Assignment
On
Implementation of Different Cipher Codes in Python

Course Title: Cryptography & Information Security Lab
Course Code: CSEL - 4110

Submitted To:
Dr. Mohammed Nasir Uddin
Professor
Department of CSE
Jagannath University, Dhaka

Submitted By:
Najnin Sultana Shirin
ID: B190305031
4th Year 1st Semester

Department of Computer Science & Engineering
Jagannath University, Dhaka
Date of Submission: 24.10.2024

Table of Contents

1. Additive Cipher 03
2. Multiplicative Cipher 05
3. Affine Cipher 07
4. Playfair Cipher 09
5. Vigenère Cipher 16
6. Data Encryption Standard (DES) 19
7. Advance Encryption Standard (AES) 20
8. RSA (Rivest-Shamir-Adleman) 21
9. El Gamal Cipher 23
11. Rabin Cryptosystem.. 25
12. Data Signature Standard (DSS).................................. 27
13. Diffie Hellman Key Exchange 29

Additive Cipher / Ceaser Cipher / Shift Cipher

A substitution cipher replaces one symbol with another. If the symbol in the plaintext are alphabetic characters, we replace one character with another. For example, we can replace letter A with letter D, and letter B with letter E. The simplest monoalphabetic substitution cipher is additive cipher. Here,

The formula of encryption is: $E(x) = (x + n) \bmod 26$
The formula of decryption is: $D(x) = (x - n) \bmod 26$
Where,
E denotes the encryption
D denotes the decryption
x denotes the letter's value
n denotes the key value (shift value)

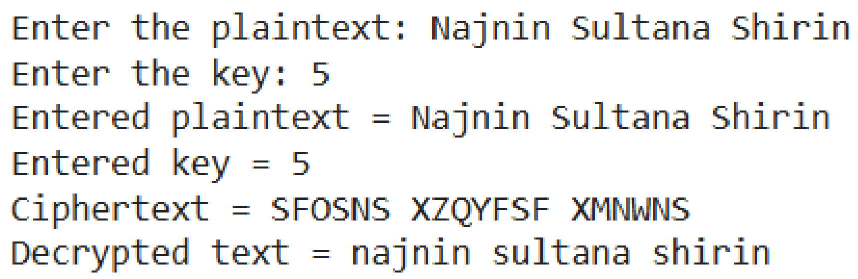
Julius Caeser used an additive cipher to communicate with his officers. For this reason, **additive cipher** is sometimes referred to as the **Caesar cipher**. Caeser used a key of 3 for his communications.

Code:

```
#Encryption Function
def encrypt(plaintext, key):
    plaintext = plaintext.upper()
    ciphertext = ""
    for i in range(len(plaintext)):
        char = plaintext[i]
        if char == ' ':
            shifted_char = char
        else:
            shifted_char = chr((ord(char) - 65 + key) % 26)
            shifted_char = chr(ord(shifted_char) + 65)
        ciphertext = ciphertext + shifted_char
    return ciphertext
```

```
#Decryption Function
def decrypt(ciphertext, key):
    ciphertext = ciphertext.lower()
    decrypted_text = ""
    for i in range(len(ciphertext)):
        char = ciphertext[i]
        if char == ' ':
            shifted_char = char
        else:
            shifted_char = chr((ord(char) - 97 - key) % 26)
            shifted_char = chr(ord(shifted_char) + 97)
        decrypted_text = decrypted_text + shifted_char
    return decrypted_text

#Input
plaintext = input("Enter the plaintext: ")
key = int(input("Enter the key: "))
#Output
print("Entered plaintext = " + plaintext)
print("Entered key = " + str(key))
ciphertext = encrypt(plaintext, key)
print("Ciphertext = " + ciphertext)
decrypted_text = decrypt(ciphertext, key)
print("Decrypted text = " + decrypted_text)
```

Output:



```
Enter the plaintext: Najnin Sultana Shirin
Enter the key: 5
Entered plaintext = Najnin Sultana Shirin
Entered key = 5
Ciphertext = SFOSNS XZQYFSF XMNWNS
Decrypted text = najnin sultana shirin
```

Multiplicative Cipher

A multiplicative cipher is a type of cipher that comes under a **monoalphabetic** cipher, in which each letter that is present in the plaintext is replaced by a corresponding letter of the ciphertext, according to a fixed multiplication key.
The decryption process is simply the reverse of the encryption process, i.e., by multiplying the numerical value of each letter in the ciphertext by multiplicative inverse of the key and then taking the result modulo of the key. Here,

The formula of encryption is: $E(x) = (P \times K) \bmod 26$
The formula of decryption is: $D(x) = (C \times K^{-1}) \bmod 26$

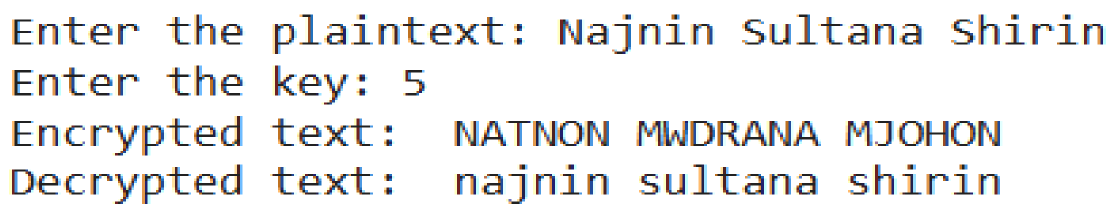
Where,
P denotes the plaintext
C denotes the ciphertext
K denotes the key value

