MINI PROJECT REPORT

TOPIC: Different types of ciphers with examples and implementation.

Submitted by: A.NAVEENA

Batch: December2022

Course: Cyber Security

OVERVIEW:

Generally, ciphers are categorized according to how they function and how their key is used for encryption and decryption. While stream ciphers employ a steady stream of symbols, block ciphers group symbols into a fixed-size message (the block).
When using a symmetric key technique or cipher, both encryption and decryption use the same key. Asymmetric key algorithms or ciphers use a distinct key for encryption and decryption. Ciphers are (an algorithm) for performing both an encryption, and the corresponding decryption. Despite might what seem to be a relatively simple concept, ciphers play a crucial role in modern technology. Technologies arguably the corner stone of cryptography. In general, a cipher is simply just a set of steps involving communication (including *the internet*, *mobile phones*, *digital television* or even *ATMs*) rely on ciphers in order to maintain both security and privacy.

Although most people claim they're not familar with cryptography, they are often familar with the concept of ciphers, whether or not they are actually concious of it. Recent films such as The Da Vinci Code and National Treature: Book of Secrets have plots centered around cryptography and ciphers, bringing these concepts to the general public.

MINI PROJECT REPORT

This section (quite appropriately) deals with individual ciphers and algorithms. They have been divided based on their era and category (i.e. *when were they used* and *how do they work*). If you're looking for a reference guide, refer to the alphabetical list to the right, otherwise continue reading.
In our effort to provide a practical approach to these, we have developed a javascript implementation for each cipher that allows encryption and decryption of abitrary text (of your choosing) using the cipher. Some history of each cipher is also included, and tips on cryptanalysis are also provided.

What Is A Cipher?:

Cipher is a frequently used algorithm in cryptology, a subject concerned with the study of cryptographic algorithms. It is a method of encrypting and decrypting data.

- The adoption of a symmetrical cipher will determine the secret or symmetric key encryption.
- The symmetric algorithm applies the same encryption key and cipher to the data in the same way.
- Symmetric ciphers are the foundation of symmetric key encryption, widely referred to as secret key encryption.
- The goal might be to transform plain text to ciphertext or vice versa.
- By transforming plaintext letters or other information into ciphertext, cyphers transform data. It is best to express the ciphertext as random information.
- The cipher analyses the original and plaintext data to generate ciphertext that seems to be random data.
- The same encryption key is applied to data, in the same way, using symmetric encryption methods, whether the goal is to convert plaintext to ciphertext or ciphertext to plaintext.

MINI PROJECT REPORT

WHAT IS A CIPHER TEXT:

The data generated by any method is known as the ciphertext.In many networking protocols like TLS, or transport layer security, which permits network traffic encryption, modern cipher techniques use private communication. Crypts are used by many communication technologies, including digital televisions, phones, and ATMs, to guarantee protection and confidentiality.

WHAT ARE CIPHERTEXT ATTACKS:

The known ciphertext attack, or ciphertext-only attack (COA), is an attack method used in cryptanalysis when the attacker has access to a specific set of ciphertext. However, in this method, the attacker doesn't have access to the corresponding cleartext, i.e., data that is transmitted or stored unencrypted. The COA succeeds when the corresponding plaintext can be determined from a given set of ciphertext. Sometimes, the key that's used to encrypt the ciphertext can be determined from this attack.

In a chosen ciphertext attack (CCA), the attacker can make the victim (who knows the secret key) decrypt any ciphertext and send back the result. By analyzing the chosen ciphertext and the corresponding plaintext they receive, the attacker tries to guess the secret key the victim used. The goal of the CCA is to gain information that diminishes the security of the encryption scheme.

Related-key attack is any form of cryptanalysis where the attacker can observe the operation of a cipher under several different keys whose values the attacker doesn't know initially. However, there is some mathematical relationship connecting the keys that the attacker does know.

Ciphers have traditionally employed two basic forms of transformation:

•**Transposition ciphers**: maintain all of the original bits of data in a byte but reverse their order.

•**Substitution ciphers**: substitute specified data sequences with alternative data sequences. For example, one type of replacement would be to convert all bits with a value of 1 to bits with a value of 0, and vice versa.

How does a Cipher Work?

An encryption technique is used by ciphers to convert plaintext, which is a legible communication, into ciphertext, which seems to be a random string of letters.In order to encrypt or decode bits in a stream, cyphers are sometimes called stream ciphers.Alternately, they can use block ciphers, which process ciphertext in uniform blocks of a predetermined amount of bits. The encryption algorithm is used by modern cipher implementations along with a secret key to modify data while it is encrypted.Longer keys (measured in bits) make ciphers more resilient against brute-force attacks.The more brute-force attacks that must be used to uncover the plaintext, the longer the key.Although key length is not always correlated with cypher strength, experts advise that modern cyphers be built with keys of at least 128 bits or more, depending on the algorithm and use case. n real-world ciphering, the key is kept secret rather than the method because a key is such a crucial component of an encryption process.Strong encryption methods are designed such that, even someone is familiar with the process, deciphering the ciphertext without the required key should be challenging.This means that in order for a cypher to work, both the sender and the receiver need to possess a key or set of keys.In symmetric key approaches, the same key is used for both data encryption and decryption.Asymmetric key algorithms use both public and private keys to encrypt and decrypt data.Utilizing enormous numbers that have been paired but are not equal,asymmetric encryption, also known as public key cryptography (asymmetric).

What is the Purpose of Ciphers?

The most used ciphers for securing internet communication are symmetric ones. They are also incorporated into several data-sharing network protocols. For example, Secure Sockets Layer and TLS use ciphers to encrypt data at the application layer, especially when combined with HTTP Secure (HTTPS).Virtual private networks that link remote workers or branches to corporate networks use protocols using symmetric key techniques to secure data transmission. Symmetric cyphers are typically used in Wi-Fi networks, online banking and shopping, and mobile phone services to protect the privacy of user data. Several protocols employ asymmetric cryptography to authenticate and encrypt endpoints. The exchange of symmetric keys for the encryption of session data is likewise protected using it. These standards include the following:

- HTTP
- TLS
- Secure Shell
- Secure/Multipurpose Internet Mail Extensions
- Open Pretty Good Privacy

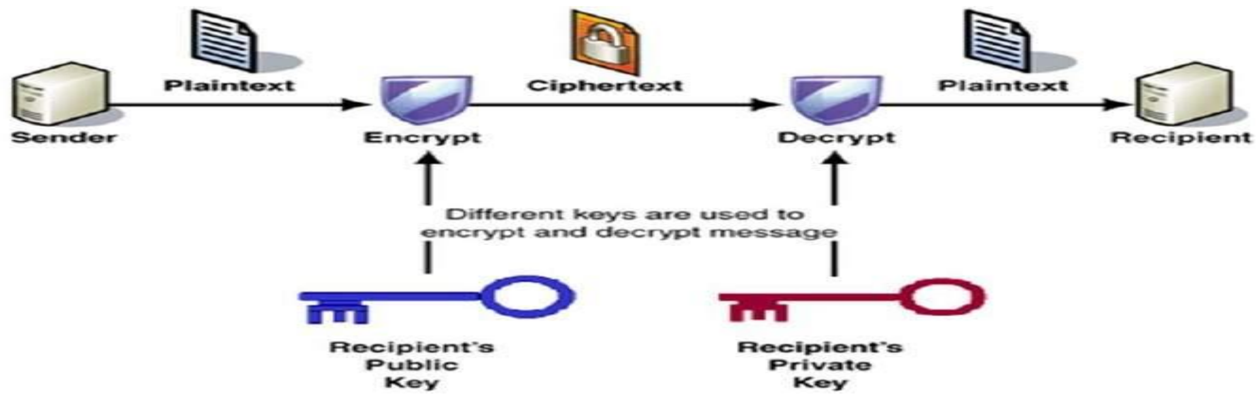
Types of Ciphers

Ciphers can be classified in a variety of ways, including the following:

<u>Public-key cryptography</u>: In this cipher, two different keys -- public key and private key -- are used for encryption and decryption. The sender uses the public key to perform the encryption, but the private key is kept secret from the receiver. This is also known as "asymmetric key algorithm." Unlike symmetric key cryptography, we do not find historical use of public-key cryptography. It is a relatively new concept.

Symmetric cryptography was well suited for organizations such as governments, military, and big financial corporations were involved in the classified communication.

With the spread of more unsecure computer networks in last few decades, a genuine need was felt to use cryptography at larger scale. The symmetric key was found to be non-practical due to challenges it faced for key management. This gave rise to the public key cryptosystems.



The most important properties of public key encryption scheme are −

- Different keys are used for encryption and decryption. This is a property which set this scheme different than symmetric encryption scheme.
- Each receiver possesses a unique decryption key, generally referred to as his private key.
- Receiver needs to publish an encryption key, referred to as his public key.

- Some assurance of the authenticity of a public key is needed in this scheme to avoid spoofing by adversary as the receiver. Generally, this type of cryptosystem involves trusted third party which certifies that a particular public key belongs to a specific person or entity only.
- Encryption algorithm is complex enough to prohibit attacker from deducing the plaintext from the ciphertext and the encryption (public) key.
- Though private and public keys are related mathematically, it is not be feasible to calculate the private key from the public key. In fact, intelligent part of any public-key cryptosystem is in designing a relationship between two keys.

There are three types of Public Key Encryption schemes. We discuss them in following sections −

1.RSA Cryptosystem

This cryptosystem is one the initial system. It remains most employed cryptosystem even today. As The system was invented by three scholars **Ron Rivest, Adi Shamir,** and **Len Adleman** and hence, it is termed as RSA cryptosystem.

We will see two aspects of the RSA cryptosystem, firstly generation of key pair and secondly encryption-decryption algorithms.

Generation of RSA Key Pair

Each person or a party who desires to participate in communication using encryption needs to generate a pair of keys, namely public key and private key.

RSA Encryption

- Suppose the sender wish to send some text message to someone whose public key is (n, e).
- The sender then represents the plaintext as a series of numbers less than n.

- To encrypt the first plaintext P, which is a number modulo n. The encryption process is simple mathematical step as −

$$C = P^e \bmod n$$

- In other words, the ciphertext C is equal to the plaintext P multiplied by itself e times and then reduced modulo n. This means that C is also a number less than n.
- Returning to our Key Generation example with plaintext P = 10, we get ciphertext C −

$$C = 10^5 \bmod 91$$

RSA Decryption

- The decryption process for RSA is also very straightforward. Suppose that the receiver of public-key pair (n, e) has received a ciphertext C.
- Receiver raises C to the power of his private key d. The result modulo n will be the plaintext P.

$$\text{Plaintext} = C^d \bmod n$$

- Returning again to our numerical example, the ciphertext C = 82 would get decrypted to number 10 using private key 29 −

$$\text{Plaintext} = 82^{29} \bmod 91 = 10$$

ElGamal Cryptosystem

Along with RSA, there are other public-key cryptosystems proposed. Many of them are based on different versions of the Discrete Logarithm Problem.

ElGamal cryptosystem, called Elliptic Curve Variant, is based on the Discrete Logarithm Problem. It derives the strength from the assumption that the discrete logarithms cannot be found in practical time frame for a given number, while the inverse operation of the power can be computed efficiently.

Let us go through a simple version of ElGamal that works with numbers modulo p. In the case of elliptic curve variants, it is based on quite different number systems.

