Lab 08
Blockchain Technology

This lab will introduce students to **Blockchain** concept which is a time-stamped decentralized series of fixed records that contains data of any size is controlled by a large network of computers that are scattered around the globe and not owned by a single organization. Every block is secured and connected with each other using hashing technology which protects it from being tampered by an unauthorized person.

At the end of lab session students will be able to understand and implement Blockchain at basic level. Students will be given challenging real world problem to solve as end term projects.

https://www.techtarget.com/searchcio/definition/blockchain

Activity Outcomes:

This lab teaches you the following topics:

- How to implement blockchain concept.
- How to find a solution of real world problems using Blockchain Technology.

Sr.No	***Allocated Time***	***Level of Complexity***	***CLO Mapping***
1	***30***	***Medium***	***CLO-6***
2	***30***	***Medium***	***CLO-6***
3	***30***	***Medium***	***CLO-6***
4	***30***	***Medium***	***CLO-6***
5	***30***	***High***	***CLO-6***
6		***High***	***CLO-6***
7		***High***	***CLO-6***

What is blockchain?

Blockchain is a record-keeping technology designed to make it impossible to hack the system or forge the data stored on the blockchain, thereby making it secure and immutable. It's a type of distributed ledger technology (DLT), a digital record-keeping system for recording transactions and related data in multiple places at the same time.

Each computer in a blockchain network maintains a copy of the ledger where transactions are recorded to prevent a single point of failure. Also, all copies are updated and validated simultaneously.

Blockchain is also considered a type of database, but it differs substantially from conventional databases in how it stores and manages information. Instead of storing data in rows, columns, tables and files as traditional databases do, blockchain stores data in blocks that are digitally chained together. In addition, a blockchain is a decentralized database managed by computers belonging to a peer-to-peer network instead of a central computer like in traditional databases.

Bitcoin, launched in 2009 on the Bitcoin blockchain, was the first cryptocurrency and popular application to successfully use blockchain. As a result, blockchain has been most often associated with Bitcoin and alternatives such as Dogecoin and Bitcoin Cash, which both use public ledgers.

How blockchain and distributed ledger technology work

Blockchain uses a multistep process that includes these five steps:

An authorized participant inputs a transaction, which must be authenticated by the technology.

That action creates a block that represents that specific transaction or data.

The block is sent to every computer node in the network.

Authorized nodes validate transactions and add the block to the existing blockchain.

The update is distributed across the network, which finalizes the transaction.

These steps take place in near real time and involve a range of elements. Nodes in public blockchain networks are referred to as miners; they're typically paid for this task -- often in processes called proof of work or proof of stake -- usually in the form of cryptocurrency.

A blockchain ledger consists of two types of records, individual transactions and blocks. The first block has a header and data that pertain to transactions taking place within a set time period. The block's timestamp is used to help create an alphanumeric string called a hash. After the first block has been created, each subsequent block in the ledger uses the previous block's hash to calculate its own hash.

Before a new block can be added to the chain, its authenticity must be verified by a computational process called validation or consensus. At this point in the blockchain process, a majority of nodes in the network must agree the new block's hash has been calculated correctly. Consensus ensures that all copies of the blockchain distributed ledger share the same state.

Once a block has been added, it can be referenced in subsequent blocks, but it can't be changed. If someone attempts to swap out a block, the hashes for previous and subsequent blocks will also change and disrupt the ledger's shared state.

When consensus is no longer possible, other computers in the network are aware that a problem has occurred, and no new blocks will be added to the chain until the problem is solved. Typically, the block causing the error will be discarded and the consensus process will be repeated. This eliminates a single point of failure.

Key features of blockchain technology

Blockchain technology is built on a foundation of unique characteristics that differentiate it from traditional databases. The following are its most important and defining characteristics:

Decentralization. Blockchain decentralization is one of the fundamental aspects of the technology. Unlike

centralized databases where a central authority, such as a bank, controls and verifies transactions, blockchain operates on a distributed ledger. This means multiple transparent participants, known as nodes, maintain, verify and update the ledger. Each node is spread across a network and contains a copy of the whole blockchain.

Immutability and security. Cryptographic algorithms are used in blockchain to provide strong security, recording transactions and making tampering nearly impossible. Information is stored in blocks that are linked together using cryptographic hashes. If someone tries to tamper or modify a block, it would require the alteration of every subsequent block, making tampering computationally infeasible. This inherent blockchain security feature ensures immutability of information and makes blockchain an ideal platform for storing sensitive data and conducting secure transactions.