Code:

```
# Input
plaintext = input("Enter the plaintext: ")
key = int(input("Enter the key: "))

# Encryption
plaintext = plaintext.upper()
ciphertext = ""
for i in range(len(plaintext)):
    ch = plaintext[i]
    if ch == " ":
        substituted_ch = ch
    else:
        substituted_ch = chr(((ord(ch) - 65) * key) % 26)
        substituted_ch = chr(ord(substituted_ch) + 65)
    ciphertext = ciphertext + substituted_ch
print("Encrypted text: ", ciphertext)
```

```python
# Finding the multiplicative inverse of key value
# Using Extended Euclidean Algorithm
r1 = 26
r2 = key
t1 = 0
t2 = 1
while r2 > 0:
    q = r1 // r2
    r = r1 - q * r2
    r1 = r2
    r2 = r
    t = t1 - q * t2
    t1 = t2
    t2 = t

if r1 == 1:
    ciphertext = ciphertext.lower()
    new_plaintext = ""
    for i in range(len(ciphertext)):
        ch = ciphertext[i]
        if ch == " ":
            substituted_ch = ch
        else:
            substituted_ch = chr(((ord(ch) - 97) * t1) % 26)
            substituted_ch = chr(ord(substituted_ch) + 97)
        new_plaintext = new_plaintext + substituted_ch
    print("Decrypted text: ", new_plaintext)

else:
    print("Multiplicative inverse of the key doesn't exist")
```

Output:



```
Enter the plaintext: Najnin Sultana Shirin
Enter the key: 5
Encrypted text:  NATNON MWDRANA MJOHON
Decrypted text:  najnin sultana shirin
```

Affine Cipher

The Affine cipher is a type of **monoalphabetic** substitution cipher, where each letter in a plaintext is mapped to its numeric equivalent, encrypted using a simple mathematical function, and converted back to a letter.

Here,

The formula of encryption is: $C = ((P \times K_1) + K_2) \bmod 26$

The formula of decryption is: $P = ((C - K_2) \times K_1^{-1}) \bmod 26$

Where,

P denotes the plaintext

C denotes the ciphertext

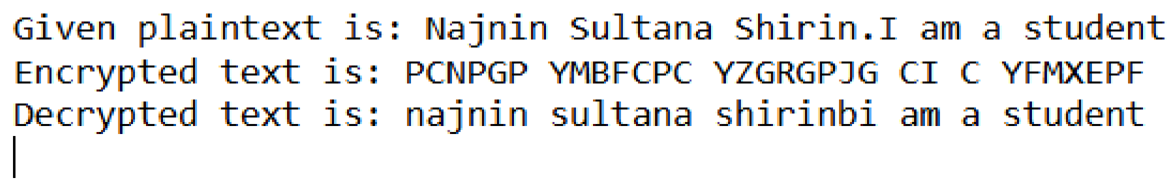
K_1 and K_2 denote the key value

Code:

```
# Encryption Function
def encryption(plaintext, key1, key2):
    ciphertext = ""
    # This for loop is for the traversing the text.
    for i in range(len(plaintext)):
        char = plaintext[i]
        # This condition is for keeping the "space" same on the cipher text.
        if char == " ":
            ciphertext = ciphertext + char
        # Ascii value of upper case letters start from 65.
        elif char.isupper():
            ciphertext = ciphertext + chr(
                ((((ord(char) - 65) * key1) + key2) % 26) + 65
            )
        # Ascii value of lower case letters start from 97.
        else:
            ciphertext = ciphertext + chr(
                ((((ord(char) - 97) * key1) + key2) % 26) + 97
            )
    return ciphertext
```

```
# Decryption Function
def decryption(ciphertext, key1, key2):
    plaintext = ""
    # Finding the multiplicative inverse of key1
    mi = pow(key1, -1, 26)
    # This for loop is for traversing the text.
    for i in range(len(ciphertext)):
        char = ciphertext[i]
        # This condition is for keeping the "space" same on the plaintext.
        if char == " ":
            plaintext = plaintext + char
        # Ascii value of upper case letters start from 65.
        elif char.isupper():
            plaintext = plaintext + chr(((((ord(char) - 65) - key2) * mi) % 26) + 65)
        # Ascii value of lower case letters start from 97.
        else:
            plaintext = plaintext + chr(((((ord(char) - 97) - key2) * mi) % 26) + 97)
    return plaintext

input = open("input.txt", "r+")
output = open("output.txt", "w")

plaintext = input.read()
key1 = 7
key2 = 2

ciphertext = encryption(plaintext, key1, key2)
decodedtext = decryption(ciphertext, key1, key2)
output.write("Given plaintext is: " + plaintext + "\n")
output.write("Encrypted text is: " + ciphertext.upper() + "\n")
output.write("Decrypted text is: " + decodedtext.lower() + "\n")

# This line is for the closing the files.
input.close()
output.close()
```

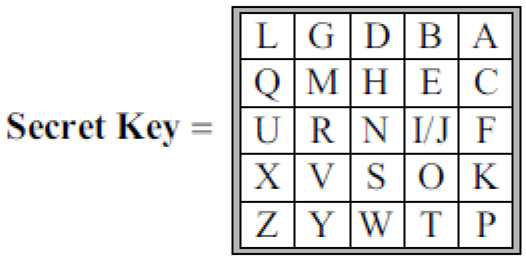
Output:



```
Given plaintext is: Najnin Sultana Shirin.I am a student
Encrypted text is: PCNPGP YMBFCPC YZGRGPJG CI C YFMXEPF
Decrypted text is: najnin sultana shirinbi am a student
```

Playfair Cipher

The Playfair cipher is a **polyalphabetic** cipher. The secret key in this cipher is made of **25 alphabet** letters arranged in a **5 × 5 matrix** (letters **I and J** are considered the same when encrypting).