MINI PROJECT REPORT

Generation of ElGamal Key Pair

Each user of ElGamal cryptosystem generates the key pair through as follows −

- **Choosing a large prime p.** Generally a prime number of 1024 to 2048 bits length is chosen.
- **Choosing a generator element g.** This number must be between 1 and p − 1, but cannot be any number.
- It is a generator of the multiplicative group of integers modulo p. This means for every integer m co-prime to p, there is an integer k such that
  g^k=a mod n.
- **Choosing the private key.** The private key x is any number bigger than 1 and smaller than p−1.
- **Computing part of the public key.** The value y is computed from the parameters p, g and the private key x as follows : $y = g^x \bmod p$
- **Obtaining Public key.** The ElGamal public key consists of the three parameters (p, g, y).

ElGamal Encryption

Suppose sender wishes to send a plaintext to someone whose ElGamal public key is (p, g, y), then −

- Sender represents the plaintext as a series of numbers modulo p.
- To encrypt the first plaintext P, which is represented as a number modulo p. The encryption process to obtain the ciphertext C is as follows −
  - Randomly generate a number k;
  - Compute two values C1 and C2, where
    $C1 = g^k \bmod p$
    $C2 = (P^* y^k) \bmod p$
- Send the ciphertext C, consisting of the two separate values (C1, C2), sent together

ElGamal Decryption

- To decrypt the ciphertext (C1, C2) using private key x, the following two steps are taken −
  - Compute the modular inverse of $(C1)^x$ modulo p, which is $(C1)^{-x}$, generally referred to as decryption factor.
  - Obtain the plaintext by using the following formula –
    $C2 \times (C1)^{-x}$ mod p = Plaintext

Elliptic Curve Cryptography (ECC)

Elliptic Curve Cryptography (ECC) is a term used to describe a suite of cryptographic tools and protocols whose security is based on special versions of the discrete logarithm problem. It does not use numbers modulo p.

ECC is based on sets of numbers that are associated with mathematical objects called elliptic curves. There are rules for adding and computing multiples of these numbers, just as there are for numbers modulo p.

ECC includes a variants of many cryptographic schemes that were initially designed for modular numbers such as ElGamal encryption and Digital Signature Algorithm.

It is believed that the discrete logarithm problem is much harder when applied to points on an elliptic curve. This prompts switching from numbers modulo p to points on an elliptic curve. Also an equivalent security level can be obtained with shorter keys if we use elliptic curve-based variants.

The shorter keys result in two benefits −

- Ease of key management
- Efficient computation

These benefits make elliptic-curve-based variants of encryption scheme highly attractive for application where computing resources are constrained.

PRIVATE-KEY CRYPTOGRAPHY:

In this cipher, the sender and receiver must have a pre-shared key. The shared key is kept secret from all other parties and is used for encryption, as well as dcryption. This cryptography is also known as "symmetric key algorithm."

A private key, also known as a *secret key*, is a variable in cryptography that is used with an algorithm to encrypt and decrypt data. Secret keys should only be shared with the key's generator or parties authorized to decrypt the data. Private keys play an important role in symmetric cryptography, asymmetric cryptography and cryptocurrencies.

A private key is typically a long, randomly or pseudo-randomly generated sequence of bits that cannot be easily guessed. The complexity and length of the private key determine how easily an attacker can execute a brute-force attack, where they try out different keys until the right one is found.

HOW THE PRIVATE KEY CIPHER WORKS? :

Private key encryption is also referred to as *symmetric encryption*, where the same private key is used for both encryption and decryption. In this case, a private key works as follows:

- **Generating a new private key.** Prior to encryption, generate a new key that is as random as possible; encryption software is typically used to generate private keys.

- **Securely storing the private key.** Once generated, the private key must be stored securely. Depending on the application, keys may be stored offline or on the computer used to generate, encrypt and decrypt data. Private keys may be protected with a password, encrypted or hashed for security -- or all three.
- **Key exchange.** The private key is used to decrypt, as well as to encrypt, so using it for symmetric encryption requires a key exchange to share that key securely with trusted parties authorized to exchange secured data. Cryptographic software is usually used to automate this process.
- **Key management.** Private key management is required to prevent any individual key from being used for too long. It helps to securely retire keys after their useful lifetime is reached.

ADVANTAGES:

Private key encryption provides several useful features. They include the following four benefits:

1. **More secure.** Private keys that are longer and have greater entropy, or randomness, are more secure from brute-force or dictionary attacks.
2. **Faster.** Symmetric key encryption is faster computationally than asymmetric encryption with its public-private key pairs.
3. **Best for encryption.** Most cryptographic processes use private key encryption to encrypt data transmissions. They typically use a public key algorithm to securely share secret keys.
4. **Work for stream and block ciphers.** Secret key ciphers -- the algorithm for encrypting and decrypting data -- generally

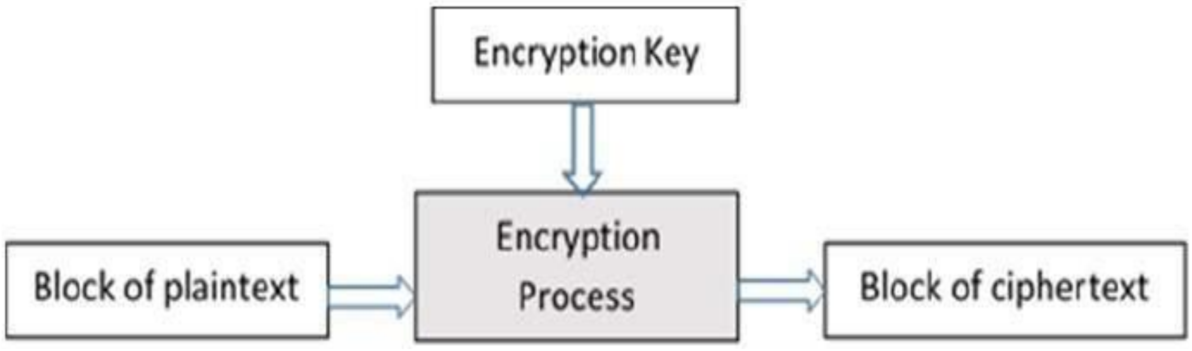
fall into one of two categories: stream ciphers or block ciphers. A block cipher applies a private key and algorithm to a block of data simultaneously, whereas a stream cipher applies the key and algorithm one bit at a time

Block ciphers:

Block ciphers encrypt data in equally sized blocks. A block cipher takes a block of plaintext bits and generates a block of ciphertext bits, generally of same size. The size of block is fixed in the given scheme. The choice of block size does not directly affect to the strength of encryption scheme. The strength of cipher depends up on the key length.

Block Size

Though any size of block is acceptable, following aspects are borne in mind while selecting a size of a block



.

- **Avoid very small block size** − Say a block size is m bits. Then the possible plaintext bits combinations are then 2^m. If the attacker discovers the plain text blocks corresponding to some previously sent ciphertext blocks, then the attacker can launch a type of ‘dictionary attack’ by building up a dictionary of plaintext/ciphertext pairs sent using that encryption key. A larger block size makes attack harder as the dictionary needs to be larger.

- **Do not have very large block size** − With very large block size, the cipher becomes inefficient to operate. Such plaintexts will need to be padded before being encrypted.
- **Multiples of 8 bit** − A preferred block size is a multiple of 8 as it is easy for implementation as most computer processor handle data in multiple of 8 bits.

Padding in Block Cipher

Block ciphers process blocks of fixed sizes (say 64 bits). The length of plaintexts is mostly not a multiple of the block size. For example, a 150-bit plaintext provides two blocks of 64 bits each with third block of balance 22 bits. The last block of bits needs to be padded up with redundant information so that the length of the final block equal to block size of the scheme. In our example, the remaining 22 bits need to have additional 42 redundant bits added to provide a complete block. The process of adding bits to the last block is referred to as **padding**.

Too much padding makes the system inefficient. Also, padding may render the system insecure at times, if the padding is done with same bits always.

Block Cipher Schemes

There is a vast number of block ciphers schemes that are in use. Many of them are publically known. Most popular and prominent block ciphers are listed below.

- **Digital Encryption Standard (DES)** − The popular block cipher of the 1990s. It is now considered as a 'broken' block cipher, due primarily to its small key size.
- **Triple DES** − It is a variant scheme based on repeated DES applications. It is still a respected block ciphers but inefficient compared to the new faster block ciphers available.

- **Advanced Encryption Standard (AES)** − It is a relatively new block cipher based on the encryption algorithm **Rijndael** that won the AES design competition.
- **IDEA** − It is a sufficiently strong block cipher with a block size of 64 and a key size of 128 bits. A number of applications use IDEA encryption, including early versions of Pretty Good Privacy (PGP) protocol. The use of IDEA scheme has a restricted adoption due to patent issues.
- **Twofish** − This scheme of block cipher uses block size of 128 bits and a key of variable length. It was one of the AES finalists. It is based on the earlier block cipher Blowfish with a block size of 64 bits.
- **Serpent** − A block cipher with a block size of 128 bits and key lengths of 128, 192, or 256 bits, which was also an AES competition finalist. It is a slower but has more secure design than other block cipher.

Encryption Process

The encryption process uses the Feistel structure consisting multiple rounds of processing of the plaintext, each round consisting of a “substitution” step followed by a permutation step.

- The input block to each round is divided into two halves that can be denoted as L and R for the left half and the right half.
- In each round, the right half of the block, R, goes through unchanged. But the left half, L, goes through an operation that depends on R and the encryption key. First, we apply an encrypting function ‘f’ that takes two input − the key K and R. The function produces the output f(R,K). Then, we XOR the output of the mathematical function with L.
- In real implementation of the Feistel Cipher, such as DES, instead of using the whole encryption key during each round, a round-dependent key (a subkey) is derived from the encryption key. This means that each round uses a

different key, although all these subkeys are related to the original key.

- The permutation step at the end of each round swaps the modified L and unmodified R. Therefore, the L for the next round would be R of the current round. And R for the next round be the output L of the current round.
- Above substitution and permutation steps form a ‘round’. The number of rounds are specified by the algorithm design.
- Once the last round is completed then the two sub blocks, ‘R’ and ‘L’ are concatenated in this order to form the ciphertext block.

The difficult part of designing a Feistel Cipher is selection of round function ‘f’.

Decryption Process

The process of decryption in Feistel cipher is almost similar. Instead of starting with a block of plaintext, the ciphertext block is fed into the start of the Feistel structure and then the process thereafter is exactly the same as described in the given illustration.

The process is said to be almost similar and not exactly same. In the case of decryption, the only difference is that the subkeys used in encryption are used in the reverse order.

The final swapping of ‘L’ and ‘R’ in last step of the Feistel Cipher is essential. If these are not swapped then the resulting ciphertext could not be decrypted using the same algorithm.

Feistel Cipher

Feistel Cipher model is a structure or a design used to develop many block ciphers such as DES. Feistel cipher may have invertible, non-invertible and self invertible components in its design. Same encryption as well as decryption algorithm is used. A separate key is used for each round. However same round keys are used for encryption as well as decryption.

Feistel cipher algorithm

- Create a list of all the Plain Text characters.
- Convert the Plain Text to Ascii and then 8-bit binary format.
- Divide the binary Plain Text string into two halves: left half (L1)and right half (R1)
- Generate a random binary keys (K1 and K2) of length equal to the half the length of the Plain Text for the two rounds.

