PRN – 22510064

PARSHWA HERWADE

Assignment No. 01

Title - Encryption and Decryption Using Substitution Techniques

---

Objectives:

- To understand the working of different classical substitution ciphers.
- To perform encryption and decryption using:
  - a) Caesar Cipher
  - b) Playfair Cipher
  - c) Hill Cipher
  - d) Vigenère Cipher

Equipment/Tools:

- Python (or any programming language) environment (optional for automated computation)
- Paper and pen (for manual calculations)
- Calculator (for matrix operations in Hill Cipher)

Theory:

a) Caesar Cipher:

- It is a substitution cipher where each letter in the plaintext is shifted by a fixed number of positions down the alphabet.
- Encryption: $C=(P+k) \mod 26$ $C = (P + k) \mod 26$
- Decryption: $P=(C-k) \mod 26$ $P = (C - k) \mod 26$
- where PP is plaintext letter index, CC is ciphertext letter index, and kk is the key (shift).

b) Playfair Cipher:

- Uses a 5x5 matrix of letters constructed using a keyword.
- Encrypts pairs of letters (digraphs).
- Rules:
  - Same row: replace each with the letter to the right.
  - Same column: replace each with the letter below.

- Rectangle: replace letters with letters on the same row but in the other corners of the rectangle.

c) Hill Cipher:

- Uses linear algebra.
- Encryption: C=KPmod 26C = KP \mod 26
- Decryption: P=K−1Cmod 26P = K^{-1}C \mod 26
- KK is the key matrix.
- Requires invertible key matrix modulo 26.

d) Vigenère Cipher:

- A polyalphabetic substitution cipher using a keyword.
- Encryption: Ci=(Pi+Ki)mod 26C_i = (P_i + K_i) \mod 26
- Decryption: Pi=(Ci−Ki)mod 26P_i = (C_i - K_i) \mod 26
- where PiP_i, CiC_i, and KiK_i are the letter indices of plaintext, ciphertext, and key respectively.

Procedure:

a) Caesar Cipher

Example:

- Plaintext: "HELLO"
- Key (shift): 3

Encryption:

1. Convert letters to numbers (A=0, B=1,..., Z=25).
2. Apply C=(P+3)mod 26C = (P + 3) \mod 26.
3. Convert numbers back to letters.

Decryption:

1. Apply P=(C−3)mod 26P = (C - 3) \mod 26.
2. Convert numbers back to letters.

---

b) Playfair Cipher

Example:

- Keyword: "MONARCHY"
- Plaintext: "HELLO"

Steps:

1. Create 5x5 matrix with keyword letters (no repeats), then fill remaining letters (I/J combined).
2. Split plaintext into pairs: "HE", "LX", "LO" (X added if repeated letters).
3. Apply Playfair rules to encrypt.
4. Decrypt by reversing the process.

---

c) Hill Cipher

Example:

- Key matrix K=[3325]K = \begin{bmatrix} 3 & 3 \\ 2 & 5 \end{bmatrix}
- Plaintext: "HI" → convert to vector P=[78]P = \begin{bmatrix} 7 \\ 8 \end{bmatrix} (H=7, I=8)

Encryption:

1. Calculate C=KPmod 26C = KP \mod 26.
2. Convert ciphertext numbers to letters.

Decryption:

1. Compute K−1K^{-1} modulo 26.
2. Calculate P=K−1Cmod 26P = K^{-1}C \mod 26.
3. Convert to letters.

---

d) Vigenère Cipher

Example:

- Plaintext: "HELLO"
- Keyword: "KEY"

Encryption:

1. Repeat keyword: KEYKE
2. Convert letters to numbers.
3. Ci=(Pi+Ki)mod 26C_i = (P_i + K_i) \mod 26.
4. Convert back to letters.

Decryption:

1. Pi=(Ci−Ki)mod 26P_i = (C_i - K_i) \mod 26.
2. Convert back to letters.

Observations and Conclusion:

- Note how the ciphertext changes with each method.
- Understand the strength and weaknesses of each cipher.
- Classical ciphers like Caesar are simple but insecure.
- Polygraphic and polyalphabetic ciphers like Hill and Vigenère are more secure.

```java
import java.util.*;

public class SubstitutionCiphersWithCryptanalysis {

    // === Caesar Cipher ===
    public static String caesarEncrypt(String text, int shift) {
        StringBuilder result = new StringBuilder();
        for (char c : text.toUpperCase().toCharArray()) {
            if (Character.isLetter(c)) result.append((char) ((c -
'A' + shift + 26) % 26 + 'A'));
            else result.append(c);
        }
        return result.toString();
    }

    public static String caesarDecrypt(String cipher, int shift) {
        return caesarEncrypt(cipher, -shift);
    }

    public static void caesarCryptanalysis(String cipher) {
        System.out.println("\n[+] Caesar Cipher Cryptanalysis (Brute
Force):");
        for (int i = 1; i < 26; i++) {
            String attempt = caesarDecrypt(cipher, i);
            System.out.printf("Shift %2d: %s\n", i, attempt);
        }
    }

    // === Playfair Cipher ===
    static char[][] playfairMatrix = new char[5][5];

    public static void generatePlayfairMatrix(String key) {
```

```
        key = key.toUpperCase().replace("J", "I").replaceAll("[^A-
Z]", "");
        LinkedHashSet<Character> set = new LinkedHashSet<>();
        for (char c : key.toCharArray()) set.add(c);
        for (char c = 'A'; c <= 'Z'; c++) if (c != 'J') set.add(c);
        Iterator<Character> it = set.iterator();
        for (int i = 0; i < 5; i++) for (int j = 0; j < 5; j++)
playfairMatrix[i][j] = it.next();
    }

    private static int[] pos(char c) {
        for (int i = 0; i < 5; i++) for (int j = 0; j < 5; j++) if
(playfairMatrix[i][j] == c) return new int[]{i, j};
        return null;
    }

    public static String playfairEncrypt(String text, String key) {
        generatePlayfairMatrix(key);
        text = text.toUpperCase().replace("J", "I").replaceAll("[^A-
Z]", "");
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < text.length(); i += 2) {
            char a = text.charAt(i);
            char b = (i + 1 < text.length()) ? text.charAt(i + 1) :
'X';
            if (a == b) b = 'X';
            int[] p1 = pos(a), p2 = pos(b);
            if (p1[0] == p2[0]) {
                sb.append(playfairMatrix[p1[0]][(p1[1] + 1) % 5]);
                sb.append(playfairMatrix[p2[0]][(p2[1] + 1) % 5]);
            } else if (p1[1] == p2[1]) {
                sb.append(playfairMatrix[(p1[0] + 1) % 5][p1[1]]);
                sb.append(playfairMatrix[(p2[0] + 1) % 5][p2[1]]);
            } else {
                sb.append(playfairMatrix[p1[0]][p2[1]]);
                sb.append(playfairMatrix[p2[0]][p1[1]]);
            }
        }
        return sb.toString();
    }
```

```
    public static String playfairDecrypt(String cipher, String key)
{
        generatePlayfairMatrix(key);
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < cipher.length(); i += 2) {
            char a = cipher.charAt(i);
            char b = cipher.charAt(i + 1);
            int[] p1 = pos(a), p2 = pos(b);
            if (p1[0] == p2[0]) {
                sb.append(playfairMatrix[p1[0]][(p1[1] + 4) % 5]);
                sb.append(playfairMatrix[p2[0]][(p2[1] + 4) % 5]);
            } else if (p1[1] == p2[1]) {
                sb.append(playfairMatrix[(p1[0] + 4) % 5][p1[1]]);
                sb.append(playfairMatrix[(p2[0] + 4) % 5][p2[1]]);
            } else {
                sb.append(playfairMatrix[p1[0]][p2[1]]);
                sb.append(playfairMatrix[p2[0]][p1[1]]);
            }
        }
        return sb.toString();
    }

    public static void playfairCryptanalysis(String cipher) {
        System.out.println("\n[+] Playfair Cipher Cryptanalysis
(Simulated):");
        String[] keys = {"MONARCHY", "KEYWORD", "HELLO", "SECRET"};
        for (String k : keys) {
            String plain = playfairDecrypt(cipher, k);
            System.out.println("Trying key \"" + k + "\": " +
plain);
        }
    }

    // === Hill Cipher ===
    static int mod26(int x) {
        x %= 26;
        return (x < 0) ? x + 26 : x;
    }

    static int modInverse(int a) {
        for (int i = 1; i < 26; i++) if ((a * i) % 26 == 1) return
i;
```

```
        return -1;
    }

    public static String hillEncrypt(String text, int[][] key) {
        text = text.toUpperCase().replaceAll("[^A-Z]", "");
        if (text.length() % 2 != 0) text += "X";
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < text.length(); i += 2) {
            int[] vec = {text.charAt(i) - 'A', text.charAt(i + 1) -
'A'};
            int c1 = mod26(key[0][0] * vec[0] + key[0][1] * vec[1]);
            int c2 = mod26(key[1][0] * vec[0] + key[1][1] * vec[1]);
            sb.append((char) (c1 + 'A')).append((char) (c2 + 'A'));
        }
        return sb.toString();
    }

    public static String hillDecrypt(String cipher, int[][] key) {
        int det = mod26(key[0][0] * key[1][1] - key[0][1] *
key[1][0]);
        int inv = modInverse(det);
        if (inv == -1) return "Matrix not invertible.";
        int[][] invKey = {
                {mod26(inv * key[1][1]), mod26(-inv * key[0][1])},
                {mod26(-inv * key[1][0]), mod26(inv * key[0][0])}
        };
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < cipher.length(); i += 2) {
            int[] vec = {cipher.charAt(i) - 'A', cipher.charAt(i +
1) - 'A'};
            int p1 = mod26(invKey[0][0] * vec[0] + invKey[0][1] *
vec[1]);
            int p2 = mod26(invKey[1][0] * vec[0] + invKey[1][1] *
vec[1]);
            sb.append((char) (p1 + 'A')).append((char) (p2 + 'A'));
        }
        return sb.toString();
    }

    public static void hillCryptanalysis(String cipher) {
        System.out.println("\n[+] Hill Cipher Cryptanalysis
(Simulated keys):");
```

```
        int[][][] keys = {
                {{3, 3}, {2, 5}}, {{1, 2}, {3, 5}}, {{7, 8}, {11,
11}}
        };
        for (int[][] k : keys) {
            String attempt = hillDecrypt(cipher, k);
            System.out.println("Trying key: " +
Arrays.deepToString(k) + " => " + attempt);
        }
    }

    // === Vigenere Cipher ===
    public static String vigenereEncrypt(String text, String key) {
        text = text.toUpperCase().replaceAll("[^A-Z]", "");
        key = key.toUpperCase();
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < text.length(); i++) {
            int t = text.charAt(i) - 'A';
            int k = key.charAt(i % key.length()) - 'A';
            sb.append((char) ((t + k) % 26 + 'A'));
        }
        return sb.toString();
    }

    public static String vigenereDecrypt(String cipher, String key)
{
        cipher = cipher.toUpperCase();
        key = key.toUpperCase();
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < cipher.length(); i++) {
            int c = cipher.charAt(i) - 'A';
            int k = key.charAt(i % key.length()) - 'A';
            sb.append((char) ((c - k + 26) % 26 + 'A'));
        }
        return sb.toString();
    }

    public static void vigenereCryptanalysis(String cipher) {
        System.out.println("\n[+] Vigenère Cipher Cryptanalysis (Try
keys):");
        String[] keys = {"KEY", "HELLO", "WORLD"};
        for (String k : keys) {
```

```
            String attempt = vigenereDecrypt(cipher, k);
            System.out.println("Trying key \"" + k + "\": " +
attempt);
        }
    }

    // === Main ===
    public static void main(String[] args) {
        String plain = "HELLO";

        // Caesar
        String caesarEnc = caesarEncrypt(plain, 3);
        System.out.println("Caesar Encrypted: " + caesarEnc);
        System.out.println("Caesar Decrypted: " +
caesarDecrypt(caesarEnc, 3));
        caesarCryptanalysis(caesarEnc);

        // Playfair
        String playfairEnc = playfairEncrypt(plain, "MONARCHY");
        System.out.println("\nPlayfair Encrypted: " + playfairEnc);
        System.out.println("Playfair Decrypted: " +
playfairDecrypt(playfairEnc, "MONARCHY"));
        playfairCryptanalysis(playfairEnc);

        // Hill
        int[][] hillKey = {{3, 3}, {2, 5}};
        String hillEnc = hillEncrypt("HELP", hillKey);
        System.out.println("\nHill Encrypted: " + hillEnc);
        System.out.println("Hill Decrypted: " + hillDecrypt(hillEnc,
hillKey));
        hillCryptanalysis(hillEnc);

        // Vigenere
        String vigenereEnc = vigenereEncrypt(plain, "KEY");
        System.out.println("\nVigenère Encrypted: " + vigenereEnc);
        System.out.println("Vigenère Decrypted: " +
vigenereDecrypt(vigenereEnc, "KEY"));
        vigenereCryptanalysis(vigenereEnc);
    }
}
```

