Experiment-1

Aim: Design and implementation of product cipher using substitution and Transposition ciphers.

Objective:

1. Design and implement substitution and Transposition ciphers.
2. To study the type of substitution and Transposition ciphers.

Outcomes:

1. Learned the importance of substitution and Transposition ciphers.
2. Learned and implemented both ciphers.

Theory:

Substitution cipher is a method of encryption by which units of plaintext are replaced with ciphertext according to a regular system; The "units" may be single letters (the most common), pairs of letters, triplets of letters, mixtures of the above, and so forth. The receiver deciphers the text by performing an inverse substitution.

Transposition cipher is a method of encryption by which the positions held by units of plaintext (which are commonly characters or groups of characters) are shifted according to a regular system, so that the ciphertext constitutes a permutation of the plaintext. That is, the order of the units is changed.

Substitution ciphers can be compared with Transposition ciphers. In a transposition cipher, the units of the plaintext are rearranged in a different and usually quite complex order, but the units themselves are left unchanged. By contrast, in a substitution cipher, the units of the plaintext are retained in the same sequence in the ciphertext, but the units themselves are altered.

Parameters	**Substitution Cipher**	**Transportation Cipher**
Definition	A substitution technique is one in which the letters of plain text are replaced by other letters or numbers or symbols.	Transposition cipher does not substitute one symbol for another instead it changes the location of the symbols

Type	Monoalphabetic and Polyalphabetic substitution cipher.	Keyless and Keyed transportation cipher.
Changes	Each letter retains its position changes its identity	Each letter retains its identity but changes its position
Disadvantage	The last letters of the alphabet which are mostly low frequency tend to stay at the end.	Keys very close to the correct key will reveal long sections of legible plaintext
Example	Caesar Cipher	Rail fence Cipher

Substitution Cipher

Additive cipher:

The simplest mono-alphabetic cipher is the additive cipher. This cipher is sometimes called a shift cipher and sometimes Caesar cipher, but the term additive cipher better reveals its mathematical nature. Assume that the plain text consists of lowercase letters and cipher text of uppercase letters. For mathematical operations on plaintext and cipher text assign numerical values to each letter.

C= (P+k) mod26 for encryption

P= (C-k) mod 26 for decryption.

C= Cipher text:

P= Plain text:

K= Key

When the cipher is additive, the plaintext, cipher text, and key are integers in Z-26

Plain text:

a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z

Cipher text:

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z

Value:

00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25

Encrypt the message **"hello"** using additive cipher with key = **15**

Apply the encryption algorithm to the plain-text, character by character

Plain text: h-07	Encryption: (07+15)mod26	Cipher text : 22→W
Plain text: e -04	Encryption: (04+15)mod26	Cipher text : 19→T
Plain text: l-11	Encryption: (11+15)mod26	Cipher text : 00→A
Plain text: l-11	Encryption: (11+15)mod26	Cipher text : 00→A
Plain text: o-14	Encryption: (14+15)mod26	Cipher text : 03→D

Transposition cipher:

Example:

The transformation can be represented by aligning two alphabets; the cipher alphabet is the plain alphabet rotated left or right by some number of positions. For instance, here is a Caesar cipher using a left rotation of three places (the shift parameter, here 3, is used as the key):

