Practical 1-

import hashlib

# Function to hash a given string (data block) def hash\_data(data):

return hashlib.sha256(data.encode('utf- 8')).hexdigest()

# Function to build a Merkle tree def build\_merkle\_tree(data\_blocks):

# Step 1: Hash the data blocks to create the leaf nodes

leaves = [hash\_data(block) for block in data\_blocks]

# Step 2: Build the Merkle tree iteratively while len(leaves) > 1:

# If odd number of leaves, duplicate the last one if len(leaves) % 2 != 0:

leaves.append(leaves[-1])

# Create the next level of the tree

leaves = [hash\_data(leaves[i] + leaves[i + 1]) for i in range(0, len(leaves), 2)]

# The root of the Merkle tree return leaves[0]

# Example data blocks

data\_blocks = ["data1", "data2", "data3", "data4"]

# Build the Merkle tree

merkle\_root = build\_merkle\_tree(data\_blocks)

# Output the Merkle root print("Merkle Root:", merkle\_root)

Practical 2 –

import networkx as nx

import matplotlib.pyplot as plt

# Create a Directed Graph

dag = nx.DiGraph()

# Add nodes (tasks or steps in a workflow)

dag.add\_nodes\_from(["A", "B", "C", "D", "E"])

# Add directed edges (dependencies)

dag.add\_edges\_from([

("A", "B"), # A must be done before B

("A", "C"),

("B", "D"),

("C", "D"),

("D", "E") # D must be done before E

])

# Check if the graph is a DAG

is\_dag = nx.is\_directed\_acyclic\_graph(dag)

print(f"Is the graph a DAG? {is\_dag}")

# Topological Sort (to get a valid order of execution)

if is\_dag:

topo\_order = list(nx.topological\_sort(dag))

print(f"Topological Order: {topo\_order}")

# Visualize the DAG

pos = nx.spring\_layout(dag)

nx.draw(dag, pos, with\_labels=True, node\_color='lightblue', edge\_color='gray', node\_size=3000, font\_size=10)

plt.title("Directed Acyclic Graph (DAG)")

plt.show()

practical 3-

import hashlib

import time

# Define the Block class

class Block:

def \_\_init\_\_(self, index, previous\_hash, timestamp, data, hash\_value):

self.index = index

self.previous\_hash = previous\_hash

self.timestamp = timestamp

self.data = data

self.hash\_value = hash\_value

def calculate\_hash(self):

# Combine all block attributes to create a string, then hash it

block\_string = f"{self.index}{self.previous\_hash}{self.timestamp}{self.data}"

return hashlib.sha256(block\_string.encode('utf-8')).hexdigest()

# Define the Blockchain class

class Blockchain:

def \_\_init\_\_(self):

self.chain = []

self.create\_genesis\_block()

def create\_genesis\_block(self):

# Manually create the first block (Genesis Block)

genesis\_block = Block(0, "0", time.time(), "Genesis Block", "")

genesis\_block.hash\_value = genesis\_block.calculate\_hash()

self.chain.append(genesis\_block)

def add\_block(self, data):

# Get the last block in the chain

last\_block = self.chain[-1]

# Create the new block based on the last block's hash

new\_block = Block(len(self.chain), last\_block.hash\_value, time.time(), data, "")

new\_block.hash\_value = new\_block.calculate\_hash()

# Add the new block to the blockchain

self.chain.append(new\_block)

def is\_chain\_valid(self):

# Verify the integrity of the blockchain

for i in range(1, len(self.chain)):

current\_block = self.chain[i]

previous\_block = self.chain[i - 1]

# Check if the hash is correct

if current\_block.hash\_value != current\_block.calculate\_hash():

print(f"Invalid hash at block {current\_block.index}")

return False

# Check if the previous hash matches

if current\_block.previous\_hash != previous\_block.hash\_value:

print(f"Invalid previous hash at block {current\_block.index}")

return False

return True

def print\_chain(self):

for block in self.chain:

print(f"Block #{block.index}")

print(f"Timestamp: {block.timestamp}")

print(f"Data: {block.data}")

print(f"Hash: {block.hash\_value}")

print(f"Previous Hash: {block.previous\_hash}")

print("-" \* 30)