Transparency and traceability. The inherent transparency of blockchain technology ensures every network participant has access to identical information. For instance, every transaction becomes part of a public ledger, visible to all participants. This transparency ensures trust and network accountability, because any inconsistency can be promptly recognized and resolved. Additionally, the blockchain's capacity to track the origin and trajectory of assets facilitates audits and decreases the likelihood of fraudulent activities.

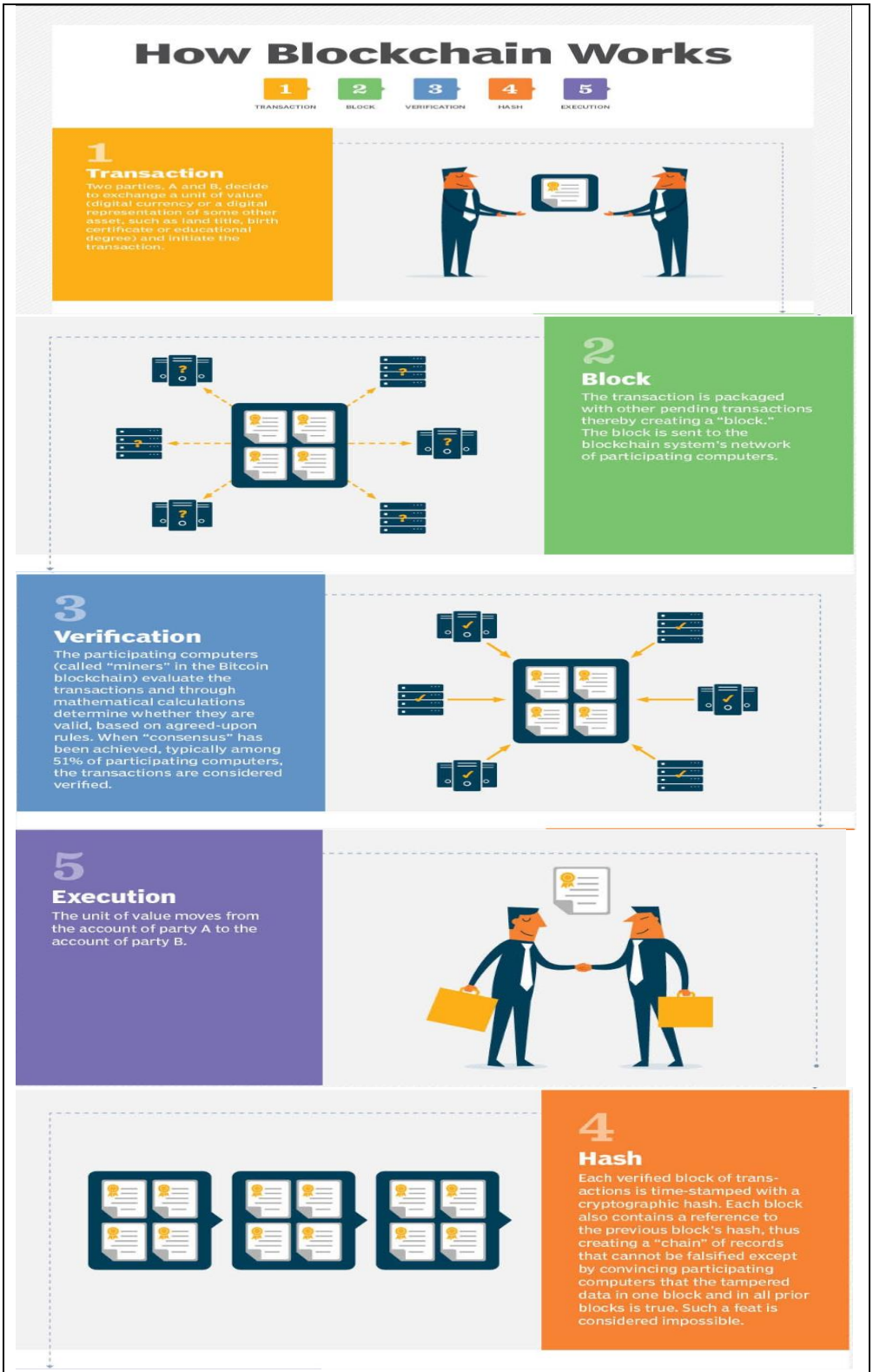
Smart contracts. These contracts are automated agreements encoded in software that execute the stipulations of a contract automatically. Smart contract codes are stored on the blockchain and carry out their functions once predetermined conditions are met. These contracts eliminate the need for intermediaries, streamline transactions, save money and speed up close times. They're used in a range of diverse sectors, including supply chain management, insurance and finance

