**Ass. No: 1 Title: Playfair Cipher**

**Exp. No: Basic A**

**Date: “21/01/25”**

**Problem Statement**

To Implement a Playfair cipher encryption and decryption using “Java”

**Problem Description**

1. **Encryption Process**

**Concept of encryption process:**

* The Playfair cipher encrypts plaintext by substituting letter pairs based on a 5x5 key matrix.
* A keyword is used to create the matrix, ensuring no repeated letters, and J is treated as I.
* Each letter pair in the plaintext is encrypted based on their positions in the matrix:
* Same row → Replace with the letter to the immediate right.
* Same column → Replace with the letter directly below.
* Rectangle rule → Replace with letters in the same row but opposite corners.

**Key steps:**

* Prepare a 5x5 key matrix using the keyword.
* Preprocess plaintext: Remove spaces, replace J with I, and add X if needed to separate duplicate letters or make the plaintext length even.
* Encrypt letter pairs according to the Playfair cipher rules.

1. **Decryption Process**

**Concept of decryption process:**

* **The Playfair cipher decryption reverses the encryption rules to retrieve the original plaintext.**
* **Decrypt letter pairs based on their positions in the matrix:**
* **Same row → Replace with the letter to the immediate left.**
* **Same column → Replace with the letter directly above.**
* **Rectangle rule → Replace with letters in the same row but opposite corners.**
* **Remove any padding X added during encryption unless it was part of the original plaintext.**

**Key steps:**

* **Use the same key matrix from encryption.**
* **Process ciphertext pairs using the decryption rules to obtain the original plaintext.**

**Algorithm:**

1. **Encryption Process**
2. Create a 5x5 matrix using the encryption key:

* Remove duplicate letters from the key.
* Replace J with I.
* Fill the matrix with the key letters and then unused alphabet letters (A-Z excluding J).

1. Preprocess the plaintext:

* Convert to uppercase, remove spaces, and replace J with I.
* Divide the plaintext into pairs of letters.
* Add X between duplicate letters in a pair and at the end if the length is odd.

1. Encrypt each letter pair:

* If both letters are in the same row, replace each with the letter to its right (wrap around if at the end).
* If both letters are in the same column, replace each with the letter below it (wrap around if at the bottom).
* If the letters form a rectangle, replace them with the letters in the same row but at opposite corners.

1. Combine the encrypted pairs to form the ciphertext.
2. **Decryption Process**
3. Use the same 5x5 matrix from encryption.
4. Divide the ciphertext into pairs of letters.
5. Decrypt each letter pair:

* If both letters are in the same row, replace each with the letter to its left (wrap around if at the start).
* If both letters are in the same column, replace each with the letter above it (wrap around if at the top).
* If the letters form a rectangle, replace them with the letters in the same row but at opposite corners.

1. Remove any padding X added during encryption (if not part of the original message).
2. Combine the decrypted pairs to retrieve the plaintext.

