**CNS LAB Experiments**

**1. Perform encryption, decryption using the following substitution techniques**

**a) Ceaser cipher -**

The Caesar Cipher shifts each letter of the plaintext by a fixed number (key) in the alphabet.

Encryption Process:

1. Choose a shift (key).
2. For each letter in the plaintext, shift it forward by the key.
3. Non-alphabet characters remain unchanged.
4. The result is the ciphertext.

Decryption Process:

1. Use the same shift (key).
2. Shift each letter in the ciphertext backward by the key.
3. The result is the original plaintext.

#include <bits/stdc++.h>

using namespace std;

string encrypt(string str, int key)

{

    int n = str.size();

    string temp;

    for(int i = 0; i < n; i++)

    {

        if(str[i] == ' ')

        {

            temp += ' ';

        }

        else

        {

            char ch = (str[i] + key);

            temp += ch;

        }

    }

    return temp;

}

string decrypt(string str, int key)

{

    int n = str.size();

    string temp;

    for(int i = 0; i < n; i++)

    {

        if(str[i] == ' ')

        {

            temp += ' ';

        }

        else

        {

            char ch = (str[i] - key);

            temp += ch;

        }

    }

    return temp;

}

int main()

{

    int shiftKey;

    cout << "Enter the shift key value: ";

    cin >> shiftKey;

    cin.ignore(); // to clear the newline character from the buffer

    string str;

    cout << "Enter the string to be encrypted: ";

    getline(cin, str);

    string newStr = encrypt(str, shiftKey);

    cout << "Encrypted string: " << newStr << endl;

    string newStr1 = decrypt(newStr, shiftKey);

    cout << "Decrypted string: " << newStr1 << endl;

    return 0;

}

**b) playfair cipher –**

The Playfair Cipher is a digraph substitution cipher where pairs of letters are encrypted using a 5x5 grid of letters based on a keyword. Here's how the execution works for encryption and decryption:

Playfair Cipher Execution:

Step 1: Create the Playfair Cipher Grid

* Keyword: Choose a keyword (e.g., "KEYWORD").
* Create a 5x5 matrix using the keyword by filling in the letters of the keyword first (without repetition) and then the remaining letters of the alphabet (combining I and J).

Example grid for keyword "KEYWORD":

K E Y W O

R D A B C

F G H I/J L

M N P Q S

T U V X Z

Step 2: Prepare the Plaintext

1. Divide the plaintext into pairs of letters (digraphs). If there are repeated letters in a pair (like "LL"), insert a filler letter (usually 'X') between them. If the text has an odd number of characters, add a filler letter at the end.

Example:

* + Plaintext: HELLO
  + Divide into pairs: HE LL O becomes HE LX LO.

Step 3: Encryption Rules

For each pair of letters:

1. Same Row: Replace each letter with the letter to its right (wrap around if needed).
2. Same Column: Replace each letter with the letter below it (wrap around if needed).
3. Different Row and Column: Form a rectangle and replace the pair with the letters at the opposite corners of the rectangle.

Example Encryption:

* Plaintext Pair: HE
  + H (row 3, col 3) and E (row 1, col 2)
  + Form a rectangle: Replace H with G (row 3, col 2) and E with Y (row 1, col 3).
  + Result: GY
* Plaintext Pair: LX
  + L (row 3, col 5) and X (row 4, col 4)
  + Form a rectangle: Replace L with Z (row 3, col 4) and X with S (row 4, col 5).
  + Result: ZS
* Plaintext Pair: LO
  + L (row 3, col 5) and O (row 1, col 5) are in the same column.
  + Replace L with S (row 4, col 5) and O with C (row 2, col 5).
  + Result: SC
* Ciphertext: GY ZS SC

Step 4: Decryption Rules

Decryption follows the reverse of the encryption rules:

1. Same Row: Replace each letter with the one to its left.
2. Same Column: Replace each letter with the one above it.
3. Different Row and Column: Form a rectangle and use the opposite corners to get the original pair.

The Playfair Cipher uses digraphs instead of single letters, making it more secure than simple substitution ciphers like Caesar.

// C++ program to implement Playfair Cipher

#include <bits/stdc++.h>

using namespace std;

#define SIZE 30

// Function to convert the string to lowercase

void toLowerCase(char plain[], int ps)

{

    int i;

    for (i = 0; i < ps; i++) {

        if (plain[i] > 64 && plain[i] < 91)

            plain[i] += 32;

    }

}

// Function to remove all spaces in a string

int removeSpaces(char\* plain, int ps)

{

    int i, count = 0;

    for (i = 0; i < ps; i++)

        if (plain[i] != ' ')

            plain[count++] = plain[i];

    plain[count] = '\0';

    return count;

}

// Function to generate the 5x5 key square

void generateKeyTable(char key[], int ks, char keyT[5][5])

{

    int i, j, k, flag = 0;

    // a 26 character hashmap

    // to store count of the alphabet

    int dicty[26] = { 0 };

    for (i = 0; i < ks; i++) {

        if (key[i] != 'j')

            dicty[key[i] - 97] = 2;

    }

    dicty['j' - 97] = 1;

    i = 0;

    j = 0;

    // Placing characters of key string in the key table

    for (k = 0; k < ks; k++) {

        if (dicty[key[k] - 97] == 2) {

            dicty[key[k] - 97] -= 1;

            keyT[i][j] = key[k];

            j++;

            if (j == 5) {

                i++;

                j = 0;

            }

        }

    }

    // PLacing remaining charachters

    for (k = 0; k < 26; k++) {

        if (dicty[k] == 0) {

            keyT[i][j] = (char)(k + 97);

            j++;

            if (j == 5) {

                i++;

                j = 0;

