RAJIV GANDHI NATIONAL INSTITUTE OF YOUTH DEVELOPMENT
(Institution of National Importance by the Act of Parliament No.35/2012)
Ministry of Youth Affairs and Sports, Government Of India

CRYPTOGRAPHY & NETWORK SECURITY PROJECT REPORT

Department of Computer Science
(1st semester, 23-25 batch)
RGNIYD, SRIPERUMBUDUR-602105

Group members

Jatin	ID- MSAI23R005
Shameel C	ID- MSCS23R007
Mathan Sethupathy	ID-MSCS23R002
Mohammed Ashjil N K	ID-MSCS23R003
Fathima Farhan K K	ID-MSCS23R001
Ninaniya Nateesh Kumar	ID-MSAI23R009

Under supervision of **Dr. P. Thiyagarajan**, Associate Professor, HOD of Computer Science (Cyber Security), RGNIYD

January 19, 2024

CERTIFICATE

This is to certify that I have examined the project entitled submitted by **Jatin, Shameel C, Mathan Sethupathy, Fathima Farhan K K, Mohammed Ashjil N K, Ninaniya Nateesh Kumar** the postgraduate students of Department of Computer Science. I hereby accord my approval of it as a study carried out and it fulfilled the requirements as per the regulations of the institute as well as course instructor and has reached the standard needed for submission.

Dr.P.Thiyagarajan
Associate Professor
HOD of Computer Science (Cyber Security)
RGNIYD, Sriperumbudur, Tamilnadu

ACKNOWLEDGEMENT

We would like to express our sincere and deep gratitude to our supervisor and guide Dr.P.Thiyagarajan, Associate Professor, HOD of Computer Science (Cyber Security), for his kind and constant support during our project work. It has been an absolute privilege to work with Dr.P.Thiyagarajan for our project. His valuable advice, critical criticism and active supervision encouraged us to sharpen our research methodology and was instrumental in shaping our professional outlook.

THANK-YOU

ABSTRACT

Cryptography plays a pivotal role in securing sensitive information and communication channels in today's digital world. Substitution techniques represent one of the fundamental building blocks in cryptographic systems, providing a method for transforming plaintext into ciphertext. This abstract provides an overview of the significance, types, and applications of substitution techniques in the context of cryptography. Substitution techniques are employed in various cryptographic applications, including securing electronic communications, protecting sensitive data in storage, and ensuring the integrity of digital signatures. These techniques are foundational in the development of more complex cryptographic algorithms and protocols.

Vigenère Cipher Implementation Documentation

MOHAMMED ASJHIK NK AND MATHAN SETHUPATHY

19 JANUARY 2024

0.1 Introduction

This document provides documentation for the Vigenère cipher implementation in Python. The Vigenère cipher is a method of encrypting alphabetic text using a simple form of polyalphabetic substitution. It uses a keyword to shift letters in the plaintext, providing a more secure encryption compared to the Caesar cipher. This document presents a Python implementation of the Vigenère cipher, including key generation, encryption, and decryption functions.

0.2 Working of 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 letters in the plaintext by different amounts at different positions.

Working Steps:

1. **Key Expansion:** Choose a keyword (a word or phrase). Repeat the keyword to match the length of the plaintext.
2. **Letter-to-Number Conversion:** Convert both the plaintext and the repeated keyword into numerical values (A=0, B=1, ..., Z=25).
3. **Encryption:** For each letter in the plaintext, shift it by the corresponding letter in the keyword. Wrap around the alphabet if needed.
4. **Ciphertext Formation:** Convert the resulting numerical values back to letters.

Example:

Consider encrypting the message "HELLO" with the keyword "KEY".

1. **Key Expansion:** KEYKEYKEY (matching the length of HELLO).
2. **Conversion:** H (7) + K (10) = Q, E (4) + E (4) = H, L (11) + Y (24) = C, L (11) + K (10) = W, O (14) + E (4) = S.
3. **Ciphertext:** QHCWS

0.3 Code Overview

The implementation consists of two main functions: `keyGeneration` and `ciphertext`, along with a `plaintext` function for decryption. The `main` section demonstrates the usage of these functions.

0.4 Functions

0.4.1 `keyGeneration`

This function generates a key that matches the length of the given message. It takes two parameters, the message (`st`) and the initial key. It ensures that the key aligns with the length of the message.

0.4.2 `ciphertext`

This function performs encryption using the Vigenère cipher. It takes a message (`st`) and a key. It uses the Vigenère cipher formula to encrypt each character and returns the encrypted message.

0.4.3 `plaintext`

This function performs decryption using the Vigenère cipher. It takes an encrypted message (`st`) and a key. It uses the Vigenère cipher decryption formula to decrypt each character and returns the decrypted message.

0.5 Main Program

The `main` section prompts the user to input a message and a key, prints the original message and key, generates the key using `keyGeneration`, encrypts the message using `ciphertext`, and then decrypts it back to the original form using `plaintext`.

0.6 Usage

To run the program, execute the Python script. Enter the message and key when prompted, and the program will display the original message, encrypted message, and decrypted message.

