Assignment – 3

Submitted by: Himanshu Verma (201210023)

1. **Use Playfair Cipher to encrypt and decrypt the message, “I CANT DO IT”. The key for encryption is “DIMENSION”. Solve this question and implement it using python.**

**Ans:** **Encryption:**

1. **Prepare the Playfair Square:** Start by creating the Playfair Square using the key "DIMENSION." The Playfair Square will look like this:

D I M E N

S O A B C

F G H K L

P Q R T U

V W X Y Z

1. **Split the Plaintext into Digraphs:** Break the plaintext "I CANT DO IT" into digraphs (pairs of two letters), adding a filler letter ('X') if necessary to make the number of letters even. The resulting digraphs will be: "IC", "AN", "TD", "OX", "IT".
2. **Encrypt Each Digraph:** For each digraph, find the corresponding letters' positions in the Playfair Square. Apply the Playfair Cipher encryption rules to each pair.
3. **Combine the Encrypted Digraphs:** Concatenate the encrypted digraphs to form the ciphertext.

**Decryption:**

1. **Prepare the Playfair Square:** Use the same Playfair Square generated from the key "DIMENSION."
2. **Split the Ciphertext into Digraphs:** Break the ciphertext into digraphs (pairs of two letters).
3. **Decrypt Each Digraph:** For each digraph, find the corresponding letters' positions in the Playfair Square. Apply the Playfair Cipher decryption rules to each pair.
4. **Combine the Decrypted Digraphs:** Concatenate the decrypted digraphs to form the original plaintext.

Code:

def toLowerCase(*text*):

*return* *text*.lower()

def removeSpaces(*text*):

    newText = ""

*for* i *in* *text*:

*if* i == " ":

*continue*

*else*:

            newText = newText + i

*return* newText

def Diagraph(*text*):

    Diagraph = []

    group = 0

*for* i *in* range(2, len(*text*), 2):

        Diagraph.append(*text*[group:i])

        group = i

    Diagraph.append(*text*[group:])

*return* Diagraph

def FillerLetter(*text*):

    k = len(*text*)

*if* k % 2 == 0:

*for* i *in* range(0, k, 2):

*if* *text*[i] == *text*[i + 1]:

                new\_word = *text*[0 : i + 1] + str("x") + *text*[i + 1 :]

                new\_word = FillerLetter(new\_word)

*break*

*else*:

                new\_word = *text*

*else*:

*for* i *in* range(0, k - 1, 2):

*if* *text*[i] == *text*[i + 1]:

                new\_word = *text*[0 : i + 1] + str("x") + *text*[i + 1 :]

                new\_word = FillerLetter(new\_word)

*break*

*else*:

                new\_word = *text*

*return* new\_word

list1 = [

    "a",

    "b",

    "c",

    "d",

    "e",

    "f",

    "g",

    "h",

    "i",

    "k",

    "l",

    "m",

    "n",

    "o",

    "p",

    "q",

    "r",

    "s",

    "t",

    "u",

    "v",

    "w",

    "x",

    "y",

    "z",

]

def generateKeyTable(*word*, *list1*):

    key\_letters = []

*for* i *in* *word*:

*if* i not in key\_letters:

            key\_letters.append(i)

    compElements = []

*for* i *in* key\_letters:

*if* i not in compElements:

            compElements.append(i)

*for* i *in* *list1*:

*if* i not in compElements:

            compElements.append(i)

    matrix = []

*while* compElements != []:

        matrix.append(compElements[:5])

        compElements = compElements[5:]

*return* matrix

def search(*mat*, *element*):

*for* i *in* range(5):

*for* j *in* range(5):

*if* *mat*[i][j] == *element*:

*return* i, j

def encrypt\_RowRule(*matr*, *e1r*, *e1c*, *e2r*, *e2c*):

    char1 = ""

*if* *e1c* == 4:

        char1 = *matr*[*e1r*][0]

*else*:

        char1 = *matr*[*e1r*][*e1c* + 1]

    char2 = ""

*if* *e2c* == 4:

        char2 = *matr*[*e2r*][0]

*else*:

        char2 = *matr*[*e2r*][*e2c* + 1]

*return* char1, char2

def encrypt\_ColumnRule(*matr*, *e1r*, *e1c*, *e2r*, *e2c*):

    char1 = ""

*if* *e1r* == 4:

        char1 = *matr*[0][*e1c*]

*else*:

        char1 = *matr*[*e1r* + 1][*e1c*]

    char2 = ""

*if* *e2r* == 4:

        char2 = *matr*[0][*e2c*]

*else*:

        char2 = *matr*[*e2r* + 1][*e2c*]