First Round of Encryption

- **a.** Generate function f1 using R1 and K1 as follows:
  f1= xor(R1, K1)
- **b.** Now the new left half(L2) and right half(R2) after round 1 are as follows:
  R2= xor(f1, L1)

  L2=R1

Second Round of Encryption

- **a.** Generate function f2 using R2 and K2 as follows:

  f2= xor(R2, K2)
- **b.** Now the new left half(L3) and right half(R3) after round 2 are as follows:
  R3= xor(f2, L2)

  L3=R2
- Concatenation of R3 to L3 is the Cipher Text
- Same algorithm is used for decryption to retrieve the Plain Text from the Cipher Text.

MINI PROJECT REPORT

EX:

```
Plain Text is: Hello

Cipher Text:  E1!w(

Retrieved Plain Text is:  b'Hello'
```

CODE:

```
import binascii

# Random bits key generation
def rand_key(p):

    import random
    key1 = ""
    p = int(p)

    for i in range(p):

        temp = random.randint(0,1)
        temp = str(temp)
        key1 = key1 + temp

    return(key1)

# Function to implement bit exor
def exor(a,b):

    temp = ""

    for i in range(n):

        if (a[i] == b[i]):
            temp += "0"

        else:
            temp += "1"

    return temp

# Defining BinarytoDecimal() function
def BinaryToDecimal(binary):

    # Using int function to convert to
    # string
    string = int(binary, 2)
```

MINI PROJECT REPORT

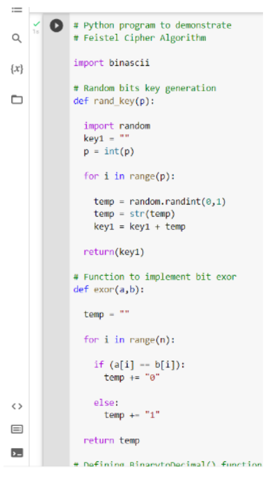
```
    return string

# Feistel Cipher
PT = "Hello"
print("Plain Text is:", PT)

# Converting the plain text to
# ASCII
PT_Ascii = [ord(x) for x in PT]

# Converting the ASCII to
# 8-bit binary format
PT_Bin = [format(y,'08b') for y in PT_Ascii]
PT_Bin = "".join(PT_Bin)

n = int(len(PT_Bin)//2)
L1 = PT_Bin[0:n]
R1 = PT_Bin[n::]
m = len(R1)

# Generate Key K1 for the
# first round
K1= rand_key(m)

# Generate Key K2 for the
# second round
K2= rand_key(m)

# first round of Feistel
f1 = exor(R1,K1)
R2 = exor(f1,L1)
L2 = R1

# Second round of Feistel
f2 = exor(R2,K2)
R3 = exor(f2,L2)
L3 = R2

# Cipher text
bin_data = L3 + R3
str_data =' '

for i in range(0, len(bin_data), 7):

    # slicing the bin_data from index range [0, 6]
```

```
        # and storing it in temp_data
        temp_data = bin_data[i:i + 7]

        # passing temp_data in BinarytoDecimal() function
        # to get decimal value of corresponding temp_data
        decimal_data = BinaryToDecimal(temp_data)

        # Decoding the decimal value returned by
        # BinarytoDecimal() function, using chr()
        # function which return the string corresponding
        # character for given ASCII value, and store it
        # in str_data
        str_data = str_data + chr(decimal_data)

    print("Cipher Text:", str_data)

    # Decryption
    L4 = L3
    R4 = R3

    f3 = exor(L4,K2)
    L5 = exor(R4,f3)
    R5 = L4

    f4 = exor(L5,K1)
    L6 = exor(R5,f4)
    R6 = L5
    PT1 = L6+R6


    PT1 = int(PT1, 2)
    RPT = binascii.unhexlify( '%x'% PT1)
    print("Retrieved Plain Text is: ", RPT)
```

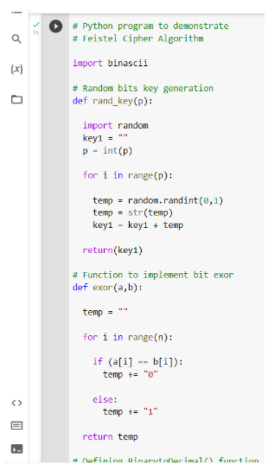
OUTPUT:

```
Plain Text is: Hello

Cipher Text:  E1!w(

Retrieved Plain Text is:  b'Hello'
```

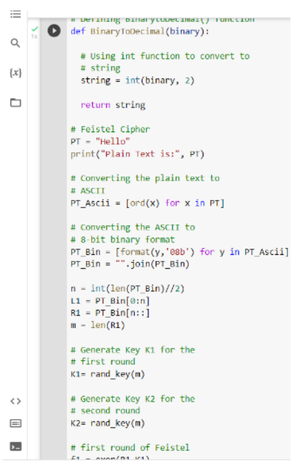
MINI PROJECT REPORT



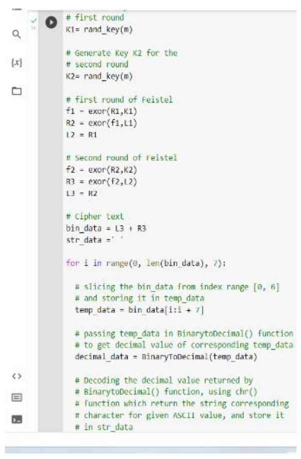
```
# Python program to demonstrate
# Feistel Cipher Algorithm

import binascii

# Random bits key generation
def rand_key(p):

  import random
  key1 = ""
  p = int(p)

  for i in range(p):

    temp = random.randint(0,1)
    temp = str(temp)
    key1 = key1 + temp

  return(key1)

# Function to implement bit exor
def exor(a,b):

  temp = ""

  for i in range(n):

    if (a[i] == b[i]):
      temp += "0"

    else:
      temp += "1"

  return temp

# Defining BinarytoDecimal() function
```



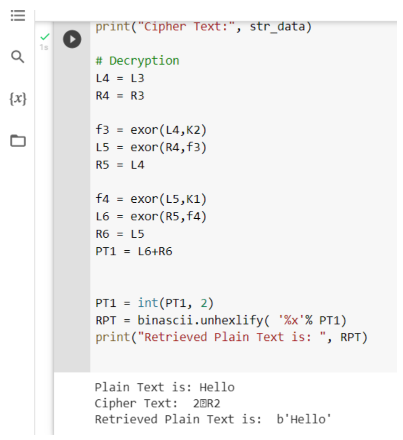
```
# Python program to demonstrate
# Feistel Cipher Algorithm

import binascii

# Random bits key generation
def rand_key(p):

  import random
  key1 = ""
  p = int(p)

  for i in range(p):

    temp = random.randint(0,1)
    temp = str(temp)
    key1 = key1 + temp

  return(key1)

# Function to implement bit exor
def exor(a,b):

  temp = ""

  for i in range(n):

    if (a[i] == b[i]):
      temp += "0"

    else:
      temp += "1"

  return temp

# Defining BinarytoDecimal() function
```



```
# Defining BinarytoDecimal() function
def BinaryToDecimal(binary):

  # Using int function to convert to
  # string
  string = int(binary, 2)

  return string

# Feistel Cipher
PT = "Hello"
print("Plain Text is:", PT)

# Converting the plain text to
# ASCII
PT_Ascii = [ord(x) for x in PT]

# Converting the ASCII to
# 8-bit binary format
PT_Bin = [format(y,'08b') for y in PT_Ascii]
PT_Bin = "".join(PT_Bin)

n = int(len(PT_Bin)//2)
L1 = PT_Bin[0:n]
R1 = PT_Bin[n::]
m = len(R1)

# Generate Key K1 for the
# first round
K1= rand_key(m)

# Generate Key K2 for the
# second round
K2= rand_key(m)

# first round of Feistel
f1 = exor(R1,K1)
```



```
# first round
K1= rand_key(m)

# Generate Key K2 for the
# second round
K2= rand_key(m)

# first round of Feistel
f1 = exor(R1,K1)
R2 = exor(f1,L1)
L2 = R1

# Second round of Feistel
f2 = exor(R2,K2)
R3 = exor(f2,L2)
L3 = R2

# Cipher text
bin_data = L3 + R3
str_data =' '

for i in range(0, len(bin_data), 7):

  # slicing the bin_data from index range [0, 6]
  # and storing it in temp_data
  temp_data = bin_data[i:i + 7]

  # passing temp_data in BinarytoDecimal() function
  # to get decimal value of corresponding temp_data
  decimal_data = BinaryToDecimal(temp_data)

  # Decoding the decimal value returned by
  # BinarytoDecimal() function, using chr()
  # function which return the string corresponding
  # character for given ASCII value, and store it
  # in str_data
```



```
print("Cipher Text:", str_data)

# Decryption
L4 = L3
R4 = R3

f3 = exor(L4,K2)
L5 = exor(R4,f3)
R5 = L4

f4 = exor(L5,K1)
L6 = exor(R5,f4)
R6 = L5
PT1 = L6+R6


PT1 = int(PT1, 2)
RPT = binascii.unhexlify( '%x'% PT1)
print("Retrieved Plain Text is: ", RPT)
```

```
Plain Text is: Hello
Cipher Text:  2R2
Retrieved Plain Text is:  b'Hello'
```

STREAM CIPHERS:

can be used on data streams that are often received and transferred via a network. In stream cipher, one byte is encrypted at a time while in block cipher ~128 bits are encrypted at a time.

Initially, a key(k) will be supplied as input to pseudorandom bit generator and then it produces a random 8-bit output which is treated as keystream.

The resulted keystream will be of size 1 byte, i.e., 8 bits.

1. Stream Cipher follows the sequence of pseudorandom number stream.
2. One of the benefits of following stream cipher is to make cryptanalysis more difficult, so the number of bits chosen in the Keystream must be long in order to make cryptanalysis more difficult.

3. By making the key more longer it is also safe against brute force attacks.
4. The longer the key the stronger security is achieved, preventing any attack.
5. Keystream can be designed more efficiently by including more number of 1s and 0s, for making cryptanalysis more difficult.
6. Considerable benefit of a stream cipher is, it requires few lines of code compared to block cipher.

Encryption :

For Encryption,

- Plain Text and Keystream produces Cipher Text (Same keystream will be used for decryption.).
- The Plaintext will undergo XOR operation with keystream bit-by-bit and produces the Cipher Text.