Listing 1: Vigenère Cipher Implementation

```python
# This function is responsible for generating a key that matches the length of the given message.
# It takes two parameters, st (the message) and key (the initial key).
# It first converts the key into a list of characters.
# If the length of the key is equal to the length of the message, it directly returns the key.
# If the length of the key is less than the length of the message,
# it shortens the key by removing characters from the end.
# If the length of the key is greater than the length of the message,
# it repeats characters from the key to match the length of the message.
def keyGeneration(st, key):
    key = list(key)  # Convert the key string into a list of characters
    try:
        if (len(st) == len(key)):  # Check if the length of the key is equal to the length of the message
            return key

        elif (len(st) < len(key)):  # Execute when the length of key is greater than the message
            size = len(st) - 1
            for i in range(len(key) - len(st)):
                del key[size - i]  # Remove excess characters from the end of the key
            return key
        else:  # Execute when the length of key is less than the message
            for i in range(len(st) - len(key)):
                key.append(key[i % len(key)])  # Repeat characters from the key to match the length of the message
            return key

    except:
        print("\nThe key is not aligning with the Message")


# This function takes a message (st) and a key, and it performs encryption using the Vigen re cipher.
```

```python
# It initializes an empty list lst to store the encrypted characters.
# It iterates through each character in the message and performs the encryption using the Vigen re cipher formula:
# (Pi + Ki) mod 26, where Pi is the plaintext character, Ki is the key character,
# and mod 26 ensures that the result remains within the range of the alphabet.
# The encrypted character is then converted back to a character and appended to the lst.
# Finally, the encrypted message is printed and returned.
def ciphertext(st, key):
    lst = []  # Initialize an empty list to store encrypted characters

    for i in range(len(st)):
        if st[i] == ' ':
            lst.append(' ')
        else:
            cipher = (ord(st[i]) + ord(key[i])) % 26  # Vigen re cipher encryption: (Pi + Ki) mod 26
            # Convert the result to ASCII value of 'A' to get the encrypted character
            cipher += ord('A')  # cipher = cipher + ord('A')
            lst.append(chr(cipher))  # Append the encrypted character to the list

    encrypted_message = "".join(lst)  # Join the list of encrypted characters to form the encrypted message
    print("\nThe Cipher TXT is", encrypted_message)
    print("THE cipher text in list :", lst)
    return encrypted_message


# This function takes an encrypted message (st) and a key, and it performs decryption using the Vigen re cipher.
# It follows a similar process as the ciphertext function, but uses the decryption formula: (Pi - Ki) mod 26.
# The decrypted character is then appended to the text list.
# The decrypted message is printed.
def plaintext(st, key):
    text = []  # Initialize an empty list to store decrypted characters

    for i in range(len(st)):
        if st[i] == ' ':
            text.append(' ')
        else:
            plaintxt = (ord(st[i]) - ord(key[i])) % 26  # Vigen re cipher decryption: (Pi - Ki) mod 26
            plaintxt += ord('A')  # Convert the result to ASCII value of 'A' to get the decrypted character
            text.append(chr(plaintxt))  # Append the decrypted character to the list

    decrypted_message = " ".join(text)  # Join the list of decrypted characters to form the decrypted message
```

```python
    print("\nThe PLAIN TXT is: ", decrypted_message, "\n")


if __name__ == "__main__":
    # User input for the message and key, converted to uppercase for consistency
    st = input("Enter the message: ").upper()
    key = input("Enter the key: ").upper()

    # Print the original message and key
    print("THE MESSAGE WAS (", st, ") and The key is :(", key, ")")

    # Generate the key with the keyGeneration function
    key = keyGeneration(st, key)

    # Encrypt the message using the Vigen re cipher and print the result
    st = ciphertext(st, key)

    # Decrypt the message back to its original form and print the result
    plaintext(st, key)
```

0.7 Output

```
THE MESSAGE WAS ( DONT GIVE HARD PROJECTS ) and The key is :( WE HAVE TO DO IT  )

The Cipher TXT is ZSGA BMOX ADFW IKKNXJTN
THE cipher text in list : ['Z', 'S', 'G', 'A', ' ', 'B', 'M', 'O', 'X', ' ', 'A', 'D', 'F', 'W', ' ', '

The PLAIN TXT is:  D O N T   G I V E   H A R D   P R O J E C T S
```

Playfair Cipher Implementation Documentation

NINANIYA NATEESH KUMAR AND FATHIMA FARHAN

January 19, 2024

Contents

0.1 Introduction

The Playfair cipher is a symmetric key substitution cipher that was designed to be more secure than simple substitution ciphers and avoids some of the weaknesses of the traditional cipher. It encrypts pairs of letters (digraphs), instead of single letters as in the case of simple substitution ciphers.

0.2 Playfair Cipher Overview

The Playfair cipher works with a key matrix, typically a 5x5 grid of letters. The key matrix is generated from a keyword provided by the user. The following steps provide an overview of the Playfair cipher:

1. **Key Matrix Generation**: The key matrix is generated by arranging the unique letters of the keyword in a 5x5 grid. The remaining letters of the alphabet are added in order, excluding 'J' (often replaced with 'I').

2. **Encryption Rules**: Pairs of letters in the plaintext (digraphs) are replaced according to the following rules: - If the letters are in the same row, replace each letter with the letter to its right (cycling to the leftmost if at the right edge). - If the letters are in the same column, replace each letter with the letter below it (cycling to the top if at the bottom edge). - If the letters are not in the same row or column, form a rectangle with the two letters and replace each with the letter at the opposite corner of the rectangle.