Example usage

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

blockchain = Blockchain()

# Add some blocks with data

blockchain.add\_block("Transaction 1: Alice sends 10 BTC to Bob")

blockchain.add\_block("Transaction 2: Bob sends 5 BTC to Charlie")

# Print the blockchain

blockchain.print\_chain()

# Validate the blockchain

if blockchain.is\_chain\_valid():

print("Blockchain is valid.")

else:

print("Blockchain is invalid.")

practical 4-

import hashlib

import json

import time

import socket

from threading import Thread

import random

# Block Class to represent each block

class Block:

def \_\_init\_\_(self, index, previous\_hash, timestamp, data, hash):

self.index = index

self.previous\_hash = previous\_hash

self.timestamp = timestamp

self.data = data

self.hash = hash

# Blockchain Class to represent the chain of blocks

class Blockchain:

def \_\_init\_\_(self):

self.chain = []

self.create\_genesis\_block()

def create\_genesis\_block(self):

# Create the first block (genesis block)

genesis\_block = Block(0, "0", int(time.time()), "Genesis Block", self.calculate\_hash(0, "0", "Genesis Block"))

self.chain.append(genesis\_block)

def add\_block(self, data):

last\_block = self.chain[-1]

new\_block = self.create\_new\_block(data, last\_block)

self.chain.append(new\_block)

self.print\_blockchain()

def create\_new\_block(self, data, last\_block):

index = last\_block.index + 1

timestamp = int(time.time())

previous\_hash = last\_block.hash

hash = self.calculate\_hash(index, previous\_hash, data)

return Block(index, previous\_hash, timestamp, data, hash)

def calculate\_hash(self, index, previous\_hash, data):

block\_content = f'{index}{previous\_hash}{data}'

return hashlib.sha256(block\_content.encode('utf-8')).hexdigest()

def get\_last\_block(self):

return self.chain[-1]

def print\_blockchain(self):

# Print the blockchain's blocks

print("\nBlockchain:")

for block in self.chain:

print(f"Block {block.index} [Data: {block.data}, Hash: {block.hash}]")

print("-" \* 50)

# Peer class that represents a node in the P2P network

class Peer:

def \_\_init\_\_(self, host, port, blockchain):

self.host = host

self.port = port

self.blockchain = blockchain

self.peers = []

def start\_server(self):

# Start the server to listen for incoming connections

server = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

server.bind((self.host, self.port))

server.listen(5)

print(f"Listening for incoming connections on {self.host}:{self.port}")

while True:

conn, addr = server.accept()

print(f"Connection established with {addr}")

Thread(target=self.handle\_client, args=(conn,)).start()

def handle\_client(self, conn):

data = conn.recv(1024)

if data:

peer\_blockchain = json.loads(data.decode('utf-8'))

print("Received blockchain data from peer...")

self.sync\_blockchain(peer\_blockchain)

def sync\_blockchain(self, peer\_blockchain):

if len(peer\_blockchain) > len(self.blockchain.chain):

print(f"Synchronizing blockchain with the peer. Peer chain length: {len(peer\_blockchain)}, Local chain length: {len(self.blockchain.chain)}")

self.blockchain.chain = peer\_blockchain

self.blockchain.print\_blockchain()

else:

print("Peer blockchain is not longer. No update needed.")

def broadcast\_block(self, new\_block):

for peer in self.peers:

peer.send\_block(new\_block)

def send\_block(self, new\_block):

client = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

client.connect((self.host, self.port))

client.send(json.dumps(new\_block).encode('utf-8'))

client.close()

def add\_peer(self, peer):

self.peers.append(peer)