            }

        }

    }

}

// Function to search for the characters of a digraph

// in the key square and return their position

void search(char keyT[5][5], char a, char b, int arr[])

{

    int i, j;

    if (a == 'j')

        a = 'i';

    else if (b == 'j')

        b = 'i';

    for (i = 0; i < 5; i++) {

        for (j = 0; j < 5; j++) {

            if (keyT[i][j] == a) {

                arr[0] = i;

                arr[1] = j;

            }

            else if (keyT[i][j] == b) {

                arr[2] = i;

                arr[3] = j;

            }

        }

    }

}

// Function to find the modulus with 5

int mod5(int a) { return (a % 5); }

// Function to make the plain text length to be even

int prepare(char str[], int ptrs)

{

    if (ptrs % 2 != 0) {

        str[ptrs++] = 'z';

        str[ptrs] = '\0';

    }

    return ptrs;

}

// Function for performing the encryption

void encrypt(char str[], char keyT[5][5], int ps)

{

    int i, a[4];

    for (i = 0; i < ps; i += 2) {

        search(keyT, str[i], str[i + 1], a);

        // if both the characters are in the same row

        if (a[0] == a[2]) {

            str[i] = keyT[a[0]][mod5(a[1] + 1)];

            str[i + 1] = keyT[a[0]][mod5(a[3] + 1)];

        }

        // if they are in the same column

        else if (a[1] == a[3]) {

            str[i] = keyT[mod5(a[0] + 1)][a[1]];

            str[i + 1] = keyT[mod5(a[2] + 1)][a[1]];

        }

        // they form a rectangle. i.e. if they are neither in the same nor in the same col

        else {

            str[i] = keyT[a[0]][a[3]];

            str[i + 1] = keyT[a[2]][a[1]];

        }

    }

}

// Function to encrypt using Playfair Cipher

void encryptByPlayfairCipher(char str[], char key[])

{

    int ps, ks;

    char keyT[5][5];

    // Key

    ks = strlen(key);

    ks = removeSpaces(key, ks);

    toLowerCase(key, ks);

    // Plaintext

    ps = strlen(str);

    toLowerCase(str, ps);

    ps = removeSpaces(str, ps);

    ps = prepare(str, ps);

    generateKeyTable(key, ks, keyT);

    encrypt(str, keyT, ps);

}

// Function to decrypt

void decrypt(char str[], char keyT[5][5], int ps)

{

    int i, a[4];

    for (i = 0; i < ps; i += 2) {

        search(keyT, str[i], str[i + 1], a);

        if (a[0] == a[2]) {

            str[i] = keyT[a[0]][mod5(a[1] - 1)];

            str[i + 1] = keyT[a[0]][mod5(a[3] - 1)];

        }

        else if (a[1] == a[3]) {

            str[i] = keyT[mod5(a[0] - 1)][a[1]];

            str[i + 1] = keyT[mod5(a[2] - 1)][a[1]];

        }

        else {

            str[i] = keyT[a[0]][a[3]];

            str[i + 1] = keyT[a[2]][a[1]];

        }

    }

}

// Function to call decrypt

void decryptByPlayfairCipher(char str[], char key[])

{

    char ps, ks, keyT[5][5];

    // Key

    ks = strlen(key);

    ks = removeSpaces(key, ks);

    toLowerCase(key, ks);

    // ciphertext

    ps = strlen(str);

    toLowerCase(str, ps);

    ps = removeSpaces(str, ps);

    generateKeyTable(key, ks, keyT);

    decrypt(str, keyT, ps);

}

// Driver code

int main()

{

    char str[SIZE], key[SIZE];

    // Key to be encrypted

    strcpy(key, "Monarchy");

    cout << "Key text: " << key << "\n";

    // Plaintext to be encrypted

    strcpy(str, "instruments");

    cout << "Plain text: " << str << "\n";

    // encrypt using Playfair Cipher

    encryptByPlayfairCipher(str, key);

    cout << "Cipher text: " << str << "\n";

    // encrypt using Playfair Cipher

    decryptByPlayfairCipher(str, key);

    cout << "Deciphered text: " << str << endl;

    return 0;

}

**c) Hill Cipher**

The Hill Cipher is a polygraphic substitution cipher that uses linear algebra to encrypt a block of plaintext letters. It operates on n-letter blocks (most commonly 2x2 or 3x3 matrices) and relies on matrix multiplication over modulo 26 (for the 26 letters of the alphabet).

Steps to Encrypt Using the Hill Cipher:

1. Key Matrix:
   * Choose an n x n key matrix (must be invertible modulo 26 for decryption to work).
   * Example of a 2x2 key matrix:

| 3 3 |

| 2 5 |

1. Plaintext Preparation:
   * Convert the plaintext into numerical form (A = 0, B = 1, ..., Z = 25).
   * Divide the plaintext into blocks of size n (same as the size of the key matrix). Add filler letters if the plaintext length is not divisible by n.

Example:

* + Plaintext: HI (H = 7, I = 8)
  + Plaintext Vector: [7, 8]

1. Matrix Multiplication:
   * Multiply each block (represented as a vector) by the key matrix, and take the result modulo 26.
   * For the example, with key matrix:

| 3 3 | | 7 | = (3\*7 + 3\*8) % 26 = 45 % 26 = 19

| 2 5 | \* | 8 | = (2\*7 + 5\*8) % 26 = 54 % 26 = 2

* + Ciphertext Vector: [19, 2] (corresponds to letters T and C).

