**Experiment-7**

**Aim:** Study and implement a program for n-gram Hill Cipher.

**Introduction:**

In classical cryptography, the hill cipher is a polygraphic substitution cipher based on Linear Algebra. It was invented by Lester S. Hill in the year 1929. In simple words, it is a cryptography algorithm used to encrypt and decrypt data for the purpose of data security.

The algorithm uses matrix calculations used in Linear Algebra. It is easier to understand if we have the basic knowledge of matrix multiplication, modulo calculation, and the inverse calculation of matrices.

In hill cipher algorithm every letter (A-Z) is represented by a number moduli 26. Usually, the simple substitution scheme is used where A = 0, B = 1, C = 2…Z = 25 to use 2x2 key matrix.

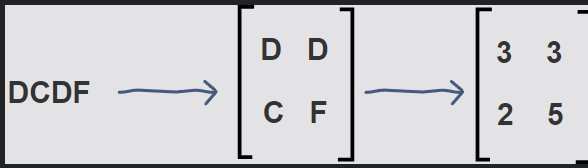
**Encryption:**

Encrypting with the Hill cipher is built on the following operation:

**E(K, P) = (K\*P) mod 26**

Where K is our key matrix and P is the plaintext in vector form. Matrix multiplying these two terms produces the encrypted ciphertext. Let us do so step by step:

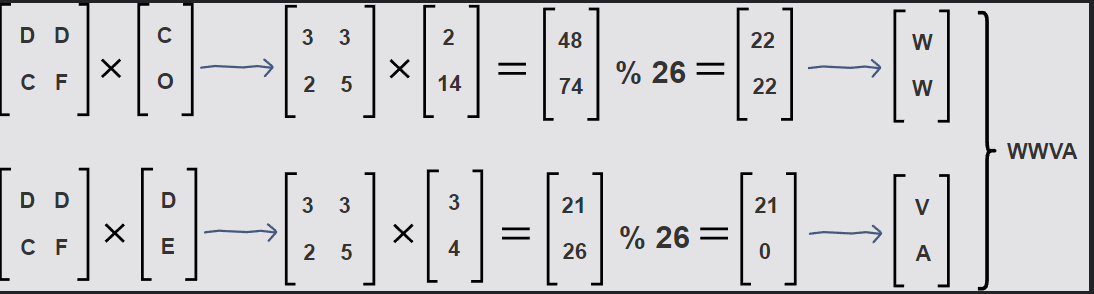
1. Pick a keyword to encrypt your plaintext message. Let us work with the random keyword “DDCF.” Convert this keyword to matrix form using your substitution scheme to convert it to a numerical 2x2 key matrix.



1. Next, we will convert our plaintext message to vector form. Since our key matrix is 2x2, the vector needs to be 2x1 for matrix multiplication to be possible. In our case, our message is four letters long so we can split it into blocks of two and then substitute to get our plaintext vectors.



1. Now, we can matrix multiply the key matrix with each 2x1 plaintext vector, take the moduli of the resulting 2x1 vectors by 26, and concatenate the results to get “WWVA”, the final ciphertext.



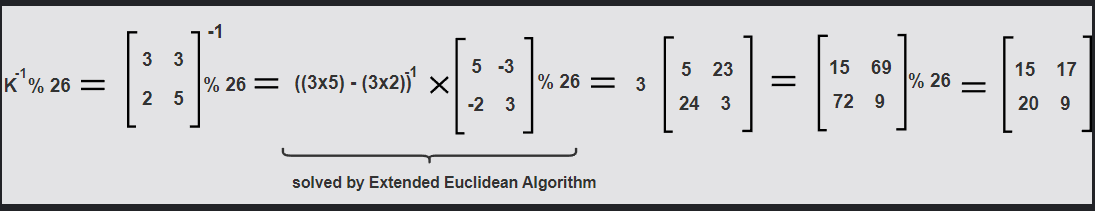
**Decryption:**

Decrypting with the Hill cipher is built on the following operation:

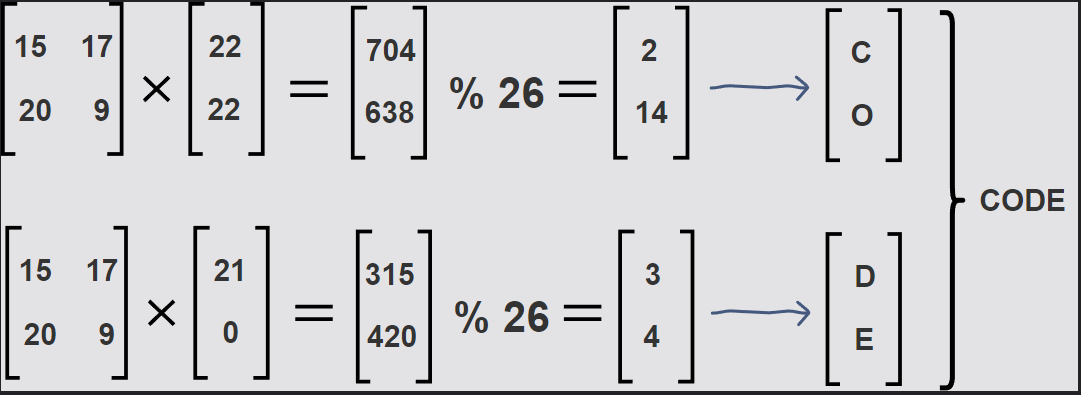
**D(K, C) = (K-1 \*C) mod 26**

Where K is our key matrix and C is the ciphertext in vector form. Matrix multiplying the inverse of the key matrix with the ciphertext produces the decrypted plaintext. Let us do this step by step with our ciphertext, "WWVA":

1. First, we calculate the inverse of the key matrix. In doing so, we must keep​ the result between 0-25 using modulo 26. For this reason, the Extended Euclidean algorithm is used to find the modular multiplicative inverse of the key matrix determinant.



1. Next, we will multiply 2x1 blocks of the ciphertext with the inverse of the key matrix to get our original plaintext message, “CODE,” back.



**Program (Source Code):**

#include <bits/stdc++.h>

using namespace std;

void performEncryption()

{

    int x, y, i, j, k, n;

    cout << "Enter the encryption key: ";

    string key;

    cin >> key;

    n = sqrt(key.length());

    vector<vector<int>> matrix(n, vector<int>(n));

    int iter = 0;

    for (i = 0; i < n; i++)

    {

        for (j = 0; j < n; j++)

        {

            int val = key[iter] - 97;

            matrix[i][j] = val;

            iter++;

        }

    }

    cout << "Key matrix:" << endl;

    for (i = 0; i < n; i++)

    {

        for (j = 0; j < n; j++)

        {

            cout << matrix[i][j] << " ";

        }

        cout << endl;

    }

    cout << "Enter the message to encrypt: ";

    string message;

    cin >> message;

    int padding = (n - message.size() % n) % n;

    for (i = 0; i < padding; i++)

    {

        message += 'x';

    }

    k = 0;

    string encryptedText = "";

    while (k < message.size())

    {

        for (i = 0; i < n; i++)

        {

            int sum = 0;

            int temp = k;

            for (j = 0; j < n; j++)

            {

                sum += (matrix[i][j] % 26 \* (message[temp++] - 'a') % 26) % 26;

                sum = sum % 26;

            }

            encryptedText += (sum + 'a');

        }

        k += n;

    }

    cout << "\nEncrypted text is: " << encryptedText << '\n';

}

int modInverse(int a, int m)

{

    a = a % m;

    for (int x = -m; x < m; x++)

        if ((a \* x) % m == 1)

            return x;

    return 0;

}

void getCofactor(vector<vector<int>> &a, vector<vector<int>> &temp, int p, int q, int n)

{

    int i = 0, j = 0;

    for (int row = 0; row < n; row++)

    {

        for (int col = 0; col < n; col++)

        {

            if (row != p && col != q)

            {

                temp[i][j++] = a[row][col];

                if (j == n - 1)

                {

                    j = 0;

                    i++;

                }

            }

        }

    }

}

int calculateDeterminant(vector<vector<int>> &a, int n, int N)

{

    int D = 0;

    if (n == 1)

        return a[0][0];

    vector<vector<int>> temp(N, vector<int>(N));

    int sign = 1;

    for (int f = 0; f < n; f++)