# Main function to initialize blockchain and P2P network

def main():

blockchain = Blockchain()

# Initialize peers with unique ports

peer1 = Peer("127.0.0.1", 5000, blockchain)

peer2 = Peer("127.0.0.1", 5001, blockchain)

# Add some blocks (transactions)

blockchain.add\_block("Transaction 1")

blockchain.add\_block("Transaction 2")

# Start P2P server for each peer

Thread(target=peer1.start\_server).start()

Thread(target=peer2.start\_server).start()

# Simulate adding a block on peer1 and broadcasting it to peer2

new\_block = {

"index": 3,

"previous\_hash": blockchain.get\_last\_block().hash,

"timestamp": int(time.time()),

"data": "Transaction 3",

"hash": blockchain.calculate\_hash(3, blockchain.get\_last\_block().hash, "Transaction 3")

}

print("\n--- Broadcasting new block from peer1 ---")

peer1.broadcast\_block(new\_block)

# Add some delay before the next transaction

time.sleep(1)

# Simulate adding another block and broadcasting it

blockchain.add\_block("Transaction 4")

new\_block\_2 = {

"index": 4,

"previous\_hash": blockchain.get\_last\_block().hash,

"timestamp": int(time.time()),

"data": "Transaction 5",

"hash": blockchain.calculate\_hash(4, blockchain.get\_last\_block().hash, "Transaction 5")

}

print("\n--- Broadcasting new block from peer1 ---")

peer1.broadcast\_block(new\_block\_2)

# Add a block to peer2's chain

peer2.blockchain.add\_block("Transaction 6")

# Sync blockchain between peers

time.sleep(1)

print("\n--- Syncing blockchain from peer2 to peer1 ---")

peer2.broadcast\_block(peer2.blockchain.chain)

# Simulate peer2 trying to sync with peer1's longer chain

time.sleep(1)

print("\n--- Syncing blockchain from peer1 to peer2 ---")

peer1.broadcast\_block(peer1.blockchain.chain)

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

main()

**practical 5-**

import secrets

from bitcoinlib.wallets import Wallet

from eth\_account import Account

def create\_bitcoin\_wallet():

"""Creates a Bitcoin wallet and returns the details"""

wallet\_name = "MyBitcoinWallet"

# Create a Bitcoin wallet

wallet = Wallet.create(wallet\_name)

# Fetch key details

btc\_address = wallet.get\_key().address

private\_key = wallet.get\_key().wif

public\_key = wallet.get\_key().public\_hex

return {

"Currency": "Bitcoin",

"Address": btc\_address,

"Private Key": private\_key,

"Public Key": public\_key

}

def create\_ethereum\_wallet():

"""Creates an Ethereum wallet and returns the details"""

private\_key = "0x" + secrets.token\_hex(32) # Generate a 256-bit random private key

account = Account.from\_key(private\_key)

return {

"Currency": "Ethereum",

"Address": account.address,

"Private Key": private\_key

}

def main():

print("Choose the cryptocurrency wallet to generate:")

print("1. Bitcoin Wallet")

print("2. Ethereum Wallet")

choice = input("Enter your choice (1 or 2): ")

if choice == "1":

wallet\_info = create\_bitcoin\_wallet()

elif choice == "2":

wallet\_info = create\_ethereum\_wallet()

else:

print("Invalid choice! Please enter 1 or 2.")

return

print("\nGenerated Wallet Details:")

for key, value in wallet\_info.items():

print(f"{key}: {value}")

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

main()

**Step 3: Running the Script in an IDE**

1. **Open an IDE** (PyCharm, VS Code, or Jupyter Notebook).
2. **Create a new Python file** (e.g., crypto\_wallet.py).
3. **Paste the code** and run the script.
4. **Choose a wallet type (Bitcoin or Ethereum)** when prompted.
5. **Save your private key securely!** This is needed to recover funds.