 11, 6, 7, 12, 0, 5, 14, 9]],

[[10, 0, 9, 14, 6, 3, 15, 5, 1, 13, 12, 7, 11, 4, 2, 8],

[13, 7, 0, 9, 3, 4, 6, 10, 2, 8, 5, 14, 12, 11, 15, 1],

[13, 6, 4, 9, 8, 15, 3, 0, 11, 1, 2, 12, 5, 10, 14, 7],

[1, 10, 13, 0, 6, 9, 8, 7, 4, 15, 14, 3, 11, 5, 2, 12]],

[[7, 13, 14, 3, 0, 6, 9, 10, 1, 2, 8, 5, 11, 12, 4, 15],

[13, 8, 11, 5, 6, 15, 0, 3, 4, 7, 2, 12, 1, 10, 14, 9],

[10, 6, 9, 0, 12, 11, 7, 13, 15, 1, 3, 14, 5, 2, 8, 4],

[3, 15, 0, 6, 10, 1, 13, 8, 9, 4, 5, 11, 12, 7, 2, 14]],

[[2, 12, 4, 1, 7, 10, 11, 6, 8, 5, 3, 15, 13, 0, 14, 9],

[14, 11, 2, 12, 4, 7, 13, 1, 5, 0, 15, 10, 3, 9, 8, 6],

[4, 2, 1, 11, 10, 13, 7, 8, 15, 9, 12, 5, 6, 3, 0, 14],

[11, 8, 12, 7, 1, 14, 2, 13, 6, 15, 0, 9, 10, 4, 5, 3]],

[[12, 1, 10, 15, 9, 2, 6, 8, 0, 13, 3, 4, 14, 7, 5, 11],

[10, 15, 4, 2, 7, 12, 9, 5, 6, 1, 13, 14, 0, 11, 3, 8],

[9, 14, 15, 5, 2, 8, 12, 3, 7, 0, 4, 10, 1, 13, 11, 6],

[4, 3, 2, 12, 9, 5, 15, 10, 11, 14, 1, 7, 6, 0, 8, 13]],

[[4, 11, 2, 14, 15, 0, 8, 13, 3, 12, 9, 7, 5, 10, 6, 1],

[13, 0, 11, 7, 4, 9, 1, 10, 14, 3, 5, 12, 2, 15, 8, 6],

[1, 4, 11, 13, 12, 3, 7, 14, 10, 15, 6, 8, 0, 5, 9, 2],

```
     [6, 11, 13, 8, 1, 4, 10, 7, 9, 5, 0, 15, 14, 2, 3, 12]],

    [[13, 2, 8, 4, 6, 15, 11, 1, 10, 9, 3, 14, 5, 0, 12, 7],
     [1, 15, 13, 8, 10, 3, 7, 4, 12, 5, 6, 11, 0, 14, 9, 2],
     [7, 11, 4, 1, 9, 12, 14, 2, 0, 6, 10, 13, 15, 3, 5, 8],
     [2, 1, 14, 7, 4, 10, 8, 13, 15, 12, 9, 0, 3, 5, 6, 11]]
]

# Final Permutation Table
final_perm = [
    40, 8, 48, 16, 56, 24, 64, 32,
    39, 7, 47, 15, 55, 23, 63, 31,
    38, 6, 46, 14, 54, 22, 62, 30,
    37, 5, 45, 13, 53, 21, 61, 29,
    36, 4, 44, 12, 52, 20, 60, 28,
    35, 3, 43, 11, 51, 19, 59, 27,
    34, 2, 42, 10, 50, 18, 58, 26,
    33, 1, 41, 9, 49, 17, 57, 25
]
def encrypt(pt, rkb, rk):
    pt = hex2bin(pt)  # Initial Permutation
    pt = permute(pt, initial_perm, 64)
    print("After initial permutation", bin2hex(pt))

    # Splitting
    left = pt[0:32]
    right = pt[32:64]

    for i in range(0, 16):
        # Expansion D-box: Expanding the 32 bits data into 48 bits
        right_expanded = permute(right, exp_d, 48)
```

```
    # XOR RoundKey[i] and right_expanded
    xor_x = xor(right_expanded, rkb[i])

    # S-boxes: substituting the value from s-box table by calculating row and column
    sbox_str = ""
    for j in range(0, 8):
        row = bin2dec(int(xor_x[j * 6] + xor_x[j * 6 + 5]))
        col = bin2dec(int(
            xor_x[j * 6 + 1] + xor_x[j * 6 + 2] +
            xor_x[j * 6 + 3] + xor_x[j * 6 + 4]
        ))
        val = sbox[j][row][col]
        sbox_str = sbox_str + dec2bin(val)

    # Straight D-box: After substituting, rearranging the bits
    sbox_str = permute(sbox_str, per, 32)

    # XOR left and sbox_str
    result = xor(left, sbox_str)
    left = result

    # Swapper
    if i != 15:
        left, right = right, left
    print("Round ", i + 1, " ", bin2hex(left),
          " ", bin2hex(right), " ", rk[i])

# Combination
combine = left + right
```

```
    # Final permutation: final rearranging of bits to get cipher text
    cipher_text = permute(combine, final_perm, 64)
    return cipher_text

pt = "123456ABCD132536"
key = "AABB09182736CCDD"
keyy = "AABB09182736CCDD"
pt1= hex2bin(pt)
# Key generation
# --hex to binary
key = hex2bin(key)

# --parity bit drop table
keyp = [57, 49, 41, 33, 25, 17, 9,
        1, 58, 50, 42, 34, 26, 18,
        10, 2, 59, 51, 43, 35, 27,
        19, 11, 3, 60, 52, 44, 36,
        63, 55, 47, 39, 31, 23, 15,
        7, 62, 54, 46, 38, 30, 22,
        14, 6, 61, 53, 45, 37, 29,
        21, 13, 5, 28, 20, 12, 4]

# Getting 56-bit key from 64-bit using the parity bits
key = permute(key, keyp, 56)

# Number of bit shifts
shift_table = [1, 1, 2, 2,
               2, 2, 2, 2,
               1, 2, 2, 2,
               2, 2, 2, 1]
```

```
# Key-Compression Table: Compression of key from 56 bits to 48 bits
key_comp = [14, 17, 11, 24, 1, 5,
            3, 28, 15, 6, 21, 10,
            23, 19, 12, 4, 26, 8,
            16, 7, 27, 20, 13, 2,
            41, 52, 31, 37, 47, 55,
            30, 40, 51, 45, 33, 48,
            44, 49, 39, 56, 34, 53,
            46, 42, 50, 36, 29, 32]

# Splitting
left = key[0:28]  # rkb for RoundKeys in binary
right = key[28:56]  # rk for RoundKeys in hexadecimal

rkb = []
rk = []

# Generating 16 keys
for i in range(16):
   # Shifting
   left = shift_left(left, shift_table[i])
   right = shift_left(right, shift_table[i])

   # Combining
   combine_str = left + right

   # Key compression
   round_key = permute(combine_str, key_comp, 48)

   rkb.append(round_key)
   rk.append(bin2hex(round_key))
```

```
print("\nCode Submitted by Prakhar Sinha 22BCI0127")
print(f"Plaintext: {pt}")
print(f"Key: {keyy}")
print("\n--- Encryption Process ---")
print(f"Initial Plaintext: {pt}")

cipher_text = bin2hex(encrypt(pt, rkb, rk))
print("Cipher Text : ", cipher_text)

print("\n--- Decryption Process ---")
rkb_rev = rkb[::-1]
rk_rev = rk[::-1]
print(f"Initial Ciphertext: {cipher_text}")
text = bin2hex(encrypt(cipher_text, rkb_rev, rk_rev))
print("Plain Text : ", text)
```