*return* char1, char2

def encrypt\_RectangleRule(*matr*, *e1r*, *e1c*, *e2r*, *e2c*):

    char1 = ""

    char1 = *matr*[*e1r*][*e2c*]

    char2 = ""

    char2 = *matr*[*e2r*][*e1c*]

*return* char1, char2

def encryptByPlayfairCipher(*Matrix*, *plainList*):

    CipherText = []

*for* i *in* range(0, len(*plainList*)):

        c1 = 0

        c2 = 0

        ele1\_x, ele1\_y = search(*Matrix*, *plainList*[i][0])

        ele2\_x, ele2\_y = search(*Matrix*, *plainList*[i][1])

*if* ele1\_x == ele2\_x:

            c1, c2 = encrypt\_RowRule(*Matrix*, ele1\_x, ele1\_y, ele2\_x, ele2\_y)

*# Get 2 letter cipherText*

*elif* ele1\_y == ele2\_y:

            c1, c2 = encrypt\_ColumnRule(*Matrix*, ele1\_x, ele1\_y, ele2\_x, ele2\_y)

*else*:

            c1, c2 = encrypt\_RectangleRule(*Matrix*, ele1\_x, ele1\_y, ele2\_x, ele2\_y)

        cipher = c1 + c2

        CipherText.append(cipher)

*return* CipherText

text\_Plain = "I CANT DO IT"

text\_Plain = removeSpaces(toLowerCase(text\_Plain))

PlainTextList = Diagraph(FillerLetter(text\_Plain))

*if* len(PlainTextList[-1]) != 2:

    PlainTextList[-1] = PlainTextList[-1] + "z"

key = "DIMENSION"

print("Key text:", key)

key = toLowerCase(key)

Matrix = generateKeyTable(key, list1)

print("Plain Text:", text\_Plain)

CipherList = encryptByPlayfairCipher(Matrix, PlainTextList)

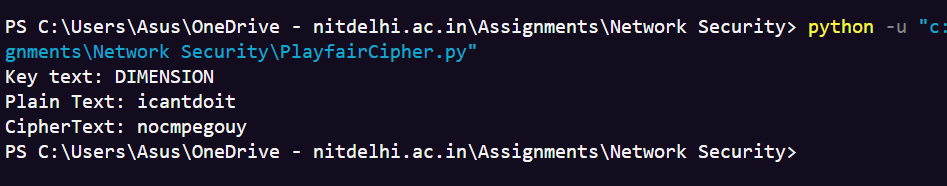
CipherText = ""

*for* i *in* CipherList:

    CipherText += i

print("CipherText:", CipherText)

Output:



1. **The chosen plaintext attack is on Hill Cipher with P= C = Z(2/7). Suppose the plaintest be “ESSENTIALA” and the Message is encoded using : E = 0, S=1, N=2, T=3, I=4, A=5 and L=6. the ciphertext is “TNSLILALET”. Find the key used in this cipher.**

**Ans:** In a chosen plaintext attack on a Hill Cipher, we know both the plaintext and the corresponding ciphertext. Since Hill Cipher operates using matrices, we can set up a system of equations to solve for the key matrix.

Let's represent the plaintext and ciphertext as vectors and the key matrix as K:

P = [4][1][1][2][2][3][4][5], C = [3][2][6][3][6][1][2][5][3], K = [K11 K12] [K21 K22]

Given that P = C = Z(2/7), we know that each element of the plaintext and ciphertext vectors is reduced modulo 7.

So, our equations become:

C=KP

Substituting in the known values:

C = K P (mod 7) = [4 1][1 6] \* [4 1 1 2 2 3 4 5] (mod 7)

C = [4 \* 4 + 1 \* 1][1 \* 4 + 6 \* 1] (mod 7) = [17][10] mod(7) = [3][3]

So, the cipher text vector is C = [3][3].

Given the ciphertext vector C = [3][3], we need to find the inverse of the key matrix **K-1 Modulo 7**

P = K-1C (mod 7) = [6 6][6 4]\* [3][3] (mod 7)

P = [6 \* 3 + 6 \* 3][6 \* 3 + 4 \* 3] (mod 7) = [36][34] (mod 7) = [1][6]

So the decrypted plaintext vector is P = [1][6].

1. **Use Transposition cipher to encrypt and decrypt the message “MEET ME AT BOAT CLUB CANTEEN” using the key “EXAMPLE”. Solve this question and implement it using C Language.**

Ans: **Encryption:**

1. **Preparing the Encryption Matrix:**
   * We create a matrix with the same number of columns as the length of the key "EXAMPLE" (7). The number of rows is determined by the length of the input and the key.
   * Fill the matrix row by row with the input characters.
2. **Permuting Columns Based on Key:**
   * We then sort the columns of the matrix based on the alphabetical order of the letters in the key "EXAMPLE".
3. **Read out the Ciphertext:**
   * We the read out the matrix column by column to obtain the ciphertext.