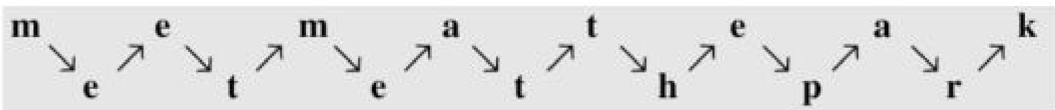
Plain: ABCDEFGHIJKLMNOPQRSTUVWXYZ

Cipher: DEFGHIJKLMNOPQRSTUVWXYZABC

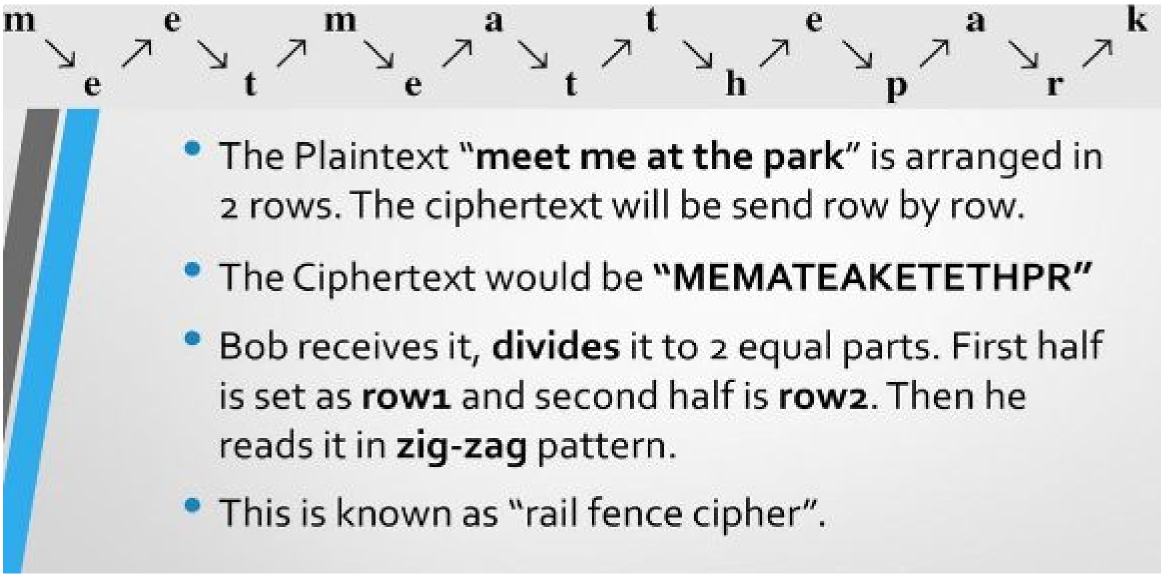
Keyless Transposition Ciphers

Simple transposition ciphers, which were used in the past, are keyless.

A good example of a keyless cipher using the first method is the rail fence cipher. The ciphertext is created reading the pattern row by row. For example, to send the message "Meet me at the park" to Bob, Alice writes



She then creates the ciphertext "MEMATEAKETETHPR".

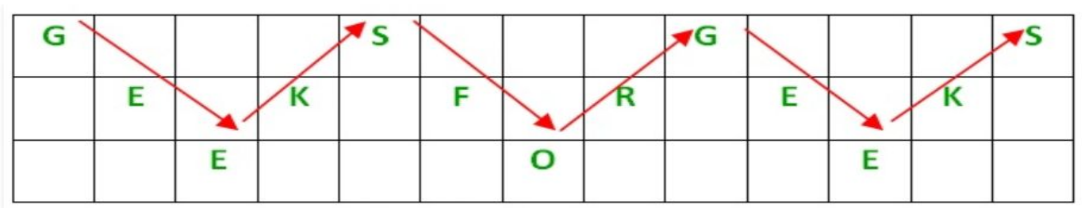


Rail Fence Cipher Techniques

Encryption

In a transposition cipher, the order of the alphabets is rearranged to obtain the cipher-text.

- In the rail fence cipher, the plain-text is written downwards and diagonally on successive rails of an imaginary fence.
- When we reach the bottom rail, we traverse upwards moving diagonally, after reaching the top rail, the direction is changed again. Thus the alphabets of the message are written in a zig-zag manner.
- After each alphabet has been written, the individual rows are combined to obtain the cipher-text.



Decryption

As we’ve seen earlier, the number of columns in rail fence cipher remains equal to the length of plain-text message. And the key corresponds to the number of rails.