1. Ciphertext:
   * Convert the resulting numbers back to letters.
   * In this case, 19 -> T and 2 -> C, so the ciphertext is TC.

Decryption:

Decryption requires finding the inverse of the key matrix (mod 26) and then multiplying the ciphertext blocks by the inverse matrix, following the same steps in reverse.

#include <iostream>

#include <vector>

using namespace std;

// Function to perform modulo operation for negative values

int mod(int a, int m)

{

    return (a % m + m) % m;

}

// Function to find the multiplicative inverse of a number mod 26

int modInverse(int a, int m)

{

    a = a % m;

    for (int x = 1; x < m; x++)

    {

        if ((a \* x) % m == 1)

            return x;

    }

    return -1; // No inverse exists

}

// Function to calculate the determinant of a 2x2 matrix

int determinant(vector<vector<int>> matrix)

{

    return mod((matrix[0][0] \* matrix[1][1] - matrix[0][1] \* matrix[1][0]), 26);

}

// Function to find the inverse of a 2x2 matrix mod 26

vector<vector<int>> inverseMatrix(vector<vector<int>> matrix)

{

    vector<vector<int>> invMatrix(2, vector<int>(2));

    int det = determinant(matrix);

    int invDet = modInverse(det, 26);

    if (invDet == -1)

    {

        cout << "Inverse does not exist." << endl;

        return invMatrix; // Return an empty matrix

    }

    // Calculate the inverse matrix mod 26

    invMatrix[0][0] = mod(matrix[1][1] \* invDet, 26);

    invMatrix[0][1] = mod(-matrix[0][1] \* invDet, 26);

    invMatrix[1][0] = mod(-matrix[1][0] \* invDet, 26);

    invMatrix[1][1] = mod(matrix[0][0] \* invDet, 26);

    return invMatrix;

}

// Function to multiply matrix with a vector

vector<int> multiplyMatrix(vector<vector<int>> keyMatrix, vector<int> textVector)

{

    vector<int> result(2);

    result[0] = (keyMatrix[0][0] \* textVector[0] + keyMatrix[0][1] \* textVector[1]) % 26;

    result[1] = (keyMatrix[1][0] \* textVector[0] + keyMatrix[1][1] \* textVector[1]) % 26;

    return result;

}

// Function to encrypt the plaintext using Hill Cipher

string hillCipherEncrypt(string plaintext, vector<vector<int>> keyMatrix)

{

    string ciphertext = "";

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

    {

        // Convert characters to numbers (A=0, B=1, ..., Z=25)

        vector<int> textVector = {plaintext[i] - 'A', plaintext[i + 1] - 'A'};

        // Multiply key matrix with the plaintext vector

        vector<int> resultVector = multiplyMatrix(keyMatrix, textVector);

        // Convert numbers back to characters

        ciphertext += (resultVector[0] + 'A');

        ciphertext += (resultVector[1] + 'A');

    }

    return ciphertext;

}

// Function to decrypt the ciphertext using Hill Cipher

string hillCipherDecrypt(string ciphertext, vector<vector<int>> keyMatrix)

{

    // Get the inverse of the key matrix

    vector<vector<int>> invMatrix = inverseMatrix(keyMatrix);

    if (invMatrix[0][0] == 0 && invMatrix[0][1] == 0)

    {

        return ""; // No valid inverse matrix found

    }

    string plaintext = "";

    for (int i = 0; i < ciphertext.length(); i += 2)

    {

        // Convert characters to numbers (A=0, B=1, ..., Z=25)

        vector<int> textVector = {ciphertext[i] - 'A', ciphertext[i + 1] - 'A'};

        // Multiply inverse matrix with the ciphertext vector

        vector<int> resultVector = multiplyMatrix(invMatrix, textVector);

        // Convert numbers back to characters

        plaintext += (resultVector[0] + 'A');

        plaintext += (resultVector[1] + 'A');

    }

    return plaintext;

}

int main()

{

    // 2x2 Key Matrix (must be invertible modulo 26)

    vector<vector<int>> keyMatrix = {{3, 3}, {2, 5}};

    // Plaintext to encrypt (must be even length)

    string plaintext = "HELLO";

    // Ensure plaintext length is even by adding filler if necessary

    if (plaintext.length() % 2 != 0)

    {

        plaintext += 'X'; // Add filler letter 'X'

    }

    // Encrypt the plaintext

    string ciphertext = hillCipherEncrypt(plaintext, keyMatrix);

    cout << "Encrypted Ciphertext: " << ciphertext << endl;

    // Decrypt the ciphertext

    string decryptedText = hillCipherDecrypt(ciphertext, keyMatrix);

    cout << "Decrypted Plaintext: " << decryptedText << endl;

    return 0;

}

**d) Vigenere cipher**

The Vigenère cipher is a method of encrypting alphabetic text using a simple form of polyalphabetic substitution. It uses a keyword to shift the characters of the plaintext, with each letter of the keyword determining the shift for the corresponding letter of the plaintext.

Steps to Encrypt Using the Vigenère Cipher:

1. Choose a Key:
   * The key is a sequence of letters (e.g., a word or phrase). If the key is shorter than the plaintext, repeat the key to match the length of the plaintext.

Example:

* + Key: KEY
  + Plaintext: HELLO

1. Prepare the Key:
   * Repeat the key to match the length of the plaintext.

For the plaintext HELLO and key KEY, the key becomes:

* + KEYKE

1. Shift Each Character:
   * Convert each letter in the plaintext and key to its corresponding number (A=0, B=1, ..., Z=25).
   * For each character in the plaintext, shift it by the number corresponding to the matching letter in the key.
   * Perform the shift using modular arithmetic (mod 26).