**Program:**

**import java.util.Scanner;**

**public class PlayfairCipher {**

**private static char[][] createMatrix(String key) {**

**char[][] matrix = new char[5][5];**

**boolean[] used = new boolean[26];**

**used['J' - 'A'] = true; // Treat J as I**

**key = key.toUpperCase().replaceAll("[^A-Z]", "").replace("J", "I");**

**int index = 0;**

**for (char c : key.toCharArray()) {**

**if (!used[c - 'A']) {**

**matrix[index / 5][index % 5] = c;**

**used[c - 'A'] = true;**

**index++;**

**}**

**}**

**for (char c = 'A'; c <= 'Z'; c++) {**

**if (!used[c - 'A']) {**

**matrix[index / 5][index % 5] = c;**

**index++;**

**}**

**}**

**return matrix;**

**}**

**private static String processText(String text) {**

**text = text.toUpperCase().replaceAll("[^A-Z]", "").replace("J", "I");**

**StringBuilder processed = new StringBuilder();**

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

**processed.append(text.charAt(i));**

**if (i < text.length() - 1 && text.charAt(i) == text.charAt(i + 1)) {**

**processed.append('X');**

**}**

**}**

**if (processed.length() % 2 != 0) {**

**processed.append('X');**

**}**

**return processed.toString();**

**}**

**private static String encrypt(String plaintext, char[][] matrix) {**

**plaintext = processText(plaintext);**

**StringBuilder ciphertext = new StringBuilder();**

**for (int i = 0; i < plaintext.length(); i += 2) {**

**char a = plaintext.charAt(i);**

**char b = plaintext.charAt(i + 1);**

**int[] posA = findPosition(a, matrix);**

**int[] posB = findPosition(b, matrix);**

**if (posA[0] == posB[0]) {**

**ciphertext.append(matrix[posA[0]][(posA[1] + 1) % 5]);**

**ciphertext.append(matrix[posB[0]][(posB[1] + 1) % 5]);**

**} else if (posA[1] == posB[1]) {**

**ciphertext.append(matrix[(posA[0] + 1) % 5][posA[1]]);**

**ciphertext.append(matrix[(posB[0] + 1) % 5][posB[1]]);**

**} else {**

**ciphertext.append(matrix[posA[0]][posB[1]]);**

**ciphertext.append(matrix[posB[0]][posA[1]]);**

**}**

**}**

**return ciphertext.toString();**

**}**

**private static int[] findPosition(char c, char[][] matrix) {**

**for (int i = 0; i < 5; i++) {**

**for (int j = 0; j < 5; j++) {**

**if (matrix[i][j] == c) {**

**return new int[]{i, j};**

**}**

**}**

**}**

**return null;**

**}**

**public static void main(String[] args) {**

**Scanner scanner = new Scanner(System.in);**

**System.out.print("Enter the key: ");**

**String key = scanner.nextLine();**

**System.out.print("Enter the plaintext: ");**

**String plaintext = scanner.nextLine();**

**char[][] matrix = createMatrix(key);**

**String ciphertext = encrypt(plaintext, matrix);**

**System.out.println("Plaintext: " + plaintext);**

**System.out.println("Ciphertext: " + ciphertext);**

**scanner.close();**

**}**

**}**

**Output:**

Key: MONARCHY

Plaintext: HELLO

Ciphertext: KFMJN

**Example 1**

**Plaintext: WORLD**

**Ciphertext: YBMAQ**

**Example 2**

**Plaintext: MEETING**

**Ciphertext: CPESBU**

**Example 3**

**Plaintext: SECRETMESSAGE**

**Ciphertext: UBEFBUUPTCBII**

**Example 4**

**Plaintext: ATTACKATDAWN**

**Ciphertext: CPXCBCPXFZNB**

**Example 5**

**Plaintext: INFORMATION**

**Ciphertext: GPCBRFUBAKBP**

**Example 6**

**Plaintext: DEFENSE**

**Ciphertext: GCRBGP**

**Example 7**

**Plaintext: PASSWORD**

**Ciphertext: CSBMBODQ**

**Ass. No: 1 Title: Hill Cipher**

**Exp. No: Basic B**

**Problem Statement**

To implement a hill cipher encryption and decryption using “Java”

**Problem Description**

1. **Encryption Process**

**Concept of encryption process:**

* The Hill Cipher encrypts plaintext by treating it as a matrix and multiplying it with a key matrix modulo 26.
* A key matrix of size NxN (e.g., 2x2 or 3x3) is used, and the plaintext is divided into chunks of size N.
* Each plaintext chunk is treated as a vector, multiplied by the key matrix, and the resulting vector is converted to ciphertext using modulo 26.

**Key steps:**

* Create the key matrix (NxN) using the encryption key.
* Divide the plaintext into chunks of size N. If the length is not a multiple of N, pad with additional letters (e.g., X).
* Encrypt each plaintext chunk by multiplying it with the key matrix modulo 26.
* Convert the resulting numbers back into letters to form the ciphertext.

1. **Decryption Process**

**Concept of decryption process:**

* **The decryption process reverses the encryption by using the inverse of the key matrix modulo 26.**
* **The ciphertext is divided into chunks of size N and treated as vectors.**
* **Each ciphertext vector is multiplied by the inverse key matrix modulo 26 to recover the original plaintext.**

**Key steps:**

* **Calculate the modular inverse of the key matrix (requires the determinant to be non-zero modulo 26).**
* **Divide the ciphertext into chunks of size N.**
* **Decrypt each ciphertext chunk by multiplying it with the inverse key matrix modulo 26.**
* **Convert the resulting numbers back into letters to recover the plaintext.**

**Algorithm:**

1. **Encryption Process**
2. Create an NxN key matrix using the encryption key:

* Convert each letter of the key into its numerical value (A = 0, B = 1, …, Z = 25).
* Arrange these values into a square matrix (NxN).

1. Preprocess the plaintext:

* Convert plaintext to uppercase, remove spaces, and divide it into chunks of size N.
* If the length is not a multiple of N, pad the plaintext with additional letters (e.g., X).