- Hence, rail matrix can be constructed accordingly. Once we’ve got the matrix we can figure-out the spots where texts should be placed (using the same way of moving diagonally up and down alternatively).
- Then, we fill the cipher-text row wise. After filling it, we traverse the matrix in zig-zag manner to obtain the original text.

Programme:

```
import random, string, sys
import math
#A custom character map table of 65 characters and which are mapped in 65 int range
char_std_65 = {'0': 0, '1': 1, '2': 2, '3': 3, '4': 4, '5': 5, '6': 6, '7': 7, '8': 8, '9': 9,
        'A': 10, 'B': 11, 'C': 12, 'D': 13, 'E': 14, 'F': 15, 'G': 16, 'H': 17, 'I': 18,
        'J': 19, 'K': 20, 'L': 21, 'M': 22, 'N': 23, 'O': 24, 'P': 25, 'Q': 26, 'R': 27,
        'S': 28, 'T': 29, 'U': 30, 'V': 31, 'W': 32, 'X': 33, 'Y': 34, 'Z': 35, 'a': 36, 'b': 37,
        'c': 38, 'd': 39, 'e': 40, 'f': 41, 'g': 42, 'h': 43, 'i': 44, 'j': 45, 'k': 46, 'l': 47,
        'm': 48, 'n': 49, 'o': 50, 'p': 51, 'q': 52, 'r': 53,'s': 54, 't': 55, 'u': 56, 'v': 57,
        'w': 58, 'x': 59, 'y': 60, 'z': 61, ' ': 62, ',': 63, '.': 64}

def _getKey(keyName):
  '''
    Function for retrieving character from the char-map table using it's numeric value
  '''
  return list(char_std_65.keys())[list(char_std_65.values()).index(keyName)]
```

```
class Encryption:
    '''
        MCA0135 Product cipher
    '''
    plain_text = ''
    key = ''
    transposition_key = ''

    def __init__(self, plain_text, key, transposition_key):
        self.plain_text = plain_text
        self.key = key
        self.transposition_key = transposition_key

    def addRoundKey(self, plain_text):
        '''
          The addRoundKey function will xor plain text with key in character level,
          Then the xore value is wrapped between 0 and 65 to match with our finite 65 character
map table'''
        xored = []
        for i in range(0, len(plain_text)):
            char_in_pt = char_std_65[plain_text[i]]
            char_in_key = char_std_65[self.key[i]]
            xored_value = _getKey((char_in_pt ^ char_in_key) % 65)

            xored.append(xored_value)
        return ''.join(xored)
```

```
    def oneTimePad(self, message):
        '''
            The One-Time Pad encrypt function will encrypt a message using the randomly generated private key that is then decrypted by the receiver using a matching one-time pad and key
        '''
        cipher = ''
        for c in range(0, len(self.key)):
            #Sum of key and message value is wrapped between 0 and 65 to use our finite char field
            subst_value = (char_std_65[message[c]] + char_std_65[self.key[c]]) % 65
            cipher = cipher + _getKey(subst_value)
        return cipher


    def rowTransposition(self, message):
        # Each string in ciphertext represents a column in the grid.
        cipher_text = [''] * self.transposition_key
        # Loop through each column in ciphertext.
        for col in range(self.transposition_key):
            pointer = col
            # Keep looping until pointer goes past the length of the message
            while pointer < len(message):
                # Place the character at pointer in message at the end of the
                # current column in the ciphertext list.
                cipher_text[col] += message[pointer]
                # move pointer over
                pointer += self.transposition_key
        return ''.join(cipher_text)
```

```
    def railFenceCipher(self, message):
        '''
            The railFenceCipher function will write message letters out diagonally
            over a number of rows. Then read off cipher by row.
        '''
        upper_row = ''
        lower_row = ''
        for m in range(1, len(message)+1):
            #Here we are reading from the grid with two rows but usually
            #as many rows as the key is, and as many columns as the length of the ciphertext.
            if (m % 2 != 0):
                upper_row = upper_row + message[m-1]
            else:
                lower_row = lower_row + message[m-1]
        return upper_row + lower_row


    def endToEndEncryptionProcess(self):
        '''
           The endToEndEncryptionProcess function will execute the whole end to end
execution of
           the algorithm round by round and provide the cipher text.
        '''
        cipher_text = self.addRoundKey(self.plain_text)
        encry_logs = []
        encry_logs.append('Cipher text after addRoundkey: "{}"'.format(cipher_text))
        '''
```

```
            first round - substitution
        '''
        cipher_text = self.oneTimePad(cipher_text)
        encry_logs.append('cipher text after first round(one-time pad):
"{}"'.format(cipher_text))
        '''
            second round - transposition
        '''
        cipher_text = self.rowTransposition(cipher_text)
        encry_logs.append('Cipher text after rowTransposition in the second round:
"{}"'.format(cipher_text))
        cipher_text = self.railFenceCipher(cipher_text)
        encry_logs.append('Final cipher text after railFenceCipher in the second round:
"{}"'.format(cipher_text))