3. **Decryption Rules**: Decryption follows similar rules to encryption, but using the reverse transformations to obtain the original plaintext.

0.3 Code Overview

The provided Python implementation of the Playfair cipher follows the described process. Let's delve into the details of the key functions:

0.3.1 generate_playfair_key

This function takes a key as input and generates the key matrix used for encryption and decryption in the Playfair cipher.

0.3.2 find_position

This function takes a matrix and a character as input and returns the row and column indices of that character in the matrix. It is used to find the positions of characters in the key matrix during encryption and decryption.

0.3.3 encrypt_playfair

This function encrypts plaintext using the Playfair cipher. It preprocesses the plaintext, divides it into pairs, and applies the Playfair rules for encryption.

0.3.4 decrypt_playfair

This function decrypts ciphertext using the Playfair cipher. It divides the ciphertext into pairs and applies the Playfair rules for decryption.

0.4 Main Program

0.4.1 main

The `main` function prompts the user to input the key and plaintext, generates the key matrix, displays the matrix, encrypts and decrypts the plaintext, and prints the results.

0.5 Usage

To run the program, execute the Python script. Enter the key and plaintext when prompted, and the program will display the key matrix, ciphertext, and decrypted plaintext.

```python
# This function takes a key as input and generates the key matrix
# used for encryption and decryption in the Playfair cipher.
def generate_playfair_key(key):
    # Convert key to uppercase and replace 'J' with 'I'
    key = key.upper().replace('J', 'I')
    # Remove any spaces from the key
    key = key.replace(' ', '')
    # Remove duplicates and maintain order of the characters
    key = ''.join(sorted(set(key), key=key.find))

    alphabet = 'ABCDEFGHIKLMNOPQRSTUVWXYZ'

    # Add remaining letters of the alphabet that are not present in the key
    # ensuring that all letters are accounted for in the key.
    for char in alphabet:
        if char not in key:
            key += char
    #  Divides the key into 5-character segments to create the 5x5 key matrix.
    key_matrix = [key[i:i + 5] for i in range(0, 25, 5)]
    return key_matrix

# This function takes a matrix and a character as input
# and returns the row and column indices of that character in the matrix.
```

```python
# It is used to find the positions of characters in the key matrix during encryption and decryption.
def find_position(matrix, char):
    for i, row in enumerate(matrix):
        if char in row:
            return i, row.index(char)
    return None

# This function encrypts plaintext using the Playfair cipher.
# It first preprocesses the plaintext by converting it to uppercase and replacing 'J' with 'I'.
# If the length of the plaintext is odd, it adds a padding character('X') to make it even.
# The plaintext is then divided into pairs (digraphs).
# For each digraph, the positions in the key matrix are determined, and the corresponding ciphertext digraph is formed.
# The resulting ciphertext is returned.
def encrypt_playfair(plaintext, key):
    # Generate the key matrix
    key_matrix = generate_playfair_key(key)

    # Preprocess plaintext
    plaintext = plaintext.upper().replace('J', 'I')
    # Remove spaces from the plaintext
    plaintext = plaintext.replace(' ', '')
    # Add padding character if the plain text length is odd
    if len(plaintext) % 2 != 0:
        plaintext += "X"
    # Divide plaintext into pairs (digraphs)
    plaintext_pairs = [plaintext[i:i + 2] for i in range(0, len(plaintext), 2)]

    # Encryption
    ciphertext = ''
    for pair in plaintext_pairs:
        # Find positions of characters in the key matrix
        row1, col1 = find_position(key_matrix, pair[0])
        row2, col2 = find_position(key_matrix, pair[1])
        # Apply Playfair rules for encryption
        if row1 == row2:
            ciphertext += key_matrix[row1][(col1 + 1) % 5] + key_matrix[row2][(col2 + 1) % 5]
        elif col1 == col2:
            ciphertext += key_matrix[(row1 + 1) % 5][col1] + key_matrix[(row2 + 1) % 5][col2]
        else:
            ciphertext += key_matrix[row1][col2] + key_matrix[row2][col1]

    return ciphertext

def decrypt_playfair(ciphertext, key):
    # Generates the key matrix using the provided key.
    key_matrix = generate_playfair_key(key)

    # Preprocess ciphertext:Divides the ciphertext into pairs (digraphs) of two characters each.
    ciphertext_pairs = [ciphertext[i:i + 2] for i in range(0, len(ciphertext), 2)]

    # Decryption
    plaintext = ''
```

```
    for pair in ciphertext_pairs:
        # Find positions of characters in the key matrix
        row1, col1 = find_position(key_matrix, pair[0])
        row2, col2 = find_position(key_matrix, pair[1])
        # Apply Playfair rules for decryption
        if row1 == row2:
            plaintext += key_matrix[row1][(col1 - 1) % 5] + key_matrix[row2][(col2 - 1) % 5]
        elif col1 == col2:
            plaintext += key_matrix[(row1 - 1) % 5][col1] + key_matrix[(row2 - 1) % 5][col2]
        else:
            plaintext += key_matrix[row1][col2] + key_matrix[row2][col1]

    return plaintext

# The user is prompted to input the key and plaintext.
# The key matrix is generated using the generate_playfair_key function.
# The key matrix is then displayed to the user.
# The plaintext is encrypted using the Playfair cipher, and the resulting ciphertext is printed.
# The ciphertext is decrypted using the Playfair cipher, and the resulting plaintext is printed.
def main():
    # Take user input for the key and plaintext
    key = input("Enter the key for Playfair cipher: ")
    plaintext = input("Enter the plaintext to encrypt: ")

    length = len(plaintext)

    # Generate the key matrix using the provided key
    key_matrix = generate_playfair_key(key)

    # Display the key matrix
    print("Key Matrix:")
    for row in key_matrix:
        print("[", end=" ")
        print(", ".join(row), end=" ")
        print("]")

    # Encrypt the plaintext using the Playfair cipher
    ciphertext = encrypt_playfair(plaintext, key)

    # Decrypt the ciphertext to verify
    decrypt_text = decrypt_playfair(ciphertext, key)
    if len(decrypt_text) != length:
        pt = decrypt_text[:-1]
    else:
        pt = decrypt_text

    # Print the results
    print("\nCiphertext:", ciphertext)
    print("\nPlaintext:", pt)

if _name_ == "_main_":
    main()
```