Example:

Plaintext: H (7), E (4), L (11), L (11), O (14)

Key: K (10), E (4), Y (24), K (10), E (4)

Shift: (7+10)%26, (4+4)%26, (11+24)%26, (11+10)%26, (14+4)%26

Result: 17 (R), 8 (I), 9 (J), 21 (V), 18 (S)

1. Ciphertext:
   * Convert the resulting numbers back to letters.
   * The final ciphertext is: RIJVS.

Decryption:

Decryption uses the same process, but instead of adding the key values to the plaintext, you subtract them and take the result modulo 26.

Example for the ciphertext - RIJVS and key- KEY

Ciphertext: R (17), I (8), J (9), V (21), S (18)

Key: K (10), E (4), Y (24), K (10), E (4)

Shift: (17-10)%26, (8-4)%26, (9-24)%26, (21-10)%26, (18-4)%26

Result: 7 (H), 4 (E), 11 (L), 11 (L), 14 (O)

* Decrypted Text: HELLO

// C++ code to implement Vigenere Cipher

#include <bits/stdc++.h>

using namespace std;

// This function generates the key in

// a cyclic manner until it's length isn't

// equal to the length of original text

string generateKey(string str, string key)

{

    int x = str.size();

    for (int i = 0;; i++) {

        if (x == i)

            i = 0;

        if (key.size() == str.size())

            break;

        key.push\_back(key[i]);

    }

    return key;

}

// This function returns the encrypted text

// generated with the help of the key

string cipherText(string str, string key)

{

    string cipher\_text;

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

        // converting in range 0-25

        char x = (str[i] + key[i]) % 26;

        // convert into alphabets(ASCII)

        x += 'A';

        cipher\_text.push\_back(x);

    }

    return cipher\_text;

}

// This function decrypts the encrypted text

// and returns the original text

string originalText(string cipher\_text, string key)

{

    string orig\_text;

    for (int i = 0; i < cipher\_text.size(); i++) {

        // converting in range 0-25

        char x = (cipher\_text[i] - key[i] + 26) % 26;

        // convert into alphabets(ASCII)

        x += 'A';

        orig\_text.push\_back(x);

    }

    return orig\_text;

}

// Driver program to test the above function

int main()

{

    string str = "GEEKSFORGEEKS";

    string keyword = "AYUSH";

    if (any\_of(str.begin(), str.end(), ::islower))

        transform(str.begin(), str.end(), str.begin(),

                  ::toupper);

    if (any\_of(keyword.begin(), keyword.end(), ::islower))

        transform(keyword.begin(), keyword.end(),

                  keyword.begin(), ::toupper);

    string key = generateKey(str, keyword);

    string cipher\_text = cipherText(str, key);

    cout << "Ciphertext : " << cipher\_text << "\n";

    cout << "Original/Decrypted Text : "

         << originalText(cipher\_text, key);

    return 0;

}

**2. Perform encryption and decryption using following transposition techniques**

**a) Rail fence**

The Rail Fence Cipher is a transposition cipher where the plaintext is written in a zigzag (rail-fence) pattern and then read row by row to create the ciphertext.

Steps for Encryption:

1. Write Plaintext in a Zigzag Pattern:
   * Choose the number of rails (rows). Write the plaintext diagonally across these rails.
   * Example: For 3 rails and plaintext "HELLO WORLD", we write:

H . . . O . . . L

. E . L . W . R .

. . L . . . O . D

1. Read Row by Row:
   * Read the letters across each row to form the ciphertext.
   * Ciphertext: HOLELWRDLO

Decryption:

To decrypt, reverse the process:

1. Write the ciphertext in a zigzag pattern (filling the characters in row by row).
2. Read the zigzag to reconstruct the plaintext.

#include <bits/stdc++.h>

using namespace std;

// function to encrypt a message

string encryptRailFence(string text, int key)

{

    char rail[key][(text.size())];

    for(int i=0;i<key;i++)

    {

        for(int j=0;j<text.size();j++)

        {

            rail[i][j] = '\n';

        }

    }

    bool dir\_down = false;

    int row=0, col=0;

    for(int i=0;i<text.length();i++)

    {

        if(row == 0 || row == key-1)

          dir\_down = !dir\_down;

        rail[row][col++] = text[i];

        dir\_down? row++ : row--;

    }

    string result;

    for(int i=0;i<key;i++)

    {

        for(int j=0;j<text.size();j++)

        {

            if(rail[i][j] != '\n')

               result.push\_back(rail[i][j]);

        }

    }

    return result;

}

// This function receives cipher-text and key

// and returns the original text after decryption

string decryptRailFence(string cipher, int key)

{

    // create the matrix to cipher plain text

    // key = rows , length(text) = columns

    char rail[key][cipher.length()];

    // filling the rail matrix to distinguish filled

    // spaces from blank ones

    for (int i=0; i < key; i++)

        for (int j=0; j < cipher.length(); j++)

            rail[i][j] = '\n';

    // to find the direction

    bool dir\_down;

    int row = 0, col = 0;

    // mark the places with '\*'

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

    {

        // check the direction of flow

        if (row == 0)

            dir\_down = true;

        if (row == key-1)

            dir\_down = false;

        // place the marker

        rail[row][col++] = '\*';

        // find the next row using direction flag

        dir\_down?row++ : row--;

    }

    // now we can construct the fill the rail matrix

    int index = 0;

    for (int i=0; i<key; i++)

        for (int j=0; j<cipher.length(); j++)

            if (rail[i][j] == '\*' && index<cipher.length())

                rail[i][j] = cipher[index++];