        _log('ENCRYPTION', encry_logs)
        return cipher_text


class Decryption:
    cipher_text = ''
    key = ''
    transposition_key = ''


    def __init__(self, cipher_text, key, transposition_key):
        self.cipher_text = cipher_text
        self.key = key
        self.transposition_key = transposition_key


    def reverseRailFenceCipher(self, message):
```

```
    '''
      The reverseRailFenceCipher function will decrypt the message.
    '''
    #The middle index for splitting the cipher
    split_index = int(len(message)/2 + 1) if len(message) % 2 != 0 else int(len(message)/2)
    reverse_text = ''
    for i in range(0, split_index):
        #Reads the character from the first half and the second half in a
        reverse_text = reverse_text + message[i]
        if (split_index + i) <= len(message)-1:
            reverse_text = reverse_text + message[split_index + i]
    return reverse_text

def reverseRowTransposition(self, message):
    '''
     The transposition decrypt function will simulate the "columns" and
     "rows" of the grid that the plaintext is written on by using a list
     of strings.
    '''
    #The number of "columns" in our transposition grid:
    numOfColumns = math.ceil(len(message) / self.transposition_key)
    # The number of "rows" in our grid will need:
    numOfRows = self.transposition_key
    # The number of "shaded boxes" in the last "column" of the grid:
    numOfShadedBoxes = (numOfColumns * numOfRows) - len(message)
    # Each string in plaintext represents a column in the grid.
    plaintext = [''] * numOfColumns
```

```
        # The col and row variables point to where in the grid the next character in the encrypted message will go.
        col = 0
        row = 0

        for symbol in message:
            plaintext[col] += symbol
            col += 1 # point to next column
            # If there are no more columns OR we're at a shaded box, go back to the first column and the next row.
            if (col == numOfColumns) or (col == numOfColumns - 1 and row >= numOfRows - numOfShadedBoxes):
                col = 0
                row += 1
        return ''.join(plaintext)

    def reverseOneTimePad(self, message):
        plain_text = ''
        for c in range(0, len(self.key)):
            rev_value = (char_std_65[message[c]] + 65) - char_std_65[self.key[c]]
            if rev_value > 65:
                rev_value = (char_std_65[message[c]] - char_std_65[self.key[c]])
            plain_text = plain_text + _getKey(rev_value)
        return plain_text

    def reverseAddRoundKey(self, message):
        xored = []
        for i in range(0, len(message)):
```

```
            char_in_ct = char_std_65[message[i]]

            char_in_key = char_std_65[self.key[i]]

            if char_in_key == 65 or char_in_key == char_in_ct:

                xored_value = _getKey((char_in_ct + 65 ^ char_in_key))

            else:

                xored_value = _getKey((char_in_ct ^ char_in_key))

            xored.append(xored_value)

        return ''.join(xored)


    def endToEndDecryptionProcess(self):

        rev_text = self.reverseRailFenceCipher(self.cipher_text)

        decry_logs = []

        decry_logs.append('Cipher text after reverseRailFenceCipher operation:
"{}"'.format(rev_text))

        rev_text = self.reverseRowTransposition(rev_text)

        decry_logs.append('Cipher text after reverseRowTransposition operation:
"{}"'.format(rev_text))

        rev_text = self.reverseOneTimePad(rev_text)

        decry_logs.append('Cipher text after reverseOneTimePad operation:
"{}"'.format(rev_text))

        rev_text = self.reverseAddRoundKey(rev_text)

        decry_logs.append('Plain text after reverseAddRoundKey operation:
"{}"'.format(rev_text))

        _log('DECRYPTION' ,decry_logs)


def _log(title, content):

    '''

       Function for logging all the traces in a wrapped box.
```

```
    '''
    msg_size = max(len(word) for word in content) #msg_size/2
    msg_half_size = int((msg_size/2)+1) if msg_size % 2 !=0 else int(msg_size/2)
    title_size = len(title)
    title_half_size = int(title_size/2)+1 if title_size % 2 !=0 else int(title_size/2)
    title_pos = (msg_half_size-title_half_size)
    print('+'+'-' * (msg_size + 2)+'+')
    print('|{}{}{}|'.format(' '*(msg_half_size-title_half_size),title, ' '*(msg_size-(title_pos+title_size)+2)))
    for word in content:
        print('| {:<{}} |'.format(word, msg_size))
    print('+'+'-' * (msg_size + 2)+'+')


if __name__ == '__main__':
    plain_text = input('Please enter a message for encryption:')
    key = ''.join(random.choice(string.ascii_uppercase + string.ascii_lowercase + string.digits) for _ in range(len(plain_text)))
    row_transposition_key = random.randrange(2 , (int(len(key)/2)+1))
    encryption = Encryption(plain_text, key, row_transposition_key)
    print('Plain message for encryption: "{}" & Key: "{}"'.format(plain_text, key)) #& rowTransposition key: {}
    cipher_text = encryption.endToEndEncryptionProcess()
    decryption = Decryption(cipher_text, key, row_transposition_key)
    decryption.endToEndDecryptionProcess()
```