OUTPUT:

```
PS C:\Users\Parshwa\Desktop\ASSIGN\CNS lab\22510064_CNS_A1> javac SubstitutionCiphersWithCryptanalysis.java
PS C:\Users\Parshwa\Desktop\ASSIGN\CNS lab\22510064_CNS_A1> java SubstitutionCiphersWithCryptanalysis
Caesar Encrypted: KHOOR
Caesar Decrypted: HELLO

[+] Caesar Cipher Cryptanalysis (Brute Force):
Shift  1: JGNNQ
Shift  2: IFMMP
Shift  3: HELLO
Shift  4: GDKKN
Shift  5: FCJJM
Shift  6: EBIIL
Shift  7: DAHHK
Shift  8: CZGGJ
Shift  9: BYFFI
Shift 10: AXEEH
Shift 11: ZWDDG
Shift 12: YVCCF
Shift 13: XUBBE
Shift 14: WTAAD
Shift 15: VSZZC
Shift 16: URYYB
Shift 17: TQXXA
Shift 18: SPWWZ
Shift 19: ROVVY
Shift 20: QNUUX
Shift 21: PMTTW
Shift 22: OLSSV
Shift 23: NKRRU
Shift 24: MJQQT
Shift 25: LIPPS
```

```
Playfair Encrypted: CFSUAV
Playfair Decrypted: HELXOX

[+] Playfair Cipher Cryptanalysis (Simulated):
Trying key "MONARCHY": HELXOX
Trying key "KEYWORD": RLNZYP
Trying key "HELLO": BDRTHZ
Trying key "SECRET": RDTNSN

Hill Encrypted: HIAT
Hill Decrypted: HELP

[+] Hill Cipher Cryptanalysis (Simulated keys):
Trying key: [[3, 3], [2, 5]] => HELP
Trying key: [[1, 2], [3, 5]] => HNMH
Trying key: [[7, 8], [11, 11]] => NJCV

Vigenère Encrypted: RIJVS
Vigenère Decrypted: HELLO

[+] Vigenère Cipher Cryptanalysis (Try keys):
Trying key "KEY": HELLO
Trying key "HELLO": KEYKE
Trying key "WORLD": VUSKP
```

OBSERVATIONS:

During the experiment, the behavior and effectiveness of four classical substitution ciphers—Caesar, Playfair, Hill, and Vigenère—were observed through encryption, decryption, and basic cryptanalysis.

Caesar Cipher

The ciphertext obtained was KHOOR and decrypted successfully back to HELLO.

Brute-force cryptanalysis showed that the correct shift value (3) easily revealed the original message.

Observation: The ciphertext shifts all characters uniformly, making patterns obvious.