AES All Round

```
# Hexadecimal to binary conversion
def hex2bin(s):
    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': "1110", 'F': "1111"
    }
    binary = ""
    for char in s.upper():
        binary += mp[char]
    return binary

# Binary to hexadecimal conversion
```

```
def bin2hex(s):
    mp = {
        "0000": '0', "0001": '1', "0010": '2', "0011": '3',
        "0100": '4', "0101": '5', "0110": '6', "0111": '7',
        "1000": '8', "1001": '9', "1010": 'A', "1011": 'B',
        "1100": 'C', "1101": 'D', "1110": 'E', "1111": 'F'
    }
    hex_result = ""
    for i in range(0, len(s), 4):
        hex_result += mp[s[i:i+4]]
    return hex_result

# Galois Field multiplication in GF(2^8)
def gmul(a, b):
    p = 0
    for i in range(8):
        if (b & 1) != 0:
            p ^= a
        hi_bit_set = (a & 0x80) != 0
        a <<= 1
        if hi_bit_set:
            a ^= 0x1B  # The irreducible polynomial x^8 + x^4 + x^3 + x + 1
        b >>= 1
    return p % 256

# AES S-box
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
]

# Inverse AES S-box
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,
    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 for key expansion
rcon = [
    0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1B, 0x36,
    0x6C, 0xD8, 0xAB, 0x4D, 0x9A, 0x2F, 0x5E, 0xBC, 0x63, 0xC6
]

# Convert a state matrix to a hex string
def state_to_hex(state):
    hex_str = ""
    for col in range(4):
        for row in range(4):
            hex_str += format(state[row][col], '02X')
    return hex_str

# Convert a hex string to a state matrix
def hex_to_state(hex_str):
    state = [[0 for _ in range(4)] for _ in range(4)]
    for i in range(16):
        byte = int(hex_str[i*2:i*2+2], 16)
        row = i % 4
        col = i // 4
        state[row][col] = byte
    return state

# AES Key Expansion
def key_expansion(key):
    key_bytes = [int(key[i:i+2], 16) for i in range(0, len(key), 2)]
```

```
    # Initialize the expanded key
    expanded_key = [0] * 176  # 44 words * 4 bytes
    for i in range(16):
        expanded_key[i] = key_bytes[i]

    for i in range(4, 44):
        temp = [expanded_key[(i-1)*4 + j] for j in range(4)]

        if i % 4 == 0:
            # Rotate
            temp = temp[1:] + temp[:1]
            # SubBytes
            for j in range(4):
                temp[j] = sbox[temp[j]]
            # XOR with round constant
            temp[0] ^= rcon[i//4 - 1]

        for j in range(4):
            expanded_key[i*4 + j] = expanded_key[(i-4)*4 + j] ^ temp[j]

    return expanded_key

# SubBytes transformation
def sub_bytes(state):
    for i in range(4):
        for j in range(4):
            state[i][j] = sbox[state[i][j]]
    return state

# InvSubBytes transformation
def inv_sub_bytes(state):
```

```
    for i in range(4):
        for j in range(4):
            state[i][j] = inv_sbox[state[i][j]]
    return state

# ShiftRows transformation
def shift_rows(state):
    state[1] = state[1][1:] + state[1][:1]
    state[2] = state[2][2:] + state[2][:2]
    state[3] = state[3][3:] + state[3][:3]
    return state

# InvShiftRows transformation
def inv_shift_rows(state):
    state[1] = state[1][3:] + state[1][:3]
    state[2] = state[2][2:] + state[2][:2]
    state[3] = state[3][1:] + state[3][:1]
    return state

# MixColumns transformation
def mix_columns(state):
    for i in range(4):
        s0 = state[0][i]
        s1 = state[1][i]
        s2 = state[2][i]
        s3 = state[3][i]

        state[0][i] = gmul(2, s0) ^ gmul(3, s1) ^ s2 ^ s3
        state[1][i] = s0 ^ gmul(2, s1) ^ gmul(3, s2) ^ s3
        state[2][i] = s0 ^ s1 ^ gmul(2, s2) ^ gmul(3, s3)
        state[3][i] = gmul(3, s0) ^ s1 ^ s2 ^ gmul(2, s3)
```

```
    return state

# InvMixColumns transformation
def inv_mix_columns(state):
    for i in range(4):
        s0 = state[0][i]
        s1 = state[1][i]
        s2 = state[2][i]
        s3 = state[3][i]

        state[0][i] = gmul(0x0E, s0) ^ gmul(0x0B, s1) ^ gmul(0x0D, s2) ^ gmul(0x09, s3)
        state[1][i] = gmul(0x09, s0) ^ gmul(0x0E, s1) ^ gmul(0x0B, s2) ^ gmul(0x0D, s3)
        state[2][i] = gmul(0x0D, s0) ^ gmul(0x09, s1) ^ gmul(0x0E, s2) ^ gmul(0x0B, s3)
        state[3][i] = gmul(0x0B, s0) ^ gmul(0x0D, s1) ^ gmul(0x09, s2) ^ gmul(0x0E, s3)

    return state

# AddRoundKey transformation
def add_round_key(state, round_key, round_num):
    for i in range(4):
        for j in range(4):
            state[i][j] ^= round_key[round_num * 16 + j * 4 + i]
    return state

# AES Encryption
def encrypt(plaintext, key):
    # Convert plaintext to state matrix
    state = hex_to_state(plaintext)

    # Key expansion
```

```
    expanded_key = key_expansion(key)

    print("Initial state:", state_to_hex(state))

    # Initial round - AddRoundKey
    state = add_round_key(state, expanded_key, 0)
    print("After initial AddRoundKey:", state_to_hex(state))

    # Main rounds
    for round_num in range(1, 10):
        state = sub_bytes(state)
        state = shift_rows(state)
        state = mix_columns(state)
        state = add_round_key(state, expanded_key, round_num)
        print(f"Round {round_num}:", state_to_hex(state))

    # Final round (no MixColumns)
    state = sub_bytes(state)
    state = shift_rows(state)
    state = add_round_key(state, expanded_key, 10)

    print("Final state:", state_to_hex(state))

    # Convert state back to hex string
    ciphertext = state_to_hex(state)
    return ciphertext

# AES Decryption
def decrypt(ciphertext, key):
    # Convert ciphertext to state matrix
    state = hex_to_state(ciphertext)
```

```
    # Key expansion
    expanded_key = key_expansion(key)

    print("Initial state:", state_to_hex(state))

    # Initial round - AddRoundKey
    state = add_round_key(state, expanded_key, 10)
    state = inv_shift_rows(state)
    state = inv_sub_bytes(state)
    print("After initial inverse transformations:", state_to_hex(state))

    # Main rounds
    for round_num in range(9, 0, -1):
        state = add_round_key(state, expanded_key, round_num)
        state = inv_mix_columns(state)
        state = inv_shift_rows(state)
        state = inv_sub_bytes(state)
        print(f"Round {10-round_num}:", state_to_hex(state))

    # Final round
    state = add_round_key(state, expanded_key, 0)

    print("Final state:", state_to_hex(state))

    # Convert state back to hex string
    plaintext = state_to_hex(state)
    return plaintext

# Main
pt = "00112233445566778899AABBCCDDEEFF"
```

```
key = "000102030405060708090A0B0C0D0E0F"
key_copy = "000102030405060708090A0B0C0D0E0F"

print(f"Plaintext: {pt}")
print(f"Key: {key_copy}")

print("\n--- Encryption Process ---")
print(f"Initial Plaintext: {pt}")

cipher_text = encrypt(pt, key)
print("Cipher Text: ", cipher_text)

print("\n--- Decryption Process ---")
print(f"Initial Ciphertext: {cipher_text}")

decrypted_text = decrypt(cipher_text, key)
print("Plain Text: ", decrypted_text)
```

