Experiment 01: Implement and design the product cipher using Substitution and Transposition ciphers

Experiment 01: (a) Substitution Cipher

Learning Objective: Implement and design the product cipher using Substitution Cipher

Tools: PyCharm

Theory:

Substitution ciphers are a method of encrypting plaintext by swapping each letter or symbol in the text with a different symbol, based on a specific key. The Caesar cipher is perhaps the simplest and most well-known of these substitution ciphers. It is named after the man who first used it. This cipher is also called a shift cipher or a mono-alphabetic cipher, which differentiates it from other more complex substitution ciphers.

In a Caesar cipher, the plaintext is represented in lowercase letters, while the ciphertext is represented in uppercase letters. Spaces are added to the ciphertext for readability, but they are removed in a real application to make attacking the ciphertext more difficult. Simple substitution of single letters separately can be demonstrated by writing out the alphabet in some order to represent the substitution. This is known as a substitution alphabet. The cipher alphabet can be shifted, reversed, or scrambled in a more complex way to create different types of substitution ciphers.

Mixed alphabets or deranged alphabets can also be used to create substitution ciphers. These are traditionally created by writing out a keyword and removing any repeated letters, then writing all the remaining letters in the alphabet in their usual order. This creates a unique mixed alphabet that can be used as the basis for the cipher. Substitution ciphers have a long history, and although they are not as secure as modern encryption methods, they are still used in some applications today.

Code:

Caesar Cipher.py

```python
# A python program for Caesar Cipher Technique
def encrypt(text, s):
    result = ""
    # traverse text
    for i in range(len(text)):
        char = text[i]
        # Encrypt uppercase characters
        if (char.isupper()):
            result += chr((ord(char) + s - 65) % 26 + 65)
        # Encrypt lowercase characters
        else:
            result += chr((ord(char) + s - 97) % 26 + 97)
    return result

def decrypt(text, s):
    result = ""
    # traverse text
    for i in range(len(text)):
        char = text[i]
        # Decrypt uppercase characters
        if (char.isupper()):
            result += chr((ord(char) - s - 65) % 26 + 65)
        # Decrypt lowercase characters
        else:
            result += chr((ord(char) - s - 97) % 26 + 97)
    return result

# Get the plain text and shift key from user input
text = input("\n"+"Enter the Plain Text: ")
s = int(input("Enter the value of the key: "))

print("\n-----------------------\n")
print("Plain Text : " + text)
print("Key: " + str(s))
a = encrypt(text, s)
print("Cipher Text: " + a)
print("Decrypted Text: " + decrypt(a, s))

print("\n-----------------------\n")
```

Output:

Run: Caesar Cipher

```
"C:\Programming Repository\PyCharm\Sem-06\CSS\venv\Scripts\python.exe" "C:\Programming Repository\PyCharm\Sem-06\CSS\Caesar Cipher.py"

Enter the Plain Text: AtTaCkAtOnCe
Enter the value of the key: 4

------------------------

Plain Text : AtTaCkAtOnCe
Key: 4
Cipher Text: ExXeGoExSrGi
Decrypted Text: AtTaCkAtOnCe

------------------------
```

Conclusion: After performing the experiment I was able to implement Substitution Cipher.

Experiment 01: (b) Transposition Cipher

Learning Objective: Implement and design the product cipher using Transposition Cipher

Tools: PyCharm

Theory:

Transposition ciphers are often used in combination with other encryption methods such as substitution ciphers to create a more secure encryption. By adding the additional layer of transposition, the resulting ciphertext becomes much more difficult to decipher without knowledge of both encryption methods. A common method of implementing transposition ciphers is through the use of a rectangular grid, where the plaintext is written out horizontally and then read vertically in a certain order to create the ciphertext. Other methods may involve shuffling the order of words or phrases in the plaintext message.

One of the most famous examples of a transposition cipher is the Rail Fence cipher, which involves writing the plaintext diagonally on alternate lines, and then reading the ciphertext vertically. This creates a zig-zag pattern that is difficult to decipher without knowledge of the exact transposition method used. Overall, transposition ciphers offer a flexible and relatively easy method of encryption that can be used in combination with other methods to create a more secure and complex encryption.

Code:

Transposition Cipher.py

```python
# Python3 implementation of Columnar Transposition
import math

# Encryption
def encryptMessage(msg):
    cipher = ""
    k_indx = 0          # track key indices
    msg_len = float(len(msg))
    msg_lst = list(msg)
    key_lst = sorted(list(key))
    # calculate column of the matrix
    col = len(key)
    # calculate maximum row of the matrix
    row = int(math.ceil(msg_len / col))
    # add the padding character '_' in empty
    # the empty cell of the matix
    fill_null = int((row * col) - msg_len)
    msg_lst.extend('_' * fill_null)
    # create Matrix and insert message and
    # padding characters row-wise
    matrix = [msg_lst[i: i + col]
              for i in range(0, len(msg_lst), col)]
    # read matrix column-wise using key
    for _ in range(col):
        curr_idx = key.index(key_lst[k_indx])
        cipher += ''.join([row[curr_idx]
                           for row in matrix])
        k_indx += 1
    return cipher

# Decryption
def decryptMessage(cipher):
    msg = ""
    # track key indices
    k_indx = 0
    # track msg indices
    msg_indx = 0
    msg_len = float(len(cipher))
    msg_lst = list(cipher)
    # calculate column of the matrix
    col = len(key)
    # calculate maximum row of the matrix
    row = int(math.ceil(msg_len / col))
```

```
    # convert key into list and sort
    # alphabetically so we can access
    # each character by its alphabetical position.
    key_lst = sorted(list(key))
    # create an empty matrix to
    # store deciphered message
    dec_cipher = []
    for _ in range(row):
        dec_cipher += [[None] * col]
    # Arrange the matrix column wise according
    # to permutation order by adding into new matrix
    for _ in range(col):
        curr_idx = key.index(key_lst[k_indx])
        for j in range(row):
            dec_cipher[j][curr_idx] = msg_lst[msg_indx]
            msg_indx += 1
        k_indx += 1
    # convert decrypted msg matrix into a string
    try:
        msg = ''.join(sum(dec_cipher, []))
    except TypeError:
        raise TypeError("This program cannot",
                        "handle repeating words.")
    null_count = msg.count('_')
    if null_count > 0:
        return msg[: -null_count]
    return msg

# Driver Code
msg = input("\n"+"Enter the Plain Text: ")
key = input("Enter the Key: ")

cipher = encryptMessage(msg)
print("Encrypted Message: {}".format(cipher))
print("Decryped Message: {}".format(decryptMessage(cipher)))
```

Output:

```
Run: Transposition Cipher
"C:\Programming Repository\PyCharm\Sem-06\CSS\venv\Scripts\python.exe" "C:\Programming Repository\PyCharm\Sem-06\CSS\Transposition Cipher.py"

Enter the Plain Text: HelloWorld
Enter the Key: 53241
Encrypted Message: odlreollHW
Decryped Message: HelloWorld

Process finished with exit code 0
```

Conclusion: After performing the experiment I was able to implement Transposition Cipher.

For Faculty Use

Correction Parameters	Formative Assessment [40%]	Timely completion of Practical [40%]	Attendance / Learning Attitude [20%]	Total
Marks Obtained				