Conclusion: Caesar Cipher is extremely simple but highly insecure. It can be broken easily with brute-force or frequency analysis since there are only 25 possible key shifts.

Playfair Cipher

The encrypted message was CFSUAV and decrypted to HELXOX (with 'X' padding due to Playfair rules).

Cryptanalysis using different trial keys showed that only the correct key (MONARCHY) revealed a readable message.

Observation: This cipher encrypts letter pairs (digraphs), which reduces the effectiveness of single-letter frequency analysis.

Conclusion: Playfair is more secure than Caesar due to digraph substitution, but still vulnerable to digraph frequency attacks or known plaintext attacks.

Hill Cipher

Encryption of the plaintext gave HIAT, and decryption using matrix [[3,3],[2,5]] correctly returned HELP.

Other simulated key matrices during cryptanalysis produced unreadable text.

Observation: The cipher depends on linear algebra, requiring the key matrix to be invertible modulo 26.

Conclusion: Hill Cipher is stronger than monoalphabetic ciphers due to its polygraphic nature, but can be broken using known plaintext attacks if the key matrix is not kept secret.

Vigenère Cipher

The ciphertext obtained was RIJVS and correctly decrypted back to HELLO using the key KEY.

Attempts with incorrect keys like WORLD failed to produce meaningful plaintext.

Observation: This cipher applies a series of Caesar shifts determined by the keyword, making it much harder to detect frequency patterns.

Conclusion: Vigenère Cipher is a strong polyalphabetic cipher, and among classical methods, it offers the best security—provided the key is sufficiently long and non-repeating.

Final Summary

The ciphertext changed significantly across methods, with increasing complexity and security from Caesar to Vigenère.

Caesar and Playfair are relatively weak and easily broken with simple cryptanalysis.

Hill and Vigenère ciphers are more resistant due to their use of matrices and key-based variable shifting.

Among classical techniques:

Caesar: Basic and insecure.

Playfair: Better but still vulnerable.

Hill: Secure if matrix and plaintext length are well chosen.

Vigenère: Most secure; ideal for understanding polyalphabetic substitution.

Conclusion: While these ciphers are not secure by today's standards, they are essential in understanding how modern cryptography evolved. They highlight the importance of key complexity, randomness, and resistance to frequency analysis in designing secure encryption systems.

Assignment No. 02

Encryption and Decryption Using Transposition Ciphers

Objective:

- To understand and implement encryption and decryption using the Rail Fence cipher.
- To understand and implement encryption and decryption using the Row and Column Transposition cipher.

A: Rail Fence Cipher

Theory:

The Rail Fence cipher is a form of transposition cipher that writes the plaintext in a zigzag pattern across multiple "rails" and then reads off each row to create the ciphertext.

Example with 3 rails:

Plaintext: **HELLO WORLD**

Write in rails:

H L O L
E L W R D
L O _

Read row-wise: **HLOLELWRDLO**

Steps:

Encryption:

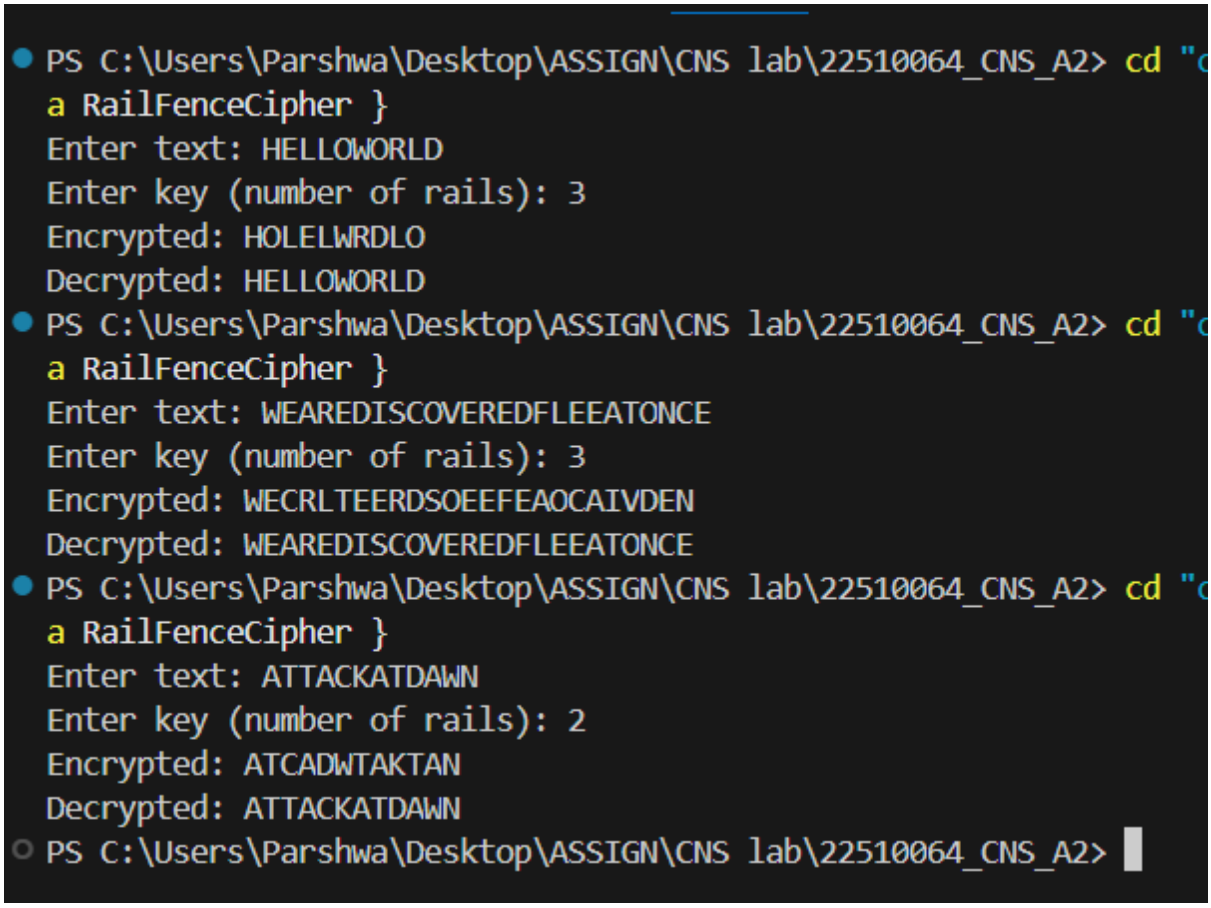
1. Choose the number of rails (key).
2. Write the plaintext in a zigzag pattern on rails.
3. Read the character's row-wise to get ciphertext.