Blockchain and smart contracts

Smart contracts are one of the most important features of blockchain technology. These are self-executing digital contracts written in code. They operate automatically according to predefined rules and conditions. Smart contracts are designed to facilitate, verify and enforce the negotiation or performance of an agreement without the need for intermediaries, such as lawyers, banks or other third parties. Once the specified conditions are met, the smart contract automatically executes the agreed-upon actions or transactions, ensuring that all parties involved adhere to the terms of the contract.

Smart contracts are typically deployed on blockchain platforms, which provide the necessary security and transparency for their execution. Ethereum is a popular blockchain platform for smart contracts. It's used for a range of applications such as financial transactions, supply chain management, real estate deals and digital identity verification.

Smart contracts have several benefits. By eliminating intermediaries, smart contract technology reduces the costs. It also cuts out complications and interference intermediaries can cause, speeding processes while also enhancing security.



Blockchain technology works in five basic steps, sometimes referred to as mining, in which transactions and data are executed and verified.

Here are several **blockchain-related programming tasks** that will introduce fundamental concepts like hashing, proof of work, and block validation, helping students understand how blockchains operate at a technical level.

Example

Create a class named Person, use the __init__() function to assign values for name and age:

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age

p1 = Person("John", 36)

print(p1.name)
print(p1.age)
```

Note: The __init__() function is called automatically every time the class is being used to create a new object.

Practice -Lab Activity 1: Block Structure Implementation

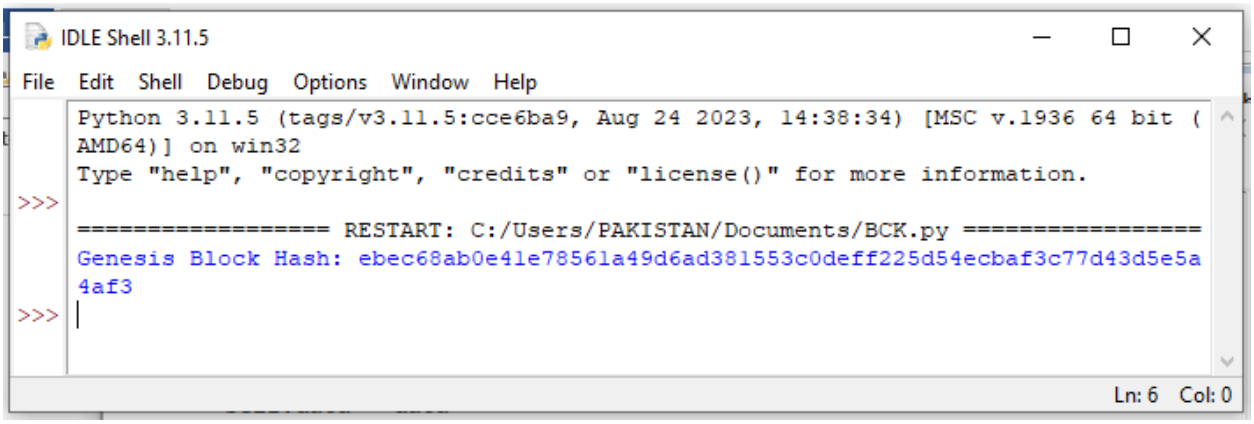
Objective: Create a basic block structure for a blockchain.

- **Step 1:** Define a `Block` class that includes properties like `index`, `timestamp`, `data`, `previous_hash`, and `hash`.
- **Step 2:** Compute the block's hash by combining the block's properties (excluding `hash`) and hashing them using SHA-256.

Example:

```
import hashlib  # Importing the hashlib library to use the SHA-256 hashing function
import time     # Importing the time module to get the current timestamp for each block

# Block class definition
class Block:
    def __init__(self, index, data, previous_hash):
        """
        Constructor to initialize a block in the blockchain.
        Parameters:
        - index: The position of the block in the blockchain (starting with 0 for the genesis block).
        - data: The transaction data or information to be stored in the block.
        - previous_hash: The hash of the previous block in the chain, ensuring continuity and security.
        """
        self.index = index                    # Index or position of the block in the chain
        self.timestamp = time.time()          # Timestamp of block creation
```

