**Artificial Intelligence**

**Practical1**

graph = {

'A': ['B', 'E', 'C'],

'B': ['A', 'D', 'E'],

'D': ['B', 'E'],

'E': ['A', 'D', 'B'],

'C': ['A', 'F', 'G'],

'F': ['C'],

'G': ['C']

}

visited = set()

stack = []

def dfs(graph, start, goal):

stack.append(start)

visited.add(start)

while stack:

node = stack.pop()

print("Node: ", node)

if node == goal:

print("Goal node found!")

return

for neighbor in graph[node]:

if neighbor not in visited:

visited.add(neighbor)

stack.append(neighbor)

dfs(graph, 'A', 'D')```

**Practical1**

graph = {

'A': ['B', 'E', 'C'],

'B': ['A', 'D', 'E'],

'D': ['B', 'E'],

'E': ['A', 'D', 'B'],

'C': ['A', 'F', 'G'],

'F': ['C'],

'G': ['C']

}

visited = []

queue = []

def bfs(visited, graph, start\_node, goal\_node):

visited.append(start\_node)

queue.append(start\_node)

while queue:

current\_node = queue.pop(0)

print(current\_node)

if current\_node == goal\_node:

print("Node is Found !!! ")

break

else:

for neighbor in graph[current\_node]:

if neighbor not in visited:

visited.append(neighbor)

queue.append(neighbor)

print("The BFS Traversal is : ")

bfs(visited, graph, 'A', 'D')

**Practical4**

**Practical5**

import random

# Function to greet the user

def greet\_user():

print("Hello! I’m Chatty Bot.")

print("I can guess your age, count numbers, and ask multiple-choice questions.")

# Function to ask the user's name and introduce

def ask\_name\_and\_introduce():

name = input("What’s your name? ")

print(f"Nice to meet you, {name}!")

# Function to guess the user's age

def guess\_age():

print("Let me guess your age.")

remainder = int(input("Enter the remainder when you divide your age by 3: "))

age = (remainder + 2) \* 3

print(f"Your age is {age} years old!")

# Function to count numbers from 1 to 10

def count\_numbers():

print("Now I will count to 10:")

for number in range(1, 11):

print(number, end=' ')

print()

# Function to ask multiple-choice questions and calculate the score

def ask\_questions():

print("Let’s test your knowledge!")

questions = {

"What is the capital of France?": "Paris",

"Which planet is known as the Red Planet?": "Mars",

"Who wrote 'Romeo and Juliet'?": "William Shakespeare"

}

correct\_answers = 0

for question, correct\_answer in questions.items():

print(question)

user\_answer = input("Enter your answer: ").strip().capitalize()

if user\_answer == correct\_answer:

print("Correct!")

correct\_answers += 1

else:

print("Incorrect!")

print(f"You got {correct\_answers} out of {len(questions)} questions correct.")

# Main function to orchestrate the conversation

def main():

greet\_user()

ask\_name\_and\_introduce()

guess\_age()

count\_numbers()

ask\_questions()

# Check if the script is executed as the main program

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

main()

**Information Security**

**Practical1**

string = "Hello World"

# Perform AND operation

and\_result = ""

for char in string:

and\_result += chr(ord(char) & 127)

# Perform XOR operation

xor\_result = ""

for char in string:

xor\_result += chr(ord(char) ^ 127)

print("Original string:", string)

print("AND result:", and\_result)

print("XOR result:", xor\_result)

**Practical2**

def encrypt(message, key):

# Create an empty string to store the encrypted message

encrypted\_message = ''

# Loop through each column in the transposition grid

for col in range(key):

# Loop through each character that falls into this column

pointer = col

while pointer < len(message):

encrypted\_message += message[pointer]

pointer += key

# Return the encrypted message

return encrypted\_message

def decrypt(ciphertext, key):