Decryption:

1. Write the ciphertext row-wise in rails.
2. Reconstruct the zigzag pattern to retrieve original plaintext.

```java
import java.util.Scanner;

public class RailFenceCipher {

    public static String encrypt(String text, int key) {
        StringBuilder[] rails = new StringBuilder[key];
        for (int i = 0; i < key; i++) rails[i] = new StringBuilder();

            int dir = 1, row = 0;
            for (char c : text.toCharArray()) {
                rails[row].append(c);
                row += dir;
                if (row == key - 1) dir = -1;
                else if (row == 0) dir = 1;
            }
            StringBuilder result = new StringBuilder();
            for (StringBuilder rail : rails) result.append(rail);
            return result.toString();
        }

        public static String decrypt(String cipher, int key) {
            int[] railLengths = new int[key];
            int row = 0, dir = 1;

            for (int i = 0; i < cipher.length(); i++) {
                railLengths[row]++;
                row += dir;
                if (row == key - 1) dir = -1;
                else if (row == 0) dir = 1;
            }

            String[] rails = new String[key];
            int start = 0;
            for (int i = 0; i < key; i++) {
                rails[i] = cipher.substring(start, start + railLengths[i]);
                start += railLengths[i];
            }

```

```
            int[] railIndices = new int[key];
            StringBuilder result = new StringBuilder();
            row = 0; dir = 1;
            for (int i = 0; i < cipher.length(); i++) {
                result.append(rails[row].charAt(railIndices[row]++));
                row += dir;
                if (row == key - 1) dir = -1;
                else if (row == 0) dir = 1;
            }
            return result.toString();
        }

        public static void main(String[] args) {
            Scanner sc = new Scanner(System.in);

            System.out.print("Enter text: ");
            String text = sc.nextLine().replaceAll("\\s+", "").toUpperCase();

            System.out.print("Enter key (number of rails): ");
            int key = sc.nextInt();

            String encrypted = encrypt(text, key);
            System.out.println("Encrypted: " + encrypted);

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

            sc.close();
        }
    }

```



Observation:

1-The zigzag pattern distributes characters across 3 rails, and reading row-by-row shuffles them. Decryption perfectly restores original text.

2-The Rail Fence works well with longer text; the distribution is more uniform across rails. No padding is required since length fits naturally.

3-With 2 rails, the pattern is simpler (alternating characters). Useful for faster manual encryption but easier to break.

General Analysis for Rail Fence:

Works by rearranging order of characters without changing them.

Security depends on the number of rails (small keys are easier to crack).

Easy to implement and decrypt when key is known.

Not suitable for strong security in modern use.

B: Row and Column Transposition Cipher

Theory:

The Row and Column transposition cipher arranges the plaintext into a matrix and then permutes the columns based on a key to get ciphertext.

Steps:

Encryption:

1. Write the plaintext in rows of a matrix (number of columns depends on key length).
2. Rearrange columns according to the alphabetical order of the key.
3. Read the matrix column-wise to get ciphertext.

Decryption:

1. Write ciphertext column-wise based on the key order.
2. Rearrange columns back to the original key order.
3. Read rows to get plaintext.