    // now read the matrix in zig-zag manner to construct

    // the resultant text

    string result;

    row = 0, col = 0;

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

    {

        // check the direction of flow

        if (row == 0)

            dir\_down = true;

        if (row == key-1)

            dir\_down = false;

        // place the marker

        if (rail[row][col] != '\*')

            result.push\_back(rail[row][col++]);

        // find the next row using direction flag

        dir\_down?row++: row--;

    }

    return result;

}

//driver program to check the above functions

int main()

{

    cout << encryptRailFence("attack at once", 2) << endl;

    cout << encryptRailFence("GeeksforGeeks ", 3) << endl;

    cout << encryptRailFence("defend the east wall", 3) << endl;

    //Now decryption of the same cipher-text

    cout << decryptRailFence("GsGsekfrek eoe",3) << endl;

    cout << decryptRailFence("atc toctaka ne",2) << endl;

    cout << decryptRailFence("dnhaweedtees alf  tl",3) << endl;

    return 0;

}

**b)row and Column Transformation**

The Row and Column Transposition Cipher is another form of transposition cipher where the plaintext is arranged into a grid (based on the key or columnar order), then rearranged by permuting the rows and columns.

Steps for Encryption:

1. Arrange Plaintext into a Grid:
   * Use a keyword to decide the number of columns.
   * Arrange the plaintext into rows, based on the number of columns defined by the keyword length.

Example: Plaintext "HELLO WORLD" with key "KEY" (length 3).

H E L

L O W

O R L

D X X

(Fill in extra spaces with X if needed.)

1. Read Column by Column (Based on Key Order):
   * Sort the columns based on the alphabetical order of the key letters and read vertically by column.

For key "KEY" (K=2, E=1, Y=3):

* + Re-order columns: (2nd column first, then 1st column, then 3rd column)
  + Ciphertext: EOXHLWRLDOX

Decryption:

1. Reconstruct Grid Based on Key:
   * Use the key to determine the column order.
   * Write the ciphertext into the columns in the given order, then read row by row to obtain the original message.

#include <bits/stdc++.h>

using namespace std;

string enc(string str, string key)

{

    bool flg = false;

    int ind = 0;

    // Find the maximum digit in the key

    int mxnum = 0;

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

    {

        mxnum = max(mxnum, key[i] - '0');

    }

    // Calculate the number of rows required

    int nrow = ceil((double)str.size() / mxnum);

    vector<vector<char>> vec(nrow, vector<char>(mxnum, '$')); // Fill with space instead of '0'

    for (int i = 0; i < nrow; i++)

    {

        for (int j = 0; j < mxnum; j++)

        {

            if (ind >= str.size())

            {

                flg = true;

                break;

            }

            vec[i][j] = str[ind++];

        }

        if (flg == true)

            break;

    }

    cout << "Matrix : \n";

    for (int i = 0; i < vec.size(); i++)

    {

        for (int j = 0; j < vec[i].size(); j++)

        {

            cout << vec[i][j] << " ";

        }

        cout << "\n";

    }

    string res;

    for (int k = 0; k < key.size(); k++)

    {

        int col = key[k] - '1'; // Adjust for 0-based index

        for (int row = 0; row < nrow; row++)

        {

            if(vec[row][col] != '$') // Avoid adding spaces from the matrix

                res += vec[row][col];

        }

    }

    return res;

}

string dec(string str, string key)

{

    bool flg = false;

    int ind = 0;

    // Find the maximum digit in the key

    int mxnum = 0;

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

    {

        mxnum = max(mxnum, key[i] - '0');

    }

    // Calculate the number of rows required

    int nrow = ceil((double)str.size() / mxnum);

    vector<vector<char>> vec(nrow, vector<char>(mxnum, '$')); // Fill with '$' instead of '0'

    for(int i=0; i<key.size(); i++)

    {

        int col = key[i] - '1';

        for(int row=0; row<nrow; row++)

        {

            if(ind >= str.size())

            {

                flg = true;

                break;

            }

            vec[row][col] = str[ind++];

        }

        if(flg == true)

          break;

    }

    string res;

    for(int i=0; i<vec.size(); i++)

    {

        for(int j=0; j<vec[i].size(); j++)

        {

            if(vec[i][j] != '$')

              res += vec[i][j];

        }

    }

    return res;

}

int main()

{

    string s;

    cout << "Input Text: " << endl;

    getline(cin, s);

    string key;

    cout << "Input Key: " << endl;

    cin >> key;

    string ciphertext = enc(s, key);

    cout << "Ciphertext: " << ciphertext << endl;

    string decryptedtext = dec(ciphertext, key);

    cout << "Decrypted text: " << decryptedtext << endl;

    return 0;

}

**3. Implementation of Euclidean and Extended Euclidean Algorithm**

Execution of Euclidean Algorithm

The Euclidean Algorithm is a method for finding the greatest common divisor (GCD) of two numbers. It repeatedly subtracts the smaller number from the larger number or uses division with remainder until one of the numbers becomes zero. The non-zero number is the GCD.

Steps of Euclidean Algorithm:

1. Input two numbers a and b.
2. While b is not zero:
   * Compute a % b (remainder when a is divided by b).
   * Set a to b and b to the remainder (a % b).
3. When b becomes 0, a will hold the GCD.