AES All States - One Round

```
import numpy as np

def print_state(state, label=""):
    """Print the state matrix in hexadecimal format with a label."""
    if label:
        print(f"State after {label}:")
    else:
        print("State:")
    for row in state:
        print(' '.join(f'{x:02x}' for x in row))
    print()

def s_box(byte):
```

```
    """S-Box substitution."""
    s_box_table = [
        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
    ]
    return s_box_table[byte]


def inv_s_box(byte):
    """Inverse S-Box substitution."""
    inv_s_box_table = [
        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,
        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
    ]
    return inv_s_box_table[byte]

def sub_bytes(state):
    """Apply S-Box substitution to each byte of the state."""
    for row in range(4):
        for col in range(4):
            state[row, col] = s_box(state[row, col])
    return state

def inv_sub_bytes(state):
    """Apply inverse S-Box substitution to each byte of the state."""
    for row in range(4):
        for col in range(4):
            state[row, col] = inv_s_box(state[row, col])
    return state

def shift_rows(state):
    """Shift rows of the state matrix."""
```

```
    for row in range(1, 4):
        state[row] = np.roll(state[row], -row)
    return state

def inv_shift_rows(state):
    """Inverse shift rows of the state matrix."""
    for row in range(1, 4):
        state[row] = np.roll(state[row], row)
    return state

def copy_to_state(input_array):
    """Convert 1D input to 4x4 state matrix."""
    state = np.zeros((4, 4), dtype=np.uint8)
    for i in range(4):
        for j in range(4):
            state[j, i] = input_array[i * 4 + j]
    return state

def state_to_output(state):
    """Convert 4x4 state matrix to 1D output."""
    output = np.zeros(16, dtype=np.uint8)
    for i in range(4):
        for j in range(4):
            output[i * 4 + j] = state[j, i]
    return output

def key_expansion(key, num_rounds):
    """Expand the key into the key schedule."""
    rcon = [0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1B, 0x36]
    key_words = [key[i:i+4] for i in range(0, len(key), 4)]
```

```
    # For AES-128, we need 44 words (11 round keys * 4 words each)
    expanded_key_words = key_words.copy()

    for i in range(len(key_words), 4 * (num_rounds + 1)):
        temp = expanded_key_words[i-1].copy()

        if i % len(key_words) == 0:
            # Rotate word
            temp = np.roll(temp, -1)
            # SubWord
            for j in range(4):
                temp[j] = s_box(temp[j])
            # XOR with round constant
            temp[0] ^= rcon[(i // len(key_words)) - 1]

        expanded_key_words.append(np.bitwise_xor(expanded_key_words[i - len(key_words)], temp))

    # Flatten the expanded key
    expanded_key = np.array(expanded_key_words).flatten()
    return expanded_key

def add_round_key(state, key_schedule, round_num):
    """XOR the state with the round key."""
    round_key = key_schedule[round_num * 16:(round_num + 1) * 16].reshape(4, 4).T
    for row in range(4):
        for col in range(4):
            state[row, col] ^= round_key[row, col]
    return state

def gmul(a, b):
    """Galois Field multiplication in GF(2^8)."""
```

```
    p = 0
    for _ in range(8):
        if b & 1:
            p ^= a
        hi_bit_set = a & 0x80
        a <<= 1
        if hi_bit_set:
            a ^= 0x1B  # x^8 + x^4 + x^3 + x + 1
        b >>= 1
    return p & 0xFF

def mix_columns(state):
    """Apply MixColumns transformation."""
    result = np.zeros_like(state)
    for col in range(4):
        result[0, col] = gmul(2, state[0, col]) ^ gmul(3, state[1, col]) ^ state[2, col] ^ state[3, col]
        result[1, col] = state[0, col] ^ gmul(2, state[1, col]) ^ gmul(3, state[2, col]) ^ state[3, col]
        result[2, col] = state[0, col] ^ state[1, col] ^ gmul(2, state[2, col]) ^ gmul(3, state[3, col])
        result[3, col] = gmul(3, state[0, col]) ^ state[1, col] ^ state[2, col] ^ gmul(2, state[3, col])

    return result

def inv_mix_columns(state):
    """Apply Inverse MixColumns transformation."""
    result = np.zeros_like(state)
    for col in range(4):
        result[0, col] = gmul(0x0E, state[0, col]) ^ gmul(0x0B, state[1, col]) ^ gmul(0x0D, state[2, col]) ^
gmul(0x09, state[3, col])
        result[1, col] = gmul(0x09, state[0, col]) ^ gmul(0x0E, state[1, col]) ^ gmul(0x0B, state[2, col]) ^
gmul(0x0D, state[3, col])
        result[2, col] = gmul(0x0D, state[0, col]) ^ gmul(0x09, state[1, col]) ^ gmul(0x0E, state[2, col]) ^
gmul(0x0B, state[3, col])
```

```
        result[3, col] = gmul(0x0B, state[0, col]) ^ gmul(0x0D, state[1, col]) ^ gmul(0x09, state[2, col]) ^
gmul(0x0E, state[3, col])

    return result

def aes_encrypt(input_data, key, num_rounds):
    """Encrypt input data using AES with detailed state output."""
    state = copy_to_state(input_data)
    key_schedule = key_expansion(key, num_rounds)

    print("Original State (4x4 Matrix):")
    print_state(state)

    # Initial round
    state = add_round_key(state, key_schedule, 0)
    print_state(state, "AddRoundKey")

    # Main rounds
    for round_num in range(1, num_rounds):
        state = sub_bytes(state)
        if round_num == 1:  # Only show SubBytes output for the first round
            print_state(state, "SubBytes")

        state = shift_rows(state)
        if round_num == 1:  # Only show ShiftRows output for the first round
            print_state(state, "ShiftRows")

        state = mix_columns(state)
        if round_num == 1:  # Only show MixColumns output for the first round
            # Extract values for first column as shown in your example
            b0, b4, b8 = state[0, 0], state[1, 0], state[2, 0]
```

```python
            print_state(state, f"MixColumns ({b0:02x}, {b4:02x}, {b8:02x})")

        state = add_round_key(state, key_schedule, round_num)

    # Final round (no MixColumns)
    state = sub_bytes(state)
    state = shift_rows(state)
    state = add_round_key(state, key_schedule, num_rounds)

    print("Encrypted State:")
    print_state(state)
    return state

def aes_decrypt(encrypted_state, key, num_rounds):
    """Decrypt input data using AES."""
    state = encrypted_state.copy()
    key_schedule = key_expansion(key, num_rounds)

    # Initial round
    state = add_round_key(state, key_schedule, num_rounds)
    state = inv_shift_rows(state)
    state = inv_sub_bytes(state)

    # Main rounds
    for round_num in range(num_rounds - 1, 0, -1):
        state = add_round_key(state, key_schedule, round_num)
        state = inv_mix_columns(state)
        state = inv_shift_rows(state)
        state = inv_sub_bytes(state)

    # Final round
```

```python
    state = add_round_key(state, key_schedule, 0)

    print("Decrypted State:")
    print_state(state)
    return state

def main():
    # Example with 128-bit key and 10 rounds (AES-128)
    key = np.array([0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08,
                0x09, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F, 0x10], dtype=np.uint8)

    # Match the input data from your images
    input_data = np.array([0x00, 0x00, 0x01, 0x00,
                    0x02, 0x03, 0x03, 0x04,
                    0x04, 0x06, 0x06, 0x08,
                    0x08, 0x07, 0x0A, 0x09], dtype=np.uint8)

    num_rounds = 10  # AES-128 uses 10 rounds
    encrypted_state = aes_encrypt(input_data, key, num_rounds)
    decrypted_state = aes_decrypt(encrypted_state, key, num_rounds)

    # Verify the decryption matches the original
    original_state = copy_to_state(input_data)
    print("Decryption Verification:")
    print("Original == Decrypted:", np.array_equal(original_state, decrypted_state))
    print("PS C:\\Users\\Prakhar>")

if __name__ == "__main__":
    main()
```

RSA

```java
import java.math.BigInteger;
```

```java
public class Main {
   // RSA Parameters
   private static BigInteger p = BigInteger.valueOf(7);
   private static BigInteger q = BigInteger.valueOf(11);
   private static BigInteger n;
   private static BigInteger phi;
   private static BigInteger e = BigInteger.valueOf(7);
   private static BigInteger d;

   public static void main(String[] args) {
      System.out.println("RSA Encryption/Decryption Example");
      System.out.println("Prakhar Sinha 22BCI0127");
      System.out.println("================================\n");

      BigInteger message = BigInteger.valueOf(9);

      generateKeys();

      BigInteger ciphertext = encrypt(message);

      BigInteger decryptedMessage = decrypt(ciphertext);

      printResults(message, ciphertext, decryptedMessage);
   }

   public static void generateKeys() {
      System.out.println("Key Generation:");
      System.out.println("--------------");

      n = p.multiply(q);
```

```
        System.out.println("Step 1:");
        System.out.println("p = " + p);
        System.out.println("q = " + q);
        System.out.println("n = p × q = " + n);

        phi = p.subtract(BigInteger.ONE).multiply(q.subtract(BigInteger.ONE));
        System.out.println("\nStep 2:");
        System.out.println("ϕ(n) = (p-1) × (q-1) = " + phi);

        System.out.println("\nStep 3:");
        System.out.println("Public key (e) = " + e);

        d = e.modInverse(phi);
        System.out.println("\nStep 4:");
        System.out.println("Private key (d) = " + d);

        System.out.println("\nPublic Key (e,n) = (" + e + "," + n + ")");
        System.out.println("Private Key (d,n) = (" + d + "," + n + ")\n");
    }

    public static BigInteger encrypt(BigInteger message) {
        System.out.println("Encryption Process:");
        System.out.println("-------------------");
        System.out.println("Using formula: c = m^e mod n");
        System.out.println("Message (m) = " + message);
        System.out.println("e = " + e);
        System.out.println("n = " + n);

        BigInteger ciphertext = message.modPow(e, n);
        System.out.println("Ciphertext (c) = " + message + "^" + e + " mod " + n + " = " +
ciphertext + "\n");
```

```
        return ciphertext;
    }

    public static BigInteger decrypt(BigInteger ciphertext) {
        System.out.println("Decryption Process:");
        System.out.println("------------------");
        System.out.println("Using formula: m = c^d mod n");
        System.out.println("Ciphertext (c) = " + ciphertext);
        System.out.println("d = " + d);
        System.out.println("n = " + n);

        BigInteger decryptedMessage = ciphertext.modPow(d, n);
        System.out.println("Decrypted message (m) = " + ciphertext + "^" + d + " mod " + n + " = " +
decryptedMessage + "\n");

        return decryptedMessage;
    }

    public static void printResults(BigInteger original, BigInteger encrypted, BigInteger
decrypted) {
        System.out.println("Final Results:");
        System.out.println("-------------");
        System.out.println("Original Message: " + original);
        System.out.println("Encrypted Message (Ciphertext): " + encrypted);
        System.out.println("Decrypted Message: " + decrypted);
        System.out.println("Decryption Successful: " + original.equals(decrypted));
    }
}
```

RSA c,e,n given - DEcryption

```
class Main {
```

```
static int modInverse(int e, int phi) {
    int m0 = phi, t, q;
    int x0 = 0, x1 = 1;

    if (phi == 1)
        return 0;

    while (e > 1) {
        q = e / phi;
        t = phi;
        phi = e % phi;
        e = t;
        t = x0;
        x0 = x1 - q * x0;
        x1 = t;
    }

    if (x1 < 0)
        x1 += m0;

    return x1;
}

static int modExp(int base, int exp, int mod) {
    int result = 1;
    base = base % mod;

    while (exp > 0) {
        if ((exp & 1) == 1)
            result = (result * base) % mod;
```

```
        exp = exp >> 1;
        base = (base * base) % mod;
    }
    return result;
}

public static void main(String[] args) {
    int e = 13;
    int n = 77;
    int C = 20;

    int p = 7;
    int q = 11;
    int phi = (p - 1) * (q - 1);

    int d = modInverse(e, phi);

    int M = modExp(C, d, n);

    System.out.println("Code Submitted by Prakhar Sinha 22BCI0127");
    System.out.println("----------------------------------------");

    System.out.println("Public Key (e, n): (" + e + ", " + n + ")");
    System.out.println("Private Key (d, n): (" + d + ", " + n + ")");
    System.out.println("Ciphertext (C): " + C);

    System.out.println("Step 1: Calculate ϕ(n) = (p - 1) * (q - 1) = " + phi);
    System.out.println("Step 2: Compute d, the modular inverse of e mod ϕ(n): " + d);
    System.out.println("Step 3: Decrypt C using the formula M ≡ C^d mod n");
```

```
    System.out.println("Decrypted Plaintext (M): " + M);
  }
}
```