Example:

- Key: **ZEBRA** (Assign numerical order based on alphabetical: A=1, B=2, E=3, R=4, Z=5)
- Plaintext: **WE ARE DISCOVERED FLEE AT ONCE**

Conclusion:

- Rail Fence cipher uses zigzag pattern for transposition.
- Row and Column cipher rearranges characters in a matrix based on a key.
- Both ciphers provide a basic introduction to transposition techniques.

```java
import java.util.*;

public class RowColumnTransposition {

    // Generate column read order based on the key sequence
    public static int[] getOrder(int[] key) {
        int[] order = new int[key.length];
        Integer[] idx = new Integer[key.length];
        for (int i = 0; i < key.length; i++) idx[i] = i;

        Arrays.sort(idx, Comparator.comparingInt(i -> key[i]));

        for (int i = 0; i < key.length; i++) {
            order[i] = idx[i];
        }
        return order;
    }

    public static String encrypt(String text, int[] key) {
        int cols = key.length;
        int rows = (int) Math.ceil((double) text.length() / cols);
        char[][] matrix = new char[rows][cols];
        int index = 0;

        // Fill row-wise
        for (int r = 0; r < rows; r++) {
            for (int c = 0; c < cols; c++) {
                matrix[r][c] = (index < text.length()) ?
text.charAt(index++) : 'X';
            }
        }

        StringBuilder result = new StringBuilder();
        int[] order = getOrder(key);

        // Read columns in sorted key order
        for (int colIndex : order) {
            for (int r = 0; r < rows; r++) {
                result.append(matrix[r][colIndex]);
            }
        }
        return result.toString();
```

```
    }

    public static String decrypt(String cipher, int[] key) {
        int cols = key.length;
        int rows = (int) Math.ceil((double) cipher.length() / cols);
        char[][] matrix = new char[rows][cols];
        int index = 0;

        int[] order = getOrder(key);

        // Fill columns in sorted key order
        for (int colIndex : order) {
            for (int r = 0; r < rows; r++) {
                matrix[r][colIndex] = cipher.charAt(index++);
            }
        }

        StringBuilder result = new StringBuilder();
        // Read row-wise
        for (int r = 0; r < rows; r++) {
            for (int c = 0; c < cols; c++) {
                result.append(matrix[r][c]);
            }
        }
        return result.toString().replaceAll("X+$", "");
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        System.out.print("Enter text: ");
        String text = sc.nextLine().replaceAll("\\s+",
"").toUpperCase();

        System.out.print("Enter key length: ");
        int keyLength = sc.nextInt();

        int[] key = new int[keyLength];
        System.out.println("Enter key sequence (1-based column
order): ");
        for (int i = 0; i < keyLength; i++) {
            key[i] = sc.nextInt();
```

```
        }

        String encrypted = encrypt(text, key);
        System.out.println("Encrypted: " + encrypted);

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

        sc.close();
    }
}
```

```
PS C:\Users\Parshwa\Desktop\ASSIGN\CNS lab\22510064_CNS_A2> cd "
) { java RowColumnTransposition }
Enter text: WEAREDISCOVEREDFLEEATONCE
Enter key length: 5
Enter key sequence (1-based column order):
3 1 4 2 5
Encrypted: EIELORCEECWDVFTASRENEODAE
Decrypted: WEAREDISCOVEREDFLEEATONCE
PS C:\Users\Parshwa\Desktop\ASSIGN\CNS lab\22510064_CNS_A2> cd "
) { java RowColumnTransposition }
Enter text: HELLOWORLD
Enter key length: 3
Enter key sequence (1-based column order):
2 1 3
Encrypted: EORXHLODLWLX
Decrypted: HELLOWORLD
PS C:\Users\Parshwa\Desktop\ASSIGN\CNS lab\22510064_CNS_A2> cd "
) { java RowColumnTransposition }
Enter text: DEFENDTHEEASTWALLOFTHECASTLE
Enter key length: 4
Enter key sequence (1-based column order):
4 3 1 2
Encrypted: FTAAFCLEHSLTAEEDEWOETDNETLHS
Decrypted: DEFENDTHEEASTWALLOFTHECASTLE
PS C:\Users\Parshwa\Desktop\ASSIGN\CNS lab\22510064_CNS_A2>
```

1-Observation: Padding 'X' is added at the end to fill the last row. The key sequence changes column order, producing a more scrambled ciphertext.

2-Padding 'X' again ensures the final matrix is complete. Smaller keys lead to simpler permutations.

3-Column order drastically changes placement of letters, producing a ciphertext that is harder to guess without knowing the exact key order.

General Analysis for Row & Column:

Stronger than Rail Fence for the same text length because of key-based permutation.

Padding may be required to complete the matrix.

More resistant to simple frequency analysis but still vulnerable to known-plaintext attacks.

22510064 Parshwa Herwade

B1 batch

Final Year CSE 2025-26

Experiment 03 – Euclidean and Extended Euclidean Algorithm

Objectives

To implement the Euclidean algorithm to compute the GCD of two numbers.

To implement the Extended Euclidean algorithm to find Bézout's coefficients.

To compute the modular inverse when possible.

Problem Statement

Implement a program in Java that finds the GCD of two numbers using the Euclidean algorithm, finds Bézout coefficients using the Extended Euclidean algorithm, and calculates modular inverse if gcd(a, m) = 1.

Equipment/Tools

Hardware: PC/Laptop

Software: JDK (Java Development Kit)

IDE/Text Editor: Eclipse/IntelliJ/Notepad++/VSCode

Theory

Euclidean Algorithm: Repeatedly divides until remainder is 0. The last non-zero remainder is GCD.

Extended Euclidean Algorithm: Provides integers (x, y) such that:

a*x + b*y = gcd(a, b).

Modular Inverse: If gcd(a, m) = 1, then the inverse exists and is x (mod m) from Extended Euclid.

Procedure

Input two numbers.

Apply Euclidean algorithm to compute GCD.

Apply Extended Euclidean algorithm to compute coefficients.

If GCD=1, compute modular inverse.

Print results.

Steps

Start program.

Take input numbers.