Secret Key =

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

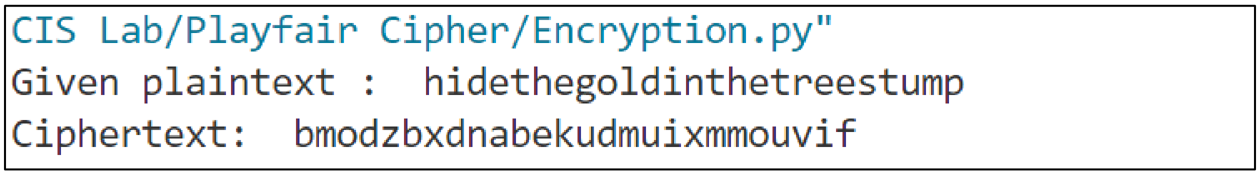
Fig: An example of a secret key in Playfair cipher

The cipher uses **three** rules for encryption:
1. If the two letters in a pair are located in the **same row** of the secret key, the corresponding encrypted character for each letter is the **next letter to the right** in the same row (with wrapping to the beginning of the row if the plaintext letter is the last character in the row).
2. If the two letters in a pair are located in the **same column** of the secret key, the corresponding encrypted character for each letter is the **letter beneath** it in the same column (with wrapping to the beginning of the column if the plaintext letter is the last character in the column).
3. If the two letters in a pair are not in the same row or column of the secret, the corresponding encrypted character for each letter is a letter that is in its own row but in the same column as the other letter.
Before encryption, if the two letters in a pair are the same, a bogus letter is inserted to separate them. After inserting bogus letters, if the number of characters in the plaintext is odd, one extra bogus character is added at the end to make the number of characters even.

Code (Encryption):

```
import numpy
# Initialize the matrix
matrix = numpy.array(
    [
        ["p", "l", "a", "y", "f"],
        ["i", "r", "e", "x", "m"],
        ["b", "c", "d", "g", "h"],
        ["k", "n", "o", "q", "s"],
        ["t", "u", "v", "w", "z"],
    ]
)
# This line is for the transpose the matrix.
result = numpy.transpose(matrix)

# Input PlainText.
plaintext = "hidethegoldinthetreestump"
# This line replace all "j" in the plaintext to "i"
plaintext = plaintext.replace("j", "i")
# This list is use for store the plaintext pair
plaintextpair = []
ciphertextpair = []

# Apply Rule 1.
# If both letter are same (or only one letter is left)
# Add on "X" after the first letter.
i = 0
while i < len(plaintext):
    a = plaintext[i]
    b = ""
    # If the letter is the last charater of the plaintext.
    if (i + 1) == len(plaintext):
        b = "x"
    else:
        b = plaintext[i + 1]
        # If the two character is not same then this makes pair.
    if a != b:
        plaintextpair.append(a + b)
        i += 2
    else:
        plaintextpair.append(a + "x")
        i += 1
ciphertext = ""
for pair in plaintextpair:
    applied_rule = True

    # Apply Rule 2.
    # If the letters appear at the same row of the table
    # Replace them with the letters to their immediate right respectively
    if applied_rule:
        for row in range(5):
            if pair[0] in matrix[row] and pair[1] in matrix[row]:
                for i in range(5):
                    if matrix[row][i] == pair[0]:
                        j0 = i
                for i in range(5):
                    if matrix[row][i] == pair[1]:
                        j1 = i
                applied_rule = False
                ciphertextpair.append(
                    (matrix[row][(j0 + 1) % 5]) + (matrix[row][(j1 + 1) % 5]) )
               ciphertext = (
                    ciphertext + (matrix[row][(j0 + 1) % 5])+(matrix[row][(j1 + 1) % 5])
      )
    # Apply rule 3.
    # If the letter appear on the same column of table.
    # Replace them with the letter immediate below respectively
    if applied_rule:
        for row in range(5):
            if pair[0] in result[row] and pair[1] in result[row]:
                for i in range(5):
                    if matrix[i][row] == pair[0]:
                        j0 = i
                    for i in range(5):
                        if matrix[i][row] == pair[1]:
                            j1 = i

                    applied_rule = False
                ciphertextpair.append(
                    (matrix[(j0 + 1) % 5][row]) + (matrix[(j1 + 1) % 5][row])
                )
                ciphertext = (
                    ciphertext
                    + (matrix[(j0 + 1) % 5][row])
                    + (matrix[(j1 + 1) % 5][row])
                )
    # Apply rule 4
    # If the letters are not in the same row or same column
    # replace them with the letters on the same row respectively but at the
    # other pair of the corners of the rectangle define by the orginal pair.

    if applied_rule:
        for row in range(5):
            for col in range(5):
                if matrix[row][col] == pair[0]:
                    x0 = row
                    y0 = col
            for row1 in range(5):
                for col1 in range(5):
                    if matrix[row1][col1] == pair[1]:
                        x1 = row1
                        y1 = col1
        ciphertextpair.append((matrix[x0][y1]) + (matrix[x1][y0]))
        ciphertext = ciphertext + (matrix[x0][y1]) + (matrix[x1][y0])

print("Given plaintext : ", plaintext)
print("Ciphertext: ", ciphertext)
```

Output (Encryption):



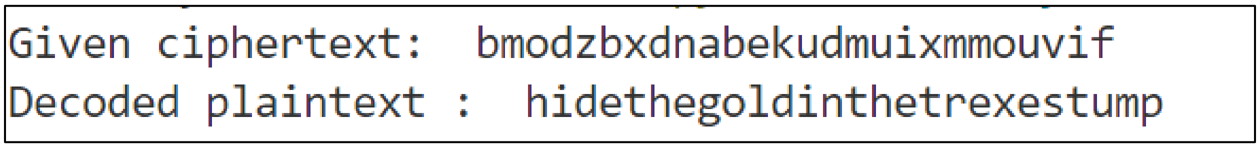
```
CIS Lab/Playfair Cipher/Encryption.py"
Given plaintext :  hidethegoldinthetreestump
Ciphertext:  bmodzbxdnabekudmuixmmouvif
```

