**Experiment 1; XOR with 0:**

#include <stdio.h>

void xorstr(char \*str){

while (\*str){

\*str= \*str ^ 0;

str++;

}

}

int main() {

char a[]="Hello";

char str[] = "Hello World";

printf("Original String: %s \n",str);

xorstr(str);

printf("XOR Result: %s \n",str);

return 0;

}

**Experiment 2: AND and XOR operation with 127:**

#include <stdio.h>

void performops(char \*str, int len){

printf("Original Text : %s \n",str);

printf("Results:\n");

for (int i=0; i<len; i++){

char andres=str[i]&127;

char xorres=str[i]^127;

printf("Character: %c, AND Result: %c \t, XOR Result: %c\n",str[i],andres,xorres);

}

}

int main(){

char text[]="Hello World";

int length=strlen(text);

performops(text,length);

return 0;

}

**Experiment 3:**

**CEASER CIPHER:** (PYTHON)

text=input("Enter Text :")

keylen=3

res=''

for i in text:

if i.isupper():

res+=chr((ord(i)+keylen-65)%26 + 65)

else:

res+=chr((ord(i)+keylen-97)%26+97)

print("Original Text",text)

print("Cipher Text",res)

**SUBSTITUTION CIPHER:** (PYTHON)

# Online Python compiler (interpreter) to run Python online.

# Write Python 3 code in this online editor and run it.

text=input("Enter Text :")

keylen=int(input("enter key length :"))

res=''

for i in text:

if i.isupper():

res+=chr((ord(i)+keylen-65)%26 + 65)

else:

res+=chr((ord(i)+keylen-97)%26+97)

print("Original Text",text)

print("Cipher Text",res)

**HILL CIPHER:** (PYTHON)

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

messageVector = [[0] for i in range(3)]

cipherMatrix = [[0] for i in range(3)]

def getKeyMatrix(key):

k = 0

for i in range(3):

for j in range(3):

keyMatrix[i][j] = ord(key[k]) % 65

k += 1

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):

getKeyMatrix(key)

for i in range(3):

messageVector[i][0] = ord(message[i]) % 65

CipherText = []

encrypt(messageVector)

for i in range(3):

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

print("Ciphertext: ", "".join(CipherText))

# Driver Code

def main():

message =input()

key = "GYBNQKURP"

HillCipher(message, key)

if \_\_name\_\_ == "\_\_main\_\_":

main()

PLAY FAIR CIPHER:

def prepare\_key(key):

key = key.upper().replace("J", "I") # Convert to uppercase and replace J with I

key\_set = set(key)

alphabet = "ABCDEFGHIKLMNOPQRSTUVWXYZ" # I and J are treated as the same letter

for char in alphabet:

if char not in key\_set:

key += char

key\_matrix = [key[i:i+5] for i in range(0, len(key), 5)]

return key\_matrix

def find\_coordinates(matrix, char):

for i, row in enumerate(matrix):

if char in row:

return i, row.index(char)

return None

def encrypt(plaintext, key\_matrix):

plaintext = plaintext.upper().replace("J", "I") # Convert to uppercase and replace J with I

# Add padding if the length of plaintext is odd

if len(plaintext) % 2 != 0:

plaintext += 'X'

plaintext\_pairs = [plaintext[i:i+2] for i in range(0, len(plaintext), 2)]

ciphertext = ""

for pair in plaintext\_pairs:

row1, col1 = find\_coordinates(key\_matrix, pair[0])

row2, col2 = find\_coordinates(key\_matrix, pair[1])

if row1 == row2:

ciphertext += key\_matrix[row1][(col1 + 1) % 5] + key\_matrix[row2][(col2 + 1) % 5]

elif col1 == col2:

ciphertext += key\_matrix[(row1 + 1) % 5][col1] + key\_matrix[(row2 + 1) % 5][col2]

else:

ciphertext += key\_matrix[row1][col2] + key\_matrix[row2][col1]