1. Encrypt each plaintext chunk:

* Represent each chunk as a column vector of numerical values.
* Multiply the key matrix by the plaintext vector.
* Take the modulo 26 of each resulting value.

1. Convert the resulting numerical values back into letters to form the ciphertext.
2. **Decryption Process**
3. Calculate the modular inverse of the key matrix:

* Find the determinant of the key matrix modulo 26.
* Compute the modular inverse of the determinant modulo 26.
* Use the modular inverse of the determinant to compute the inverse key matrix modulo 26.

1. Divide the ciphertext into chunks of size N.
2. Decrypt each ciphertext chunk:

* Represent each chunk as a column vector of numerical values.
* Multiply the inverse key matrix by the ciphertext vector.
* Take the modulo 26 of each resulting value.

1. Convert the resulting numerical values back into letters to recover the plaintext.

**Program:**

**import java.util.Scanner;**

**public class HillCipher {**

**private static int[][] createKeyMatrix(String key, int size) {**

**int[][] keyMatrix = new int[size][size];**

**int index = 0;**

**for (int i = 0; i < size; i++) {**

**for (int j = 0; j < size; j++) {**

**keyMatrix[i][j] = key.charAt(index) - 'A';**

**index++;**

**}**

**}**

**return keyMatrix;**

**}**

**private static int[] createTextMatrix(String text, int size) {**

**int[] textMatrix = new int[size];**

**for (int i = 0; i < size; i++) {**

**textMatrix[i] = text.charAt(i) - 'A';**

**}**

**return textMatrix;**

**}**

**private static int[] multiplyMatrix(int[][] keyMatrix, int[] textMatrix, int size) {**

**int[] result = new int[size];**

**for (int i = 0; i < size; i++) {**

**for (int j = 0; j < size; j++) {**

**result[i] += keyMatrix[i][j] \* textMatrix[j];**

**}**

**result[i] %= 26;**

**}**

**return result;**

**}**

**private static String matrixToString(int[] matrix) {**

**StringBuilder result = new StringBuilder();**

**for (int value : matrix) {**

**result.append((char) (value + 'A'));**

**}**

**return result.toString();**

**}**

**public static void main(String[] args) {**

**Scanner scanner = new Scanner(System.in);**

**System.out.print("Enter the key (9 characters for 3x3 matrix): ");**

**String key = scanner.nextLine();**

**System.out.print("Enter the plaintext (3 characters): ");**

**String plaintext = scanner.nextLine();**

**int size = 3;**

**int[][] keyMatrix = createKeyMatrix(key, size);**

**int[] textMatrix = createTextMatrix(plaintext, size);**

**int[] ciphertextMatrix = multiplyMatrix(keyMatrix, textMatrix, size);**

**String ciphertext = matrixToString(ciphertextMatrix);**

**System.out.println("Plaintext: " + plaintext);**

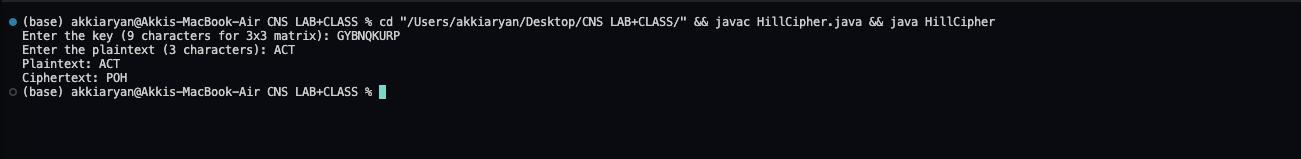
**System.out.println("Ciphertext: " + ciphertext);**

**scanner.close();**

**}**

**}**

**Output:**

**Ass. No: 1 Title: Data Encryption Standard (DES)**

**Exp. No: 1**

**Problem Statement:**

To find the output of the initial permutation box, Final permutation box and S-box using “Java”

1. **Initial permutation box and Final Permutation Box**

**Problem Description**

**Concept of Initial Permutation (IP):**

* The Initial Permutation (IP) is the first step in the DES algorithm.
* It rearranges the 64-bit plaintext according to a predefined permutation table.
* This step is performed to provide confusion and ensure the plaintext is sufficiently scrambled before encryption.

**Concept of Final Permutation (FP):**

* The Final Permutation (FP) is the last step in DES.
* It reverses the Initial Permutation, rearranging the encrypted 64-bit data back to its final format.
* It is used after all 16 rounds of DES encryption/decryption.