Code (Decryption):

```python
import numpy
# Initialize the matrix
matrix = numpy.array(
    [
        ["p", "l", "a", "y", "f"],
        ["i", "r", "e", "x", "m"],
        ["b", "c", "d", "g", "h"],
        ["k", "n", "o", "q", "s"],
        ["t", "u", "v", "w", "z"],
    ]
)
# This line is for the transpose the matrix.
result = numpy.transpose(matrix)
# Given Ciphertext
ciphertext = "bmodzbxdnabekudmuixmmouvif"
# This line replace all "j" in the plaintext to "i"
ciphertext = ciphertext.replace("j", "i")
# This list is use for store the ciphertext and plaintext pair
ciphertextpair = []
plaintextpair = []

# Apply Rule 1.
# make the pair of two letter.
i = 0
while i < len(ciphertext):
    a = ciphertext[i]
    b = ciphertext[i + 1]
    ciphertextpair.append(a + b)
    i = i + 2
```

```
plaintext = ""
for pair in ciphertextpair:
    applied_rule = True

    # Apply Rule 2.
    # If the letters appear at the same row of the table
    # Replace them with the letters to their immediate left respectively
    if applied_rule:
        for row in range(5):
            if pair[0] in matrix[row] and pair[1] in matrix[row]:
                for i in range(5):
                    if matrix[row][i] == pair[0]:
                        j0 = i

                for i in range(5):
                    if matrix[row][i] == pair[1]:
                        j1 = i

                applied_rule = False
                plaintextpair.append(
                    (matrix[row][(j0 - 1) % 5]) + (matrix[row][(j1 - 1) % 5])
                )
                plaintext += (matrix[row][(j0 - 1) % 5]) + (matrix[row][(j1 - 1) % 5])
    # Apply rule 3.
    # If the letter appear on the same column of table.
    # Replace them with the letter immediate upper respectively
    if applied_rule:
        for row in range(5):
            if pair[0] in result[row] and pair[1] in result[row]:
                for i in range(5):
                    if matrix[i][row] == pair[0]:
                        j0 = i

                    for i in range(5):
                        if matrix[i][row] == pair[1]:
                            j1 = i

                    applied_rule = False
                plaintextpair.append(
                    (matrix[(j0 - 1) % 5][row]) + (matrix[(j1 - 1) % 5][row])
                )
                plaintext += (matrix[(j0 - 1) % 5][row]) + (matrix[(j1 - 1) % 5][row])

    # Apply rule 4.
    # If the letters are not in the same row or same column
    # replace them with the letters on the same row respectively but at the
    # other pair of the corners of the rectangle define by the orginal pair.

    if applied_rule:
        for row in range(5):
            for col in range(5):
                if matrix[row][col] == pair[0]:
                    x0 = row
                    y0 = col
            for row1 in range(5):
                for col1 in range(5):
                    if matrix[row1][col1] == pair[1]:
                        x1 = row1
                        y1 = col1
        plaintextpair.append((matrix[x0][y1]) + (matrix[x1][y0]))
        plaintext += (matrix[x0][y1]) + (matrix[x1][y0])

print("Given ciphertext: ", ciphertext)
print("Decoded plaintext : ", plaintext)
```

Output (Decryption):



```
Given ciphertext:  bmodzbxdnabekudmuixmmouvif
Decoded plaintext :  hidethegoldinthetrexestump
```

Vigenère Cipher

Vigenère cipher is a simple form of **polyalphabetic** substitution cipher.
Here,
The formula of encryption is: $C_i = (P_i + K_i) \bmod 26$
The formula of decryption is: $P_i = (C_i - K_i) \bmod 26$

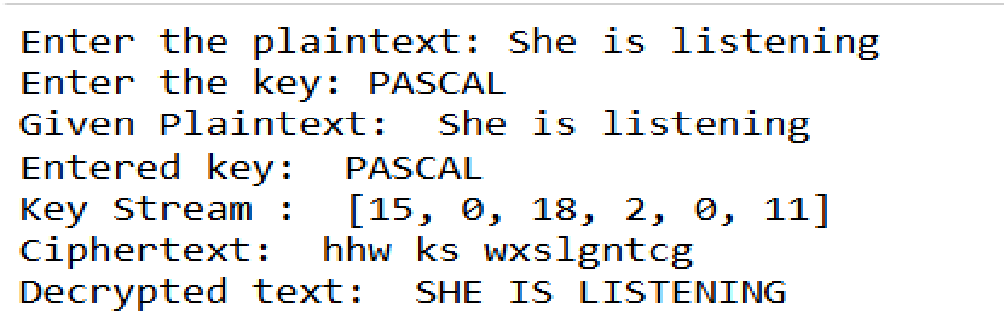
Where,
P_i denotes the plaintext stream
C_i denotes the ciphertext stream
$K = [\ (k_1, k_2, k_3,\dots,k_m), (k_1, k_2, k_3,\dots,k_m),..]$
For generating key, the given keyword is repeated in a circular manner until it matches the length of the plain text.

Code:

```
# This function generates the key in a cyclic manner
# until it's length isn't equal to the length of original text
#Key Generation
def key_generation(key):
    key_len = len(key)
    key_stream = [0]*key_len
    key = key.lower()
    for i in range(key_len):
        order = ord(key[i]) - 97
        key_stream[i] = order
    return key_stream
```

```
# Encryption Function
def encryption(plaintext, key_stream):
    text = plaintext.lower()
    key_size = len(key_stream)
    ciphertext = ""
    j = 0
    for char in text:
        order = ord(char)
        if order>=97 and order<=122:
            #Storing the key for current plaintext character
            key = key_stream[j]
            if j==(key_size-1):
                j = 0
            else:
                j = j+1
            #Calculating the ciphertext charater
            order = order - 97
            order = (order + key) % 26
            order = order + 97
            new_char = chr(order)
            ciphertext = ciphertext + new_char
        else:
            ciphertext = ciphertext + char
    return ciphertext

#Decryption Function
def decryption(ciphertext, key_stream):
    text = ciphertext.upper()
    key_size = len(key_stream)
    plaintext = ""
    j = 0
    for char in text:
        order = ord(char)
        if order>=65 and order<=90:
            #Storing the key for current ciphertext character
            key = key_stream[j]
            if j==(key_size-1):
                j = 0
            else:
                j = j+1
            #Calculating the plaintext charater
            order = order - 65
            order = (order - key) % 26
            order = order + 65
            new_char = chr(order)
            plaintext = plaintext + new_char
        else:
            plaintext = plaintext + char
    return plaintext

#Input Section
plaintext = input("Enter the plaintext: ")
key = input("Enter the key: ")

#Function Calling
key_stream = key_generation(key)
ciphertext = encryption(plaintext, key_stream)
decrypted_text = decryption(ciphertext, key_stream)

#Output Section
print("Given Plaintext: ", plaintext)
print("Entered key: ", key)
print("Key Stream : ", key_stream)
print("Ciphertext: ", ciphertext)
print("Decrypted text: ", decrypted_text)
```