return ciphertext

def decrypt(ciphertext, key\_matrix):

ciphertext\_pairs = [ciphertext[i:i+2] for i in range(0, len(ciphertext), 2)]

plaintext = ""

for pair in ciphertext\_pairs:

row1, col1 = find\_coordinates(key\_matrix, pair[0])

row2, col2 = find\_coordinates(key\_matrix, pair[1])

if row1 == row2:

plaintext += key\_matrix[row1][(col1 - 1) % 5] + key\_matrix[row2][(col2 - 1) % 5]

elif col1 == col2:

plaintext += key\_matrix[(row1 - 1) % 5][col1] + key\_matrix[(row2 - 1) % 5][col2]

else:

plaintext += key\_matrix[row1][col2] + key\_matrix[row2][col1]

return plaintext

def main():

key = "KEYWORD"

plaintext = "HELLO"

key\_matrix = prepare\_key(key)

ciphertext = encrypt(plaintext, key\_matrix)

decrypted\_text = decrypt(ciphertext, key\_matrix)

print("Plaintext:", plaintext)

print("Ciphertext:", ciphertext)

print("Decrypted Text:", decrypted\_text)

if \_\_name\_\_ == "\_\_main\_\_":

main()

RSA:

from math import gcd

# defining a function to perform RSA approach

def RSA(p: int, q: int, message: int):

# calculating n

n = p \* q

# calculating totient, t

t = (p - 1) \* (q - 1)

# selecting public key, e

for i in range(2, t):

if gcd(i, t) == 1:

e = i

break

# selecting private key, d

j = 0

while True:

if (j \* e) % t == 1:

d = j

break

j += 1

# performing encryption

ct = (message \*\* e) % n

print(f"Encrypted message is {ct}")

# performing decryption

mes = (ct \*\* d) % n

print(f"Decrypted message is {mes}")

# Testcase - 1

RSA(p=53, q=59, message=89)

# Testcase - 2

RSA(p=3, q=7, message=12)

DIFFIE HELLMAN:

<!DOCTYPE html>

<html>

<head>

<title>Diffie-Hellman Key Exchange</title>

</head>

<body>

<h2>Diffie-Hellman Key Exchange Simulation</h2>

<div>

<label for="prime">Shared Prime (p):</label>

<input type="text" id="prime" value="23">

</div>

<div>

<label for="generator">Generator (g):</label>

<input type="text" id="generator" value="5">

</div>

<div>

<label for="alice-private">Alice's Private Key (a):</label>

<input type="text" id="alice-private" value="">

</div>

<div>

<label for="bob-private">Bob's Private Key (b):</label>

<input type="text" id="bob-private" value="">

</div>

<div>

<button onclick="generateKeys()">Generate Keys</button>

</div>

<h3>Shared Secret Key:</h3>

<div id="shared-key"></div>

<script>

function generateKeys() {

const prime = parseInt(document.getElementById("prime").value);

const generator = parseInt(document.getElementById("generator").value);

const alicePrivate = parseInt(document.getElementById("alice-private").value);

const bobPrivate = parseInt(document.getElementById("bob-private").value);

// Calculate public keys

const alicePublic = (generator \*\* alicePrivate) % prime;

const bobPublic = (generator \*\* bobPrivate) % prime;

// Calculate shared secret

const sharedSecretAlice = (bobPublic \*\* alicePrivate) % prime;

const sharedSecretBob = (alicePublic \*\* bobPrivate) % prime;

// Display the shared secret

document.getElementById("shared-key").innerHTML = `Alice's Shared Secret: ${sharedSecretAlice}<br>Bob's Shared Secret: ${sharedSecretBob}`;

}

</script>

</body>

</html>

**MD5 Algorithm:**

# importing the required libraries

import hashlib

# making a message

inputstring = "This is a message sent by a computer user."