**Key Points:**

* Both IP and FP use fixed permutation tables.
* These tables map each input bit to a new position in the output.

**Algorithm:**

**Initial Permutation (IP):**

1. **Input**: A 64-bit binary string (plaintext).
2. Apply the **Initial Permutation (IP)** table:

* Use the predefined table to rearrange the bits.

1. **Output**: A permuted 64-bit binary string.

**Final Permutation (FP):**

1. **Input**: A 64-bit binary string (after DES processing).
2. Apply the **Final Permutation (FP)** table:
   * Use the predefined table to rearrange the bits back to their final format.
3. **Output**: A permuted 64-bit binary string.

**Program:**

**import java.util.Scanner;**

**public class DESPermutation {**

**private static final int[] IP = {**

**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**

**};**

**private static final int[] FP = {**

**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**

**};**

**private static String permute(String input, int[] table) {**

**StringBuilder output = new StringBuilder();**

**for (int position : table) {**

**output.append(input.charAt(position - 1)); // Positions in the table are 1-based**

**}**

**return output.toString();**

**}**

**public static void main(String[] args) {**

**Scanner scanner = new Scanner(System.in);**

**System.out.print("Enter a 64-bit binary string: ");**

**String input = scanner.nextLine();**

**if (input.length() != 64 || !input.matches("[01]+")) {**

**System.out.println("Invalid input. Please enter exactly 64 binary digits (0s and 1s).");**

**return;**

**}**

**String initialPermutation = permute(input, IP);**

**String finalPermutation = permute(initialPermutation, FP);**

**// Display results**

**System.out.println("Input (64-bit): " + input);**

**System.out.println("After Initial Permutation: " + initialPermutation);**

**System.out.println("After Final Permutation: " + finalPermutation);**

**scanner.close();**

**}**

**}**

**Output:**

**Example 1:**

**Enter a 64-bit binary string: 0001001000110100010101101010101111001101000100110011010001110111**

**Input (64-bit): 0001001000110100010101101010101111001101000100110011010001110111**

**After Initial Permutation: 1100110000000000110011001111111111110000101010101111000010101010**

**After Final Permutation: 0001001000110100010101101010101111001101000100110011010001110111**

**Example 2:**

**Enter a 64-bit binary string: 1111111100000000111111110000000011111111000000001111111100000000**

**Input (64-bit): 1111111100000000111111110000000011111111000000001111111100000000**

**After Initial Permutation: 0000000011111111111111110000000000000000111111111111111100000000**

**After Final Permutation: 1111111100000000111111110000000011111111000000001111111100000000**

1. **S - Box**

**Problem Description**

**Concept of S-Box**

* **S-Box (Substitution Box)** is a key component in DES for non-linear substitution.
* DES uses **8 different S-Boxes** (S1 to S8), each mapping a 6-bit input to a 4-bit output.
* Each S-Box has a predefined lookup table with 64 entries arranged in a 4x16 grid.
* The **first and last bits** of the 6-bit input determine the row, while the **middle 4 bits** determine the column.
* The output of the S-Box substitution is a 4-bit value, which is combined with the output of other S-Boxes to form a 32-bit block.

**Algorithm:**

1. **Input**:

* A 48-bit binary string is divided into **8 chunks of 6 bits** each.

1. **For Each 6-Bit Chunk**:

* Determine the **row** using the first and last bits of the chunk.
* Determine the **column** using the middle 4 bits of the chunk.
* Use the S-Box lookup table to find the 4-bit output based on the row and column.

1. **Combine Outputs**:

* Concatenate the 4-bit outputs of all 8 S-Boxes to form a **32-bit output**.

**Program:**

**import java.util.Scanner;**

**public class DESSBox {**

**// Example S-Boxes (for S1, S5, and S8 only)**

**private static final int[][] S1 = {**

**{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}**

**};**

**private static final int[][] S5 = {**

**{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}**

**};**

**private static final int[][] S8 = {**

**{13, 1, 2, 15, 8, 13, 4, 10, 11, 7, 6, 0, 14, 9, 3, 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, 6, 9, 0, 12, 11, 7, 13, 15, 1, 3, 14, 5, 2, 8, 4}**