Output:



```
Enter the plaintext: She is listening
Enter the key: PASCAL
Given Plaintext:  She is listening
Entered key:  PASCAL
Key Stream :  [15, 0, 18, 2, 0, 11]
Ciphertext:  hhw ks wxslgntcg
Decrypted text:  SHE IS LISTENING
```

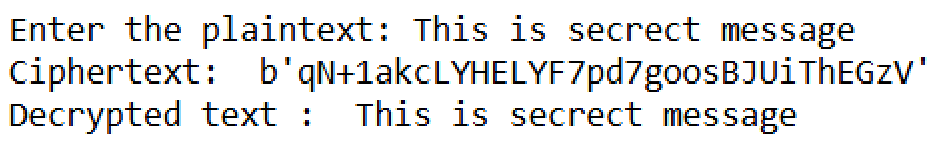
Digital Encryption Standard (DES)

DES is a block cipher and encrypts data in blocks of size of **64 bits** each, which means 64 bits of plain text go as the input to DES, which produces 64 bits of ciphertext. The same algorithm and key are used for encryption and decryption, with minor differences. The key length is **56 bits**.

Encryption & Decryption:

```
import base64
from Crypto.Cipher import DES
from Crypto.Random import get_random_bytes

#Input plaintext
plaintext = input("Enter the plaintext: ");
#Padding the plaintext
while len(plaintext) % 8 != 0:
    plaintext = plaintext + " "
#Create a random key
key = get_random_bytes(8)

#Create model of the cipher
des = DES.new(key, DES.MODE_ECB)

#Encryption Part
ciphertext = des.encrypt(plaintext.encode('utf-8'))
print("Ciphertext: ", base64.b64encode(ciphertext))
#Decryptiom Part
decryptedtext = des.decrypt(ciphertext)
print("Decrypted text : ", decryptedtext.decode()
```

Output:



```
Enter the plaintext: This is secrect message
Ciphertext:  b'qN+1akcLYHELYF7pd7goosBJUiThEGzV'
Decrypted text :  This is secrect message
```

Advance Encryption Standard (AES)

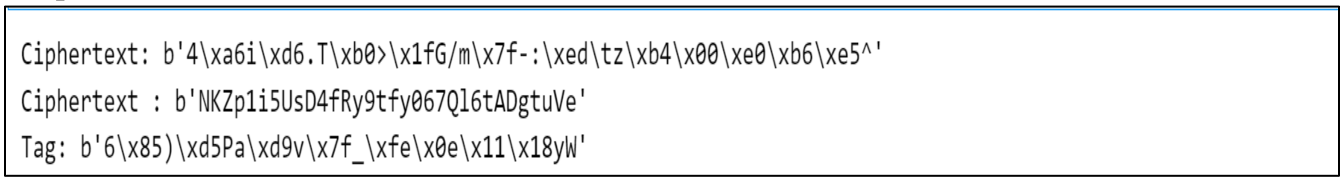
- AES is a **block** cipher.
- The key size can be 128/192/256 bits.
- Encrypts data in blocks of **128 bits** each.

That means it takes 128 bits as input and outputs 128 bits of encrypted cipher text as output. AES relies on substitution-permutation network principle which means it is performed using a series of linked operations which involves replacing and shuffling of the input data.

Encryption:

```
import base64
from Crypto.Cipher import AES
from Crypto.Random import get_random_bytes

plaintext = b'This is a secret message'
# The get_random_bytes() function takes a parameter that specifies the desired length
of the random byte string to generate. In this case, the parameter is 16 corresponds to
16 bytes or 128 bits.
key = get_random_bytes(16)

# Create an AES cipher object with a 128-bit key
cipher = AES.new(key, AES.MODE_EAX)
# The authentication tag provides a unique identifier for the encrypted data, ensuring
that it has not been tampered with or modified.
ciphertext, tag = cipher.encrypt_and_digest(plaintext)

# Print the encrypted ciphertext and tag
print("Ciphertext:", ciphertext)
print("Ciphertext :",base64.b64encode(ciphertext))
print("Tag:", tag)
```

Output:

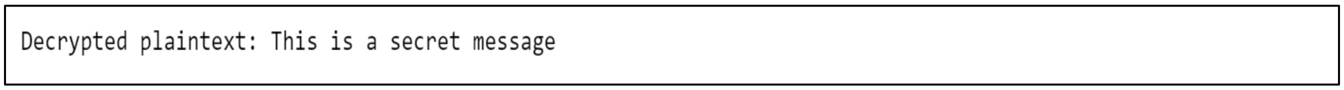


```
Ciphertext: b'4\xa6i\xd6.T\xb0>\x1fG/m\x7f-:\xed\tz\xb4\x00\xe0\xb6\xe5^'
Ciphertext : b'NKZp1i5UsD4fRy9tfy067Ql6tADgtuVe'
Tag: b'6\x85)\xd5Pa\xd9v\x7f_\xfe\x0e\x11\x18yW'
```

Decryption:

```
# Nonce serves as an additional input to the encryption algorithm and helps ensure the uniqueness and security of the ciphertext produced.
decrypt_cipher = AES.new(key, AES.MODE_EAX, nonce=cipher.nonce)
# Decrypt the ciphertext
decrypted_plaintext = decrypt_cipher.decrypt_and_verify(ciphertext, tag)
# Print the decrypted plaintext
print("Decrypted plaintext:", decrypted_plaintext.decode())
```

Output:



```
Decrypted plaintext: This is a secret message
```

RSA (Rivest-Shamir-Adleman)

The most common **public-key algorithm** is the RSA cryptosystem, named for its inventors (Rivest, Shamir, and Adleman). RSA uses **two exponents**, **e** and **d**.
Where,
e is public
d is private

Suppose P is the plaintext and C is the ciphertext. Alice uses,

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

to create ciphertext C from plaintext P;
Bob uses,

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

to retrieve the plaintext sent by Alice.