# encoding the message using the library function

output = hashlib.md5(inputstring.encode())

# printing the hash function

print("Hash of the input string:")

print(output.hexdigest())

SHA-1: Replace md5 in output= line with sha1

**RC4 LOGIC:**

def initialize\_s\_box(key):

key\_length = len(key)

s\_box = list(range(256))

j = 0

for i in range(256):

j = (j + s\_box[i] + key[i % key\_length]) % 256

s\_box[i], s\_box[j] = s\_box[j], s\_box[i]

return s\_box

def rc4\_encrypt(plaintext, key):

s\_box = initialize\_s\_box(key)

i = 0

j = 0

ciphertext = []

for char in plaintext:

i = (i + 1) % 256

j = (j + s\_box[i]) % 256

s\_box[i], s\_box[j] = s\_box[j], s\_box[i]

k = s\_box[(s\_box[i] + s\_box[j]) % 256]

ciphertext.append(bytes([char ^ k]))

return b''.join(ciphertext)

def rc4\_decrypt(ciphertext, key):

return rc4\_encrypt(ciphertext, key) # RC4 decryption is the same as encryption

# Example usage

if \_\_name\_\_ == "\_\_main\_\_":

key = b"SecretKey"

plaintext = "Hello, RC4!"

ciphertext = rc4\_encrypt(plaintext.encode('utf-8'), key)

decrypted\_text = rc4\_decrypt(ciphertext, key)

print(f"Original Text: {plaintext}")

print(f"Ciphertext: {ciphertext}")

print(f"Decrypted Text: {decrypted\_text.decode('utf-8')}")  
  
  
  
  
  
DES:

# Initial permutation (IP) table

initial\_permutation = [

58, 50, 42, 34, 26, 18, 10, 2,

60, 52, 44, 36, 28, 20, 12, 4,

62, 54, 46, 38, 30, 22, 14, 6,

64, 56, 48, 40, 32, 24, 16, 8,

57, 49, 41, 33, 25, 17, 9, 1,

59, 51, 43, 35, 27, 19, 11, 3,

61, 53, 45, 37, 29, 21, 13, 5,

63, 55, 47, 39, 31, 23, 15, 7

]

# Initial permutation of data

def initial\_permute(data):

# Ensure the length is a multiple of 64 by zero-padding

data += '0' \* (64 - len(data) % 64)

return [data[i - 1] for i in initial\_permutation]

if \_\_name\_\_ == "\_\_main\_\_":

plaintext = input("Enter the plaintext: ")

key = input("Enter the key: ")

# Convert plaintext and key to binary

plaintext\_binary = ''.join(format(ord(char), '08b') for char in plaintext)

key\_binary = ''.join(format(ord(char), '08b') for char in key)

# Initial permutation of plaintext

initial\_permuted\_text = initial\_permute(plaintext\_binary)

# Perform DES encryption rounds (not shown here for brevity)

# Display the formatted output

print("\nPlainText:", plaintext)

print("Key:", key)

print("CipherText:", ''.join(initial\_permuted\_text))

AES:

import javax.crypto.Cipher;

import javax.crypto.KeyGenerator;

import javax.crypto.SecretKey;

import javax.crypto.spec.SecretKeySpec;

public class AES\_Builtin {

public static void main(String[] args)

throws Exception

{

String message = "This is my input string";

System.out.println("Input: " + message);

//Get the keyGenerator

KeyGenerator kgen = KeyGenerator.getInstance("AES");

kgen.init(128);

//Generate secret key specs

SecretKey skey = kgen.generateKey();

byte[] raw = skey.getEncoded();

SecretKeySpec skeySpec = new SecretKeySpec(raw, "AES");

//Instantiate the cipher

Cipher cipher = Cipher.getInstance("AES");

cipher.init(Cipher.ENCRYPT\_MODE, skeySpec);

// Encryption

byte[] encrypted = cipher.doFinal(message.getBytes());

String cipherString = new String(encrypted);

System.out.println("Encrypted String: " + cipherString);