Output screenshot:

```
C:\Users\91937\Desktop\py4e>python producttransposition.py
Please enter a message for encryption:meet me in the park
Plain message for encryption: "meet me in the park" & Key: "nOi69fGxfEVvbhJLJaA"
+-------------------------------------------------------------------------------------+
|                                     ENCRYPTION                                      |
| Cipher text after addRoundkey: "1m4ntPu55,XEE3jctHa"                                |
| cipher text after first round(one-time pad): "o7mt.17.kC.6pk.x9rk"                  |
| Cipher text after rowTransposition in the second round: "o7pk7.kmk.tCx..916r"       |
| Final cipher text after railFenceCipher in the second round: "op7kktx.1r7k.m.C.96"  |
+-------------------------------------------------------------------------------------+
+-----------------------------------------------------------------------------+
|                                 DECRYPTION                                  |
| Cipher text after reverseRailFenceCipher operation: "o7pk7.kmk.tCx..916r"   |
| Cipher text after reverseRowTransposition operation: "o7mt.17.kC.6pk.x9rk"  |
| Cipher text after reverseOneTimePad operation: "1m4ntPu55,XEE3jctHa"        |
| Plain text after reverseAddRoundKey operation: "meet me in the park"        |
+-----------------------------------------------------------------------------+

C:\Users\91937\Desktop\py4e>
```

Conclusion: We were able to implement product transposition successfully in python. We also learned the theory and implementation of Substitution and Transposition cipher.

References:

1. https://www.geeksforgeeks.org/
2. https://www.tutorialspoint.com/index.htm