#include <iostream>

using namespace std;

// Function to return

// gcd of a and b

int gcd(int a, int b)

{

    if (a == 0)

        return b;

    return gcd(b % a, a);

}

int main() {

    int num1, num2;

    cout << "Enter two integers: ";

    cin >> num1 >> num2;

    int result = gcd(num1, num2);

    cout << "GCD of " << num1 << " and " << num2 << " is: " << result << endl;

    return 0;

}

**Execution of Extended Euclidean Algorithm**

The Extended Euclidean Algorithm not only finds the GCD of two numbers but also finds coefficients x and y such that:

a⋅x+b⋅y=GCD(a,b)

These coefficients can be used to solve linear Diophantine equations.

Steps of Extended Euclidean Algorithm:

1. Input two numbers a and b.
2. Use the Euclidean algorithm to find the GCD of a and b.
3. During the process, keep track of the coefficients x and y that satisfy the equation.

Example of Extended Euclidean Algorithm:

Find the GCD of 56 and 98 and the coefficients x and y.

1. Run the Euclidean Algorithm:
   * GCD(56, 98) = 14 (as seen above).
2. Backtrack to find x and y:
   * From the steps of Euclidean Algorithm:
     + 98 = 1 \* 56 + 42
     + 56 = 1 \* 42 + 14
     + 42 = 3 \* 14 + 0
   * Now, backtrack:
     + From 14 = 56 - 1 \* 42 (substitute 42 from the first equation):
     + 14 = 56 - 1 \* (98 - 1 \* 56) = 2 \* 56 - 1 \* 98.

Thus, the coefficients are x = 2 and y = -1.

This means:

56⋅2+98⋅(−1)= 1456\*2+98\*(−1)=14

// C++ program to demonstrate working of

// extended Euclidean Algorithm

#include <bits/stdc++.h>

using namespace std;

// Function for extended Euclidean Algorithm

int gcdExtended(int a, int b, int \*x, int \*y)

{

    // Base Case

    if (a == 0)

    {

        \*x = 0;

        \*y = 1;

        return b;

    }

    int x1, y1; // To store results of recursive call

    int gcd = gcdExtended(b%a, a, &x1, &y1);

    // Update x and y using results of

    // recursive call

    \*x = y1 - (b/a) \* x1;

    \*y = x1;

    return gcd;

}

// Driver Code

int main()

{

    int x, y, a = 35, b = 15;

    int g = gcdExtended(a, b, &x, &y);

    cout << "GCD(" << a << ", " << b

         << ") = " << g << endl;

    return 0;

}

**4. Implementation of Chinese Remainder Theorem (CRT)**

// A C++ program to demonstrate

// working of Chinese remainder

// Theorem

#include <bits/stdc++.h>

using namespace std;

// Returns modulo inverse of a

// with respect to m using

// extended Euclid Algorithm.

// multiplicative-inverse-under-modulo-m/

int inv(int a, int m)

{

    int m0 = m, t, q;

    int x0 = 0, x1 = 1;

    if (m == 1)

        return 0;

    // Apply extended Euclid Algorithm

    while (a > 1) {

        // q is quotient

        q = a / m;

        t = m;

        // m is remainder now, process same as

        // euclid's algo

        m = a % m, a = t;

        t = x0;

        x0 = x1 - q \* x0;

        x1 = t;

    }

    // Make x1 positive

    if (x1 < 0)

        x1 += m0;

    return x1;

}

// k is size of num[] and rem[]. Returns the smallest

// number x such that:

// x % num[0] = rem[0],

// x % num[1] = rem[1],

// ..................

// x % num[k-2] = rem[k-1]

// Assumption: Numbers in num[] are pairwise coprime

// (gcd for every pair is 1)

int findMinX(int num[], int rem[], int k)

{

    // Compute product of all numbers

    int prod = 1;

    for (int i = 0; i < k; i++)

        prod \*= num[i];

    // Initialize result

    int result = 0;

    // Apply above formula

    for (int i = 0; i < k; i++) {

        int pp = prod / num[i];

        result += rem[i] \* inv(pp, num[i]) \* pp;

    }

    return result % prod;

}

// Driver method

int main(void)

{

    int num[] = { 3, 4, 5 };

    int rem[] = { 2, 3, 1 };

    int k = sizeof(num) / sizeof(num[0]);

    cout << "x is " << findMinX(num, rem, k);

    return 0;

}

**5. Apply DES algorithm for practical applications**

Data Encryption Standard (DES) is a symmetric-key algorithm for the encryption of digital data. Here’s how it works:

Execution of DES

1. Key Generation: DES uses a 56-bit key (often represented as a 64-bit key with every eighth bit used for parity). The key undergoes permutation and is divided into two halves.
2. Initial Permutation (IP): The plaintext undergoes an initial permutation to rearrange its bits.
3. Rounds: DES has 16 rounds of processing. Each round involves:
   * Expansion: Expanding the right half of the data from 32 bits to 48 bits.
   * Key Mixing: Mixing the expanded data with a round key derived from the original key.
   * Substitution: The mixed data is substituted using S-boxes to obtain a 32-bit output.
   * Permutation: The output undergoes a final permutation.
   * The two halves are then swapped.
4. Final Permutation (IP-1): The output from the 16th round undergoes a final permutation to produce the ciphertext.

Practical Applications of DES

* Secure Communications: Used in secure email and file encryption.
* Banking Systems: For encrypting sensitive transactions and data.
* Legacy Systems: Some older systems still rely on DES for backward compatibility, although it is considered insecure by modern standards.