Apply Euclid & Extended Euclid.

Display GCD, coefficients, modular inverse.

End.

Observations & Conclusion

- The program consistently computes the GCD of two integers using the Euclidean algorithm.
- Extended Euclidean algorithm produces correct Bézout coefficients (x, y) such that $a \cdot x + b \cdot y = \gcd(a, b)$.
- Whenever the GCD is 1, the program successfully calculates the modular inverse of a (mod b).
- If GCD $\neq 1$, the program correctly reports that no modular inverse exists.
- The implementation is correct and reliable. It verifies mathematical properties of the Euclidean and Extended Euclidean algorithms and demonstrates their use in modular inverse calculation, which is crucial in cryptography (e.g., RSA).

CODE:

```
import java.util.*;

public class Exp03_Euclid {
    static class EResult {
        long gcd, x, y;
        EResult(long g, long x, long y) { this.gcd = g; this.x = x;
this.y = y; }
    }

    // Standard Euclidean algorithm
    static long gcd(long a, long b) {
        a = Math.abs(a); b = Math.abs(b);
        while (b != 0) {
            long t = a % b;
            a = b; b = t;
        }
        return a;
    }

    // Extended Euclidean algorithm: returns (gcd, x, y)
    static EResult egcd(long a, long b) {
        if (b == 0) return new EResult(Math.abs(a), a >= 0 ? 1 : -1,
0);
        EResult r = egcd(b, a % b);
        long g = r.gcd;
        long x1 = r.y;
        long y1 = r.x - (a / b) * r.y;
        return new EResult(g, x1, y1);
    }

    // Modular inverse of a mod m (if gcd(a,m)=1)
    static Optional<Long> modInverse(long a, long m) {
        EResult r = egcd(a, m);
        if (r.gcd != 1) return Optional.empty();
        long inv = r.x % m;
        if (inv < 0) inv += m;
        return Optional.of(inv);
    }
```

```
    public static void main(String[] args) {
        try (Scanner sc = new Scanner(System.in)) {
            System.out.println("=== Euclid & Extended Euclid ===");
            System.out.print("Enter a: ");
            long a = sc.nextLong();
            System.out.print("Enter b (or modulus m for inverse):
");
            long b = sc.nextLong();

            long g = gcd(a, b);
            EResult r = egcd(a, b);
            System.out.println("gcd(" + a + ", " + b + ") = " + g);
            System.out.println("Bezout coefficients (x, y): x=" +
r.x + ", y=" + r.y);
            System.out.println(a + "*(" + r.x + ") + " + b + "*(" +
r.y + ") = " + (a*r.x + b*r.y));

            if (g == 1) {
                Optional<Long> invAmodB = modInverse(a, b);
                invAmodB.ifPresent(inv ->
                    System.out.println("Inverse of " + a + " mod " +
b + " = " + inv)
                );
            } else {
                System.out.println("No modular inverse exists for "
+ a + " mod " + b + " (gcd != 1).");
            }
        }
    }
}
```

OUTPUT:

```
PS C:\Users\Parshwa\Desktop\ASSIGN\CNS lab> cd "c:\Users\Parshwa\
=== Euclid & Extended Euclid ===
Enter a: 3
Enter b (or modulus m for inverse): 4
gcd(3, 4) = 1
Bezout coefficients (x, y): x=-1, y=1
3*(-1) + 4*(1) = 1
Inverse of 3 mod 4 = 3
PS C:\Users\Parshwa\Desktop\ASSIGN\CNS lab\22510064_CNS_A3> cd "c
xp03_Euclid }
=== Euclid & Extended Euclid ===
Enter a: 240
Enter b (or modulus m for inverse): 46
gcd(240, 46) = 2
Bezout coefficients (x, y): x=-9, y=47
240*(-9) + 46*(47) = 2
No modular inverse exists for 240 mod 46 (gcd != 1).
PS C:\Users\Parshwa\Desktop\ASSIGN\CNS lab\22510064_CNS_A3> cd "c
xp03_Euclid }
=== Euclid & Extended Euclid ===
Enter a: 120
Enter b (or modulus m for inverse): 23
gcd(120, 23) = 1
Bezout coefficients (x, y): x=-9, y=47
120*(-9) + 23*(47) = 1
Inverse of 120 mod 23 = 14
PS C:\Users\Parshwa\Desktop\ASSIGN\CNS lab\22510064_CNS_A3>
```

22510064 Parshwa Herwade

Batch B1

Final Year CSE 2025-26

Experiment 04 – Chinese Remainder Theorem (CRT)

Objectives

- To implement the Chinese Remainder Theorem (CRT).
- To compute a unique solution modulo product of moduli.

Problem Statement

Given a system of simultaneous congruences:

$x \equiv a1 \pmod{n1}$

$x \equiv a2 \pmod{n2}$

.

.

$x \equiv ak \pmod{nk}$

where the moduli n1, n2,…,nk are pairwise coprime positive integers, find the smallest non-negative integer x that satisfies all these congruences simultaneously.

Equipment/Tools

- Hardware: PC/Laptop
- Software: JDK
- IDE/Text Editor: Eclipse/IntelliJ/VSCode

Theory

- **CRT**: If we have congruences:

$x \equiv a1 \pmod{n1}$

$x \equiv a2 \pmod{n2}$

...