// Decryption

cipher.init(Cipher.DECRYPT\_MODE, skeySpec);

byte[] original = cipher.doFinal(encrypted);

String originalString = new String(original);

System.out.println("Decrypted: " + originalString);

}

}

BLOWFISH:

p = [0] \* 18

s = [[0] \* 256 for \_ in range(4)]

key = [0x01234567, 0x89abcdef, 0xfedcba98, 0x76543210, 0x01234567, 0x89abcdef, 0xfedcba98, 0x76543210,

0x01234567, 0x89abcdef, 0xfedcba98, 0x76543210, 0x01234567, 0x89abcdef, 0xfedcba98, 0x76543210]

p\_new = p.copy()

def swap(a, b):

temp = a

a = b

b = temp

return a, b

def driver():

for i in range(0, 18):

p[i] = p[i] ^ key[i % 14]

k = 0

data = 0

for i in range(0, 9):

temp = encryption(data)

p[k] = temp >> 32

k += 1

p[k] = temp & 0xffffffff

k += 1

data = temp

encrypt\_data = int(input("Enter data to encrypt: "))

encrypted\_data = encryption(encrypt\_data)

print("Encrypted data:", encrypted\_data)

decrypted\_data = decryption(encrypted\_data)

print("Decrypted data:", decrypted\_data)

def encryption(data):

L = data >> 32

R = data & 0xffffffff

for i in range(0, 16):

L = p[i] ^ L

L1 = func(L)

R = R ^ func(L1)

L, R = swap(L, R)

L, R = swap(L, R)

L = L ^ p[17]

R = R ^ p[16]

encrypted = (L << 32) ^ R

return encrypted

def func(L):

temp = s[0][L >> 24]

temp = (temp + s[1][L >> 16 & 0xff]) % 2 \*\* 32

temp = temp ^ s[2][L >> 8 & 0xff]

temp = (temp + s[3][L & 0xff]) % 2 \*\* 32

return temp

def decryption(data):

L = data >> 32

R = data & 0xffffffff

for i in range(17, 1, -1):

L = p[i] ^ L

L1 = func(L)

R = R ^ func(L1)

L, R = swap(L, R)

L, R = swap(L, R)

L = L ^ p[0]

R = R ^ p[1]

decrypted\_data1 = (L << 32) ^ R

return decrypted\_data1

driver()

TRANSPOSITION:

class Adv\_Columnar:

def main(self):

message = "Hello CSE5"

key = "megabuck"

iterations = 9

advC = Adv\_Columnar()

cipher = advC.encrypt(message, key, iterations)

decrypted = advC.decrypt(cipher, key, iterations)

print("Message:", message)

print("Key:", key)

print("Iterations:", iterations)

print("CipherText:", cipher)

print("Decrypted:", decrypted)

def encrypt(self, message, key, iterations):

cipher = message

for i in range(iterations):

cipher = Columnar.encrypt(cipher, key)

return cipher

def decrypt(self, cipher, key, iterations):

message = cipher

for i in range(iterations):

message = Columnar.decrypt(message, key)

message = message.replace('-', ' ')

return message

class Columnar:

@staticmethod

def encrypt(message, key):

key\_order = sorted(range(len(key)), key=lambda k: key[k])

encrypted\_message = [''] \* len(key)

for i, k in enumerate(key\_order):

for j in range(i, len(message), len(key)):

encrypted\_message[k] += message[j]

return ''.join(encrypted\_message)

@staticmethod

def decrypt(message, key):

key\_order = sorted(range(len(key)), key=lambda k: key[k])

col\_size = len(message) // len(key)

decrypted\_message = [''] \* col\_size

for i, k in enumerate(key\_order):

for j in range(i \* col\_size, (i + 1) \* col\_size):

decrypted\_message[j % col\_size] += message[j]

return ''.join(decrypted\_message)

if \_\_name\_\_ == "\_\_main\_\_":

adv\_columnar = Adv\_Columnar()

adv\_columnar.main()