# Python3 code for the above approach

# 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"}

    bin = ""

    for i in range(len(s)):

        bin = bin + mp[s[i]]

    return bin

# 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 = ""

    for i in range(0, len(s), 4):

        ch = ""

        ch = ch + s[i]

        ch = ch + s[i + 1]

        ch = ch + s[i + 2]

        ch = ch + s[i + 3]

        hex = hex + mp[ch]

    return hex

# Binary to decimal conversion

def bin2dec(binary):

    binary1 = binary

    decimal, i, n = 0, 0, 0

    while(binary != 0):

        dec = binary % 10

        decimal = decimal + dec \* pow(2, i)

        binary = binary//10

        i += 1

    return decimal

# Decimal to binary conversion

def dec2bin(num):

    res = bin(num).replace("0b", "")

    if(len(res) % 4 != 0):

        div = len(res) / 4

        div = int(div)

        counter = (4 \* (div + 1)) - len(res)

        for i in range(0, counter):

            res = '0' + res

    return res

# Permute function to rearrange the bits

def permute(k, arr, n):

    permutation = ""

    for i in range(0, n):

        permutation = permutation + k[arr[i] - 1]

    return permutation

# shifting the bits towards left by nth shifts

def shift\_left(k, nth\_shifts):

    s = ""

    for i in range(nth\_shifts):

        for j in range(1, len(k)):

            s = s + k[j]

        s = s + k[0]

        k = s

        s = ""

    return k

# calculating xow of two strings of binary number a and b

def xor(a, b):

    ans = ""

    for i in range(len(a)):

        if a[i] == b[i]:

            ans = ans + "0"

        else:

            ans = ans + "1"

    return ans

# Table of Position of 64 bits at initial level: 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-boxex: 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"

# 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 = []

for i in range(0, 16):

    # Shifting the bits by nth shifts by checking from shift table

    left = shift\_left(left, shift\_table[i])

    right = shift\_left(right, shift\_table[i])

    # Combination of left and right string

    combine\_str = left + right

    # Compression of key from 56 to 48 bits

    round\_key = permute(combine\_str, key\_comp, 48)

    rkb.append(round\_key)

    rk.append(bin2hex(round\_key))

print("Encryption")

cipher\_text = bin2hex(encrypt(pt, rkb, rk))

print("Cipher Text : ", cipher\_text)

print("Decryption")

rkb\_rev = rkb[::-1]

rk\_rev = rk[::-1]

text = bin2hex(encrypt(cipher\_text, rkb\_rev, rk\_rev))

print("Plain Text : ", text)

from Crypto.Cipher import DES

from secrets import token\_bytes

# Function to generate a random key for DES encryption

def generate\_key():

    return token\_bytes(8)  # DES uses 8-byte (64-bit) keys

# Function to pad the text to a multiple of 8 bytes (since DES works on 64-bit blocks)

def pad(text):

    while len(text) % 8 != 0:

        text += ' '

    return text

# Encrypt function

def encrypt(message, key):

    cipher = DES.new(key, DES.MODE\_ECB)  # Using ECB mode for simplicity

    padded\_text = pad(message)

    encrypted\_text = cipher.encrypt(padded\_text.encode('utf-8'))

    return encrypted\_text

# Decrypt function

def decrypt(encrypted\_message, key):

    cipher = DES.new(key, DES.MODE\_ECB)  # Using ECB mode for simplicity

    decrypted\_text = cipher.decrypt(encrypted\_message).decode('utf-8').rstrip()

    return decrypted\_text

# Example usage

if \_\_name\_\_ == "\_\_main\_\_":

    # Generate a key

    key = generate\_key()

    # Message to be encrypted

    message = "DES Sample Text"

    # Encrypt the message

    encrypted\_message = encrypt(message, key)

    print(f"Encrypted Message: {encrypted\_message}")

    # Decrypt the message

    decrypted\_message = decrypt(encrypted\_message, key)

    print(f"Decrypted Message: {decrypted\_message}")

**6. Apply AES algorithm for practical applications**

Advanced Encryption Standard (AES) is a symmetric encryption standard that replaced DES. It supports key sizes of 128, 192, and 256 bits.

Execution of AES

1. Key Expansion: The original key is expanded into a key schedule, which contains the round keys used in each round.
2. Initial Round: The plaintext is added to the first round key.
3. Main Rounds (9, 11, or 13 rounds):
   * SubBytes: Each byte of the state is replaced with its corresponding byte from the S-box.
   * ShiftRows: Rows of the state are cyclically shifted.
   * MixColumns: Columns of the state are mixed.
   * AddRoundKey: The state is XORed with the round key.
4. Final Round: Similar to main rounds, but without the MixColumns step.
5. Output: The final state after the last round is the ciphertext.

Practical Applications of AES

* Data Protection: Used in securing sensitive data, including government communications, financial transactions, and personal data.
* Secure Wi-Fi: AES is used in WPA2 for encrypting wireless communications.
* Disk Encryption: Commonly used for full disk encryption in various operating systems and storage devices.

from Crypto.Cipher import AES

from Crypto.Random import get\_random\_bytes

from Crypto.Util.Padding import pad, unpad

# Function to encrypt a message using AES

def aes\_encrypt(message, key):

    cipher = AES.new(key, AES.MODE\_CBC)  # CBC mode requires an IV (Initialization Vector)

    iv = cipher.iv  # Generate random IV

    padded\_message = pad(message.encode('utf-8'), AES.block\_size)  # Pad the message to a multiple of 16 bytes

    encrypted\_message = cipher.encrypt(padded\_message)

    return iv + encrypted\_message  # Return IV concatenated with encrypted message

# Function to decrypt a message using AES

def aes\_decrypt(encrypted\_message, key):

    iv = encrypted\_message[:16]  # Extract the IV (first 16 bytes)

    cipher = AES.new(key, AES.MODE\_CBC, iv)

    decrypted\_message = unpad(cipher.decrypt(encrypted\_message[16:]), AES.block\_size)  # Decrypt and unpad the message

    return decrypted\_message.decode('utf-8')