Listing 1: Playfair Cipher Implementation

0.6 Output

```
Enter the key for Playfair cipher:  hello
Enter the plaintext to encrypt: how are you

Key Matrix:
[ H, E, L, O, A ]
[ B, C, D, F, G ]
[ I, K, M, N, P ]
[ Q, R, S, T, U ]
[ V, W, X, Y, Z ]
Ciphertext: EAZEWCOFSZ
Plaintext: HOWAREYOU
```

Hill Cipher Implementation Documentation

JATIN AND SHAMEEL C

19 JANUARY 2024

0.1 Hill Cipher

The Hill Cipher is a classical symmetric encryption algorithm that operates on blocks of text, providing a polygraphic substitution approach to secure communication. Developed by Lester S. Hill in 1929, it represents a departure from traditional substitution ciphers by working with matrices and linear algebra concepts. It uses linear algebraic principles to encrypt and decrypt messages. Unlike traditional substitution ciphers, which replace individual letters with other letters or symbols, the Hill Cipher operates on groups of letters.

0.2 Working Principle

The Hill Cipher operates on the principle of linear algebraic transformations applied to blocks of plaintext to produce corresponding blocks of ciphertext. Unlike traditional substitution ciphers, which replace individual letters with others, the Hill Cipher considers multiple letters at once, making it a polygraphic substitution cipher. The core idea is to use a key matrix for encryption and its inverse for decryption.

0.2.1 Matrix Representation

Let's denote the plaintext as a column vector P, the key matrix as K, and the ciphertext as C. The encryption process is represented by the equation:

$$C = K \times P \; mod 26$$

Here, $\times$ denotes matrix multiplication, and $mod 26$ is applied element-wise to ensure the result stays within the range of the alphabet (assuming a standard 26-letter English alphabet). The key matrix K must be a square matrix of size $n \times n$ (where n is the key length) and must be invertible.

0.2.2 Block Processing

The plaintext is divided into blocks of size n, where n is the size of the key matrix. If the length of the plaintext is not a multiple of n, padding is applied. Each block is then treated as a column vector, and the matrix multiplication is performed for each block independently.

0.2.3 Decryption Process

The decryption process involves multiplying the ciphertext by the modular inverse of the key matrix:

$$P = K^{-1} \times C \; mod 26$$

Here, K^{-1} is the modular inverse of K. The modular inverse is crucial, and it exists if and only if the determinant of K is invertible modulo 26. This requirement ensures that the decryption process can recover the original plaintext.

0.2.4 Key Generation

Generating a suitable key for the Hill Cipher involves selecting a square matrix K that is invertible modulo 26. The key space in the Hill Cipher is vast, but not all matrices are suitable keys. The determinant of the key matrix must be coprime to 26, ensuring the existence of a modular inverse.

Finding the Inverse of a Matrix

The modular inverse of a matrix $\mathbf{K}$ can be found using various methods. One common approach involves using the adjugate matrix.

Calculate the determinant $|\mathbf{K}|$.

If $|\mathbf{K}|$ is not relatively prime to the modulus (in this case, 26), the matrix is not invertible.

Calculate the adjugate matrix $adj(\mathbf{K})$.

Find the modular inverse $\mathbf{K}^{-1}$ using:

$$\mathbf{K}^{-1} = |\mathbf{K}|^{-1} \cdot adj(\mathbf{K}) \pmod{26}$$

0.2.5 Security Considerations

The Hill Cipher provides security through the complexity of matrix inversion and the requirement of an invertible key matrix. However, it is vulnerable to certain attacks, such as known-plaintext attacks when an adversary has access to both plaintext and corresponding ciphertext. Careful key management and the use of larger key sizes can enhance the security of the Hill Cipher in practice.

In summary, the Hill Cipher leverages linear algebraic principles to achieve encryption and decryption, making it a unique and interesting member of classical ciphers.

The Hill Cipher uses matrix multiplication over a finite field to transform blocks of plaintext into ciphertext and vice versa. The key to the Hill Cipher is represented as a square matrix. The size of the matrix depends on the key length, and it must be invertible.

Let's denote the plaintext as a column vector P and the key matrix as K. The encryption process is given by the equation:

$$C = K \times P \ mod 26$$

Where: - C is the ciphertext column vector, - $\times$ denotes matrix multiplication, - $mod 26$ is applied element-wise to ensure the result stays within the range of the alphabet.