EX:

```
Plain Text : 10011001

Keystream  : 11000011
`````````````````````

Cipher Text : 01011010
```

**Decryption :**

For Decryption,

- Cipher Text and Keystream gives the original Plain Text (Same keystream will be used for encryption.).
- The Ciphertext will undergo XOR operation with keystream bit-by-bit and produces the actual Plain Text.
- Decryption is just the reverse process of Encryption i.e. performing XOR with Cipher Text.

EX:

```
     Cipher Text : 01011010

     Keystream   : 11000011
     ~~~

     Plain Text : 1001100
```
     ~~~

# MINI PROJECT REPORT

**Substitution ciphers**: Replace bits, characters, or character blocks in plaintext with alternate bits, characters or character blocks to produce ciphertext. A substitution cipher may
be monoalphabetic or polyalphabetic*:*

- A single alphabet is used to encrypt the entire plaintext message. For example, if the letter A is enciphered as the letter K, this will be the same for the entire message.
- A more complex substitution using a mixed alphabet to encrypt each bit, character or character block of a plaintext message. For instance, the letter A may be encoded as the letter K for part of the message, but later it might be encoded as the letter W

- Hiding some data is known as encryption. When plain text is encrypted it becomes unreadable and is known as ciphertext. In a Substitution cipher, any character of plain text from the given fixed set of characters is substituted by some other character from the same set depending on a key. For example with a shift of 1, A would be replaced by B, B would become C, and so on.
- **Note:** Special case of Substitution cipher is known as Caesar cipher where the key is taken as 3.

Mathematical representation:

- The encryption can be represented using modular arithmetic by first transforming the letters into numbers, according to the scheme, A = 0, B = 1,…, Z = 25. Encryption of a letter by a shift n can be described mathematically as:

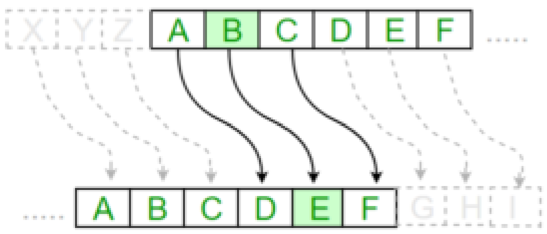
- “$E_n(x) = (x + n) mod\ 26$”

(Encryption Phase with shift n):

$D_n(x) = (x - n) mod\ 26$

(Decryption Phase with shift n)

# MINI PROJECT REPORT



**Algorithm for Substitution Cipher:**

**Input:**

- A String of both lower and upper case letters, called PlainText.
- An Integer denoting the required key.

**Procedure:**

- Create a list of all the characters.
- Create a dictionary to store the substitution for all characters.
- For each character, transform the given character as per the rule, depending on whether we're encrypting or decrypting the text.
- Print the new string generated

**Examples:**

**Plain Text:** `I am studying Data Encryption`
**Key:** `4`
**Output:** `M eq wxyhCmrk Hexe IrgvCtxmsr`

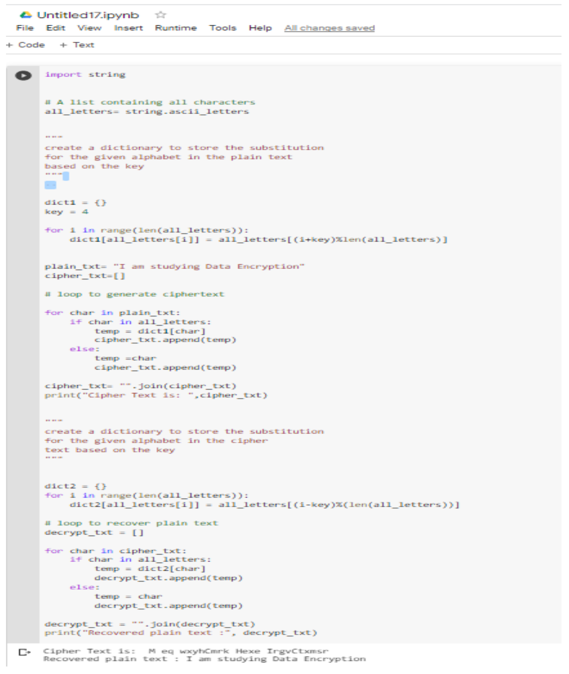
CODE IN PYTHON:

```python
import string
all_letters= string.ascii_letters

dict1 = {}
key = 4

for i in range(len(all_letters)):
    dict1[all_letters[i]] = all_letters[(i+key)%len(all_letters)]


plain_txt= "I am studying Data Encryption"
cipher_txt=[]
```

MINI PROJECT REPORT

```
for char in plain_txt:
    if char in all_letters:
        temp = dict1[char]
        cipher_txt.append(temp)
    else:
        temp =char
        cipher_txt.append(temp)

cipher_txt= "".join(cipher_txt)
print("Cipher Text is: ",cipher_txt)

dict2 = {}
for i in range(len(all_letters)):
    dict2[all_letters[i]] = all_letters[(i-key)%(len(all_letters))]

decrypt_txt = []

for char in cipher_txt:
    if char in all_letters:
        temp = dict2[char]
        decrypt_txt.append(temp)
    else:
        temp = char
        decrypt_txt.append(temp)

decrypt_txt = "".join(decrypt_txt)
print("Recovered plain text :", decrypt_txt)
```

OUTPUT:



```
import string

# A list containing all characters
all_letters= string.ascii_letters

"""
create a dictionary to store the substitution
for the given alphabet in the plain text
based on the key
"""

dict1 = {}
key = 4

for i in range(len(all_letters)):
    dict1[all_letters[i]] = all_letters[(i+key)%len(all_letters)]


plain_txt= "I am studying Data Encryption"
cipher_txt=[]

# loop to generate ciphertext

for char in plain_txt:
    if char in all_letters:
        temp = dict1[char]
        cipher_txt.append(temp)
    else:
        temp =char
        cipher_txt.append(temp)

cipher_txt= "".join(cipher_txt)
print("Cipher Text is: ",cipher_txt)


"""
create a dictionary to store the substitution
for the given alphabet in the cipher
text based on the key
"""


dict2 = {}
for i in range(len(all_letters)):
    dict2[all_letters[i]] = all_letters[(i-key)%(len(all_letters))]

# loop to recover plain text
decrypt_txt = []

for char in cipher_txt:
    if char in all_letters:
        temp = dict2[char]
        decrypt_txt.append(temp)
    else:
        temp = char
        decrypt_txt.append(temp)

decrypt_txt = "".join(decrypt_txt)
print("Recovered plain text :", decrypt_txt)
```

```
Cipher Text is:  M eq wxyhCmrk Hexe IrgvCtxmsr
Recovered plain text : I am studying Data Encryption
```

MINI PROJECT REPORT

TRANSPOSITION CIPHERS:

Unlike substitution ciphers that replaceletters with other letters, transposition ciphers keep the letters the same, but rearrange their order according to a specific algorithm. For instance, in a simple columnar transposition cipher, a message might be read horizontally but would be written vertically to produce the ciphertext. Given a plain-text message and a numeric key, cipher/de-cipher the given text using Columnar Transposition Cipher

The Columnar Transposition Cipher is a form of transposition cipher just like Rail Fence Cipher. Columnar Transposition involves writing the plaintext out in rows, and then reading the ciphertext off in columns one by one.

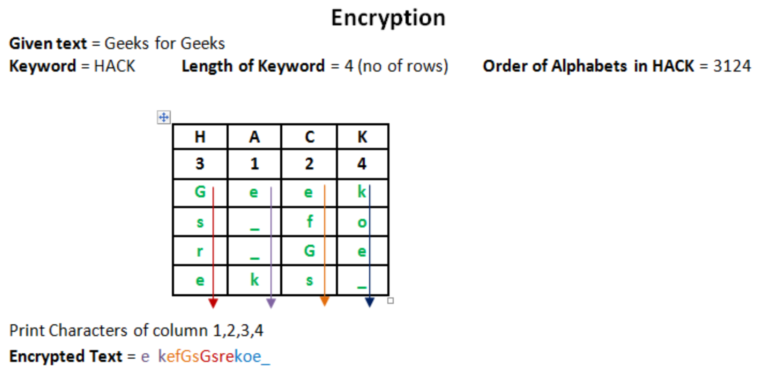
EXAMPLE:

```
Encryption
Input : Geeks for Geeks
Key = HACK
Output : e  kefGsGsrekoe_
Decryption
Input : e  kefGsGsrekoe_
Key = HACK
Output : Geeks for Geeks
```

Encryption:
In a transposition cipher, the order of the alphabets is re-arranged to obtain the cipher-text.

1. The message is written out in rows of a fixed length, and then read out again column by column, and the columns are chosen in some scrambled order.
2. Width of the rows and the permutation of the columns are usually defined by a keyword.
3. For example, the word HACK is of length 4 (so the rows are of length 4), and the permutation is defined by the alphabetical order of the letters in the keyword. In this case, the order would be “3 1 2 4”.

4. Any spare spaces are filled with nulls or left blank or placed by a character (Example: _).
5. Finally, the message is read off in columns, in the order specified by the keyword.



Encryption

Given text = Geeks for Geeks
Keyword = HACK **Length of Keyword** = 4 (no of rows) **Order of Alphabets in HACK** = 3124

H	A	C	K
3	1	2	4
G	e	e	k
s	_	f	o
r	_	G	e
e	k	s	_

Print Characters of column 1,2,3,4
Encrypted Text = e kefGsGsrekoe_

Decryption:

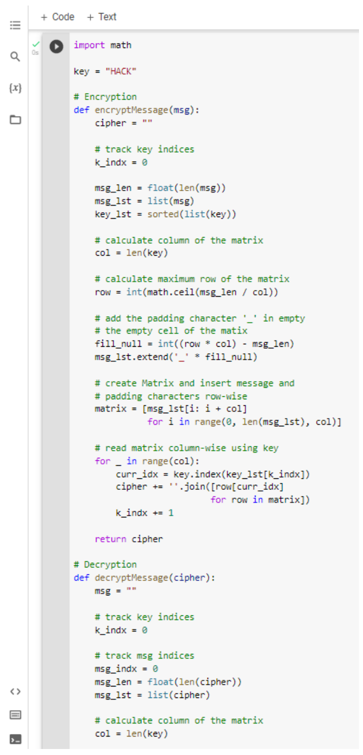
1. To decipher it, the recipient has to work out the column lengths by dividing the message length by the key length.
2. Then, write the message out in columns again, then re-order the columns by reforming the key word.

CODE:

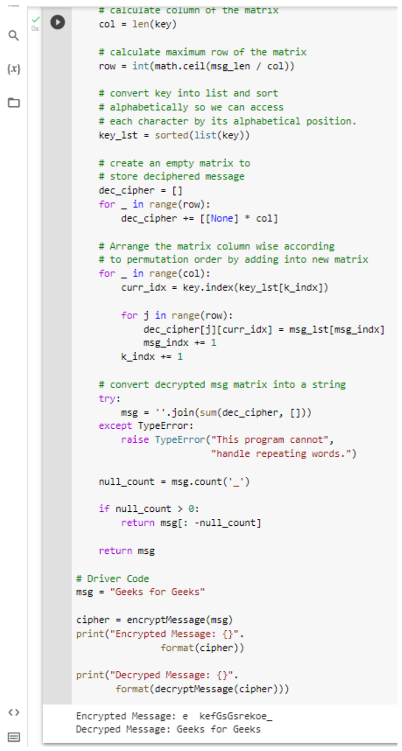
```
import math

key = "HACK"

# Encryption
def encryptMessage(msg):
    cipher = ""

    # track key indices
    k_indx = 0

    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 = "Geeks for Geeks"

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

print("Decryped Message: {}".
       format(decryptMessage(cipher)))
```

OUTPUT:

MINI PROJECT REPORT



```python
import math

key = "HACK"

# Encryption
def encryptMessage(msg):
    cipher = ""

    # track key indices
    k_indx = 0

    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)
```



```python
    # 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 = "Geeks for Geeks"