# Example usage

if \_\_name\_\_ == "\_\_main\_\_":

    # AES uses 16, 24, or 32 byte keys (128, 192, or 256 bits)

    key = get\_random\_bytes(16)  # Generate a random 16-byte key (128-bit)

    # message = "This is a secret message."

     # Take input from the user

    message = input("Enter the message to encrypt: ")

    # Encrypt the message

    encrypted\_message = aes\_encrypt(message, key)

    print(f"Encrypted Message: {encrypted\_message}")

    # Decrypt the message

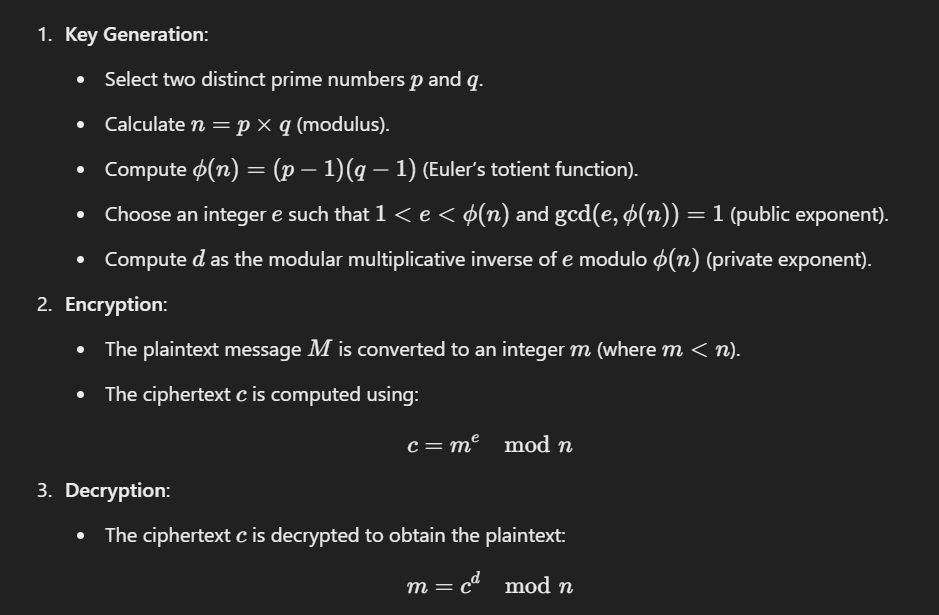
    decrypted\_message = aes\_decrypt(encrypted\_message, key)

    print(f"Decrypted Message: {decrypted\_message}")

**7. Implementation of RSA Algorithm**

RSA (Rivest-Shamir-Adleman) is an asymmetric cryptographic algorithm used for secure data transmission.

Execution of RSA



Practical Applications of RSA

* Secure Data Transmission: Used in HTTPS for secure web communications.
* Digital Signatures: RSA is widely used for creating digital signatures that verify the authenticity and integrity of messages.
* Key Exchange: RSA can be used for securely exchanging symmetric keys.

# and RSA la p,q cha numbers mothe ghyayche ahet

# ani prime numbers 2048 bits che ghya for rsa -> rsa numbers takla ki yeta

# ani rsa la plain text ghyaych aani number madhe convert karaycha

from Crypto.PublicKey import RSA

from Crypto.Cipher import PKCS1\_OAEP

from Crypto.Util.number import bytes\_to\_long, long\_to\_bytes

# Function to generate RSA keys (private and public)

def generate\_keys():

    key = RSA.generate(2048)  # RSA key size of 2048 bits

    private\_key = key.export\_key()

    public\_key = key.publickey().export\_key()

    return private\_key, public\_key

# Function to encrypt a plaintext message using RSA and converting it to a number

def rsa\_encrypt(plaintext, public\_key):

    rsa\_key = RSA.import\_key(public\_key)

    cipher = PKCS1\_OAEP.new(rsa\_key)

    # Convert plaintext to bytes and then to a long number

    plaintext\_bytes = plaintext.encode('utf-8')

    plaintext\_number = bytes\_to\_long(plaintext\_bytes)

    # Encrypt the plaintext number using RSA

    encrypted\_message = cipher.encrypt(long\_to\_bytes(plaintext\_number))

    return encrypted\_message

# Function to decrypt an RSA encrypted message and convert it back to plaintext

def rsa\_decrypt(encrypted\_message, private\_key):

    rsa\_key = RSA.import\_key(private\_key)

    cipher = PKCS1\_OAEP.new(rsa\_key)

    # Decrypt the message and convert back to a long number

    decrypted\_message\_bytes = cipher.decrypt(encrypted\_message)

    decrypted\_message\_number = bytes\_to\_long(decrypted\_message\_bytes)

    # Convert the number back to bytes and decode it to a string

    decrypted\_message = long\_to\_bytes(decrypted\_message\_number).decode('utf-8')

    return decrypted\_message

# Example usage

if \_\_name\_\_ == "\_\_main\_\_":

    # Generate RSA keys

    private\_key, public\_key = generate\_keys()

    # Message to encrypt

    message = "This is a secret message."

    # Encrypt the message

    encrypted\_message = rsa\_encrypt(message, public\_key)

    print(f"Encrypted Message (in bytes): {encrypted\_message}")

    # Decrypt the message

    decrypted\_message = rsa\_decrypt(encrypted\_message, private\_key)

    print(f"Decrypted Message: {decrypted\_message}")