The decryption process involves multiplying the ciphertext by the modular inverse of the key matrix:

$$P = K^{-1} \times C \ mod 26$$

Here, K^{-1} is the modular inverse of K.

0.3 Functions

0.3.1 up

The `up` function converts lowercase letters in a string to uppercase, ignoring non-alphabetic characters.

Listing 1: Uppercase Function

```python
def up(letter):
    text1 = ""
    for i in letter:
        if ord('a') <= ord(i) <= ord('z'):
            text2 = chr(ord(i) - 32)
            text1 = text1 + text2
        elif ord('A') <= ord(i) <= ord('Z'):
            text2 = i
            text1 = text1 + text2
        else:
            print("Not Alphabet")
    return text1
```

0.3.2 space

The `space` function removes spaces from a given string.

Listing 2: Remove Spaces Function

```
def space(s):
    text = ""
    for i in s:
        if ord(i) != 32:
            text += i
    return text
```

0.3.3 mat

The `mat` function performs matrix multiplication between two matrices.

Listing 3: Matrix Multiplication Function

```
def mat(a, b):
    row1 = len(a)
    row2 = len(b)
    col1 = len(a[0])
    col2 = len(b[0])

    result = [[0, 0, 0], [0, 0, 0], [0, 0, 0]]

    for i in range(row1):
        for j in range(col2):
            for k in range(col1):
                result[i][j] += a[i][k] * b[k][j]

    res = [[0, 0, 0], [0, 0, 0], [0, 0, 0]]
    m = 0
    for row in result:
        res[m] = row
        m += 1

    return res
```

0.3.4 letter_to_number

Converts an uppercase letter to its corresponding numerical value (A=0, B=1, ..., Z=25).

Listing 4: Letter to Number Conversion

```
def letter_to_number(letter):
    return ord(letter) - ord('A')
```

0.3.5 number_to_letter

Converts a numerical value to its corresponding uppercase letter.

Listing 5: Number to Letter Conversion

```
def number_to_letter(number):
    return chr(number + ord('A'))
```

0.3.6 `print_matrix`

Prints a 3x3 matrix.

Listing 6: Print Matrix Function

```
def print_matrix(matrix):
    for row in matrix:
        print(row)
```

0.3.7 `mod_inverse`

Calculates the modular inverse of a modulo m.

Listing 7: Modular Inverse Function

```
def mod_inverse(a, m):
    if m == 0:
        raise ValueError("Cannot perform modular inverse with modulus 0.")
    q = a//m
    m0, x0, x1 = m, 0, 1

    while a > 1:
        q = a // m
        m, a = a % m, m
        x0, x1 = x1 - q * x0, x0

    return x1 + m0 if x1 < 0 else x1
```

0.3.8 `det_inverse`

Calculates the modular inverse of the determinant *det* modulo *mod*.

Listing 8: Determinant Inverse Function

```
def det_inverse(det, mod):
     if mod == 0:
        raise ValueError("Determinant is zero. Cannot compute modular inverse.")
    det_inv = mod_inverse(det, mod)
    return det_inv % mod
```

0.3.9 `adjugate`

Calculates the adjugate (or adjoint) of a 3x3 matrix.

Listing 9: Adjugate Function

```
def adjugate(matrix):
    return [
        [matrix[1][1]*matrix[2][2] - matrix[1][2]*matrix[2][1],
           matrix[0][2]*matrix[2][1] - matrix[0][1]*matrix[2][2],
           matrix[0][1]*matrix[1][2] - matrix[0][2]*matrix[1][1]],
        [matrix[1][2]*matrix[2][0] - matrix[1][0]*matrix[2][2],
           matrix[0][0]*matrix[2][2] - matrix[0][2]*matrix[2][0],
           matrix[0][2]*matrix[1][0] - matrix[0][0]*matrix[1][2]],
```

```
        [matrix[1][0]*matrix[2][1] - matrix[1][1]*matrix[2][0],
           matrix[0][1]*matrix[2][0] - matrix[0][0]*matrix[2][1],
           matrix[0][0]*matrix[1][1] - matrix[0][1]*matrix[1][0]]
    ]
```

0.3.10 inverse_matrix

Calculates the inverse of a 3x3 matrix modulo a given modulus (*mod*).

Listing 10: Inverse Matrix Function

```
def inverse_matrix(matrix, mod):
    det = (matrix[0][0] * (matrix[1][1] * matrix[2][2] - matrix[1][2] *
       matrix[2][1]) -
           matrix[0][1] * (matrix[1][0] * matrix[2][2] - matrix[1][2] *
              matrix[2][0]) +
           matrix[0][2] * (matrix[1][0] * matrix[2][1] - matrix[1][1] *
              matrix[2][0])) % mod
    if det == 0:
        raise ValueError("Determinant is zero. Cannot compute inverse 
           matrix.")

    det_inv = det_inverse(det, mod)

    adj = adjugate(matrix)

    inv_matrix = [[((det_inv * adj[i][j]) % mod + mod) % mod for j in
       range(3)] for i in range(3)]

    return inv_matrix
```

0.3.11 create_key

Creates a 3x3 matrix from the key string.