# Calculate the number of columns in the transposition grid

num\_cols = (len(ciphertext) + key - 1) // key

# Calculate the number of rows in the transposition grid

num\_rows = key

# Calculate the number of empty cells in the last column of the transposition grid

num\_empty\_cells = num\_cols \* num\_rows - len(ciphertext)

# Initialize an empty grid to store the decrypted message

grid = [''] \* num\_cols

col = 0

row = 0

# Loop through each character in the ciphertext

for char in ciphertext:

# Add the character to the grid

grid[col] += char

# Move to the next column or row

col += 1

if (col == num\_cols) or (col == num\_cols - 1 and row >= num\_rows - num\_empty\_cells):

col = 0

row += 1

# Concatenate the rows of the grid to get the decrypted message

decrypted\_message = ''.join(grid)

# Return the decrypted message

return decrypted\_message

# Test the functions

message = "Hello World"

key = 3

encrypted\_message = encrypt(message, key)

print("Encrypted:", encrypted\_message)

decrypted\_message = decrypt(encrypted\_message, key)

print("Decrypted:", decrypted\_message)

**Practical3**

from Crypto.Cipher import DES

from Crypto.Random import get\_random\_bytes

from Crypto.Util.Padding import pad, unpad

def encrypt(message, key):

cipher = DES.new(key, DES.MODE\_ECB)

padded\_message = pad(message.encode(), DES.block\_size)

ciphertext = cipher.encrypt(padded\_message)

return ciphertext

def decrypt(ciphertext, key):

cipher = DES.new(key, DES.MODE\_ECB)

decrypted\_message = cipher.decrypt(ciphertext)

unpadded\_message = unpad(decrypted\_message, DES.block\_size)

return unpadded\_message.decode()

# Example usage

message = "Hello World"

key = get\_random\_bytes(8) # 8 bytes key for DES

print("Original message:", message)

encrypted\_message = encrypt(message, key)

print("Encrypted message:", encrypted\_message)

decrypted\_message = decrypt(encrypted\_message, key)

print("Decrypted message:", decrypted\_message)

**Practical4**

from Crypto.Cipher import AES

from Crypto.Random import get\_random\_bytes

from Crypto.Util.Padding import pad, unpad

def encrypt(message, key):

cipher = AES.new(key, AES.MODE\_CBC)

padded\_message = pad(message.encode(), AES.block\_size)

ciphertext = cipher.encrypt(padded\_message)

return ciphertext

def decrypt(ciphertext, key):

cipher = AES.new(key, AES.MODE\_CBC)

decrypted\_message = cipher.decrypt(ciphertext)

unpadded\_message = unpad(decrypted\_message, AES.block\_size)

return unpadded\_message.decode()

# Example usage

message = "Hello World"

key = get\_random\_bytes(16) # 16 bytes key for AES

print("Original message:", message)

encrypted\_message = encrypt(message, key)

print("Encrypted message:", encrypted\_message)

decrypted\_message = decrypt(encrypted\_message, key)

print("Decrypted message:", decrypted\_message)

**Practical5**

from Crypto.Util.number import getPrime, inverse # Import necessary functions

import random # Import random module for secure random number generation

class RSA:

def \_\_init\_\_(self, bit\_length):

self.generate\_keys(bit\_length)

def generate\_keys(self, bit\_length):

p = getPrime(bit\_length // 2 )

q = getPrime(bit\_length // 2)

self.modulus = p \* q

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

self.public\_key = 65537

self.private\_key = inverse(self.public\_key, phi)

def encrypt(self, message):

return pow(message, self.public\_key, self.modulus)

def decrypt(self, encrypted\_message):

return pow(encrypted\_message, self.private\_key, self.modulus)

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

rsa = RSA(1024

message = "Hello, World!"

plaintext = int.from\_bytes(message.encode(), "big

encrypted = rsa.encrypt(plaintext)

print("Encrypted message:", encrypted)

decrypted = rsa.decrypt(encrypted

decrypted\_message = decrypted.to\_bytes((decrypted.bit\_length() + 7) // 8, "big").decode print("Decrypted message:", decrypted\_message)