**};**

**public static String applySBox(String input, int[][] sBox) {**

**StringBuilder output = new StringBuilder();**

**for (int i = 0; i < input.length(); i += 6) {**

**String chunk = input.substring(i, i + 6);**

**// Row: First and Last bit**

**int row = Integer.parseInt("" + chunk.charAt(0) + chunk.charAt(5), 2);**

**// Column: Middle 4 bits**

**int col = Integer.parseInt(chunk.substring(1, 5), 2);**

**// Lookup S-Box value**

**int value = sBox[row][col];**

**output.append(String.format("%4s", Integer.toBinaryString(value)).replace(' ', '0'));**

**}**

**return output.toString();**

**}**

**public static void main(String[] args) {**

**Scanner scanner = new Scanner(System.in);**

**// Input the 6-bit binary strings**

**System.out.print("Enter a 6-bit binary string for S1: ");**

**String s1Input = scanner.nextLine();**

**System.out.print("Enter a 6-bit binary string for S5: ");**

**String s5Input = scanner.nextLine();**

**System.out.print("Enter a 6-bit binary string for S8: ");**

**String s8Input = scanner.nextLine();**

**// Validate input**

**if (s1Input.length() != 6 || !s1Input.matches("[01]+") ||**

**s5Input.length() != 6 || !s5Input.matches("[01]+") ||**

**s8Input.length() != 6 || !s8Input.matches("[01]+")) {**

**System.out.println("Invalid input. Each input must be exactly 6 binary digits (0s and 1s).");**

**return;**

**}**

**String s1Output = applySBox(s1Input, S1);**

**String s5Output = applySBox(s5Input, S5);**

**String s8Output = applySBox(s8Input, S8);**

**System.out.println("S1 Output: " + s1Output);**

**System.out.println("S5 Output: " + s5Output);**

**System.out.println("S8 Output: " + s8Output);**

**scanner.close();**

**}**

**}**

**Output:**

**Example 1 (S-Box 1):**

**Enter a 6-bit binary string for S1: 011000**

**Enter a 6-bit binary string for S5: 110101**

**Enter a 6-bit binary string for S8: 101111**

**S1 Output: 0101**

**S5 Output: 1001**

**S8 Output: 1110**

**Example 2 (S-Box 5):**

**Enter a 6-bit binary string for S1: 001111**

**Enter a 6-bit binary string for S5: 111000**

**Enter a 6-bit binary string for S8: 100101**

**S1 Output: 0010**

**S5 Output: 0111**

**S8 Output: 1011**

**Ass. No: 1 Title: Advanced Encryption Standard (AES)-128**

**Exp. No: 2**

**Problem Statement:**

To find the output of the Mix column operation and generate the Key for Round1, Round2 and Round3 from the Master key using “Java”

1. **Mix column Operation**

**Problem Description**

**Concept of Mix Column Operation**:

* The **Mix Column** operation is a transformation applied to each column of the state matrix in AES.
* Each column of the 4x4 state matrix is treated as a polynomial over GF(2⁸) and multiplied with a fixed matrix:

[02 03 01 01]

[01 02 03 01]

[01 01 02 03]

[03 01 01 02]

* This step provides diffusion by combining the values in each column.

**Algorithm:**

1. **Input**:

* A 4x4 state matrix after the Shift Row operation.
* The fixed Mix Column matrix used for transformation.

1. **For Each Column**:

* Multiply the column vector with the fixed Mix Column matrix.
* Perform modulo reduction over GF(2⁸) using the irreducible polynomial x⁸ + x⁴ + x³ + x + 1.

1. **Output**:

* A new 4x4 state matrix after the Mix Column transformation.