RSA Encryption Decryption

```
public class Main {
  static int modExp(int base, int exp, int mod) {
    int result = 1;
    base = base % mod;

    while (exp > 0) {
      if ((exp & 1) == 1)
        result = (result * base) % mod;

      exp = exp >> 1;
      base = base * base % mod;
    }
    return result;
  }

  static int modInverse(int e, int phi) {
    int m0 = phi, t, q;
    int x0 = 0, x1 = 1;

    if (phi == 1)
      return 0;

    while (e > 1) {
      q = e / phi;
      t = phi;
      phi = e % phi;
      e = t;
```

```
        t = x0;
        x0 = x1 - q * x0;
        x1 = t;
    }

    if (x1 < 0)
        x1 += m0;

    return x1;
}

static int encrypt(int M, int e, int n) {
    return modExp(M, e, n);
}

static int decrypt(int C, int d, int n) {
    return modExp(C, d, n);
}

public static void main(String[] args) {
    int p = 5;
    int q = 13;
    int e = 5;
    int M = 8;

    int n = p * q;
    int phi = (p - 1) * (q - 1);
    int d = modInverse(e, phi);

    System.out.println("Encryption");
    System.out.println("----------");
```

```
    int C = encrypt(M, e, n);
    System.out.println("Step 1: Compute the modulus n = p * q");
    System.out.println("n = " + n);
    System.out.println("Step 2: Compute Euler's Totient ϕ(n) = (p-1) * (q-1)");
    System.out.println("ϕ(n) = " + phi);
    System.out.println("Step 3: Encrypt the original message M using the public key (e, n)");
    System.out.println("Original Message (M) = " + M);
    System.out.println("Encrypted Ciphertext (C) = " + C);
System.out.println("Decryption");
System.out.println("----------");
int decryptedM = decrypt(C, d, n);
System.out.println("Step 1: Compute the private key exponent d = e^(-1) mod ϕ(n)");
System.out.println("d = " + d);
System.out.println("Step 2: Decrypt the ciphertext C using the private key (d, n)");
System.out.println("Decrypted Plaintext (M) = " + decryptedM);
}
}
```

RSA Text

```
public class Main {
  static int modExp(int base, int exp, int mod) {
    int result = 1;
    base = base % mod;

    while (exp > 0) {
      if ((exp & 1) == 1)
        result = (result * base) % mod;

      exp = exp >> 1;
      base = base * base % mod;
    }
    return result;
```

```
}

static int modInverse(int e, int phi) {
    int m0 = phi, t, q;
    int x0 = 0, x1 = 1;

    if (phi == 1)
        return 0;

    while (e > 1) {
        q = e / phi;
        t = phi;
        phi = e % phi;
        e = t;
        t = x0;
        x0 = x1 - q * x0;
        x1 = t;
    }

    if (x1 < 0)
        x1 += m0;

    return x1;
}

static int[] encryptString(String message, int e, int n) {
    int[] encrypted = new int[message.length()];

    for (int i = 0; i < message.length(); i++) {
        encrypted[i] = modExp((int)message.charAt(i), e, n);
    }
```

```
    return encrypted;
}

static String decryptString(int[] encrypted, int d, int n) {
    StringBuilder decrypted = new StringBuilder();

    for (int i = 0; i < encrypted.length; i++) {
        decrypted.append((char)modExp(encrypted[i], d, n));
    }

    return decrypted.toString();
}

public static void main(String[] args) {
    // RSA key parameters
    int p = 61; // Larger prime
    int q = 53; // Larger prime
    int e = 17; // Common value for e

    // Original message
    String message = "Hello, RSA!";

    int n = p * q;
    int phi = (p - 1) * (q - 1);
    int d = modInverse(e, phi);

    System.out.println("Encryption");
    System.out.println("----------");
    int[] encryptedMessage = encryptString(message, e, n);
```

```
        System.out.println("Step 1: Compute the modulus n = p * q");
        System.out.println("n = " + n);
        System.out.println("Step 2: Compute Euler's Totient ϕ(n) = (p-1) * (q-1)");
        System.out.println("ϕ(n) = " + phi);
        System.out.println("Step 3: Encrypt the original message using the public key (e, n)");
        System.out.println("Original Message = \"" + message + "\"");

        System.out.print("Encrypted Ciphertext = [");
        for (int i = 0; i < encryptedMessage.length; i++) {
            System.out.print(encryptedMessage[i]);
            if (i < encryptedMessage.length - 1) {
                System.out.print(", ");
            }
        }
        System.out.println("]");

        System.out.println("\nDecryption");
        System.out.println("----------");
        String decryptedMessage = decryptString(encryptedMessage, d, n);

        System.out.println("Step 1: Compute the private key exponent d = e^(-1) mod ϕ(n)");
        System.out.println("d = " + d);
        System.out.println("Step 2: Decrypt the ciphertext using the private key (d, n)");
        System.out.println("Decrypted Plaintext = \"" + decryptedMessage + "\"");
    }
}
```

MD5 - hash

```
public class Main {
    private static final int[] SHIFT = {
        7, 12, 17, 22, 7, 12, 17, 22, 7, 12, 17, 22, 7, 12, 17, 22,
        5, 9, 14, 20, 5, 9, 14, 20, 5, 9, 14, 20, 5, 9, 14, 20,
```

```
    4, 11, 16, 23, 4, 11, 16, 23, 4, 11, 16, 23, 4, 11, 16, 23,
    6, 10, 15, 21, 6, 10, 15, 21, 6, 10, 15, 21, 6, 10, 15, 21
};

private static final int[] TABLE = new int[64];

static {
    for (int i = 0; i < 64; i++) {
        TABLE[i] = (int) (long) ((1L << 32) * Math.abs(Math.sin(i + 1)));
    }
}

public static String getMd5(String input) {
    byte[] message = input.getBytes();
    int messageLenBytes = message.length;
    int numBlocks = ((messageLenBytes + 8) >>> 6) + 1;
    int totalLen = numBlocks << 6;
    byte[] paddingBytes = new byte[totalLen - messageLenBytes];
    paddingBytes[0] = (byte) 0x80;

    long messageLenBits = (long) messageLenBytes << 3;
    for (int i = 0; i < 8; i++) {
        paddingBytes[paddingBytes.length - 8 + i] = (byte) messageLenBits;
        messageLenBits >>>= 8;
    }

    int a0 = 0x67452301;
    int b0 = 0xefcdab89;
    int c0 = 0x98badcfe;
    int d0 = 0x10325476;
```

```
        int[] buffer = new int[16];
        for (int i = 0; i < numBlocks; i++) {
            int index = i << 6;
            for (int j = 0; j < 16; j++) {
                buffer[j] = 0;
                for (int k = 0; k < 4; k++) {
                    int offset = index + (j << 2) + k;
                    int value = offset < messageLenBytes ? message[offset] : paddingBytes[offset -
messageLenBytes];
                    buffer[j] |= ((value & 0xff) << (k << 3));
                }
            }

            int A = a0;
            int B = b0;
            int C = c0;
            int D = d0;

            for (int j = 0; j < 64; j++) {
                int F;
                int bufferIndex;

                if (j < 16) {
                    F = (B & C) | (~B & D);
                    bufferIndex = j;
                } else if (j < 32) {
                    F = (D & B) | (~D & C);
                    bufferIndex = (5 * j + 1) & 0x0F;
                } else if (j < 48) {
                    F = B ^ C ^ D;
                    bufferIndex = (3 * j + 5) & 0x0F;
```

```
        } else {
            F = C ^ (B | ~D);
            bufferIndex = (7 * j) & 0x0F;
        }

        int temp = B + Integer.rotateLeft(A + F + buffer[bufferIndex] + TABLE[j], SHIFT[j]);
        A = D;
        D = C;
        C = B;
        B = temp;
    }

    a0 += A;
    b0 += B;
    c0 += C;
    d0 += D;
}

byte[] digest = new byte[16];
int[] buf = new int[]{a0, b0, c0, d0};
for (int i = 0; i < 4; i++) {
    for (int j = 0; j < 4; j++) {
        digest[i * 4 + j] = (byte) ((buf[i] >>> (j * 8)) & 0xFF);
    }
}

StringBuilder hexString = new StringBuilder();
for (byte b : digest) {
    String hex = Integer.toHexString(0xFF & b);
    if (hex.length() == 1) {
        hexString.append('0');
```

```
        }
        hexString.append(hex);
      }
      return hexString.toString();
    }

    public static void main(String[] args) {
      String input = "Cryptography";
      System.out.println("Your HashCode Generated by MD5 is: " + getMd5(input));
    }
}
```