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

print("Decryped Message: {}".
       format(decryptMessage(cipher)))
```

```
Encrypted Message: e  kefGsGsrekoe_
Decryped Message: Geeks for Geeks
```

ADVANTAGES & DISADVANTAGES:

In the case of columnar transposition, the message is addressed out in rows of a fixed length, and then put out again column by column, and the columns are chosen in any scrambled order.

- A keyword normally defines both the width of the rows and the permutation of the columns.
- In a regular columnar transposition cipher, any spare places are filled with nulls; in an irregular columnar transposition cipher, the areas are left blank.

- Finally, the information made off in columns in the form defined by the keyword.

A disadvantage of such ciphers is considerably more difficult and error-prone than simpler cipher.

MINI PROJECT REPORT

PERMUTATION CIPHER:

a permutation cipher is a transposition cipher in which the key is a permutation. To apply a cipher, a random permutation of size e is generated (the larger the value of e the more secure the cipher). The plaintext is then broken into segments of size e and the letters within that segment are permuted according to this key.In theory, any transposition cipher can be viewed as a permutation cipher where e is equal to the length of the plaintext; this is too cumbersome a generalisation to use in actual practice, however. As It is similar to Columnar Transposition in some ways, in that the columns are written in the same way, including how the keyword is used. However, the Permutation Cipher acts on blocks of letters (the lengths of the keyword), rather than the whole ciphertext. Mathematically, a permutation is a rule that tells you how to rearrange a set of elements. For example, the permutation shown to the left (this is how we write a permutation mathematically), tells us that the first element is moved to the third position, the second element is moved to the first position and the third element is moved to the second position.

Encryption:

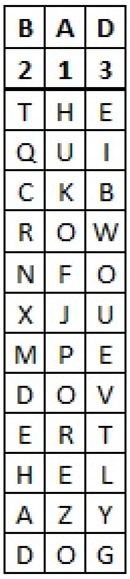
We choose a keyword, and split the plaintext into blocks that are the same length as the keyword. We write this in columns beneath the keyword. We then label each keyword letter in alphabetical order (if there are duplicates we take them in order of appearance). So far this is identical to Columnar Transposition. Now we reorder the columns, so that the numbers are in order (the letters of the keyword are in alphabetical order). We now read across the rows.

As an example we shall encrypt the plaintext "the quick brown fox jumped over the lazy dog" using the keyword bad.

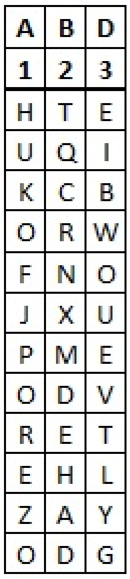
MINI PROJECT REPORT

We start by creating a grid that has 3 columns (as the keyword has 3 letters). In this grid we write out the plaintext beneath the keyword. We add a row with the numbers representing the alphabetical order of the letters of the keyword.

We then reorder the columns so that the numbers are in order, like in the image to the far right.



B	A	D
2	1	3
T	H	E
Q	U	I
C	K	B
R	O	W
N	F	O
X	J	U
M	P	E
D	O	V
E	R	T
H	E	L
A	Z	Y
D	O	G



A	B	D
1	2	3
H	T	E
U	Q	I
K	C	B
O	R	W
F	N	O
J	X	U
P	M	E
O	D	V
R	E	T
E	H	L
Z	A	Y
O	D	G

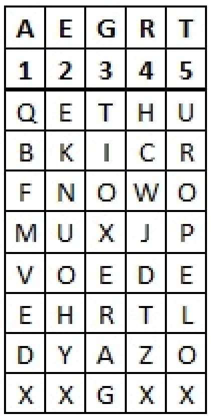
DECRYPTION:

To decrypt a ciphertext encoded with the Permutation Cipher, we have to write out the ciphertext in columns (the same number as the length of the keyword). We then order the keyword alphabetically, and write the ordered keyword at the top of the columns. We then rearrange the columns to reform the keyword, and read of the plaintext in rows.

As an example, we shall decipher the ciphertext "QETHU BKICR FNOWO MUXJP VOEDE EHRTL DYAZO XXGXX" with keyword great.

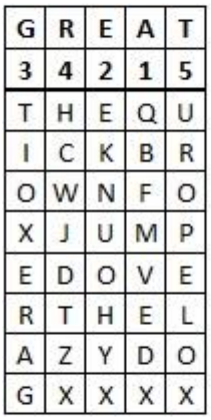
We start by writing a grid with 5 columns (the length of the keyword), and filling it with the ciphertext, row by row. We label these with the numbers

1-5, and rearrange the keyword into alphabetical order before adding this to the top (AEGRT).



A	E	G	R	T
1	2	3	4	5
Q	E	T	H	U
B	K	I	C	R
F	N	O	W	O
M	U	X	J	P
V	O	E	D	E
E	H	R	T	L
D	Y	A	Z	O
X	X	G	X	X

We now reorder the letters of the keyword to form the actual keyword, and at the same time, reorder the columns in the same way.



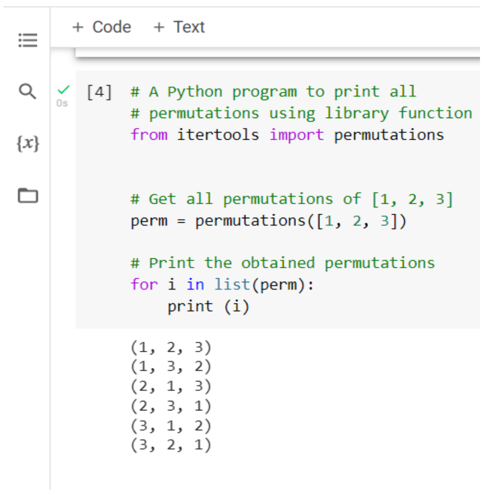
G	R	E	A	T
3	4	2	1	5
T	H	E	Q	U
I	C	K	B	R
O	W	N	F	O
X	J	U	M	P
E	D	O	V	E
R	T	H	E	L
A	Z	Y	D	O
G	X	X	X	X

CODE:

```python
# A Python program to print all
# permutations using library function
from itertools import permutations


# Get all permutations of [1, 2, 3]
perm = permutations([1, 2, 3])

# Print the obtained permutations
for i in list(perm):
    print (i)
```

OUTPUT:

MINI PROJECT REPORT



```
# A Python program to print all
# permutations using library function
from itertools import permutations


# Get all permutations of [1, 2, 3]
perm = permutations([1, 2, 3])

# Print the obtained permutations
for i in list(perm):
    print (i)
```

```
(1, 2, 3)
(1, 3, 2)
(2, 1, 3)
(2, 3, 1)
(3, 1, 2)
(3, 2, 1)
```

Mono-alphabetic Cipher –

In mono-alphabetic ciphers, each symbol in plain-text (eg; ‘o’ in ‘follow’) is mapped to one cipher-text symbol. No matter how many times a symbol occurs in the plain-text, it will correspond to the same cipher-text symbol. For example, if the plain-text is ‘follow’ and the mapping is :

- f -> g
- o -> p
- l -> m
- w -> x

The cipher-text is ‘gpmmpx’.

Types of mono-alphabetic ciphers are:

(a). Additive Cipher (Shift Cipher / Caesar Cipher) –

The simplest mono-alphabetic cipher is additive cipher. It is also referred to as ‘Shift Cipher’ or ‘Caesar Cipher’. As the name suggests, ‘addition modulus 2’ operation is performed on the plain-text to obtain a cipher-text.

C = (M + k) mod n

M = (C – k) mod n

where,

C -> cipher-text

M -> message/plain-text
k -> key

The key space is 26. Thus, it is not very secure. It can be broken by brute-force attack.

(b). Multiplicative Cipher –
The multiplicative cipher is similar to additive cipher except the fact that the key bit is multiplied to the plain-text symbol during encryption. Likewise, the cipher-text is multiplied by the multiplicative inverse of key for decryption to obtain back the plain-text
$C = (M * k) \bmod n$
$M = (C * k^{-1}) \bmod n$
where,
k^{-1} -> multiplicative inverse of k (key)
The key space of multiplicative cipher is 12. Thus, it is also not very secure.

(c). Affine Cipher –
The affine cipher is a combination of additive cipher and multiplicative cipher. The key space is 26 * 12 (key space of additive * key space of multiplicative) i.e. 312. It is relatively secure than the above two as the key space is larger.
Here two keys k_1 and k_2 are used.
$C = [(M * k_1) + k_2] \bmod n$
$M = [(C - k_2) * k_1^{-1}] \bmod n.$

Poly-alphabetic Cipher –
In poly-alphabetic ciphers, every symbol in plain-text is mapped to a different cipher-text symbol regardless of its occurrence. Every different occurrence of a symbol has different mapping to a cipher-text. For example, in the plain-text ‘follow’, the mapping is :
f -> q
o -> w
l -> e

l -> r
o -> t
w -> y

Thus, the cipher text is ‘qwerty’.

RAIL FENCE CIPHER:

Given a plain-text message and a numeric key, cipher/de-cipher the given text using Rail Fence algorithm.
The rail fence cipher (also called a zigzag cipher) is a form of transposition cipher. It derives its name from the way in which it is encoded.

EX:
Encryption
Input : "GeeksforGeeks "
Key = 3
Output : GsGsekfrek eoe
Decryption
Input : GsGsekfrek eoe
Key = 3
Output : "GeeksforGeeks "

Encryption:

- n 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.

For example, if the message is “GeeksforGeeks” and the number of rails = 3 then cipher is prepared as:

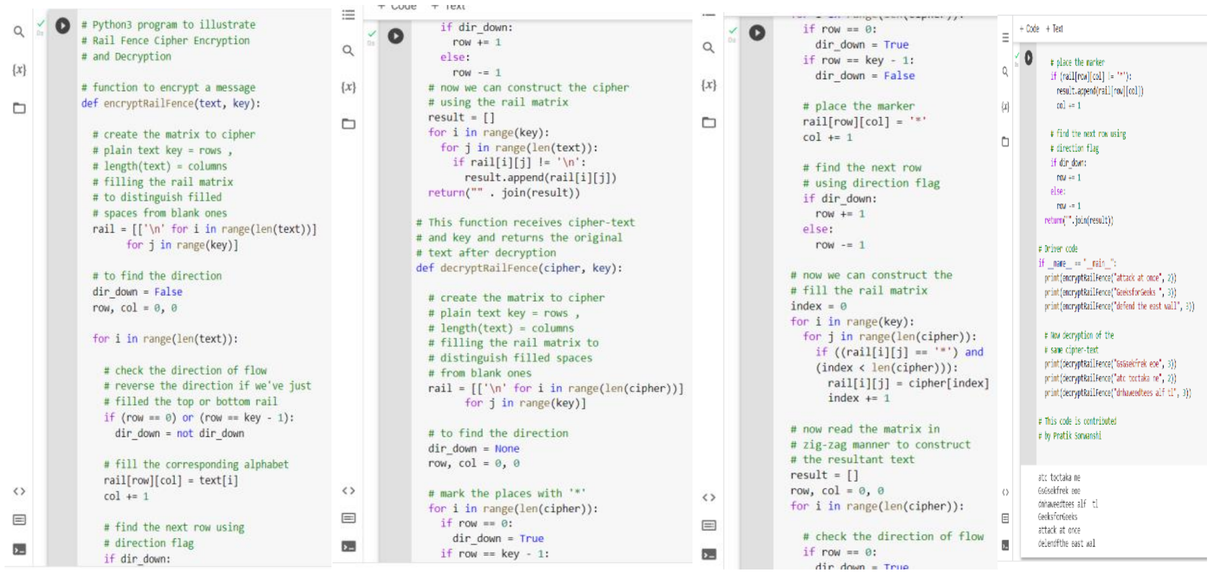
Decryption

MINI PROJECT REPORT

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.

CODE &OUTPUT:



```
# Python3 program to illustrate
# Rail Fence Cipher Encryption
# and Decryption