The modulus n, a very large number, is created during the key generation process. Encryption and decryption use modular exponentiation.

Code:

```
#A Python Code for Encryption Using RSA Algorithm
from Crypto.PublicKey import RSA
from Crypto.Cipher import PKCS1_OAEP

#Function for generating public and private key
def generate_key_pair():
    key = RSA.generate(2048)
    public_key = key.publickey().export_key()
    private_key = key.export_key()
    return public_key, private_key

#Encryption Function
def encrypt(message, public_key):
    cipher = PKCS1_OAEP.new(RSA.import_key(public_key))
    encrypted_message = cipher.encrypt(message)
    return encrypted_message

#Decryption Function
def decrypt(encrypted_message, private_key):
    cipher = PKCS1_OAEP.new(RSA.import_key(private_key))
    decrypted_message = cipher.decrypt(encrypted_message)
    return decrypted_message

# Example usage
plaintext = b"This is a secret message from TAJ"
print("Plaintext:", plaintext)
print("----output----")

# Generate key pair
public_key, private_key = generate_key_pair()

# Encrypt the message
encrypted_message = encrypt(plaintext, public_key)
print("Encrypted message:", encrypted_message.hex())

# Decrypt the message
decrypted_message = decrypt(encrypted_message, private_key)
print("Decrypted message:", decrypted_message.decode())
```

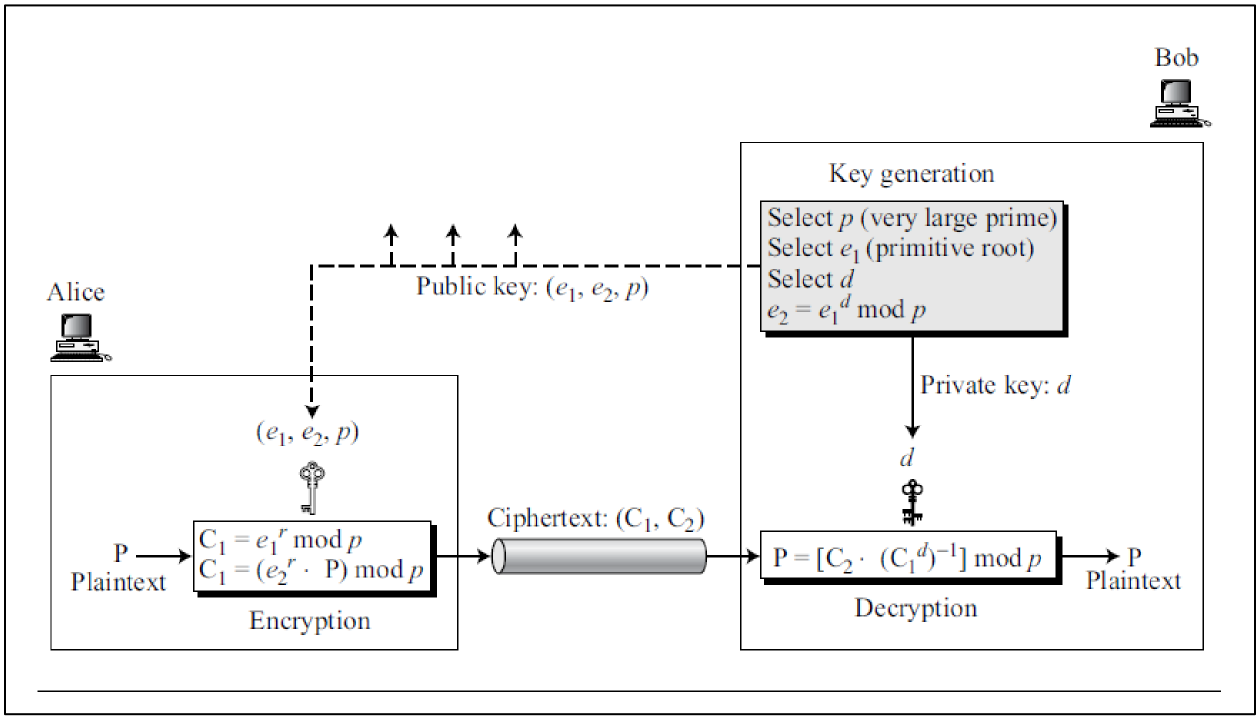
Output:

```
Enter the plain text: This is a secret message from TAJ
Plaintext: This is a secret message from TAJ
Plaintext bytes: b'This is a secret message from TAJ'
----output----
Encrypted message: 233963506c0d79cffee876c691dc7674470d13d9163060fc94cc399a6d66e0aa629261574240fd405544f5ff537ae970bf6a790c394a
497d913ece25337ccb0529c03a103a9809e69a0785e19cb5d6c64911b2a604264dbb971e9614867af392ab44f775776833bac809ba0d58e5d0acfa23a230c46
1ecd3715f2a15c46e75fd4eb64d0cbec8368bb4ad003446142117660a8195b96620a93862fcc503c4162ed76bb2dfa01f71daf03542efcaa6bce44067df0cce
ceb3a234ce01e9ee1cbff9701b33c2ffabd53c12a1c2f34251861740485bd57a05c152900c5d1797feaaccf4faf7860dbdb34043779b7e3c9458cc1895aef2c
45284714880aa987627501b
Decrypted message: This is a secret message from TAJ
```

El Gamal Cipher

Besides RSA and Rabin, another **public-key** cryptosystem is ElGamal, named after its inventor, Taher ElGamal. ElGamal is based on the discrete logarithm problem.