Listing 11: Create Key Function

```
def create_key(key):
    key = space(key)
    key = up(key)

    matrix = [[0, 0, 0], [0, 0, 0], [0, 0, 0]]
    s = 0
    for i in range(3):
        for j in range(3):
            matrix[i][j] = letter_to_number(key[s])
            s = s + 1

    print("Key Matrix:")
    print_matrix(matrix)

    return matrix
```

0.3.12 encryption

Performs Hill Cipher encryption.

Listing 12: Encryption Function

```
def encryption(plaintext, key):
    plaintext = space(plaintext)
    plaintext = up(plaintext)

    if len(plaintext) % 3 != 0:
        plaintext += 'X' * (3 - len(plaintext) % 3)

    keymat = create_key(key)

    numtext = [letter_to_number(letter) for letter in plaintext]
    et = ""

    for i in range(0, len(numtext), 3):
        block = [[numtext[i]], [numtext[i + 1]], [numtext[i + 2]]]
        encrypted_block = mat(keymat, block)
        encrypted_block = [[num % 26 for num in row] for row in
            encrypted_block]
        et += number_to_letter(encrypted_block[0][0])
        et += number_to_letter(encrypted_block[1][0])
        et += number_to_letter(encrypted_block[2][0])

    return et
```

0.3.13 decryption

Performs Hill Cipher decryption.

Listing 13: Decryption Function

```
def decryption(ct, key):
    ct = space(ct)
    ct = up(ct)

    keymat = create_key(key)
    keymat_inv = inverse_matrix(keymat, 26)

    numtext = [letter_to_number(letter) for letter in ct]
    dt = ""

    for i in range(0, len(numtext), 3):
        block = [[numtext[i]], [numtext[i + 1]], [numtext[i + 2]]]
        decrypted_block = mat(keymat_inv, block)
        decrypted_block = [[num % 26 for num in row] for row in
            decrypted_block]
        dt += number_to_letter(decrypted_block[0][0])
        dt += number_to_letter(decrypted_block[1][0])
        dt += number_to_letter(decrypted_block[2][0])

    return dt
```

0.4 Main Program

Listing 14: Main Program

```
key = input("ENTER THE KEY (9 CHARACTERS)")

while True:
    key_matrix = create_key(key)

    plaintext = input("Enter Plain Text: ")

    ciphertext = encryption(plaintext, key)
    print("Cipher Text: ", ciphertext)

    decrypted_text = decryption(ciphertext, key)
    print("Decrypted Text: ", decrypted_text)
```

0.5 Usage

To run the program, execute the Python script. Enter the key and plaintext when prompted, and the program will display the key matrix, ciphertext, and decrypted plaintext.

Listing 15: Playfair Cipher Implementation

```
# uppercase() function takes a string letter as input and converts any lowercase letters to uppercase.
# It ignores non-alphabetic characters.
# It uses ASCII values to check if a character is a lowercase letter and then converts it to uppercase if needed.
def up(letter):
    text1 = ""  # Initialize an empty string to store the result
    for i in letter:
        # Condition ord('a') <= ord(i) <= ord('z') checks
        # if the ASCII value of i is within the range of lowercase letters
        # If this condition is true,it means 'i' is a lowercase letter

        if ord('a') <= ord(i) <= ord('z'):              # ord() returns the ASCII value.
            # If 'i' is a lowercase letter, this line converts it to uppercase.
            text2 = chr(ord(i) - 32)                    # EX: ASCII value of a-97
            text1 = text1 + text2                       # 97-32=65(ASCII value of A)
        elif ord('A') <= ord(i) <= ord('Z'):
            # If the character is already uppercase,keep it same
            text2 = i
            text1 = text1 + text2
        else:
            # If a character is not an alphabet,print a message
            print("Not Alphabet")
    return text1

# This function removes spaces from a given string s.
```

```python
# It iterates through each character in the string and appends non-space characters to a new string.
def space(s):
    text = ""  # Initialize an empty string to store characters without spaces
    for i in s:
        if ord(i) != 32:  # Check if the ASCII value of the character is not equal to 32 (the ASCII value for space)
            text += i  # Add the non-space character to the 'text' string
    return text  # Return the modified string without spaces


# This function performs matrix multiplication between two matrices a and b.
# It initializes a result matrix result with zeros and then performs the matrix multiplication using nested loops.
# The resulting matrix is then copied to a new matrix res.
def mat(a,b):
    # EX:
    # matrix_a = [[1, 2, 3, 4], [4, 5, 6, 4], [7, 8, 9, 4]]
    # row=len(matrix_a)
    # column=len(matrix_a[0])
    # print(row)
    # print(column)
    # OUTPUT:3 and 4
    row1 = len(a)       # Number of rows in matrix a
    row2 = len(b)       # Number of rows in matrix b
    col1 = len(a[0])    # Number of columns in matrix a
    col2 = len(b[0])    # Number of columns in matrix b

    # Initialize a result matrix with zeros
    result = [[0, 0, 0], [0, 0, 0], [0, 0, 0]]

    # Perform matrix multiplication
    for i in range(row1):
        for j in range(col2):
            for k in range(col1):
                result[i][j] += a[i][k] * b[k][j]            # result[i][j] = result[i][j] + a[i][k] * b[k][j]

    # Copy the result to a new matrix
    res = [[0, 0, 0], [0, 0, 0], [0, 0, 0]]
    m = 0
    for row in result:
        res[m] = row
        m += 1

    return res


# This function converts an uppercase letter to its corresponding numerical value (A=0, B=1, ..., Z=25).
def letter_to_number(letter):
    return ord(letter) - ord('A')
```

```python


# This function converts a numerical value to its corresponding uppercase letter.
def number_to_letter(number):
    return chr(number + ord('A'))


# Prints a 3x3 matrix.
def print_matrix(matrix):
    for row in matrix:
        print(row)


# Calculates the modular inverse of a modulo m. This function is used in the determinant inverse calculation.
def mod_inverse(a, m):
    if m == 0:
        raise ValueError("Cannot perform modular inverse with modulus 0.")

    q = a // m      # Floor operator
    m0, x0, x1 = m, 0, 1

    while a > 1:
        q = a // m  # Floor operator
        m, a = a % m, m
        x0, x1 = x1 - q * x0, x0

    return x1 + m0 if x1 < 0 else x1
# EX:Suppose we want to find the modular inverse of a = 3 modulo m = 11.
# Initialize m0 = 11, x0 = 0, and x1 = 1.
# Enter the loop since a > 1:
# Calculate q = 3 // 11, which is 0.
# Update m = 3 % 11, which is 3, and a = 11.
# Update x0 and x1 using the extended Euclidean algorithm: x0, x1 = 1, -3.
# The loop continues:
# Calculate q = 11 // 3, which is 3.
# Update m = 11 % 3, which is 2, and a = 3.
# Update x0 and x1 using the extended Euclidean algorithm: x0, x1 = -3, 10.
# The loop continues:
# Calculate q = 3 // 2, which is 1.
# Update m = 3 % 2, which is 1, and a = 2.
# Update x0 and x1 using the extended Euclidean algorithm: x0, x1 = 10, -13.
# The loop continues:
# Calculate q = 2 // 1, which is 2.
# Update m = 2 % 1, which is 0, and a = 1.
# Update x0 and x1 using the extended Euclidean algorithm: x0, x1 = -13, 23.
# The loop exits since a is now 1. Finally, the function returns x1 + m0 because x1 is negative.
# RESULT:23+11=