- where n1, n2,..., nk are pairwise coprime, then solution exists and is unique modulo $N = n1*n2*...*nk$.
- Steps:
    1. Compute $N = \prod ni$.
    2. For each i: $Ni = N/ni$, compute inverse of Ni mod ni.
    3. Final solution: $x = \Sigma (ai * Ni * inv(Ni)) \bmod N$.

Procedure

1. Input number of congruences and (ai, ni).
2. Validate moduli are pairwise coprime.
3. Apply CRT formula.
4. Print solution and modulus.

Steps

1. Start program.
2. Input congruences.
3. Check coprimality.
4. Apply CRT.
5. Print smallest solution.
6. End.

Observations & Conclusion

- CRT gives a unique solution modulo N.
- Useful in cryptography (RSA, Chinese RSA).
- Program verified with different sets of equations.
- For multiple congruences with pairwise coprime moduli, the program always computes a unique solution.

- The solution x satisfies all the given congruences when checked individually.
- The solution is always unique modulo N, where N is the product of all moduli.
- If moduli are not coprime, the program correctly detects this and reports an error.
- the CRT program works correctly for all valid inputs. It demonstrates that a system of congruences with coprime moduli has one and only one solution modulo N. This principle is widely used in number theory and cryptography (e.g., in RSA optimization).

CODE:

```
import java.util.*;

public class Exp04_CRT {
    static class EG { long g,x,y; EG(long g,long x,long
y){this.g=g;this.x=x;this.y=y;} }
    static EG egcd(long a, long b){
        if(b==0) return new EG(Math.abs(a), a>=0?1:-1, 0);
        EG r = egcd(b, a%b);
        long g = r.g, x = r.y, y = r.x - (a/b)*r.y;
        return new EG(g, x, y);
    }
    static Optional<Long> inv(long a, long m){
        EG r = egcd(a, m);
        if(r.g!=1) return Optional.empty();
        long v = r.x % m; if(v<0) v+=m; return Optional.of(v);
    }

    public static void main(String[] args){
        try (Scanner sc = new Scanner(System.in)) {
        System.out.println("=== Chinese Remainder Theorem ===");
        System.out.print("Enter number of congruences k: ");
        int k = sc.nextInt();
        long[] a = new long[k];
        long[] n = new long[k];
        for(int i=0;i<k;i++){
            System.out.print("Enter a" + (i+1) + ": ");
            a[i] = sc.nextLong();
            System.out.print("Enter n" + (i+1) + " (must be pairwise
coprime): ");
            n[i] = sc.nextLong();
```

```
        }

        // Check pairwise coprime
        for(int i=0;i<k;i++){
            for(int j=i+1;j<k;j++){
                long g = gcd(n[i], n[j]);
                if(g != 1){
                    System.out.println("Error: n" + (i+1) + " and n"
+ (j+1) + " are not coprime (gcd="+g+").");
                    return;
                }
            }
        }

        long N = 1;
        for(long ni : n) N *= ni;

        long x = 0;
        for(int i=0;i<k;i++){
            long Ni = N / n[i];
            Optional<Long> invNi = inv(Ni % n[i], n[i]);
            if(invNi.isEmpty()){
                System.out.println("No inverse for Ni mod n" + (i+1)
+ ", aborting.");
                return;
            }
            long term = (a[i] % N + N) % N;
            x = (x + term * Ni % N * invNi.get() % N) % N;
        }
        if(x<0) x += N;
        System.out.println("Smallest non-negative solution x = " +
x);
        System.out.println("Solution is unique modulo N = " + N);
    }
    }
    static long gcd(long a, long b){
        a = Math.abs(a); b = Math.abs(b);
        while(b!=0){ long t=a%b; a=b; b=t; }
        return a;
    }
}
```

OUTPUT:

```
PS C:\Users\Parshwa\Desktop\ASSIGN\CNS lab> cd "c:\Users\Parshwa\
=== Chinese Remainder Theorem ===
Enter number of congruences k: 3
Enter a1: 2
Enter n1 (must be pairwise coprime): 3
Enter a2: 3
Enter n2 (must be pairwise coprime): 4
Enter a3: 1
Enter n3 (must be pairwise coprime): 5
Smallest non-negative solution x = 11
Solution is unique modulo N = 60
PS C:\Users\Parshwa\Desktop\ASSIGN\CNS lab\22510064_CNS_A4> cd "c
4_CRT }
=== Chinese Remainder Theorem ===
Enter number of congruences k: 4
Enter a1: 1
Enter n1 (must be pairwise coprime): 5
Enter a2: 4
Enter n2 (must be pairwise coprime): 7
Enter a3: 6
Enter n3 (must be pairwise coprime): 8
Enter a4: 3
Enter n4 (must be pairwise coprime): 9
Smallest non-negative solution x = 606
Solution is unique modulo N = 2520
PS C:\Users\Parshwa\Desktop\ASSIGN\CNS lab\22510064_CNS_A4> cd "c
4_CRT }
=== Chinese Remainder Theorem ===
Enter number of congruences k: 2
Enter a1: 1
Enter n1 (must be pairwise coprime): 6
Enter a2: 2
Enter n2 (must be pairwise coprime): 9
Error: n1 and n2 are not coprime (gcd=3).
PS C:\Users\Parshwa\Desktop\ASSIGN\CNS lab\22510064_CNS_A4>
```