Basic idea of ElGamal cryptosystem is showed in the following figure:



Code:

```
# Sympy is a Python library for symbolic mathematics.
from sympy import primitive_root,randprime
import random

# The number for which you want to find the primitive root
prime = randprime(124,10**3)
root = primitive_root(prime)


d=random.randint(1,(prime-2)) # It is private key.
e=(pow(root,d)%prime)  # It is public key.
r=random.randint(1,10)  # Select a random integer.

#Define the plaintext.
plaintext = "This is a secret message"
```

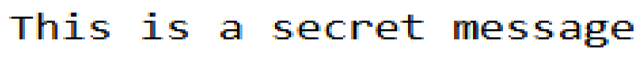
```
# Encryption Algorithm.
ciphertext=[]

for char in plaintext:
  ciphertext1=(pow(root,r)%prime)
  ciphertext2=((ord(char)*pow(e,r))%prime)
  ciphertext.append((ciphertext1,ciphertext2))

print(ciphertext)
```

Output:

```
[(535, 163), (535, 373), (535, 24), (535, 129), (535, 336), (535, 24), (535, 129), (53
5, 336), (535, 659), (535, 336), (535, 129), (535, 701), (535, 680), (535, 478), (535,
701), (535, 499), (535, 336), (535, 66), (535, 701), (535, 129), (535, 129), (535, 65
9), (535, 3), (535, 701)]
```

```
#Decryption Algorithm
plaintext=""

for pair in ciphertext:
  ciphertext1,ciphertext2=pair
  value=pow(ciphertext1,d)
  multinv = pow(value,-1,prime)
  decrypt_char = (ciphertext2*multinv) % prime
  plaintext += chr(decrypt_char)

print(plaintext)
```

Output:



```
This is a secret message
```

Rabin Cryptosystem

The Rabin Cryptosystem is an **asymmetric** (public-key) cryptosystem based on the difficulty of factoring large integers, similar to RSA. It was invented by Michael Rabin in 1979.

1. Key Generation:

- Step 1: Choose two large distinct prime numbers, p and q.

- Step 2: Compute (**n = p*q**). n is the public modulus.

- Step 3: The **public** key is **n**, and the **private** key is the pair (**p, q**).

2. Encryption:

- Step 1: Let the plaintext message m be an integer where **$0 \le m < n$**.

- Step 2: The ciphertext c is computed using the formula: **[$c = m^2 \bmod n$]**

- Step 3: The ciphertext c is then sent to the recipient.

3. Decryption:

- Step 1: Upon receiving the ciphertext c, the recipient computes four possible square roots of c modulo n. This requires using the **Chinese Remainder Theorem** (CRT) and the private key (p, q).

- Step 2: Compute the square roots modulo p and q (since both p and q are prime, this is straightforward).

- Step 3: Combine these results using CRT to find the four possible values of m, since squaring a number modulo n could yield four results.

- Step 4: The correct plaintext m is identified by additional information (e.g., the original message format or context).

Code:

```
import random
def key_generation(p, q):
    n = p * q
    return (p, q), n

def encryption(n, M):
    C = pow(M, 2, n)
    return C

def extended_gcd(a, b):
    if b == 0:
        return a, 1, 0
    gcd, x1, y1 = extended_gcd(b, a % b)
    x = y1
    y = x1 - (a // b) * y1
    return gcd, x, y
def decryption(p, q, C):

    a1 = pow(C, (p + 1) // 4, p)
    b1 = pow(C, (q + 1) // 4, q)

    gcd, M1, M2 = extended_gcd(p, q)
    r1 = (M1 * p * b1 + M2 * q * a1) % (p * q)
    r2 = (M1 * p * b1 - M2 * q * a1) % (p * q)
    r3 = (-r1) % (p * q)
    r4 = (-r2) % (p * q)
    return r1, r2, r3, r4
```

```
p = int(input("Enter the value of p : "))
q = int(input("Enter the value of q : "))

privateKey, publicKey = key_generation(p, q)
print(f"Public key (n): {publicKey}, Private key (p, q): {privateKey}")

M = int(input("Enter the plaintext (as an integer): "))
C = encryption(publicKey, M)
print("Ciphertext: "+str(C))

r1, r2, r3, r4 = decryption(privateKey[0], privateKey[1], C)
print(f"Possible plaintexts: {r1}, {r2}, {r3}, {r4}")
```

Output:

```
Enter the value of p : 7
Enter the value of q : 3
Public key (n): 21, Private key (p, q): (7, 3)
Enter the plaintext (as an integer): 123
Ciphertext: 9
Possible plaintexts: 18, 3, 3, 18
```

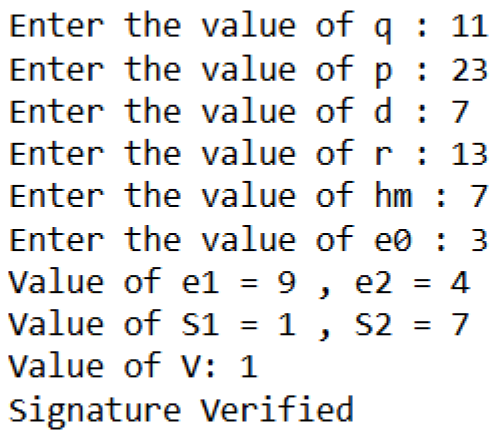
Data Signature Standard (DSS)

DSS is a way of authenticating a digital data coming from a trusted source. Digital Signature Standard (DSS) is a Federal Information Processing Standard (FIPS) which defines algorithms that are used to generate digital signatures with the help of Secure Hash Algorithm (SHA) for the authentication of electronic documents. DSS only provides us with the digital signature function and not with any encryption or key exchanging strategies.