```
        self.data = data                          # Data stored in the block
(e.g., transactions)
        self.previous_hash = previous_hash    # Hash of the previous block,
linking to it
        self.hash = self.compute_hash()       # Hash of the current block,
generated using compute_hash method

    def compute_hash(self):
        """
        Method to calculate the SHA-256 hash of the block's contents.
        The hash is generated using the block's index, timestamp, data, and
the previous block's hash.
        """
        # Combine the block's properties into a single string
        block_string =
f"{self.index}{self.timestamp}{self.data}{self.previous_hash}"
        # Compute and return the SHA-256 hash of the block string
        return hashlib.sha256(block_string.encode()).hexdigest()

# Creating the genesis block (the first block in the blockchain)
genesis_block = Block(0, "Genesis Block", "0")
# Output the hash of the genesis block
print(f"Genesis Block Hash: {genesis_block.hash}")
```



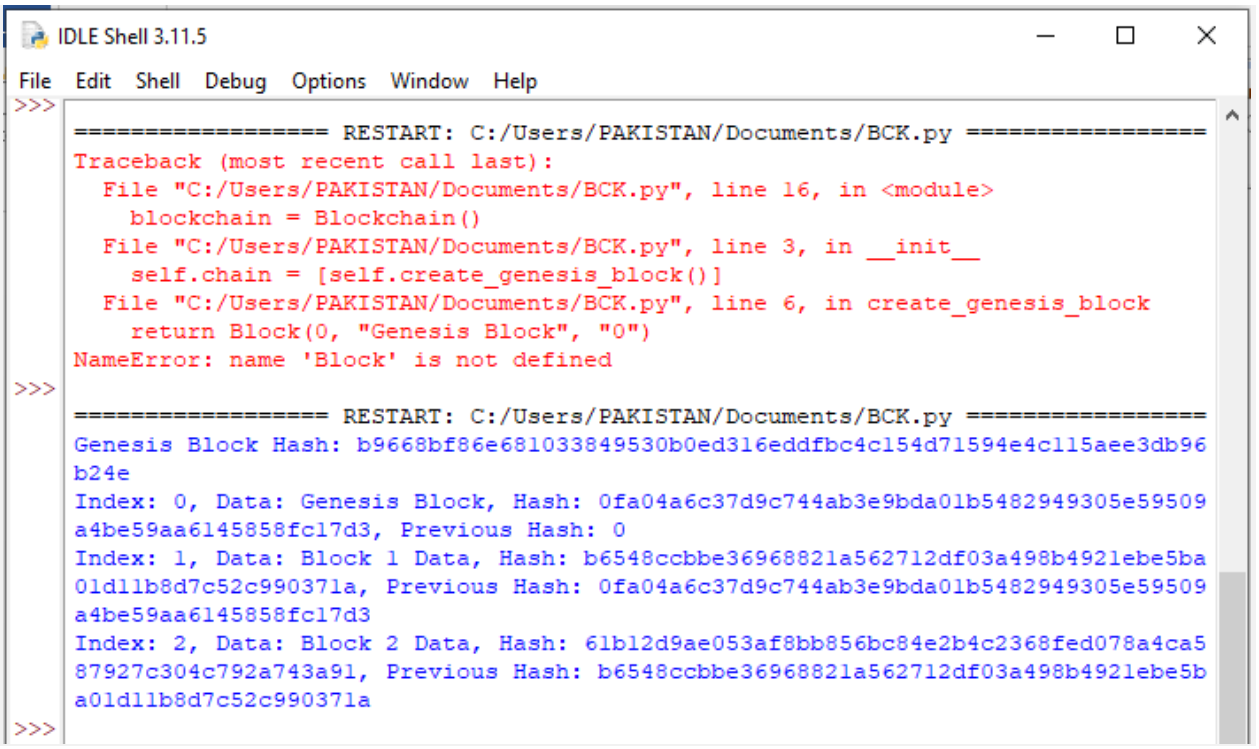
Practice -Lab Activity 2: Blockchain Construction

Objective: Create a blockchain by linking blocks together.

- **Step 1:** Create a `Blockchain` class that stores a list of blocks.
- **Step 2:** Implement a method to add new blocks to the chain.
- **Step 3:** Ensure that each block correctly references the previous block's hash.