MD5 - Binary

```
import java.math.BigInteger;
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;

public class Main {
    private static final int[] SHIFT_AMOUNTS = {
      7, 12, 17, 22,  7, 12, 17, 22,  7, 12, 17, 22,  7, 12, 17, 22,
      5,  9, 14, 20,  5,  9, 14, 20,  5,  9, 14, 20,  5,  9, 14, 20,
      4, 11, 16, 23,  4, 11, 16, 23,  4, 11, 16, 23,  4, 11, 16, 23,
      6, 10, 15, 21,  6, 10, 15, 21,  6, 10, 15, 21,  6, 10, 15, 21
    };

    private static final int[] K_TABLE = new int[64];
    static {
      for (int i = 0; i < 64; i++) {
        K_TABLE[i] = (int)(Math.floor(Math.abs(Math.sin(i + 1)) * Math.pow(2, 32)));
      }
    }
```

```
    public static void showSteps(String input) {
        try {
            System.out.println("MD5 Hash Computation Steps");
            System.out.println("=========================");
            System.out.println("Input message: \"" + input + "\"");

            System.out.println("\nStep 1: Converting message to binary");
            byte[] messageBytes = input.getBytes();
            System.out.println("Message length: " + messageBytes.length + " bytes (" +
(messageBytes.length * 8) + " bits)");
            System.out.println("Binary representation (first 8 bytes):");
            for (int i = 0; i < Math.min(8, messageBytes.length); i++) {
                System.out.printf("%8s ", String.format("%8s",
                    Integer.toBinaryString(messageBytes[i] & 0xFF)).replace(' ', '0'));
            }
            System.out.println("...");

            System.out.println("\nStep 2: Calculating and applying MD5 padding");
            int messageLengthBits = messageBytes.length * 8;
            int totalBitsNeeded = ((messageLengthBits + 64 + 1 + 511) / 512) * 512;
            int paddingBits = totalBitsNeeded - messageLengthBits - 64;

            int paddedLength = totalBitsNeeded / 8;
            byte[] paddedMessage = new byte[paddedLength];

            System.arraycopy(messageBytes, 0, paddedMessage, 0, messageBytes.length);

            paddedMessage[messageBytes.length] = (byte) 0x80;

            long originalLength = messageLengthBits & 0xFFFFFFFFFFFFFFFFL;
            for (int i = 0; i < 8; i++) {
```

```
    paddedMessage[paddedMessage.length - 8 + i] = (byte) originalLength;
    originalLength >>>= 8;
}

System.out.println("Original length in bits: " + messageLengthBits);
System.out.println("Padding bits needed: " + paddingBits);
System.out.println("Final length in bits: " + totalBitsNeeded);
System.out.println("\nPadding structure:");
System.out.println("- Original message: " + messageLengthBits + " bits");
System.out.println("- Single '1' bit");
System.out.println("- " + (paddingBits - 1) + " zero bits");
System.out.println("- 64 bits for length");

System.out.println("\nStep 3: Message blocks structure");
int numBlocks = totalBitsNeeded / 512;
System.out.println("Number of 512-bit blocks: " + numBlocks);
System.out.println("Block structure:");
for (int i = 0; i < numBlocks; i++) {
    System.out.printf("Block %d: %d-%d bytes\n",
        i + 1, i * 64, Math.min((i + 1) * 64, paddedMessage.length));
}

System.out.println("\nStep 4: First operation in Round 1");
int A = 0x67452301;
int B = 0xEFCDAB89;
int C = 0x98BADCFE;
int D = 0x10325476;

System.out.println("Initial buffer values:");
System.out.printf("A = %08x\n", A);
System.out.printf("B = %08x\n", B);
```

```java
        System.out.printf("C = %08x\n", C);
        System.out.printf("D = %08x\n", D);

        System.out.println("\nFirst round operation:");
        System.out.println("F(B,C,D) = (B AND C) OR (NOT B AND D)");
        int firstF = F(B, C, D);
        System.out.printf("F(%08x, %08x, %08x) = %08x\n", B, C, D, firstF);

        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;
        }

        System.out.println("\nFinal MD5 Hash:");
        System.out.println("---------------");
        System.out.println(hashtext);

    } catch (NoSuchAlgorithmException e) {
        throw new RuntimeException(e);
    }
}

private static int F(int x, int y, int z) {
    return (x & y) | (~x & z);
}

private static int G(int x, int y, int z) {
    return (x & z) | (y & ~z);
```

```
    }

    private static int H(int x, int y, int z) {
        return x ^ y ^ z;
    }

    private static int I(int x, int y, int z) {
        return y ^ (x | ~z);
    }

    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 input = "cryptography and network security.";
        showSteps(input);
    }
}
```

MD5 - MAC

```java
import java.nio.charset.StandardCharsets;

public class Main {
    private static final int[] SHIFT = {
        7, 12, 17, 22, 7, 12, 17, 22, 7, 12, 17, 22, 7, 12, 17, 22,
        5, 9, 14, 20, 5, 9, 14, 20, 5, 9, 14, 20, 5, 9, 14, 20,
        4, 11, 16, 23, 4, 11, 16, 23, 4, 11, 16, 23, 4, 11, 16, 23,
        6, 10, 15, 21, 6, 10, 15, 21, 6, 10, 15, 21, 6, 10, 15, 21
    };

    private static final int[] TABLE = new int[64];

    static {
        for (int i = 0; i < 64; i++) {
            TABLE[i] = (int) (long) ((1L << 32) * Math.abs(Math.sin(i + 1)));
        }
    }

    public static byte[] getMd5Bytes(byte[] message) {
        int messageLenBytes = message.length;
        int numBlocks = ((messageLenBytes + 8) >>> 6) + 1;
        int totalLen = numBlocks << 6;
        byte[] paddingBytes = new byte[totalLen - messageLenBytes];
        paddingBytes[0] = (byte) 0x80;

        long messageLenBits = (long) messageLenBytes << 3;
        for (int i = 0; i < 8; i++) {
            paddingBytes[paddingBytes.length - 8 + i] = (byte) messageLenBits;
            messageLenBits >>>= 8;
        }
```

```
        int a0 = 0x67452301;
        int b0 = 0xefcdab89;
        int c0 = 0x98badcfe;
        int d0 = 0x10325476;

        int[] buffer = new int[16];
        for (int i = 0; i < numBlocks; i++) {
            int index = i << 6;
            for (int j = 0; j < 16; j++) {
                buffer[j] = 0;
                for (int k = 0; k < 4; k++) {
                    int offset = index + (j << 2) + k;
                    int value = offset < messageLenBytes ? message[offset] : paddingBytes[offset -
messageLenBytes];
                    buffer[j] |= ((value & 0xff) << (k << 3));
                }
            }

            int A = a0;
            int B = b0;
            int C = c0;
            int D = d0;

            for (int j = 0; j < 64; j++) {
                int F;
                int bufferIndex;

                if (j < 16) {
                    F = (B & C) | (~B & D);
                    bufferIndex = j;
                } else if (j < 32) {
```

```
            F = (D & B) | (~D & C);
            bufferIndex = (5 * j + 1) & 0x0F;
        } else if (j < 48) {
            F = B ^ C ^ D;
            bufferIndex = (3 * j + 5) & 0x0F;
        } else {
            F = C ^ (B | ~D);
            bufferIndex = (7 * j) & 0x0F;
        }

        int temp = B + Integer.rotateLeft(A + F + buffer[bufferIndex] + TABLE[j], SHIFT[j]);
        A = D;
        D = C;
        C = B;
        B = temp;
    }

    a0 += A;
    b0 += B;
    c0 += C;
    d0 += D;
}

byte[] digest = new byte[16];
int[] buf = new int[]{a0, b0, c0, d0};
for (int i = 0; i < 4; i++) {
    for (int j = 0; j < 4; j++) {
        digest[i * 4 + j] = (byte) ((buf[i] >>> (j * 8)) & 0xFF);
    }
}
```

```
    return digest;
}

public static String hmacMd5(String message, String key) {
    byte[] keyBytes = key.getBytes(StandardCharsets.UTF_8);
    byte[] messageBytes = message.getBytes(StandardCharsets.UTF_8);

    // HMAC-MD5 implementation
    byte[] adjustedKey = adjustKey(keyBytes);
    byte[] ipad = xorBytes(adjustedKey, 0x36);
    byte[] opad = xorBytes(adjustedKey, 0x5c);

    // Inner hash
    byte[] inner = concatenate(ipad, messageBytes);
    byte[] innerHash = getMd5Bytes(inner);

    // Outer hash
    byte[] outer = concatenate(opad, innerHash);
    byte[] mac = getMd5Bytes(outer);

    return bytesToHex(mac);
}

private static byte[] adjustKey(byte[] key) {
    if (key.length > 64) {
        return getMd5Bytes(key);
    }
    byte[] adjusted = new byte[64];
    System.arraycopy(key, 0, adjusted, 0, key.length);
    return adjusted;
}
```

```
private static byte[] xorBytes(byte[] data, int value) {
    byte[] result = new byte[data.length];
    for (int i = 0; i < data.length; i++) {
        result[i] = (byte) (data[i] ^ value);
    }
    return result;
}

private static byte[] concatenate(byte[] a, byte[] b) {
    byte[] result = new byte[a.length + b.length];
    System.arraycopy(a, 0, result, 0, a.length);
    System.arraycopy(b, 0, result, a.length, b.length);
    return result;
}

private static String bytesToHex(byte[] bytes) {
    StringBuilder hexString = new StringBuilder();
    for (byte b : bytes) {
        String hex = Integer.toHexString(0xFF & b);
        if (hex.length() == 1) {
            hexString.append('0');
        }
        hexString.append(hex);
    }
    return hexString.toString();
}

public static void main(String[] args) {
    String message = "The quick brown fox jumps over the lazy dog";
    String key = "key";
```

```
        System.out.println("HMAC-MD5: " + hmacMd5(message, key));
    }
}
```

