Q Shift Cipher

Ans
```
def shift_cipher(text, shift, mode='encrypt'):
    result = ''

    if mode == 'decrypt':
        shift = -shift

    for char in text:
        if char.isupper():
            result += chr((ord(char) + shift - 65)
                          % 26 + 65)
        elif char.islower():
            result += chr((ord(char) + shift - 97)
                          % 26 + 97)
        else:
            result += char

    return result

plain_text = input("Enter the text:").
shift_value = 3

cipher_text = shift_cipher(plain_text, shift_value,
                           mode='encrypt')
```

```
print ("Encrypted:", ciphertext)

decrypted_text = shift_cipher (ciphertext, shift value,
                               mode = 'decrypt')
print ("Decrypted:", decrypted_text)
```

* Mono-alphabetic Substitution Cipher :-

```
def encrypt (plaintext, substitution_key):
    alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
    ciphertext = ""

    for char in plaintext.upper():
        if char.isalpha():
            index = alphabet.index(char)
            ciphertext += substitution_key[index]

        else:
            ciphertext += char

    return ciphertext

def decrypt (ciphertext, substitution_key):
    alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
    plaintext = ""
```

```
for char in ciphertext.upper():
    if char.isalpha():
        index = substitution_key.index(char)
        plaintext += alphabet[index]

    else:
        plaintext += char

return plaintext

substitution_key = "QWERTYUIOPASDFGHJKLZXCVBNM"

plaintext = input("Enter the text:")

ciphertext = encrypt(plaintext, substitution_key)
print("Encrypted:", ciphertext)

decrypted_text = decrypt(ciphertext, substitution_key)
print("Decrypted:", decrypted_text)
```

Date : Page No

Name : Class : Group Roll No

WAP to implement one-time pad cipher

```
import os

def generate_random_key(length):
    return os.urandom(length)

def one_time_pad_encrypt(plaintext):
    plaintext_bytes = plaintext.encode()
    key = generate_random_key(len(plaintext_bytes))
    ciphertext = bytes([p ^ k for p, k in zip
                        (plaintext_bytes, key)])
    return ciphertext, key

def one_time_pad_decrypt(ciphertext, key):
    plaintext_bytes = bytes([c ^ k for c, k in zip
                             (ciphertext, key)])
    return plaintext_bytes.decode()

if __name__ == "__main__":
    plaintext = "This is a top-secret message!"
    ciphertext, key = one_time_pad_encrypt(plaintext)
    print(f"Ciphertext (hex): {ciphertext.hex()}")
    print(f"Key (hex): {key.hex()}")
    decrypted_text = one_time_pad_decrypt(ciphertext, key)
    print(f"Decrypted text: {decrypted_text}")
```

Garg Brothers

Date : Page No

Name : Class : Group Roll No.

WAP to implement message authentication codes.

```
import hmac
import hashlib

def generate_mac(message, key):
    """
    Generate a message authentication code using HMAC-SHA256
    Parameters:
        message (str): The message to authenticate
        key (str): The secret key
    Returns:
        str: The hexadecimal representation of MAC.
    """
    key_bytes = key.encode()
    message_bytes = message.encode()
    mac = hmac.new(key_bytes, message_bytes, hashlib.sha256)

    return mac.hexdigest()

if __name__ == "__main__":
    secret_key = "supersecretkey"
    message = "This is a secure message."

    mac = generate_mac(message, secret_key)
    print(f"Generated MAC: {mac}")
```

Date : ..... Page No .....

Name : ..... Class : ..... Group ..... Roll No. .....

WAP to implement AES algorithm logic.

```
from Crypto.Cipher import AES
from Crypto.util.Padding import pad, unpad
from Crypto.Random import get_random_bytes

key = get_random_bytes(16)
iv = get_random_bytes(16)

plaintext = "This is a secret message."

cipher = AES.new(key, AES.MODE_CBC, iv)
ciphertext = cipher.encrypt(pad(plaintext.encode('utf-8'),
                                AES.block_size))
decipher = AES.new(key, AES.MODE_CBC, iv)
decrypted_text = unpad(decipher.decrypt(ciphertext),
                       AES.block_size).decode('utf-8')
print("Original:", plaintext)
print("Encrypted:", ciphertext.hex())
print("Decrypted:", decrypted_text)
```

Output

Original: This is a secret message.
Encrypted: 8fa7eb9245e5fa17d47ab401b249ddf4.....
Decrypted: This is a ~~decrypted~~ Secret message.

Date : ..... Page No .....

Name : ..... Class : ..... Group ..... Roll No. .....

For DES:

```
from Crypto.Cipher import DES
from Crypto.Util.Padding import pad, unpad
from Crypto.Random import get_random_bytes

key = get_random_bytes(8)
iv = get_random_bytes(8)

plaintext = "This is a secret message."
cipher = DES.new(key, DES.MODE_CBC, iv)
ciphertext = cipher.encrypt(pad(plaintext.encode('utf-8'),
                                DES.block_size))
decipher = DES.new(key, DES.MODE_CBC, iv)
decrypted_text = unpad(decipher.decrypt(ciphertext),
                       DES.block_size).decode('utf-8')
print("Original:", plaintext)
print("Encrypted:", ciphertext.hex())
print("Decrypted:", decrypted_text)
```

Output:

```
Original: This is a secret message.
Encrypted: 35f89a56a01c7c73cc04a6bfe3ec28333
Decrypted: This is a secret message.
```

* Diffie-Hellman Key Exchange

Ans

```
import random

def mod_exp(base, exp, modulus):
    return pow(base, exp, modulus)

def diffie_hellman():
    p = 23
    g = 5
# Party A (Alice) selects a private key
    a = random.randint(1, p-1)
A = mod_exp(g, a, p)

# Party B (bob) selects a private key
    b = random.randint(1, p-1)
B = mod_exp(g, b, p)

shared_secret_A = mod_exp(B, a, p)
shared_secret_B = mod_exp(A, b, p)

print("Public parameters : p =", p, "g =", g)
print("Alice's private key a:", a)
print("Bob's private key b:", b)
print("Alice's public value A:", A)
print("Bob's public value B:", B)
```

```
print("Shared secret computed by Alice:", shared_secret_A)
print("Shared secret computed by Bob:", shared_secret_B)

if shared_secret_A == shared_secret_B:
    print("The shared secret keys match! Key established
                                    shared_secret_A)
else:
    print("Something went wrong, the shared keys do not
                                    match.")
diffie_hellman()
```

Output

```
Public parameters : p = 23  g = 5
Alice's private key a : 10
Bob's private key b : 3
Alice's public value A : 9
Bob's public value B : 10

Shared secret computed by Alice : 18
Shared secret computed by Bob : 18

The shared secret keys match! Key established : 18
```

* WAP to implement digital signature.

→
```
from Crypto.PublicKey import RSA
from Crypto.Signature import pkcs1_15
from Crypto.Hash import SHA256
from Crypto.Random import get_random_bytes.

key = RSA.generate(2048)
private_key = key
public_key = key.publickey()

message = "Hello World"
hash_obj = SHA256.new(message.encode('utf-8'))
signature = pkcs1_15.new(private_key).sign(hash_obj)
try:
    pkcs1_15.new(public_key).verify(hash_obj, signature)
    print("Signature is valid.")
except(ValueError, TypeError):
    print("Signature is invalid.")

print("Original Message:", message)
print("Signature (hex):", signature.hex())
```

Output:

```
Signature is valid.
Original Message: Hello World
Signature (hex): a962c984fa7c14c9c9073d40907b232f0
8e1b5da91efcb1d161d29f46c33b9
```