```
# Blockchain class definition
class Blockchain:
    def __init__(self):
        """
        Constructor to initialize the blockchain.
        It starts with a list containing only the genesis block.
```

```
        """
        # The blockchain starts with the genesis block
        self.chain = [self.create_genesis_block()]

    def create_genesis_block(self):
        """
        Creates the first block in the blockchain (the genesis block).
        The genesis block has an index of 0, default data "Genesis Block",
        and a previous hash of "0" since it's the first block.
        """
        return Block(0, "Genesis Block", "0")

    def add_block(self, data):
        """
        Adds a new block to the blockchain.
        Parameters:
        - data: The data to be stored in the new block.
        The new block links to the previous block by including its hash.
        """
        # Get the last block in the current chain (previous block)
        last_block = self.chain[-1]
        # Create a new block with the next index, the provided data, and the
hash of the last block
        new_block = Block(len(self.chain), data, last_block.hash)
        # Append the new block to the blockchain
        self.chain.append(new_block)

    def print_blockchain(self):
        """
        Prints out each block in the blockchain.
        Displays the block's index, data, hash, and the hash of the previous
block.
        """
        # Iterate over all blocks in the chain and print their details
        for block in self.chain:
            print(f"Index: {block.index}, Data: {block.data}, Hash:
{block.hash}, Previous Hash: {block.previous_hash}")

# Example usage of the Blockchain class
blockchain = Blockchain()               # Create a new blockchain with a
genesis block
blockchain.add_block("Block 1 Data")     # Add a block with data "Block 1
Data"
blockchain.add_block("Block 2 Data")     # Add a block with data "Block 2
Data"
blockchain.print_blockchain()            # Print the details of the
blockchain
```



Practice -Lab Activity 3: Proof of Work

Purpose of PoW

The purpose of a consensus mechanism is to bring all the nodes in agreement, that is, trust one another, in an environment where the nodes don't trust each other.

- All the transactions in the new block are then validated and the new block is then added to the blockchain.
- The block will get added to the chain which has the longest block height (see blockchain forks to understand how multiple chains can exist at a point in time).
- Miners(special computers on the network) perform computation work in solving a complex mathematical problem to add the block to the network, hence named, Proof-of-Work.
- With time, the mathematical problem becomes more complex.

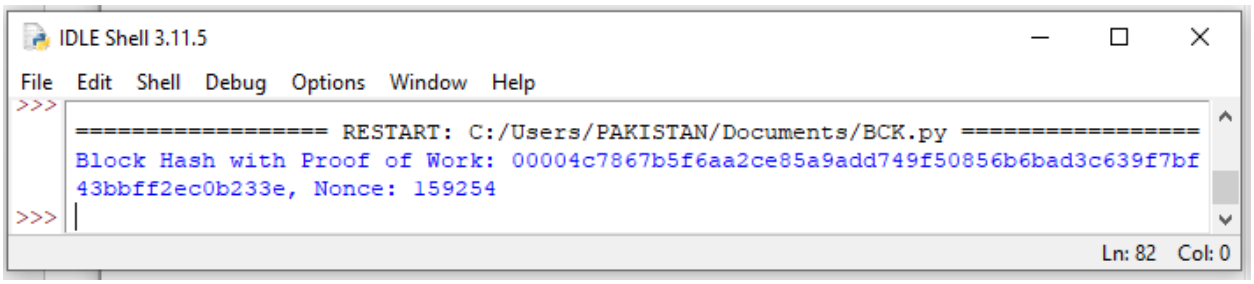
Objective: Implement a basic Proof of Work (PoW) algorithm.

- **Step 1:** Modify the `Block` class to include a `nonce` and a difficulty target.
- **Step 2:** Implement the PoW mechanism that requires the block's hash to start with a certain number of zeros.

```
import hashlib  # Importing hashlib to use the SHA-256 hashing function
import time     # Importing time to capture the current timestamp for each block