    {

        getCofactor(a, temp, 0, f, n);

        D += sign \* a[0][f] \* calculateDeterminant(temp, n - 1, N);

        sign = -sign;

    }

    return D;

}

void getAdjoint(vector<vector<int>> &a, vector<vector<int>> &adj, int N)

{

    if (N == 1)

    {

        adj[0][0] = 1;

        return;

    }

    int sign = 1;

    vector<vector<int>> temp(N, vector<int>(N));

    for (int i = 0; i < N; i++)

    {

        for (int j = 0; j < N; j++)

        {

            getCofactor(a, temp, i, j, N);

            sign = ((i + j) % 2 == 0) ? 1 : -1;

            adj[j][i] = (sign) \* (calculateDeterminant(temp, N - 1, N));

        }

    }

}

bool calculateInverse(vector<vector<int>> &a, vector<vector<int>> &inv, int N)

{

    int det = calculateDeterminant(a, N, N);

    if (det == 0)

    {

        cout << "Inverse does not exist";

        return false;

    }

    int invDet = modInverse(det, 26);

    cout << det % 26 << ' ' << invDet << '\n';

    vector<vector<int>> adj(N, vector<int>(N));

    getAdjoint(a, adj, N);

    for (int i = 0; i < N; i++)

        for (int j = 0; j < N; j++)

            inv[i][j] = (adj[i][j] \* invDet) % 26;

    return true;

}

void performDecryption()

{

    int x, y, i, j, k, n;

    cout << "Enter the decryption key: ";

    string key;

    cin >> key;

    n = key.length() / 2;

    vector<vector<int>> a(n, vector<int>(n));

    vector<vector<int>> adj(n, vector<int>(n));

    vector<vector<int>> inv(n, vector<int>(n));

    int iter = 0;

    for (i = 0; i < n; i++)

    {

        for (j = 0; j < n; j++)

        {

            int val = key[iter] - 97;

            a[i][j] = val;

            iter++;

        }

    }

    if (calculateInverse(a, inv, n))

    {

        cout << "Inverse exists\n";

    }

    cout << "Enter the message to decrypt\n";

    string message;

    cin >> message;

    k = 0;

    string decryptedText;

    while (k < message.size())

    {

        for (i = 0; i < n; i++)

        {

            int sum = 0;

            int temp = k;

            for (j = 0; j < n; j++)

            {

                sum += ((inv[i][j] + 26) % 26 \* (message[temp++] - 'a') % 26) % 26;

                sum = sum % 26;

            }

            decryptedText += (sum + 'a');

        }

        k += n;

    }

    int lastCharIndex = decryptedText.size() - 1;

    while (decryptedText[lastCharIndex] == 'x')

    {

        lastCharIndex--;

    }

    cout << "\nDecrypted text is: ";

    for (i = 0; i <= lastCharIndex; i++)

    {

        cout << decryptedText[i];

    }

    cout << '\n';

}

int main()

{

    int choice;

    cout << "Enter your choice :\n";

    cout << "1. Encryption:\n2.Decryption:\n";

    cin >> choice;

    switch (choice)

    {

    case 1:

        performEncryption();

        break;

    case 2:

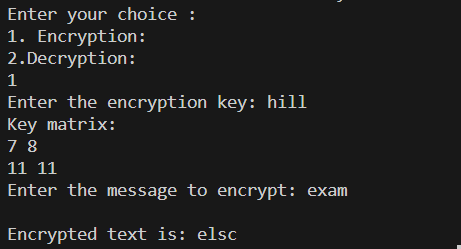
        performDecryption();

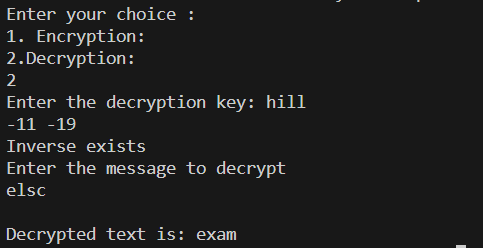
        break;