```

```
# Calculates the modular inverse of the determinant det modulo mod.
def det_inverse(det, mod):
    if det == 0:
        raise ValueError("Determinant is zero. Cannot compute modular inverse.")
    # Calculate the modular inverse of the determinant
    det_inv = mod_inverse(det, mod)

    # Return the modular inverse modulo the given modulus
    # The modulo operation ensures that the result is within the range [0, mod - 1].
    return det_inv % mod


# The adjugate function calculates the adjugate (or adjoint) of a 3x3 matrix.
# The adjugate matrix is used in the process of finding the inverse of a matrix.
# It returns a new 3x3 matrix where each element is calculated based on the formula for the adjugate:
# adjugate( A[i,j] ) = (-1)^(i+j).minor(A[i,j])
def adjugate(matrix):
    return [
        [matrix[1][1]*matrix[2][2] - matrix[1][2]*matrix[2][1], matrix[0][2]*matrix[2][1] - matrix[0][1]*matrix[2][2], matrix[0][1]*matrix[1][2] - matrix[0][2]*matrix[1][1]],
        [matrix[1][2]*matrix[2][0] - matrix[1][0]*matrix[2][2], matrix[0][0]*matrix[2][2] - matrix[0][2]*matrix[2][0], matrix[0][2]*matrix[1][0] - matrix[0][0]*matrix[1][2]],
        [matrix[1][0]*matrix[2][1] - matrix[1][1]*matrix[2][0], matrix[0][1]*matrix[2][0] - matrix[0][0]*matrix[2][1], matrix[0][0]*matrix[1][1] - matrix[0][1]*matrix[1][0]]
    ]


# The inverse_matrix function calculates the inverse of a 3x3 matrix modulo a given modulus (mod).
# The steps involve calculating the determinant, finding its modular inverse, computing the adjugate matrix,
# and finally obtaining the inverse matrix.
def inverse_matrix(matrix, mod):
    # Calculate the determinant of the 3x3 matrix
    det = (matrix[0][0] * (matrix[1][1] * matrix[2][2] - matrix[1][2] * matrix[2][1]) -
           matrix[0][1] * (matrix[1][0] * matrix[2][2] - matrix[1][2] * matrix[2][0]) +
           matrix[0][2] * (matrix[1][0] * matrix[2][1] - matrix[1][1] * matrix[2][0])) % mod

    if det == 0:
        raise ValueError("Determinant is zero. Cannot compute inverse matrix.")