# Block class definition
class Block:
```

```
    def __init__(self, index, data, previous_hash, difficulty=2):
        """
        Constructor to initialize a block in the blockchain.
        Parameters:
        - index: The position of the block in the blockchain.
        - data: The data or information stored in the block (e.g.,
transactions).
        - previous_hash: The hash of the previous block in the chain,
ensuring blockchain continuity.
        - difficulty: The difficulty level for the Proof of Work (PoW),
default is 2.
        """
        self.index = index                    # The block's position in the
chain
        self.timestamp = time.time()          # Timestamp of block creation
        self.data = data                      # Data stored in the block
        self.previous_hash = previous_hash    # Hash of the previous block
        self.nonce = 0                        # Nonce (number used in PoW to
find valid hash)
        self.hash = self.compute_proof_of_work(difficulty)  # Find the valid
hash using Proof of Work

    def compute_hash(self):
        """
        Compute the SHA-256 hash of the block's contents.
        The hash includes the block's index, timestamp, data, previous hash,
and nonce.
        """
        # Combine the block's attributes into a single string
        block_string =
f"{self.index}{self.timestamp}{self.data}{self.previous_hash}{self.nonce}"
        # Compute and return the SHA-256 hash of the block string
        return hashlib.sha256(block_string.encode()).hexdigest()

    def compute_proof_of_work(self, difficulty):
        """
        Implements the Proof of Work (PoW) algorithm.
        PoW requires finding a hash that starts with a certain number of
leading zeros,
        defined by the difficulty parameter.
        """
        # The required hash prefix is a string of '0' repeated difficulty
times (e.g., "00" for difficulty=2)
        prefix = '0' * difficulty
        # Loop until we find a hash that starts with the required number of
leading zeros
        while True:
            self.hash = self.compute_hash()  # Compute the block's hash
            if self.hash.startswith(prefix): # Check if the hash satisfies
the difficulty requirement
                return self.hash             # Return the valid hash if it
meets the condition
            self.nonce += 1                  # Increment the nonce and try
again (to find a new hash)

# Example usage with Proof of Work
```

```
block = Block(1, "Some Data", "0", difficulty=4)  # Create a new block with
difficulty level 4
# Print the block's hash and the nonce value found through Proof of Work
print(f"Block Hash with Proof of Work: {block.hash}, Nonce: {block.nonce}")
```



Practice -Lab Activity 4: Blockchain Validation

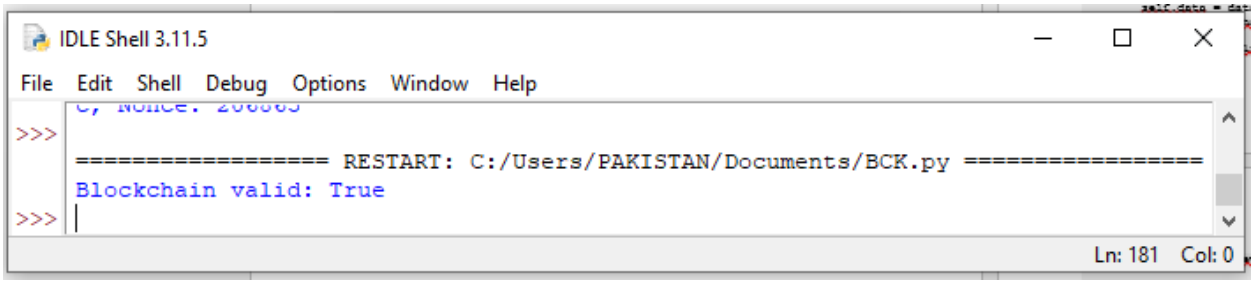
Objective: Implement a method to validate the integrity of the blockchain.

- **Step 1:** Ensure that each block's `previous_hash` matches the hash of the previous block.
- **Step 2:** Recompute the hash of each block and verify it matches the stored hash.

```
import hashlib  # Import hashlib to use the SHA-256 hashing algorithm for
block creation
import time     # Import time to capture the timestamp for each block

# Block class definition
class Block:
    def __init__(self, index, data, previous_hash, difficulty=2):
        """
        Constructor to initialize a block in the blockchain.
        Parameters:
        - index: The position of the block in the blockchain.
        - data: The information or transactions stored in the block.
        - previous_hash: The hash of the previous block in the blockchain,
ensuring continuity.
        - difficulty: The difficulty level for Proof of Work (PoW) that
defines how hard it is to mine the block.
        """
        self.index = index                      # Block's position in the
blockchain
        self.timestamp = time.time()            # Current timestamp for block
creation
        self.data = data                        # Data stored in the block
        self.previous_hash = previous_hash      # Hash of the previous block in
the chain
        self.nonce = 0                          # Nonce used for Proof of Work,
starts at 0
        self.hash = self.compute_proof_of_work(difficulty)  # Compute the
block's hash using PoW with the given difficulty

    def compute_hash(self):
        """
        Compute the SHA-256 hash of the block.
```