**Program:**

**import java.util.Scanner;**

**public class AESMixColumns {**

**private static final int[][] MIX\_COLUMN\_MATRIX = {**

**{0x02, 0x03, 0x01, 0x01},**

**{0x01, 0x02, 0x03, 0x01},**

**{0x01, 0x01, 0x02, 0x03},**

**{0x03, 0x01, 0x01, 0x02}**

**};**

**private static int galoisMultiply(int a, int b) {**

**int result = 0;**

**while (b > 0) {**

**if ((b & 1) == 1) {**

**result ^= a;**

**}**

**a = (a << 1);**

**if ((a & 0x100) != 0) {**

**a ^= 0x1b; // Irreducible polynomial**

**}**

**b >>= 1;**

**}**

**return result & 0xFF; // Ensure result is 8-bit**

**}**

**private static int[][] mixColumns(int[][] state) {**

**int[][] result = new int[4][4];**

**for (int col = 0; col < 4; col++) {**

**for (int row = 0; row < 4; row++) {**

**result[row][col] = galoisMultiply(MIX\_COLUMN\_MATRIX[row][0], state[0][col])**

**^ galoisMultiply(MIX\_COLUMN\_MATRIX[row][1], state[1][col])**

**^ galoisMultiply(MIX\_COLUMN\_MATRIX[row][2], state[2][col])**

**^ galoisMultiply(MIX\_COLUMN\_MATRIX[row][3], state[3][col]);**

**}**

**}**

**return result;**

**}**

**public static void main(String[] args) {**

**Scanner scanner = new Scanner(System.in);**

**// Input the 4x4 state matrix**

**int[][] state = new int[4][4];**

**System.out.println("Enter the 4x4 state matrix (in hexadecimal): ");**

**for (int i = 0; i < 4; i++) {**

**for (int j = 0; j < 4; j++) {**

**state[i][j] = Integer.parseInt(scanner.next(), 16);**

**}**

**}**

**int[][] mixedState = mixColumns(state);**

**// Display the result**

**System.out.println("After Mix Column Transformation:");**

**for (int[] row : mixedState) {**

**for (int val : row) {**

**System.out.printf("%02X ", val);**

**}**

**System.out.println();**

**}**

**scanner.close();**

**}**

**}**

**Output:**

****

**Enter the 4x4 state matrix (in hexadecimal):**

**19 A0 9A E9**

**3D F4 C6 F8**

**E3 E2 8D 48**

**BE 2B 2A 08  
  
After Mix Column Transformation:**

**D4 E0 B8 1E**

**BF B4 41 27**

**5D 52 11 98**

**30 AE F1 E5**

1. **Key Expansion Algorithm**

**Problem Description**

**Concept of Key Expansion**:

* The **Key Expansion Algorithm** generates round keys from the master key for all rounds of AES encryption.
* AES-128 requires **11 round keys** (1 master key + 10 round keys), each of size 16 bytes (4x4 matrix).
* Round keys are derived using:
* **Rotation**: The last word is rotated (left circular shift).
* **Substitution**: Each byte is replaced using the AES S-Box.
* **Round Constant (RCON) XOR**: The first byte is XORed with a round constant.

**Algorithm:**

1. **Input**:

* A 4x4 master key matrix (16 bytes).

1. **Generate Round Keys**:

* Divide the master key into four 32-bit words (W0, W1, W2, W3).
* For each subsequent round key:
  + Apply **Key Schedule Core**:
    - Rotate the last word (W[i-1]).
    - Substitute each byte using the AES S-Box.
    - XOR the first byte with the **round constant (RCON)**.
  + Generate the next words:
  + W[i] = W[i-4] XOR KeyScheduleCore(W[i-1])
  + Repeat for W[i+1], W[i+2], and W[i+3].

1. **Output**:

* Round keys for all rounds.

**Program:**

**import java.util.Arrays;**

**public class AESKeyExpansion {**

**private static final int[] RCON = {0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1B, 0x36};**

**private static final int[] S\_BOX = {**

**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**

**};**

**public static byte[] keyExpansion(byte[] masterKey) {**

**byte[] expandedKeys = new byte[176];**

**System.arraycopy(masterKey, 0, expandedKeys, 0, 16);**

**for (int i = 16; i < 176; i += 4) {**

**byte[] temp = Arrays.copyOfRange(expandedKeys, i - 4, i);**

**if (i % 16 == 0) {**

**temp = keyScheduleCore(temp, i / 16 - 1);**

**}**

**for (int j = 0; j < 4; j++) {**

**expandedKeys[i + j] = (byte) (expandedKeys[i - 16 + j] ^ temp[j]);**

**}**

**}**

**return expandedKeys;**

**}**

**private static byte[] keyScheduleCore(byte[] word, int rconIndex) {**

**byte temp = word[0];**

**word[0] = word[1];**

**word[1] = word[2];**

**word[2] = word[3];**

**word[3] = temp;**

**for (int i = 0; i < 4; i++) {**

**word[i] = (byte) (S\_BOX[word[i] & 0xFF]);**

**}**

**word[0] ^= RCON[rconIndex];**

**return word;**

**}**

**public static void main(String[] args) {**

**byte[] masterKey = "1234567890123456".getBytes(); // Example 16-byte key**

**byte[] expandedKeys = keyExpansion(masterKey);**

**System.out.println("Master Key: " + Arrays.toString(masterKey));**

**System.out.println("Expanded Keys (176 bytes): " + Arrays.toString(expandedKeys));**

**}**

**}**

**Output:**