Diffie - MITM

```
public class Main {
    static long modPow(long base, long exponent, long modulus) {
        long 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;
    }

    public static void main(String[] args) {
        long p = 23;
        long g = 5;
        long a = 6;
        long b = 15;
        long e = 10;

        System.out.println("Diffie-Hellman Key Exchange with MITM Attack\n");
        System.out.println("Public Values:");
        System.out.println("Prime number p = " + p);
        System.out.println("Primitive root g = " + g + "\n");
```

```
System.out.println("Private Keys:");
System.out.println("Alice's private key a = " + a);
System.out.println("Bob's private key b = " + b);
System.out.println("Eve's private key e = " + e + "\n");

long A = modPow(g, a, p);
System.out.println("1. Alice's public key:");
System.out.println("A = " + g + "^" + a + " mod " + p);
System.out.println("A = " + A + "\n");

long B = modPow(g, b, p);
System.out.println("2. Bob's public key:");
System.out.println("B = " + g + "^" + b + " mod " + p);
System.out.println("B = " + B + "\n");

long E = modPow(g, e, p);
System.out.println("3. Eve's intercepted public key:");
System.out.println("E = " + g + "^" + e + " mod " + p);
System.out.println("E = " + E + "\n");

long KA = modPow(E, a, p);
System.out.println("4. Key Alice derives with Eve:");
System.out.println("KA = " + E + "^" + a + " mod " + p);
System.out.println("KA = " + KA + "\n");

long KB = modPow(E, b, p);
System.out.println("5. Key Bob derives with Eve:");
System.out.println("KB = " + E + "^" + b + " mod " + p);
System.out.println("KB = " + KB + "\n");

long KEA = modPow(A, e, p);
```

```
        System.out.println("6. Keys Eve derives:");
        System.out.println("With Alice (KEA = A^e mod p):");
        System.out.println("KEA = " + A + "^" + e + " mod " + p);
        System.out.println("KEA = " + KEA);

        long KEB = modPow(B, e, p);
        System.out.println("\nWith Bob (KEB = B^e mod p):");
        System.out.println("KEB = " + B + "^" + e + " mod " + p);
        System.out.println("KEB = " + KEB + "\n");

        System.out.println("7. Key Verification:");
        System.out.println("Alice's derived key (KA) = " + KA);
        System.out.println("Bob's derived key (KB) = " + KB);
        System.out.println("Eve's key with Alice (KEA) = " + KEA);
        System.out.println("Eve's key with Bob (KEB) = " + KEB);
        System.out.println("\nResult: " + (KA != KB ? "Alice and Bob have different keys - MITM Attack Successful!" : "Alice and Bob have same key - MITM Attack Failed!"));
    }
}
```

Diffie

```
public class Main {

    public static void main(String[] args) {
        // 1. Hardcoded values for p and g
        long p = 23; // Example prime number
        long g = 5;  // Example generator (primitive root modulo p)

        System.out.println("Hardcoded Public Values:");
        System.out.println("Prime number (p): " + p);
        System.out.println("Generator (g): " + g);
```

```java
        // 2. Alice's Secret Key
        long aliceSecret = 6; // Hardcoded Alice's secret key
        System.out.println("Alice's Secret Key (a): " + aliceSecret);

        // 3. Alice's Public Key
        long alicePublic = modPow(g, aliceSecret, p);
        System.out.println("Alice's Public Key (A): " + alicePublic);

        // 4. Bob's Secret Key
        long bobSecret = 15; // Hardcoded Bob's secret key
        System.out.println("Bob's Secret Key (b): " + bobSecret);

        // 5. Bob's Public Key
        long bobPublic = modPow(g, bobSecret, p);
        System.out.println("Bob's Public Key (B): " + bobPublic);

        // 6. Alice calculates the shared secret key
        long aliceSharedSecret = modPow(bobPublic, aliceSecret, p);
        System.out.println("Alice's calculated shared secret key: " + aliceSharedSecret);

        // 7. Bob calculates the shared secret key
        long bobSharedSecret = modPow(alicePublic, bobSecret, p);
        System.out.println("Bob's calculated shared secret key: " + bobSharedSecret);

        // 8. Verify that the keys are the same
        if (aliceSharedSecret == bobSharedSecret) {
            System.out.println("Shared secret keys match. Key exchange successful!");
        } else {
            System.out.println("Shared secret keys do not match. Key exchange failed.");
        }
    }
```

```
    // Modular Exponentiation without using BigInteger.modPow()
    private static long modPow(long base, long exponent, long modulus) {
        long result = 1;
        base = base % modulus;  // Ensure base is within modulus range

        while (exponent > 0) {
            if (exponent % 2 == 1) {
                result = (result * base) % modulus;
            }
            base = (base * base) % modulus;
            exponent = exponent / 2;
        }

        return result;
    }
}
```

Diffie Text

```
public class Main {

    public static void main(String[] args) {
        // 1. Hardcoded values for p and g
        long p = 23; // Example prime number
        long g = 5;  // Example generator (primitive root modulo p)

        StringBuilder output = new StringBuilder();

        output.append("Hardcoded Public Values:\n");
        output.append("Prime number (p): ").append(p).append("\n");
        output.append("Generator (g): ").append(g).append("\n\n");
```

```
        // 2. Alice's Secret Key
        long aliceSecret = 6; // Hardcoded Alice's secret key
        output.append("Alice's Secret Key (a): ").append(aliceSecret).append("\n\n");

        // 3. Alice's Public Key
        long alicePublic = modPow(g, aliceSecret, p);
        output.append("Alice's Public Key (A): ").append(alicePublic).append("\n\n");

        // 4. Bob's Secret Key
        long bobSecret = 15; // Hardcoded Bob's secret key
        output.append("Bob's Secret Key (b): ").append(bobSecret).append("\n\n");

        // 5. Bob's Public Key
        long bobPublic = modPow(g, bobSecret, p);
        output.append("Bob's Public Key (B): ").append(bobPublic).append("\n\n");

        // 6. Alice calculates the shared secret key
        long aliceSharedSecret = modPow(bobPublic, aliceSecret, p);
        output.append("Alice's calculated shared secret key: 
").append(aliceSharedSecret).append("\n\n");

        // 7. Bob calculates the shared secret key
        long bobSharedSecret = modPow(alicePublic, bobSecret, p);
        output.append("Bob's calculated shared secret key: 
").append(bobSharedSecret).append("\n\n");

        // 8. Verify that the keys are the same
        output.append("Shared secret keys ");
        if (aliceSharedSecret == bobSharedSecret) {
            output.append("match. Key exchange successful!\n");
        } else {
            output.append("do not match. Key exchange failed.\n");
```

```
        }

        // Message Simulation
        String message = "hey";
        output.append("\nMessage to be sent: ").append(message).append("\n");

        output.append("\nCharacter Breakdown:\n");
        for (int i = 0; i < message.length(); i++) {
            char c = message.charAt(i);
            int asciiValue = (int) c;
            output.append("Character: '").append(c).append("', ASCII Value:
").append(asciiValue).append("\n");
        }



        System.out.println(output.toString());
    }

    // Modular Exponentiation without using BigInteger.modPow()
    private static long modPow(long base, long exponent, long modulus) {
        long result = 1;
        base = base % modulus;  // Ensure base is within modulus range

        while (exponent > 0) {
            if (exponent % 2 == 1) {
                result = (result * base) % modulus;
            }
            base = (base * base) % modulus;
            exponent = exponent / 2;
        }
```

```
        return result;
    }
}
```

Diffie MITM Text

```
public class Main {
    static long modPow(long base, long exponent, long modulus) {
        long 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;
    }

    public static void main(String[] args) {
        long p = 23;
        long g = 5;
        long a = 6;
        long b = 15;
        long e = 10;

        long A = modPow(g, a, p);
        long B = modPow(g, b, p);
        long E = modPow(g, e, p);

        long KA = modPow(E, a, p); // Alice's key with Eve
```

```
        long KB = modPow(E, b, p); // Bob's key with Eve
        long KEA = modPow(A, e, p); // Eve's key with Alice
        long KEB = modPow(B, e, p); // Eve's key with Bob

        String originalMessage = "hey";
        StringBuilder output = new StringBuilder();

        output.append("Diffie-Hellman Key Exchange with MITM Attack\n\n");
        output.append("Message Simulation: \"").append(originalMessage).append("\"\n\n");

        // Alice encrypts
        output.append("1. Alice encrypts with KA = ").append(KA).append(":\n");
        int[] aliceEncrypted = new int[originalMessage.length()];
        for (int i = 0; i < originalMessage.length(); i++) {
            aliceEncrypted[i] = originalMessage.charAt(i) ^ (int) KA;
            output.append("  ").append(originalMessage.charAt(i)).append(" (").append((int)
originalMessage.charAt(i)).append(") XOR ").append(KA).append(" =
").append(aliceEncrypted[i]).append("\n");
        }
        output.append("Alice sends:
").append(java.util.Arrays.toString(aliceEncrypted)).append("\n\n");

        // Eve intercepts and decrypts
        output.append("2. Eve intercepts and decrypts with KEA = ").append(KEA).append(":\n");
        StringBuilder eveDecryptedMessage = new StringBuilder();
        for (int i = 0; i < aliceEncrypted.length; i++) {
            char decryptedChar = (char) (aliceEncrypted[i] ^ (int) KEA);
            eveDecryptedMessage.append(decryptedChar);
            output.append("  ").append(aliceEncrypted[i]).append(" XOR ").append(KEA).append(" =
").append((int) decryptedChar).append(" (").append(decryptedChar).append(")\n");
        }
        output.append("Eve decrypts: \"").append(eveDecryptedMessage.toString()).append("\"\n\n");
```

```
        // Eve re-encrypts
        output.append("3. Eve re-encrypts with KB = ").append(KB).append(":\n");
        int[] eveReEncrypted = new int[originalMessage.length()];
        for (int i = 0; i < originalMessage.length(); i++) {
            eveReEncrypted[i] = originalMessage.charAt(i) ^ (int) KB;
            output.append("  ").append(originalMessage.charAt(i)).append(" (").append((int) originalMessage.charAt(i)).append(") XOR ").append(KB).append(" = ").append(eveReEncrypted[i]).append("\n");
        }
        output.append("Eve sends: ").append(java.util.Arrays.toString(eveReEncrypted)).append("\n\n");


        // Bob decrypts
        output.append("4. Bob decrypts with KB = ").append(KB).append(":\n");
        StringBuilder bobDecryptedMessage = new StringBuilder();
        for (int i = 0; i < eveReEncrypted.length; i++) {
            char decryptedChar = (char) (eveReEncrypted[i] ^ (int) KB);
            bobDecryptedMessage.append(decryptedChar);
            output.append("  ").append(eveReEncrypted[i]).append(" XOR ").append(KB).append(" = ").append((int) decryptedChar).append(" (").append(decryptedChar).append(")\n");
        }
        output.append("Bob decrypts: \"").append(bobDecryptedMessage.toString()).append("\"\n\n");


        System.out.println(output.toString());
    }
}
```