# function to encrypt a message
def encryptRailFence(text, key):

  # create the matrix to cipher
  # plain text key = rows ,
  # length(text) = columns
  # filling the rail matrix
  # to distinguish filled
  # spaces from blank ones
  rail = [['\n' for i in range(len(text))]
        for j in range(key)]

  # to find the direction
  dir_down = False
  row, col = 0, 0

  for i in range(len(text)):

    # check the direction of flow
    # reverse the direction if we've just
    # filled the top or bottom rail
    if (row == 0) or (row == key - 1):
      dir_down = not dir_down

    # fill the corresponding alphabet
    rail[row][col] = text[i]
    col += 1

    # find the next row using
    # direction flag
    if dir_down:
```

```
    if dir_down:
      row += 1
    else:
      row -= 1
  # now we can construct the cipher
  # using the rail matrix
  result = []
  for i in range(key):
    for j in range(len(text)):
      if rail[i][j] != '\n':
        result.append(rail[i][j])
  return("" . join(result))

# This function receives cipher-text
# and key and returns the original
# text after decryption
def decryptRailFence(cipher, key):

  # create the matrix to cipher
  # plain text key = rows ,
  # length(text) = columns
  # filling the rail matrix to
  # distinguish filled spaces
  # from blank ones
  rail = [['\n' for i in range(len(cipher))]
        for j in range(key)]

  # to find the direction
  dir_down = None
  row, col = 0, 0

  # mark the places with '*'
  for i in range(len(cipher)):
    if row == 0:
      dir_down = True
    if row == key - 1:
```

```
    if row == 0:
      dir_down = True
    if row == key - 1:
      dir_down = False

    # place the marker
    rail[row][col] = '*'
    col += 1

    # find the next row
    # using direction flag
    if dir_down:
      row += 1
    else:
      row -= 1

  # now we can construct the
  # fill the rail matrix
  index = 0
  for i in range(key):
    for j in range(len(cipher)):
      if ((rail[i][j] == '*') and
      (index < len(cipher))):
        rail[i][j] = cipher[index]
        index += 1

  # now read the matrix in
  # zig-zag manner to construct
  # the resultant text
  result = []
  row, col = 0, 0
  for i in range(len(cipher)):

    # check the direction of flow
    if row == 0:
      dir_down = True
```

```
    # place the marker
    if (rail[row][col] != '*'):
      result.append(rail[row][col])
      col += 1

    # find the next row using
    # direction flag
    if dir_down:
      row += 1
    else:
      row -= 1
  return("".join(result))

# Driver code
if __name__ == "__main__":
  print(encryptRailFence("attack at once", 2))
  print(encryptRailFence("GeeksforGeeks ", 3))
  print(encryptRailFence("defend the east wall", 3))

  # Now decryption of the
  # same cipher-text
  print(decryptRailFence("GsGsekfrek eoe", 3))
  print(decryptRailFence("atc toctaka ne", 2))
  print(decryptRailFence("dnhaweedtees alf tl", 3))

# This code is contributed
# by Pratik Somwanshi
```

```
atc toctaka ne
GsGsekfrek eoe
dnhaweedtees alf  tl
GeeksforGeeks
attack at once
delendfthe east wal
```

PLAYFAIR CIPHER:

The Playfair cipher or Playfair square or Wheatstone–Playfair cipher is a manual symmetric encryption technique and was the first literal diagram substitution cipher. The scheme was invented in 1854 by Charles Wheatstone, but bears the name of Lord Playfair for promoting its use.

The **Playfair cipher** was the first practical digraph substitution cipher. The scheme was invented in **1854** by **Charles Wheatstone** but was named after Lord Playfair who promoted the use of the cipher. In playfair cipher unlike traditional cipher we encrypt a pair of alphabets(digraphs) instead of a single alphabet. It was used for tactical purposes by British forces in the Second Boer War and in World War I and for the same purpose by the Australians during World War II. This was because Playfair is reasonably fast to use and requires no special equipment.

Rules for Encryption

1.Two plaintext letters in the same row of the matrix are each replaced by the letter to the right, with the first element of the row circularly following the last.

eb would be replaced by sd

ng would be replaced by gi/gj

2.Two plaintext letters that fall in the same column of the matrix are replaced by the letters beneath, with the top element of the column circularly following the bottom.

dt would be replaced by my

ty would be replaced by yr

3. Otherwise, each plaintext letter in a pair is replaced by the letter that lies in its own row and the column occupied by the other plaintext letter.

me would be replaced by gd

et would be replaced by do

Following these rules, the ciphertext becomes 'gd do gd rq pr sd hm *em bv*'.

MINI PROJECT REPORT

This cipher is more secure than simple substitution, but is still susceptible to ciphertext-only attacks by doing statistical frequency counts of pairs of letters, since each pair of letters always gets encrypted in the same fashion. Moreover, short keywords make the Playfair cipher even easier to crack.
CODE & OUTPUT:

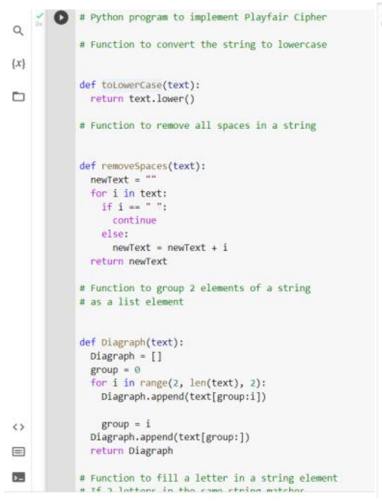
```python
# Python program to implement Playfair Cipher

# Function to convert the string to lowercase

def toLowerCase(text):
    return text.lower()
# Function to remove all spaces in a string
def removeSpaces(text):
    newText = ""
    for i in text:
        if i == " ":
            continue
        else:
            newText = newText + i
    return newText

# Function to group 2 elements of a string
# as a list element
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

# Function to fill a letter in a string element
# If 2 letters in the same string matches
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']

# Function to generate the 5x5 key square matrix
 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):
```

MINI PROJECT REPORT

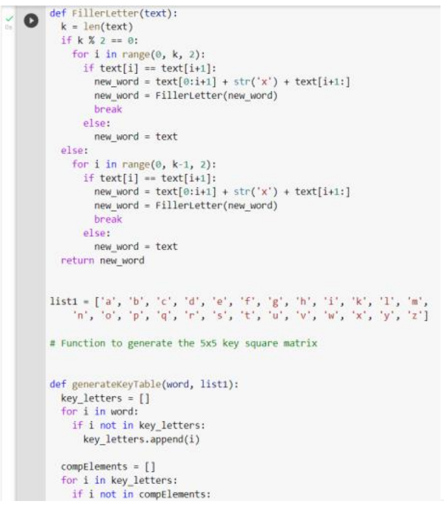
```
    char1 = ''
    if e1c == 4:
        char1 = matr[e1r][0]
    else:
        char1 = matr[e1r][e1c+1]
   char2 = ''
    if e2c == 4:
        char2 = matr[e2r][0]
    else:
        char2 = matr[e2r][e2c+1]

    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 = 'instruments'
text_Plain = removeSpaces(toLowerCase(text_Plain))
PlainTextList = Diagraph(FillerLetter(text_Plain))
if len(PlainTextList[-1]) != 2:
    PlainTextList[-1] = PlainTextList[-1]+'z'

key = "Monarchy"
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:
Key text: Monarchy

Plain text: instruments

Cipher text: gatlmzclrqtx
```

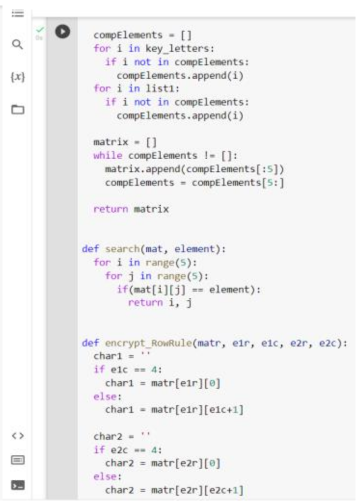
MINI PROJECT REPORT



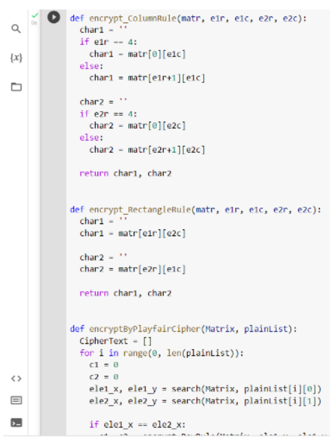
```
# Python program to implement Playfair Cipher

# Function to convert the string to lowercase


def toLowerCase(text):
  return text.lower()

# Function to remove all spaces in a string


def removeSpaces(text):
  newText = ""
  for i in text:
    if i == " ":
      continue
    else:
      newText = newText + i
  return newText

# Function to group 2 elements of a string
# as a list element


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

# Function to fill a letter in a string element
```



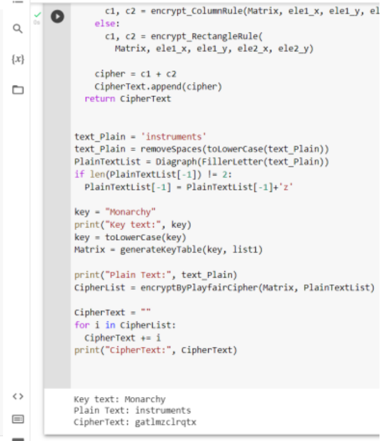
```
def FillerLetter(text):
  k = len(text)
  if k % 2 == 0:
    for i in range(0, k, 2):
      if text[i] == text[i+1]:
        new_word = text[0:i+1] + str('x') + text[i+1:]
        new_word = FillerLetter(new_word)
        break
      else:
        new_word = text
  else:
    for i in range(0, k-1, 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']

# Function to generate the 5x5 key square matrix


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 = []
  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]
```