```
        The hash is generated from the block's index, timestamp, data,
previous hash, and nonce.
        """
        # Concatenate the block's attributes into a string
        block_string =
f"{self.index}{self.timestamp}{self.data}{self.previous_hash}{self.nonce}"
        # Return the SHA-256 hash of the concatenated string
        return hashlib.sha256(block_string.encode()).hexdigest()

    def compute_proof_of_work(self, difficulty):
        """
        Implements Proof of Work (PoW).
        PoW requires finding a hash that starts with a certain number of
leading zeros (determined by the difficulty).
        """
        prefix = '0' * difficulty  # String of '0's, the length of which is
the difficulty level
        while True:
            self.hash = self.compute_hash()  # Compute the block's hash
            if self.hash.startswith(prefix):  # Check if the hash satisfies
the difficulty condition (starts with the required number of leading zeros)
                return self.hash             # Return the valid hash if it
meets the condition
            self.nonce += 1                  # Increment the nonce and try
again to find a valid hash

# Blockchain class definition
class Blockchain:
    def __init__(self):
        """
        Constructor to initialize the blockchain.
        The blockchain starts with the genesis block (the first block).
        """
        self.chain = [self.create_genesis_block()]  # Initialize the chain
with the genesis block

    def create_genesis_block(self):
        """
        Creates the genesis block (the first block in the chain).
        The genesis block has an index of 0, default data "Genesis Block",
and a previous hash of "0".
        """
        return Block(0, "Genesis Block", "0")  # Return the first block with
index 0

    def add_block(self, data, difficulty=2):
        """
        Adds a new block to the blockchain.
        Parameters:
        - data: The data or transactions to be stored in the block.
        - difficulty: The difficulty level for Proof of Work (PoW) for the
new block.
        """
        last_block = self.chain[-1]  # Get the last block in the chain
        new_block = Block(len(self.chain), data, last_block.hash, difficulty)
# Create a new block with the current index, data, and previous block's hash
```

```
        self.chain.append(new_block)  # Append the newly created block to the
blockchain

    def is_chain_valid(self):
        """
        Validates the entire blockchain by ensuring the hashes and block
linking are correct.
        """
        # Loop through the blockchain from the second block (index 1) onward
        for i in range(1, len(self.chain)):
            current_block = self.chain[i]
            previous_block = self.chain[i-1]

            # Check if the current block's hash is correct (recompute and
compare)
            if current_block.hash != current_block.compute_hash():
                print(f"Invalid block at index {i}")  # Print error if the
current block's hash is invalid
                return False  # Return False if the blockchain is invalid

            # Check if the current block's previous hash matches the previous
block's hash
            if current_block.previous_hash != previous_block.hash:
                print(f"Invalid chain at index {i}")  # Print error if the
chain is broken
                return False  # Return False if the chain linking is invalid

        return True  # Return True if the entire blockchain is valid

# Example usage of Blockchain class
blockchain = Blockchain()                          # Create a new blockchain
with the genesis block
blockchain.add_block("Block 1 Data")               # Add a block with data
"Block 1 Data"
blockchain.add_block("Block 2 Data", difficulty=4)  # Add another block with
difficulty level 4 for PoW

# Validate the blockchain and print whether it's valid
print("Blockchain valid:", blockchain.is_chain_valid())  # Output whether the
blockchain is valid
```



Practice -Lab Activity 5: Simulate a 51% Attack

A 51% attack refers to a scenario in which a single entity or group of attackers gains control over more than 50% of the computational power (hashing power) in a blockchain network, particularly in proof-of-work (PoW) blockchains like Bitcoin. This gives the attacker the ability to manipulate the blockchain by performing actions that compromise its integrity. Some of the risks associated with a 51% attack include:

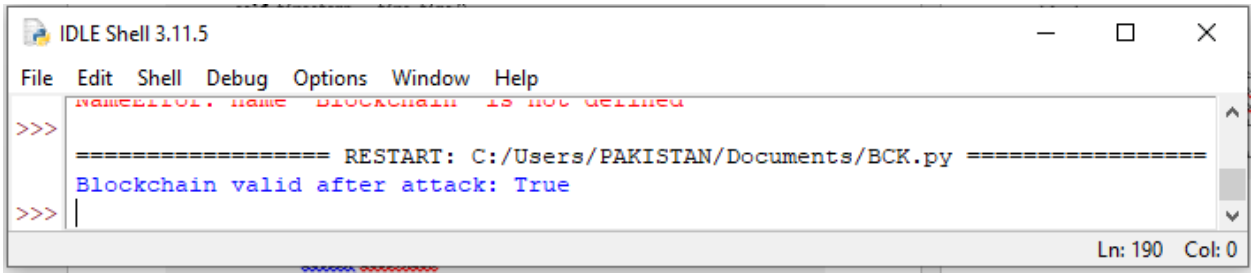
Objective: Simulate a scenario where the blockchain is compromised by a malicious actor who controls over 51% of the mining power.

- **Step 1:** Create two versions of a blockchain: one valid and one modified by an attacker.
- **Step 2:** The attacker rewrites history by modifying the data in an earlier block and recalculating all subsequent blocks' hashes.