Code:

```
q=int(input("Enter the value of p : "))
p=int(input("Enter the value of q : "))
d=int(input("Enter the value of d : "))
r=int(input("Enter the value of r : "))
hm=int(input("Enter the value of hm : "))
e0=int(input("Enter the value of e0 : "))
e1 = pow(e0, (p-1)//q, p)
e2 = pow(e1, d, p)
print("Value of e1 =",e1,", e2 =",e2)
s1 = pow(e1, r, p) % q
inverse_r = pow(r, -1, q)
s2 = ((hm + d*s1)*inverse_r) % q
print("Value of S1 =",s1,", S2 =",s2)
inverse_s2 = pow(s2, -1, q)
v = ((pow(e1, hm*inverse_s2)*pow(e2, s1*inverse_s2))%p)%q
print("Value of V:", v)
if v==s1:
    print("Signature Verified")
else:
    print("Not Verified")
```

Output:

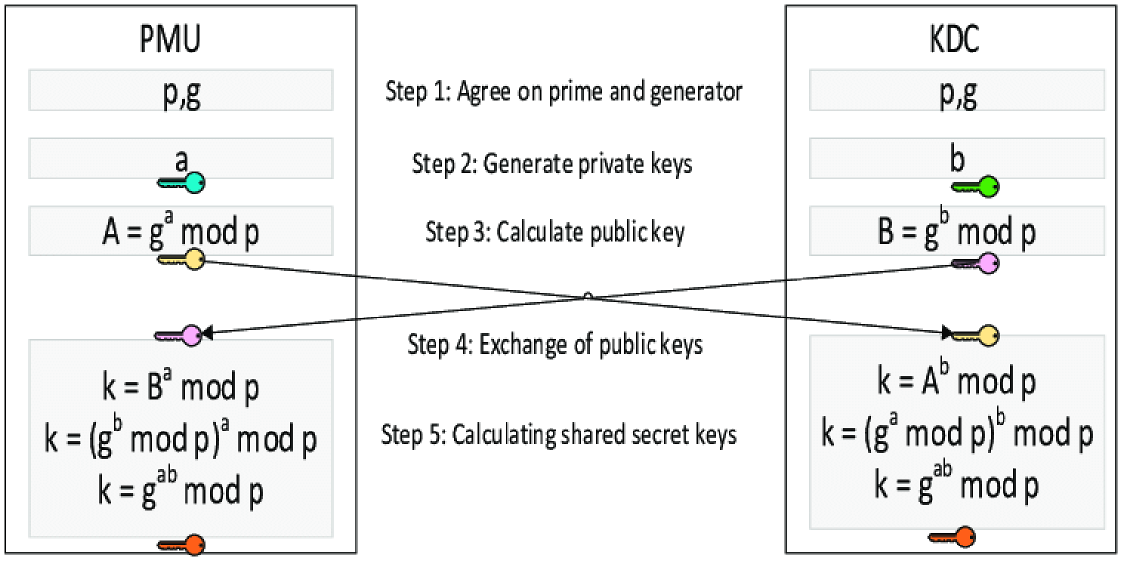


```
Enter the value of q : 11
Enter the value of p : 23
Enter the value of d : 7
Enter the value of r : 13
Enter the value of hm : 7
Enter the value of e0 : 3
Value of e1 = 9 , e2 = 4
Value of S1 = 1 , S2 = 7
Value of V: 1
Signature Verified
```

Diffie Hellman Key Exchange

Diffie-Hellman key exchange is a type of digital encryption in which two different parties securely exchange cryptographic keys over a public channel without their communication being sent over the internet. Both parties use symmetric cryptography to encrypt and decrypt their messages.

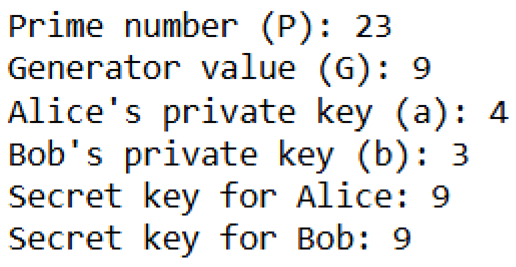
Basic idea of Diffie Hellman Key Exchange is showed in the following figure:


PMU
p,g
a
$A = g^a \bmod p$
$k = B^a \bmod p$
$k = (g^b \bmod p)^a \bmod p$
$k = g^{ab} \bmod p$
Step 1: Agree on prime and generator
Step 2: Generate private keys
Step 3: Calculate public key
Step 4: Exchange of public keys
Step 5: Calculating shared secret keys
KDC
p,g
b
$B = g^b \bmod p$
$k = A^b \bmod p$
$k = (g^a \bmod p)^b \bmod p$
$k = g^{ab} \bmod p$

Code:

```python
def calculate_power(base, exponent, modulus):
    """Power function to return value of (base ^ exponent) mod modulus."""
    if exponent == 1:
        return base
    else:
        return pow(base, exponent) % modulus
# Driver code
if __name__ == "__main__":
    prime = 23
    generator = 9
    private_key_alice = 4
    private_key_bob = 3
    # Generating public keys
    x = calculate_power(generator, private_key_alice, prime)
    y = calculate_power(generator, private_key_bob, prime)

    # Generating secret keys after the exchange of keys
    secret_key_alice = calculate_power(y, private_key_alice, prime)
    secret_key_bob = calculate_power(x, private_key_bob, prime)

    # Output
    print("Prime number (P):", prime)
    print("Generator value (G):", generator)
    print("Alice's private key (a):", private_key_alice)
    print("Bob's private key (b):", private_key_bob)
    print("Secret key for Alice:", secret_key_alice)
    print("Secret key for Bob:", secret_key_bob)
```

Output:



```
Prime number (P): 23
Generator value (G): 9
Alice's private key (a): 4
Bob's private key (b): 3
Secret key for Alice: 9
Secret key for Bob: 9
```