```
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_ColumnRule(Matrix, ele1_x, ele1_y, el
    else:
      c1, c2 = encrypt_RectangleRule(
        Matrix, ele1_x, ele1_y, ele2_x, ele2_y)

    cipher = c1 + c2
    CipherText.append(cipher)
  return CipherText


text_Plain = 'instruments'
text_Plain = removeSpaces(toLowerCase(text_Plain))
PlainTextList = Diagraph(FillerLetter(text_Plain))
if len(PlainTextList[-1]) != 2:
  PlainTextList[-1] = PlainTextList[-1]+'z'

key = "Monarchy"
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)
```

```
Key text: Monarchy
Plain Text: instruments
CipherText: gatlmzclrqtx
```

Hill Cipher:

Hill cipher is a polygraphic substitution cipher based on linear algebra.Each letter is represented by a number modulo 26. Often the simple scheme A = 0, B = 1, …, Z = 25 is used, but this is not an essential feature of the cipher. To encrypt a message, each block of n letters (considered as an n-component vector) is multiplied by an invertible n × n matrix, against modulus 26. To decrypt the message, each block is multiplied by the inverse of the matrix used for encryption.

The matrix used for encryption is the cipher key, and it should be chosen randomly from the set of invertible n × n matrices (modulo 26).

Examples:

Input : Plaintext: ACT

MINI PROJECT REPORT

Key: GYBNQKURP

Output : Ciphertext: POH

Encryption:

We have to encrypt the message 'ACT' (n=3).The key is 'GYBNQKURP' which can be written as the nxn matrix:

$$\begin{bmatrix} 6 & 24 & 1 \\ 13 & 16 & 10 \\ 20 & 17 & 15 \end{bmatrix}$$

The message 'ACT' is written as vector:

$$\begin{bmatrix} 0 \\ 2 \\ 19 \end{bmatrix}$$

The enciphered vector is given as:

$$\begin{bmatrix} 6 & 24 & 1 \\ 13 & 16 & 10 \\ 20 & 17 & 15 \end{bmatrix} \begin{bmatrix} 0 \\ 2 \\ 19 \end{bmatrix} = \begin{bmatrix} 67 \\ 222 \\ 319 \end{bmatrix} \equiv \begin{bmatrix} 15 \\ 14 \\ 7 \end{bmatrix} (\text{mod } 26)$$

which corresponds to ciphertext of 'POH'

Decryption

To decrypt the message, we turn the ciphertext back into a vector, then simply multiply by the inverse matrix of the key matrix (IFKVIVVMI in letters).The inverse of the matrix used in the previous example is:

$$\begin{bmatrix} 6 & 24 & 1 \\ 13 & 16 & 10 \\ 20 & 17 & 15 \end{bmatrix}^{-1} \equiv \begin{bmatrix} 8 & 5 & 10 \\ 21 & 8 & 21 \\ 21 & 12 & 8 \end{bmatrix} (\text{mod } 26)$$

For the previous Ciphertext 'POH':

$$\begin{bmatrix} 8 & 5 & 10 \\ 21 & 8 & 21 \\ 21 & 12 & 8 \end{bmatrix} \begin{bmatrix} 15 \\ 14 \\ 7 \end{bmatrix} \equiv \begin{bmatrix} 260 \\ 574 \\ 539 \end{bmatrix} \equiv \begin{bmatrix} 0 \\ 2 \\ 19 \end{bmatrix} (\text{mod } 26)$$

which gives us back 'ACT'.
Assume that all the alphabets are in upper case.
Below is the implementation of the above idea for n=3

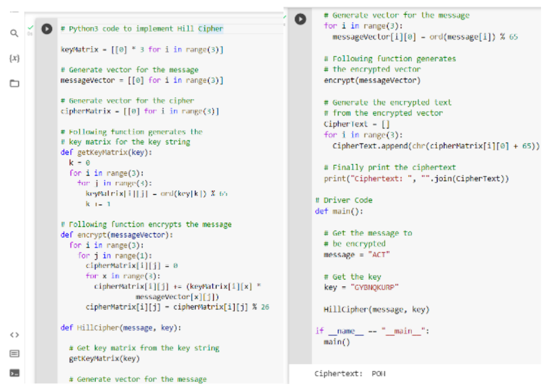
CODE:

MINI PROJECT REPORT

```
# Python3 code to implement Hill Cipher

keyMatrix = [[0] * 3 for i in range(3)]

# Generate vector for the message
messageVector = [[0] for i in range(3)]

# Generate vector for the cipher
cipherMatrix = [[0] for i in range(3)]

# Following function generates the
# key matrix for the key string
def getKeyMatrix(key):
    k = 0
    for i in range(3):
        for j in range(3):
            keyMatrix[i][j] = ord(key[k]) % 65
            k += 1

# Following function encrypts the message
def encrypt(messageVector):
    for i in range(3):
        for j in range(1):
            cipherMatrix[i][j] = 0
            for x in range(3):
                cipherMatrix[i][j] += (keyMatrix[i][x] *
                                       messageVector[x][j])
            cipherMatrix[i][j] = cipherMatrix[i][j] % 26

def HillCipher(message, key):

    # Get key matrix from the key string
    getKeyMatrix(key)

    # Generate vector for the message
    for i in range(3):
        messageVector[i][0] = ord(message[i]) % 65

    # Following function generates
    # the encrypted vector
    encrypt(messageVector)

    # Generate the encrypted text
    # from the encrypted vector
    CipherText = []
    for i in range(3):
```

MINI PROJECT REPORT

```
            CipherText.append(chr(cipherMatrix[i][0] + 65))

    # Finally print the ciphertext
    print("Ciphertext: ", "".join(CipherText))

# Driver Code
def main():

    # Get the message to
    # be encrypted
    message = "ACT"

    # Get the key
    key = "GYBNQKURP"

    HillCipher(message, key)

if __name__ == "__main__":
    main()
OUTPUT:
Ciphertext: POH
```



```
# Python3 code to implement Hill Cipher

keyMatrix = [[0] * 3 for i in range(3)]

# Generate vector for the message
messageVector = [[0] for i in range(3)]

# Generate vector for the cipher
cipherMatrix = [[0] for i in range(3)]

# Following function generates the
# key matrix for the key string
def getKeyMatrix(key):
  k = 0
  for i in range(3):
    for j in range(3):
      keyMatrix[i][j] = ord(key[k]) % 65
      k += 1

# Following function encrypts the message
def encrypt(messageVector):
  for i in range(3):
    for j in range(1):
      cipherMatrix[i][j] = 0
      for x in range(3):
        cipherMatrix[i][j] += (keyMatrix[i][x] *
                    messageVector[x][j])
      cipherMatrix[i][j] = cipherMatrix[i][j] % 26

def HillCipher(message, key):

  # Get key matrix from the key string
  getKeyMatrix(key)

  # Generate vector for the message
```

```
  # Generate vector for the message
  for i in range(3):
    messageVector[i][0] = ord(message[i]) % 65

  # Following function generates
  # the encrypted vector
  encrypt(messageVector)

  # Generate the encrypted text
  # from the encrypted vector
  CipherText = []
  for i in range(3):
    CipherText.append(chr(cipherMatrix[i][0] + 65))

  # Finally print the ciphertext
  print("Ciphertext: ", "".join(CipherText))

# Driver Code
def main():

  # Get the message to
  # be encrypted
  message = "ACT"

  # Get the key
  key = "GYBNQKURP"

  HillCipher(message, key)

if __name__ == "__main__":
  main()

Ciphertext:  POH
```

VIGENERE CIPHER:

Vigenere Cipher is a method of encrypting alphabetic text. It uses a simple form of polyalphabetic substitution. A polyalphabetic cipher is any cipher based on substitution, using multiple substitution alphabets. The encryption of the original text is done using the *Vigenère square or Vigenère table*.

- The table consists of the alphabets written out 26 times in different rows, each alphabet shifted cyclically to the left compared to the previous alphabet, corresponding to the 26 possible Caesar Ciphers.

MINI PROJECT REPORT

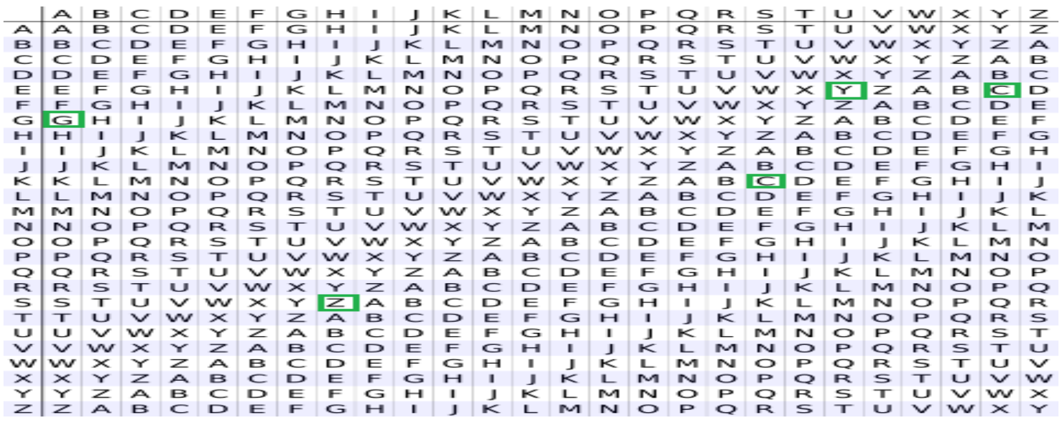
- At different points in the encryption process, the cipher uses a different alphabet from one of the rows.
- The alphabet used at each point depends on a repeating keyword.

Example:

```
Input : Plaintext :   GEEKSFORGEEKS

Keyword :  AYUSH

Output : Ciphertext :  GCYCZFMLYLEIM
```

Encryption:

The first letter of the plaintext, G is paired with A, the first letter of the key. So use row G and column A of the Vigenère square, namely G. Similarly, for the second letter of the plaintext, the second letter of the key is used, the letter at row E, and column Y is C. The rest of the plaintext is enciphered in a similar fashion.



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

Decryption:

Decryption is performed by going to the row in the table corresponding to the key, finding the position of the ciphertext letter in this row, and then using the column's label as the plaintext. For example, in row A (from AYUSH), the ciphertext G appears in column G, which is the first plaintext letter. Next, we go to row Y (from AYUSH), locate the ciphertext C which is found in column E, thus E is the second plaintext letter.

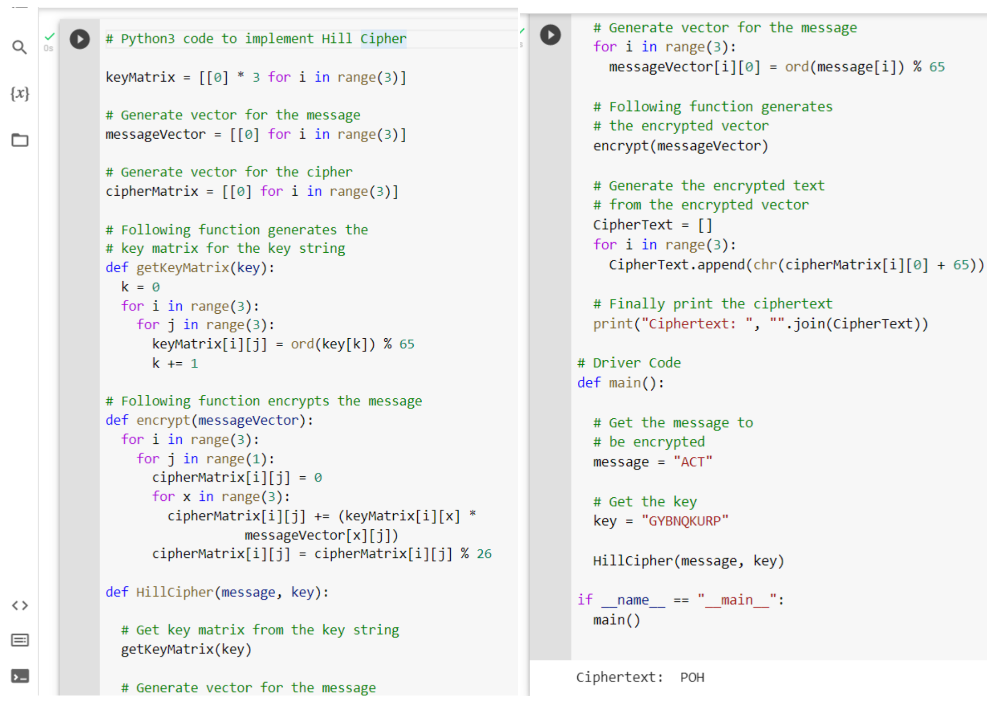
CODE:

```
# Python code to implement
# Vigenere Cipher
```

MINI PROJECT REPORT

```
# This function generates the
# key in a cyclic manner until
# it's length isn't equal to
# the length of original text
def generateKey(string, key):
  key = list(key)
  if len(string) == len(key):
    return(key)
  else:
    for i in range(len(string) -
          len(key)):
      key.append(key[i % len(key)])
  return("" . join(key))