Elgamal

```
import java.security.SecureRandom;
import java.util.Random;


public class Main {
```

```
public static void main(String[] args) {
    Random random = new SecureRandom();

    // 1. Hardcoded Public Parameters
    long p = 23;  // Example prime number
    long g = 5;   // Example generator (primitive root modulo p)

    System.out.println("Hardcoded Public Parameters:");
    System.out.println("Prime number (p): " + p);
    System.out.println("Generator (g): " + g);

    // 2. Alice's Private and Public Key Generation
    long x = 4; // Alice's private key (1 < x < p-1) - Hardcoded
    System.out.println("Private key (x): " + x);

    long y = modPow(g, x, p); // Alice's public key
    System.out.println("Public key (y): " + y);

    // 3. Encryption (Bob encrypts a message for Alice)
    long message = 10; // Example message to encrypt (must be less than p) - Hardcoded
    System.out.println("Original Message: " + message);

    long k = 7; // Ephemeral key (0 < k < p-1) - Hardcoded
    System.out.println("Ephemeral key (k): " + k);

    long c1 = modPow(g, k, p);
    long c2 = (message * modPow(y, k, p)) % p;

    System.out.println("Ciphertext (c1, c2): (" + c1 + ", " + c2 + ")");
```

```
        // 4. Decryption (Alice decrypts the message)
        long inverse_c1 = modPow(c1, p - 1 - x, p);
        long decryptedMessage = (c2 * inverse_c1) % p;

        System.out.println("Decrypted message: " + decryptedMessage);
    }



    // Modular exponentiation function (without using BigInteger)
    private static long modPow(long base, long exponent, long modulus) {
        long result = 1;
        base = base % modulus;

        while (exponent > 0) {
            if (exponent % 2 == 1) {
                result = (result * base) % modulus;
            }
            base = (base * base) % modulus;
            exponent /= 2;
        }

        return result;
    }
}
```

SHA512 Inbuilt

```
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
import java.nio.charset.StandardCharsets;

public class Min {
    public static void main(String[] args) {
```

```
        String input = "Hello, SHA-512!";

        try {
            String hash = getSHA512Hash(input);
            System.out.println("Input: " + input);
            System.out.println("SHA-512 Hash: " + hash);
        } catch (NoSuchAlgorithmException e) {
            System.err.println("SHA-512 algorithm not available: " + e.getMessage());
        }
    }

    public static String getSHA512Hash(String input) throws NoSuchAlgorithmException {
        MessageDigest md = MessageDigest.getInstance("SHA-512");
        byte[] hashBytes = md.digest(input.getBytes(StandardCharsets.UTF_8));
        return bytesToHex(hashBytes);
    }

    private static String bytesToHex(byte[] bytes) {
        StringBuilder hexString = new StringBuilder();
        for (byte b : bytes) {
            String hex = Integer.toHexString(0xff & b);
            if (hex.length() == 1) hexString.append('0');
            hexString.append(hex);
        }
        return hexString.toString();
    }
}
```

SHA

```
public class SHA512 {
    // Initial hash values (first 64 bits of fractional parts of square roots of first 8 primes)
    private static final long[] INITIAL_HASH = {
```

```java
    0x6a09e667f3bcc908L, 0xbb67ae8584caa73bL,
    0x3c6ef372fe94f82bL, 0xa54ff53a5f1d36f1L,
    0x510e527fade682d1L, 0x9b05688c2b3e6c1fL,
    0x1f83d9abfb41bd6bL, 0x5be0cd19137e2179L
};

// Round constants (first 64 bits of fractional parts of cube roots of first 80 primes)
private static final long[] K = {
    0x428a2f98d728ae22L, 0x7137449123ef65cdL, 0xb5c0fbcfec4d3b2fL, 0xe9b5dba58189dbbcL,
    0x3956c25bf348b538L, 0x59f111f1b605d019L, 0x923f82a4af194f9bL, 0xab1c5ed5da6d8118L,
    0xd807aa98a3030242L, 0x12835b0145706fbeL, 0x243185be4ee4b28cL, 0x550c7dc3d5ffb4e2L,
    0x72be5d74f27b896fL, 0x80deb1fe3b1696b1L, 0x9bdc06a725c71235L, 0xc19bf174cf692694L,
    0xe49b69c19ef14ad2L, 0xefbe4786384f25e3L, 0x0fc19dc68b8cd5b5L, 0x240ca1cc77ac9c65L,
    0x2de92c6f592b0275L, 0x4a7484aa6ea6e483L, 0x5cb0a9dcbd41fbd4L, 0x76f988da831153b5L,
    0x983e5152ee66dfabL, 0xa831c66d2db43210L, 0xb00327c898fb213fL, 0xbf597fc7beef0ee4L,
    0xc6e00bf33da88fc2L, 0xd5a79147930aa725L, 0x06ca6351e003826fL, 0x142929670a0e6e70L,
    0x27b70a8546d22ffcL, 0x2e1b21385c26c926L, 0x4d2c6dfc5ac42aedL, 0x53380d139d95b3dfL,
    0x650a73548baf63deL, 0x766a0abb3c77b2a8L, 0x81c2c92e47edaee6L, 0x92722c851482353bL,
    0xa2bfe8a14cf10364L, 0xa81a664bbc423001L, 0xc24b8b70d0f89791L, 0xc76c51a30654be30L,
    0xd192e819d6ef5218L, 0xd69906245565a910L, 0xf40e35855771202aL, 0x106aa07032bbd1b8L,
    0x19a4c116b8d2d0c8L, 0x1e376c085141ab53L, 0x2748774cdf8eeb99L, 0x34b0bcb5e19b48a8L,
    0x391c0cb3c5c95a63L, 0x4ed8aa4ae3418acbL, 0x5b9cca4f7763e373L, 0x682e6ff3d6b2b8a3L,
    0x748f82ee5defb2fcL, 0x78a5636f43172f60L, 0x84c87814a1f0ab72L, 0x8cc702081a6439ecL,
    0x90befffa23631e28L, 0xa4506cebde82bde9L, 0xbef9a3f7b2c67915L, 0xc67178f2e372532bL,
    0xca273eceea26619cL, 0xd186b8c721c0c207L, 0xeada7dd6cde0eb1eL, 0xf57d4f7fee6ed178L,
    0x06f067aa72176fbaL, 0x0a637dc5a2c898a6L, 0x113f9804bef90daeL, 0x1b710b35131c471bL,
    0x28db77f523047d84L, 0x32caab7b40c72493L, 0x3c9ebe0a15c9bebcL, 0x431d67c49c100d4cL,
    0x4cc5d4becb3e42b6L, 0x597f299cfc657e2aL, 0x5fcb6fab3ad6faecL, 0x6c44198c4a475817L
};

public static String hash(byte[] input) {
```

```
    // Initial hash values
    long[] hash = INITIAL_HASH.clone();

    // Pre-processing
    byte[] padded = padMessage(input);

    // Process each 1024-bit block
    for (int i = 0; i < padded.length; i += 128) {
        processBlock(padded, i, hash);
    }

    // Convert hash to hex string
    StringBuilder hexString = new StringBuilder();
    for (long h : hash) {
        hexString.append(String.format("%016x", h));
    }
    return hexString.toString();
}

private static byte[] padMessage(byte[] message) {
    int length = message.length;
    long bitLength = (long) length * 8;
    int paddingLength = (int) ((112 - (length % 128) + 128) % 128);
    if (paddingLength < 1) paddingLength += 128;

    byte[] padded = new byte[length + paddingLength + 16];
    System.arraycopy(message, 0, padded, 0, length);
    padded[length] = (byte) 0x80;

    // Add bit length at end (128-bit big-endian)
    for (int i = 0; i < 16; i++) {
```

```
            padded[padded.length - 16 + i] = (byte) (bitLength >>> (120 - i * 8));
        }
        return padded;
    }

    private static void processBlock(byte[] block, int offset, long[] hash) {
        long[] W = new long[80];

        // Convert 128-byte block to 16 64-bit words
        for (int i = 0; i < 16; i++) {
            W[i] = bytesToLong(block, offset + i * 8);
        }

        // Expand to 80 words
        for (int i = 16; i < 80; i++) {
            W[i] = sigma1(W[i - 2]) + W[i - 7] + sigma0(W[i - 15]) + W[i - 16];
        }

        // Initialize working variables
        long a = hash[0], b = hash[1], c = hash[2], d = hash[3];
        long e = hash[4], f = hash[5], g = hash[6], h = hash[7];

        // Compression loop
        for (int i = 0; i < 80; i++) {
            long temp1 = h + bigSigma1(e) + ch(e, f, g) + K[i] + W[i];
            long temp2 = bigSigma0(a) + maj(a, b, c);
            h = g;
            g = f;
            f = e;
            e = d + temp1;
            d = c;
```

```
        c = b;
        b = a;
        a = temp1 + temp2;
    }

    // Update hash values
    hash[0] += a; hash[1] += b; hash[2] += c; hash[3] += d;
    hash[4] += e; hash[5] += f; hash[6] += g; hash[7] += h;
}

// Helper functions for 64-bit operations
private static long bytesToLong(byte[] bytes, int offset) {
    long value = 0;
    for (int i = 0; i < 8; i++) {
        value = (value << 8) | (bytes[offset + i] & 0xff);
    }
    return value;
}

private static long ch(long x, long y, long z) {
    return (x & y) ^ (~x & z);
}

private static long maj(long x, long y, long z) {
    return (x & y) ^ (x & z) ^ (y & z);
}

private static long bigSigma0(long x) {
    return Long.rotateRight(x, 28) ^ Long.rotateRight(x, 34) ^ Long.rotateRight(x, 39);
}
```

```
    private static long bigSigma1(long x) {
        return Long.rotateRight(x, 14) ^ Long.rotateRight(x, 18) ^ Long.rotateRight(x, 41);
    }

    private static long sigma0(long x) {
        return Long.rotateRight(x, 1) ^ Long.rotateRight(x, 8) ^ (x >>> 7);
    }

    private static long sigma1(long x) {
        return Long.rotateRight(x, 19) ^ Long.rotateRight(x, 61) ^ (x >>> 6);
    }

    public static void main(String[] args) {
        String input = "Hello, SHA-512!";
        String hash = hash(input.getBytes());
        System.out.println("SHA-512 hash: " + hash);
    }
}
```