```

```
    # Calculate the modular inverse of the determinant
    det_inv = det_inverse(det, mod)

    # Calculate the adjugate matrix
    adj = adjugate(matrix)

    # Calculate the inverse matrix using modular arithmetic
    # + mod) % mod: Add the modulus and take the result modulo the modulus again.
    # This step ensures that the result is non-negative and within the range [0, mod - 1].
    inv_matrix = [[((det_inv * adj[i][j]) % mod + mod) % mod for j in range(3)] for i in range(3)]

    return inv_matrix


# This function takes a key string as input, removes spaces, and converts it to uppercase using the up function.
# It then creates a 3x3 matrix from the key, where each element of the matrix corresponds to the numerical value
# of a letter in the key.
def create_key(key):
    # Remove spaces and convert to uppercase
    key = space(key)
    key = up(key)

    # Initialize a 3x3 matrix with zeros
    matrix = [[0, 0, 0], [0, 0, 0], [0, 0, 0]]
    # Initializes a variable s to keep track of the position in the key string.
    s = 0
    for i in range(3):
        for j in range(3):
            # Assign the numerical value of the corresponding character to the matrix
            matrix[i][j] = letter_to_number(key[s])
            s = s + 1          # Updates the position in the key string.

    # Print the key matrix
    print("Key Matrix:")
    print_matrix(matrix)

    return matrix


# This function performs Hill Cipher encryption.
# It takes a plaintext and a key as input.
# The plaintext is processed to remove spaces and convert to uppercase.
# If the length of the plaintext is not a multiple of 3, it pads the plaintext with 'X' to make it a multiple of 3.
# The key is converted to a matrix using the create_key function.
# The plaintext is divided into blocks of 3 letters each,
# and each block is encrypted using matrix multiplication with the key
```

```
matrix.
# The result is converted back to letters using the number_to_letter function.
def encryption(plaintext, key):
    # Remove spaces and convert to uppercase
    plaintext = space(plaintext)
    plaintext = up(plaintext)

    # Pad the plaintext with 'X' if its length is not a multiple of 3
    if len(plaintext) % 3 != 0:
        plaintext += 'X' * (3 - len(plaintext) % 3)             # plaintext = plaintext + 'X' * (3 - len(plaintext) % 3)

    # Create the key matrix
    keymat = create_key(key)

    # Convert the plaintext to a list of numerical values
    numtext = [letter_to_number(letter) for letter in plaintext]
    # EX:
    # If plaintext is "HELLO", and letter_to_number maps 'A' to 0, 'B' to 1, and so on,
    # then numtext might become [7, 4, 11, 11, 14] based on the numerical values of 'H', 'E', 'L', 'L', 'O' respectively

    # Initialize an empty string for the ciphertext
    et = ""

    # Process the plaintext in blocks of 3
    for i in range(0, len(numtext), 3):
        # Create a 3x1 block from the numerical values
        block = [[numtext[i]], [numtext[i + 1]], [numtext[i + 2]]]

        # Encrypt the block using the key matrix and mat function
        encrypted_block = mat(keymat, block)

        # Apply modulo 26 to each element in the encrypted block
        encrypted_block = [[num % 26 for num in row] for row in encrypted_block]

        # Convert the numerical values back to letters and append to the ciphertext
        et += number_to_letter(encrypted_block[0][0])                # et = et + number_to_letter(encrypted_block[0][0])
        et += number_to_letter(encrypted_block[1][0])
        et += number_to_letter(encrypted_block[2][0])

    # Return the ciphertext
    return et


# The decryption function is the decryption part of the Hill Cipher algorithm.
# It takes a ciphertext (ct) and a key (key) as input and returns the corresponding plaintext (dt).
```

```
def decryption(ct, key):
    # Process the ciphertext
    ct = space(ct)  # Remove spaces from the ciphertext
    ct = up(ct)  # Convert the ciphertext to uppercase

    # Create the key matrix and its inverse
    keymat = create_key(key)
    keymat_inv = inverse_matrix(keymat, 26)

    # Convert the ciphertext letters to numbers
    numtext = [letter_to_number(letter) for letter in ct]

    # Initialize an empty string for the decrypted text
    dt = ""

    # Decrypt the text in blocks of 3 letters
    for i in range(0, len(numtext), 3):
        # Create a block of numbers from the ciphertext
        block = [[numtext[i]], [numtext[i + 1]], [numtext[i + 2]]]

        # Decrypt the block using the inverse key matrix
        decrypted_block = mat(keymat_inv, block)

        # Apply modulo 26 to each element of the decrypted block
        decrypted_block = [[num % 26 for num in row] for row in
           decrypted_block]

        # Convert the numbers back to letters and append to the decrypted
           text
        dt += number_to_letter(decrypted_block[0][0])
        dt += number_to_letter(decrypted_block[1][0])
        dt += number_to_letter(decrypted_block[2][0])

    # Return the decrypted text
    return dt


key = input("ENTER THE KEY(9 CHARACTERS)")

while True:
    # Create the key matrix
    key_matrix = create_key(key)

    # Input plaintext from the user
    plaintext = input("Enter Plain Text: ")

    # Encrypt the plaintext
    ciphertext = encryption(plaintext, key)
    print("Cipher Text: ", ciphertext)

    # Decrypt the ciphertext and print
    decrypted_text = decryption(ciphertext, key)
    print("Decrypted Text: ", decrypted_text)
```

0.6 Output

```
ENTER THE KEY(9 CHARACTERS)TBHIDONTU
Key Matrix:
[19, 1, 7]
[8, 3, 14]
[13, 19, 20]
Enter Plain Text: MY NAME IS JATIN
Key Matrix:
[19, 1, 7]
[8, 3, 14]
[13, 19, 20]
Cipher Text:  FMOOOWZKCXNBPBA
Key Matrix:
[19, 1, 7]
[8, 3, 14]
[13, 19, 20]
Decrypted Text:  MYNAMEISJATINXX
Key Matrix:
[19, 1, 7]
[8, 3, 14]
[13, 19, 20]
Enter Plain Text:
```