# This function returns the
# encrypted text generated
# with the help of the key
def cipherText(string, key):
  cipher_text = []
  for i in range(len(string)):
    x = (ord(string[i]) +
      ord(key[i])) % 26
    x += ord('A')
    cipher_text.append(chr(x))
  return("" . join(cipher_text))

# This function decrypts the
# encrypted text and returns
# the original text
def originalText(cipher_text, key):
  orig_text = []
  for i in range(len(cipher_text)):
    x = (ord(cipher_text[i]) -
      ord(key[i]) + 26) % 26
    x += ord('A')
    orig_text.append(chr(x))
  return("" . join(orig_text))

# Driver code
if __name__ == "__main__":
  string = "GEEKSFORGEEKS"
  keyword = "AYUSH"
  key = generateKey(string, keyword)
  cipher_text = cipherText(string,key)
  print("Ciphertext :", cipher_text)
  print("Original/Decrypted Text :",
    originalText(cipher_text, key))
```

```
OUTPUT:

Ciphertext : GCYCZFMLYLEIM

Original/Decrypted Text : GEEKSFORGEEKS
```



```
# Python3 code to implement Hill Cipher

keyMatrix = [[0] * 3 for i in range(3)]

# Generate vector for the message
messageVector = [[0] for i in range(3)]

# Generate vector for the cipher
cipherMatrix = [[0] for i in range(3)]

# Following function generates the
# key matrix for the key string
def getKeyMatrix(key):
  k = 0
  for i in range(3):
    for j in range(3):
      keyMatrix[i][j] = ord(key[k]) % 65
      k += 1

# Following function encrypts the message
def encrypt(messageVector):
  for i in range(3):
    for j in range(1):
      cipherMatrix[i][j] = 0
      for x in range(3):
        cipherMatrix[i][j] += (keyMatrix[i][x] *
                   messageVector[x][j])
      cipherMatrix[i][j] = cipherMatrix[i][j] % 26

def HillCipher(message, key):

  # Get key matrix from the key string
  getKeyMatrix(key)

  # Generate vector for the message
```

```
  # Generate vector for the message
  for i in range(3):
    messageVector[i][0] = ord(message[i]) % 65

  # Following function generates
  # the encrypted vector
  encrypt(messageVector)

  # Generate the encrypted text
  # from the encrypted vector
  CipherText = []
  for i in range(3):
    CipherText.append(chr(cipherMatrix[i][0] + 65))

  # Finally print the ciphertext
  print("Ciphertext: ", "".join(CipherText))

# Driver Code
def main():

  # Get the message to
  # be encrypted
  message = "ACT"

  # Get the key
  key = "GYBNQKURP"

  HillCipher(message, key)

if __name__ == "__main__":
  main()
```

```
Ciphertext:  POH
```

CAESAR CIPHER:

The Caesar Cipher technique is one of the earliest and simplest methods of encryption technique. It's simply a type of substitution cipher, i.e., each letter of a given text is replaced by a letter with a fixed number of positions down the alphabet. For example with a shift of 1, A would be replaced by B, B would become C, and so on. The method is apparently named after Julius Caesar, who apparently used it to communicate with his officials.

Thus to cipher a given text we need an integer value, known as a shift which indicates the number of positions each letter of the text has been moved down.
The encryption can be represented using modular arithmetic by first transforming the letters into numbers, according to the scheme, A = 0, B = 1,…, Z = 25. Encryption of a letter by a shift n can be described mathematically as:

MINI PROJECT REPORT

Examples :

```
Text : ABCDEFGHIJKLMNOPQRSTUVWXYZ
Shift: 23
Cipher: XYZABCDEFGHIJKLMNOPQRSTUVW

Text : ATTACKATONCE
Shift: 4
Cipher: EXXEGOEXSRGI
```

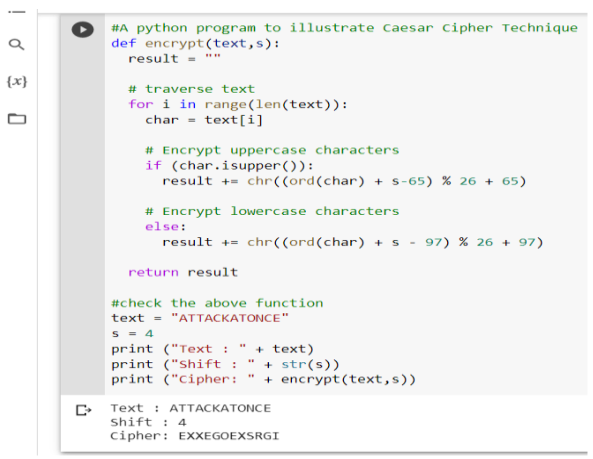
Algorithm for Caesar Cipher:
Input:

1. A String of lower case letters, called Text.
2. An Integer between 0-25 denoting the required shift.

```
CODE:
#A python program to illustrate 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

#check the above function
text = "ATTACKATONCE"
s = 4
print ("Text : " + text)
print ("Shift : " + str(s))
print ("Cipher: " + encrypt(text,s))
```

```
OUTPUT:
Text : ATTACKATONCE

Shift: 4

Cipher: EXXEGOEXSRGI
```

MINI PROJECT REPORT



```
#A python program to illustrate 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

#check the above function
text = "ATTACKATONCE"
s = 4
print ("Text : " + text)
print ("Shift : " + str(s))
print ("Cipher: " + encrypt(text,s))
```

```
Text : ATTACKATONCE
Shift : 4
Cipher: EXXEGOEXSRGI
```

KEYWORD CIPHER:

Keyword cipher is a form of monoalphabetic substitution. A keyword is used as the key, and it determines the letter matchings of the cipher alphabet to the plain alphabet. Repeats of letters in the word are removed, then the cipher alphabet is generated with the keyword matching to A, B, C, etc. until the keyword is used up, whereupon the rest of the ciphertext letters are used in alphabetical order, excluding those already used in the key.

Encryption:

The first line of input contains the keyword which you wish to enter. The second line of input contains the string which you have to encrypt.

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

With KRYPTOS as the keyword, all As become Ks, all Bs becoming Rs, and so on. Encrypting the message "knowledge is power" using the keyword "Kryptos":

Encrypting the message: Knowledge is Power

Encoded message: IlmWjbaEb GQ NmWbp

EX:

```
Input :
Keyword : secret
Message : Zombie Here
```

```
Output :
Ciphered String : ZLJEFT DTOT
```

NOTE:

- All the messages are encoded in uppercase.
- Whitespace, special characters, and numbers do not take into consideration keywords although you can put them in there.
- While encrypting the message, whitespace, special characters and numbers remain unaffected

Decryption:

To decode the message you check the position of the given message in encrypting text with the plain text.

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

Deciphered Text: DECIPHERED
```

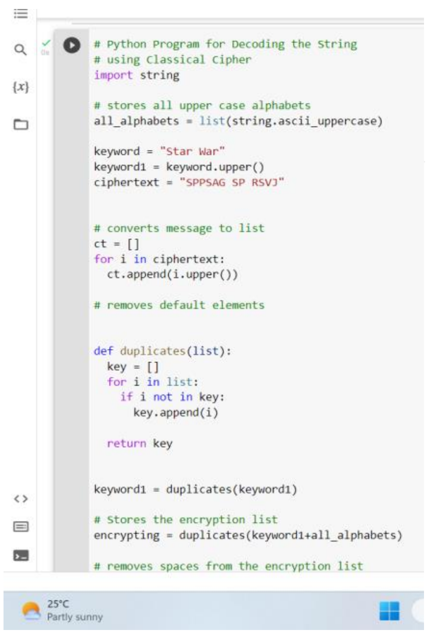
CODE:

```
# Python Program for Decoding the String
# using Classical Cipher
import string

# stores all upper case alphabets
all_alphabets = list(string.ascii_uppercase)

keyword = "Star War"
keyword1 = keyword.upper()
ciphertext = "SPPSAG SP RSVJ"


# converts message to list
ct = []
for i in ciphertext:
    ct.append(i.upper())

# removes default elements


def duplicates(list):
    key = []
    for i in list:
        if i not in key:
            key.append(i)

    return key


keyword1 = duplicates(keyword1)

# Stores the encryption list
encrypting = duplicates(keyword1+all_alphabets)

# removes spaces from the encryption list
for i in encrypting:
    if(i == ' '):
        encrypting.remove(' ')

# maps each element of the message to the encryption list and stores it
in ciphertext
message = ""
for i in range(len(ct)):
    if(ct[i] != ' '):
        message = message+all_alphabets[encrypting.index(ct[i])]
    else:
        message = message+' '

print("Keyword : ", keyword)
print("Ciphered Text : ", ciphertext)
print("Message before Ciphering : ", message)
```

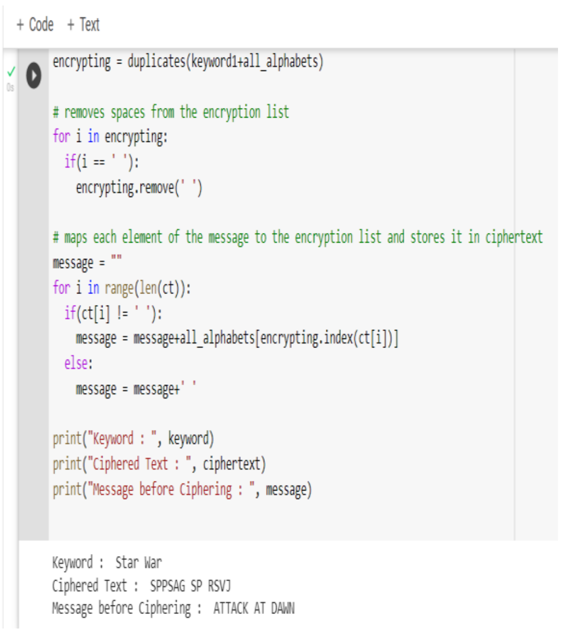
OUTPUT:

```
('Keyword : ', 'Star War')

('Ciphered Text : ', 'SPPSAG SP RSVJ')

('Message before Ciphering : ', 'ATTACK AT DAWN')
```

MINI PROJECT REPORT



```
# Python Program for Decoding the String
# using Classical Cipher
import string

# stores all upper case alphabets
all_alphabets = list(string.ascii_uppercase)

keyword = "Star War"
keyword1 = keyword.upper()
ciphertext = "SPPSAG SP RSVJ"


# converts message to list
ct = []
for i in ciphertext:
  ct.append(i.upper())

# removes default elements


def duplicates(list):
  key = []
  for i in list:
    if i not in key:
      key.append(i)

  return key


keyword1 = duplicates(keyword1)

# Stores the encryption list
encrypting = duplicates(keyword1+all_alphabets)

# removes spaces from the encryption list
```



```
encrypting = duplicates(keyword1+all_alphabets)

# removes spaces from the encryption list
for i in encrypting:
  if(i == ' '):
    encrypting.remove(' ')

# maps each element of the message to the encryption list and stores it in ciphertext
message = ""
for i in range(len(ct)):
  if(ct[i] != ' '):
    message = message+all_alphabets[encrypting.index(ct[i])]
  else:
    message = message+' '

print("Keyword : ", keyword)
print("Ciphered Text : ", ciphertext)
print("Message before Ciphering : ", message)
```

```
Keyword :  Star War
Ciphered Text :  SPPSAG SP RSVJ
Message before Ciphering :  ATTACK AT DAWN
```

MINI PROJECT REPORT