MAC - Size

```
import time
import random
import string

def sha_128(message):
    h0 = 0x67452301
    h1 = 0xEFCDAB89
    h2 = 0x98BADCFE
    h3 = 0x10325476

    ml = len(message)
    message = bytearray(message.encode())
```

```
padding = bytearray()
padding.append(0x80)
pad_len = 64 - ((ml + 1 + 8) % 64)
padding.extend([0] * pad_len)

ml_bits = ml * 8
padding.extend(ml_bits.to_bytes(8, 'big'))
message.extend(padding)

for i in range(0, len(message), 64):
    chunk = message[i:i+64]
    w = [0] * 64

    for j in range(16):
        w[j] = int.from_bytes(chunk[j*4:(j+1)*4], 'big')

    for j in range(16, 64):
        s0 = (w[j-15] >> 7 | w[j-15] << 25) ^ (w[j-15] >> 18 | w[j-15] << 14) ^ (w[j-15] >> 3)
        s1 = (w[j-2] >> 17 | w[j-2] << 15) ^ (w[j-2] >> 19 | w[j-2] << 13) ^ (w[j-2] >> 10)
        w[j] = (w[j-16] + s0 + w[j-7] + s1) & 0xFFFFFFFF

    a, b, c, d = h0, h1, h2, h3

    for j in range(64):
        if j < 16:
            f = (b & c) | ((~b) & d)
            k = 0x5A827999
        elif j < 32:
            f = b ^ c ^ d
            k = 0x6ED9EBA1
```

```
        elif j < 48:
            f = (b & c) | (b & d) | (c & d)
            k = 0x8F1BBCDC
        else:
            f = b ^ c ^ d
            k = 0xCA62C1D6

        temp = ((a << 5) | (a >> 27)) + f + k + w[j]
        e = d
        d = c
        c = ((b << 30) | (b >> 2))
        b = a
        a = temp & 0xFFFFFFFF

    h0 = (h0 + a) & 0xFFFFFFFF
    h1 = (h1 + b) & 0xFFFFFFFF
    h2 = (h2 + c) & 0xFFFFFFFF
    h3 = (h3 + d) & 0xFFFFFFFF

  return '%08x%08x%08x%08x' % (h0, h1, h2, h3)

def sha_256(message):
  h0 = 0x6a09e667
  h1 = 0xbb67ae85
  h2 = 0x3c6ef372
  h3 = 0xa54ff53a
  h4 = 0x510e527f
  h5 = 0x9b05688c
  h6 = 0x1f83d9ab
  h7 = 0x5be0cd19
```

```
k = [
    0x428a2f98, 0x71374491, 0xb5c0fbcf, 0xe9b5dba5,
    0x3956c25b, 0x59f111f1, 0x923f82a4, 0xab1c5ed5,
    0xd807aa98, 0x12835b01, 0x243185be, 0x550c7dc3,
    0x72be5d74, 0x80deb1fe, 0x9bdc06a7, 0xc19bf174,
    0xe49b69c1, 0xefbe4786, 0x0fc19dc6, 0x240ca1cc,
    0x2de92c6f, 0x4a7484aa, 0x5cb0a9dc, 0x76f988da,
    0x983e5152, 0xa831c66d, 0xb00327c8, 0xbf597fc7,
    0xc6e00bf3, 0xd5a79147, 0x06ca6351, 0x14292967
]

message = bytearray(message.encode())
ml = len(message) * 8

message.append(0x80)
while (len(message) + 8) % 64 != 0:
    message.append(0x00)

message += ml.to_bytes(8, 'big')

for i in range(0, len(message), 64):
    chunk = message[i:i+64]
    w = [0] * 64

    for j in range(16):
        w[j] = int.from_bytes(chunk[j*4:(j+1)*4], 'big')

    for j in range(16, 64):
        s0 = ((w[j-15] >> 7) | (w[j-15] << 25)) ^ ((w[j-15] >> 18) | (w[j-15] << 14)) ^ (w[j-15] >> 3)
        s1 = ((w[j-2] >> 17) | (w[j-2] << 15)) ^ ((w[j-2] >> 19) | (w[j-2] << 13)) ^ (w[j-2] >> 10)
        w[j] = (w[j-16] + s0 + w[j-7] + s1) & 0xffffffff
```

```
    a, b, c, d, e, f, g, h = h0, h1, h2, h3, h4, h5, h6, h7

    for j in range(64):
        S1 = ((e >> 6) | (e << 26)) ^ ((e >> 11) | (e << 21)) ^ ((e >> 25) | (e << 7))
        ch = (e & f) ^ ((~e) & g)
        temp1 = h + S1 + ch + k[j % 32] + w[j]
        S0 = ((a >> 2) | (a << 30)) ^ ((a >> 13) | (a << 19)) ^ ((a >> 22) | (a << 10))
        maj = (a & b) ^ (a & c) ^ (b & c)
        temp2 = S0 + maj

        h = g
        g = f
        f = e
        e = (d + temp1) & 0xffffffff
        d = c
        c = b
        b = a
        a = (temp1 + temp2) & 0xffffffff

    h0 = (h0 + a) & 0xffffffff
    h1 = (h1 + b) & 0xffffffff
    h2 = (h2 + c) & 0xffffffff
    h3 = (h3 + d) & 0xffffffff
    h4 = (h4 + e) & 0xffffffff
    h5 = (h5 + f) & 0xffffffff
    h6 = (h6 + g) & 0xffffffff
    h7 = (h7 + h) & 0xffffffff

return '%08x%08x%08x%08x%08x%08x%08x%08x' % (h0, h1, h2, h3, h4, h5, h6, h7)
```

```
def generate_random_message(size):
    return ''.join(random.choices(string.ascii_letters + string.digits, k=size))

print("Code submitted by Prakhar Sinha 22BCI0127")
print("\nSHA-128 Analysis")
print("=" * 80)
print("Message Size(bytes) | Execution Time(seconds) | MAC (first 32 characters)")
print("-" * 80)

message_sizes = [100, 1000, 10000, 100000]

for size in message_sizes:
    message = generate_random_message(size)
    start_time = time.time()
    mac_128 = sha_128(message)
    time_128 = time.time() - start_time
    print(f"{size:15} | {time_128:19.6f} | {mac_128[:32]}")

print("\nSHA-256 Analysis")
print("=" * 80)
print("Message Size(bytes) | Execution Time(seconds) | MAC (first 32 characters)")
print("-" * 80)

for size in message_sizes:
    message = generate_random_message(size)
    start_time = time.time()
    mac_256 = sha_256(message)
    time_256 = time.time() - start_time
    print(f"{size:15} | {time_256:19.6f} | {mac_256[:32]}")

print("\nPerformance Comparison")
```

```
print("=" * 80)

print("- SHA-256 generally takes more time due to additional rounds and complexity")

print("- MAC length: SHA-128 produces 32-character MAC, SHA-256 produces 64-character MAC")

print("- As message size increases, the time difference between SHA-128 and SHA-256 becomes more noticeable")
```