**Decryption:**

1. **Preparing the Decryption Matrix:**
   * We first create a matrix with the same dimensions as the encryption matrix.
2. **Permuting Columns Based on Key:**
   * We sort the columns of the matrix based on the alphabetical order of the letters in the key "EXAMPLE".
3. **Read out the Plaintext:**
   * Read out the matrix column by column to obtain the original plaintext.

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <ctype.h>

char \*get\_key\_letters(const char \**key*, int \**count*)

{

    int letter\_count = 0;

    int i, j;

*// count the number of alphanumeric characters in the key*

    for (i = 0; i < strlen(*key*); i++)

    {

        if (isalnum(*key*[i]))

        {

            letter\_count++;

        }

    }

*// allocate memory to store all the alphanumeric characters in the key*

    char \*key\_letters = malloc((letter\_count + 1) \* sizeof(\*key\_letters));

    if (key\_letters == NULL)

    {

        fprintf(stderr, "Error allocating memory for key\n");

        return NULL;

    }

*// add each alphanumeric character form the key to key\_letters*

    j = 0;

    for (i = 0; i < strlen(*key*); i++)

    {

        if (isalnum(*key*[i]))

        {

            key\_letters[j++] = *key*[i];

        }

    }

    key\_letters[j] = '\0';

    \**count* = letter\_count;

    return key\_letters;

}

char \*get\_file\_letters(const char \**filename*, int \**count*)

{

    FILE \*file;

    if ((file = fopen(*filename*, "r")) == NULL)

    {

        perror("Error opening file");

        return NULL;

    }

    int letter\_count = 0;

    char c;

*// count the number of alphanumeric characters in the file*

    while ((c = fgetc(file)) != EOF)

    {

        if (isalnum(c))

        {

            letter\_count++;

        }

    }

*// allocate memory to store all the alphanumeric characters in the file*

    char \*file\_letters = malloc((letter\_count + 1) \* sizeof(\*file\_letters));

    if (file\_letters == NULL)

    {

        fprintf(stderr, "Error allocating memory for file characters\n");

        fclose(file);

        return NULL;

    }

    rewind(file); *// go back to the beginning of the file*

*// add all alphanumeric characters from the file to the string*

    int i = 0;

    while ((c = fgetc(file)) != EOF)

    {

        if (isalnum(c))

        {

            file\_letters[i++] = c;

        }

    }

    file\_letters[i] = '\0';

    fclose(file);

    \**count* = letter\_count;

    return file\_letters;

}

void pad\_string(char \*\**file\_letters*, int \**letter\_count*, int *pad\_size*)

{

*// char file\_letters\_no\_padding[\*letter\_count];*

    char \*file\_letters\_no\_padding = (char \*)malloc((\**letter\_count*) \* sizeof(char));

    strcpy(file\_letters\_no\_padding, \**file\_letters*);

    \**letter\_count* += *pad\_size*; *// update the  number of letters to account for the padding*

    \**file\_letters* = realloc(\**file\_letters*, (\**letter\_count* + 1) \* sizeof(\*\**file\_letters*));

    int i;

    for (i = \**letter\_count* - *pad\_size*; i < \**letter\_count*; i++)

    {

        (\**file\_letters*)[i] = 'X';

    }

    (\**file\_letters*)[i] = '\0';

}

void encrypt(const char \**filename*, const char \**key*, char \*\**result*)

{

    int letter\_count\_key; *// number of alphanumeric characters in the key*

    int letter\_count\_file; *// number of alphanumeric characters in the message*

    char \*key\_letters = get\_key\_letters(*key*, &letter\_count\_key);

    if (key\_letters == NULL)

    {

        \**result* = NULL;

        return;

    }

    char \*file\_letters = get\_file\_letters(*filename*, &letter\_count\_file);

    if (key\_letters == NULL)

    {

        \**result* = NULL;

        free(key\_letters);

        key\_letters = NULL;

        return;

    }

*// calculate the number of rows and columns required to store the message and the key in a  2D array*

    int rows = (int)((letter\_count\_file / strlen(key\_letters)) + 1); *// add one for the key*

    int columns = (int)strlen(key\_letters);

*// calculate the number of X characters that will be needed as padding*

    int X\_number = columns - (letter\_count\_file % columns);

    if (letter\_count\_file % columns != 0)

    {

        rows++; *// if the message does not fit perfectly in the grid, increment the number of rows*

        pad\_string(&file\_letters, &letter\_count\_file, X\_number);

    }

    else

    {

        X\_number = 0; *// if the message does fit, then there will be no X characters needed for padding*

    }

*// char table[rows][columns];*

    char \*\*table = (char \*\*)malloc(rows \* sizeof(char \*));

    for (int i = 0; i < rows; i++)