    }

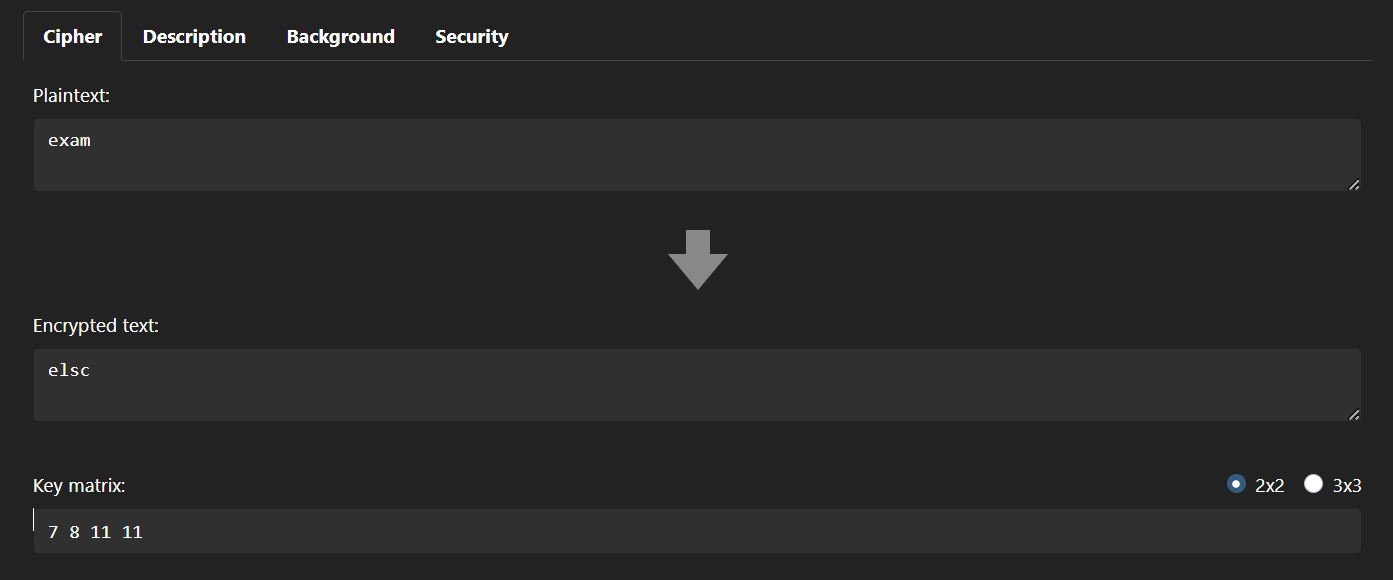
}

**Output (Program):**





**Output (Cryptool):**



**Cryptanalysis:**

**1) Known Plaintext Attack:**

In a known-plaintext attack, the attacker has access to some examples of plaintext and their corresponding ciphertext pairs. This information helps in deducing the key.

**2) Meet-in-the-Middle Attack:**

This attack involves encrypting all possible plaintexts using half of the key space, and decrypting all possible ciphertexts using the other half. If a match is found between the two sets, the key can be recovered.

**3) Linear Algebra:** The Hill cipher is based on matrix multiplication. With enough known plaintext-ciphertext pairs, the attacker can set up a system of linear equations to solve for the key matrix. The attacker needs at least as many pairs as the size of the key matrix (e.g., 4 pairs for a 2x2 matrix).

**Applications:**

While the Hill Cipher has certain vulnerabilities, it still finds application in various domains due to its matrix-based encryption. Some applications include:

**Educational Purposes:** The Hill Cipher is often used as an introductory example of a polygraphic cipher that utilizes matrix operations. It helps students learn about encryption, linear algebra, and modular arithmetic.

**Historical Significance:** The Hill Cipher is historically significant as one of the earliest attempts to enhance the security of classical ciphers. It paved the way for more advanced encryption techniques, including modern block ciphers.

**Secure Communication Protocols:** While not suitable for modern cryptographic standards, Hill Cipher's matrix-based approach can inspire more complex encryption algorithms used in secure communication protocols.

**Basic Encryption:** In scenarios where moderate security is sufficient, the Hill Cipher can be used for basic encryption of small texts, especially when education or historical context is the primary objective.

**References:**

1. GeeksforGeeks
2. www.javatpoint.com
3. www.educative.io
4. www.cryptool.org/en/cto/hill