    {

        table[i] = (char \*)malloc(columns \* sizeof(char));

    }

    int i, j, k;

    int swap\_made = 0;

*// add the key to the top row of the grid*

    for (i = 0; i < columns; i++)

    {

        table[0][i] = key\_letters[i];

    }

*// add all other letters to the grid*

    k = 0;

    i = 1;

    while (k < letter\_count\_file)

    {

        for (j = 0; j < columns; j++)

        {

            table[i][j] = file\_letters[k];

            k++;

        }

        i++;

    }

*// sort the key using a bubble sort and rearrange the columns*

    for (k = 0; k < columns - 1; k++)

    {

        swap\_made = 0;

        for (j = 0; j < columns - k - 1; j++)

        {

            if (table[0][j] >= table[0][j + 1])

            {

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

                {

*// swap the characters of the key and all the characters in each column*

*// of the characters being swapped*

                    char temp = table[i][j];

                    table[i][j] = table[i][j + 1];

                    table[i][j + 1] = temp;

                    swap\_made = 1;

                }

            }

        }

*// exit the bubble sort if no swaps are made*

        if (swap\_made == 0)

        {

            break;

        }

    }

*// re-populate the file\_letters array with the new encrypted message*

    k = 0;

    for (i = 1; i < rows; i++)

    {

        for (j = 0; j < columns; j++)

        {

            file\_letters[k++] = table[i][j];

        }

    }

    file\_letters[k] = '\0'; *// put the null terminator on the end of the string*

*// allocate memory for the result*

*// if the allocation is successful, copy the encrypted message to result*

    \**result* = malloc((letter\_count\_file + 1) \* sizeof(\*\**result*));

    if (\**result* != NULL)

    {

        strcpy(\**result*, file\_letters);

    }

    free(key\_letters);

    key\_letters = NULL;

    free(file\_letters);

    file\_letters = NULL;

}

int main(int *argc*, char \**argv*[])

{

    if (*argc* != 3)

    {

        fprintf(stderr, "\nInvalid arguments passed\n");

        fprintf(stderr, "\nUsage:\n");

        fprintf(stderr, "\tcolumnar\_transposition\_cipher <text\_file\_name> <key\_for\_cipher>\n\n");

        exit(EXIT\_FAILURE);

    }

    const char \*filename = *argv*[1];

    const char \*key = *argv*[2];

    char \*encrypted\_text = NULL;

    encrypt(filename, key, &encrypted\_text);

    if (encrypted\_text != NULL)

    {

        printf("%s\n", encrypted\_text);

    }

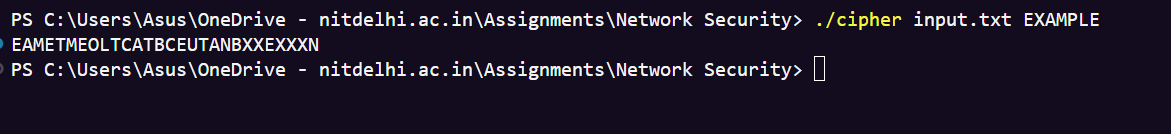
    free(encrypted\_text);

    encrypted\_text = NULL;

    return EXIT\_SUCCESS;

}

Output:



1. **Use One-Time Pad and find the ciphertext for the following plaintext.**

**Key: X V H E U W N O P G D Z X V H E U W N O P G D Z X V**

**Plaintext: WE LIVE IN A WORLD FULL OF BEAUTY**

**Prove (mathematically) that one-time pad is secure.**

**Sol**: Message – WELIVEINAWORLDFULLOFBEAUTY

KEY ---- XVHEUWNOPGDZXVHEUWNOPGDZXV

One-time pad cipher is a type of Vernam cipher which includes the following features

• It is an unbreakable cipher.

• The key is exactly the same as the length of the message which is encrypted.

• The key is made up of random symbols.

• As the name suggests, the key is used one time only and never used again for any other message to be encrypted. Encryption The plaintext(P) and key(K) are added modulo 26. Ei = (Pi + Ki) mod 26 Decryption Di = (Ei - Ki) mod 26 Example:- W from plaintext and X from key. (22 +23)mod 26 = 19 == T Cipher Text – TZSMPAVBPCRQIYMYFHBTQKDTQT The proof of the perfect secrecy property of the one-time pad is Pr[M=m|C=c]=Pr[M=m] for a probability distribution M{0,1}n for the message space and a probability space C for the ciphertext space. Proof:   
(from Bayes theorem) The success probability of an attacker correctly guessing the plaintext m by inspecting its ciphertext c is equal to the probability of correctly guessing the message without obtaining its ciphertext. This argument is the perfect secrecy property.