```
import hashlib  # Import hashlib to use the SHA-256 hashing algorithm
import time     # Import time to generate timestamps for each block

# Block class definition
class Block:
    def __init__(self, index, data, previous_hash, difficulty=2):
        """
        Constructor to initialize a block in the blockchain.
        Parameters:
        - index: The position of the block in the blockchain.
        - data: The information or transactions stored in the block.
        - previous_hash: The hash of the previous block in the blockchain.
        - difficulty: The difficulty level for the Proof of Work (PoW)
algorithm.
        """
        self.index = index                    # The index of the block in the
chain
        self.timestamp = time.time()          # Timestamp when the block is
created
        self.data = data                      # Data stored in the block
        self.previous_hash = previous_hash    # Hash of the previous block
        self.nonce = 0                        # Nonce used for the PoW
algorithm, starts at 0
        self.hash = self.compute_proof_of_work(difficulty)  # Compute the
block's hash using PoW

    def compute_hash(self):
        """
        Compute the SHA-256 hash of the block.
        This method concatenates the block's attributes and returns the hash
of the concatenated string.
        """
        # Combine block properties into a string and hash it
        block_string =
f"{self.index}{self.timestamp}{self.data}{self.previous_hash}{self.nonce}"
        return hashlib.sha256(block_string.encode()).hexdigest()

    def compute_proof_of_work(self, difficulty):
        """
        Implements Proof of Work (PoW) by finding a hash that starts with a
certain number of leading zeros.
        The number of zeros is determined by the 'difficulty' parameter.
        """
```

```
        prefix = '0' * difficulty  # The target hash must start with a
specific number of leading zeros
        while True:
            self.hash = self.compute_hash()  # Compute the hash
            if self.hash.startswith(prefix):  # Check if the hash meets the
difficulty requirement
                return self.hash             # If valid, return the hash
            self.nonce += 1                  # If not, increment the nonce
and try again

# Blockchain class definition
class Blockchain:
    def __init__(self):
        """
        Constructor to initialize the blockchain with the genesis block (the
first block in the chain).
        """
        self.chain = [self.create_genesis_block()]  # Start the chain with
the genesis block

    def create_genesis_block(self):
        """
        Creates the genesis block, the first block in the blockchain.
        The genesis block has an index of 0, fixed data, and no previous
block (previous hash = "0").
        """
        return Block(0, "Genesis Block", "0")  # The first block with index 0
and default previous hash

    def add_block(self, data, difficulty=2):
        """
        Adds a new block to the blockchain.
        Parameters:
        - data: The data to be stored in the new block.
        - difficulty: The difficulty level for the Proof of Work (PoW)
algorithm.
        """
        last_block = self.chain[-1]  # Get the last block in the blockchain
        new_block = Block(len(self.chain), data, last_block.hash, difficulty)
# Create a new block
        self.chain.append(new_block)  # Append the new block to the chain

    def is_chain_valid(self):
        """
        Validates the integrity of the blockchain by checking each block's
hash and previous hash linkage.
        """
        # Loop through the chain starting from the second block (index 1)
        for i in range(1, len(self.chain)):
            current_block = self.chain[i]
            previous_block = self.chain[i-1]

            # Check if the current block's hash is correct (by recomputing
the hash)
            if current_block.hash != current_block.compute_hash():
                print(f"Invalid block at index {i}")  # If the hash is
incorrect, print an error message
```

```
                return False  # Return False as the chain is invalid

            # Check if the current block's previous hash matches the hash of
the previous block
            if current_block.previous_hash != previous_block.hash:
                print(f"Invalid chain at index {i}")  # If the chain linkage
is broken, print an error
                return False  # Return False as the chain is invalid

        return True  # If all checks pass, return True (the chain is valid)

# Simulating a 51% attack
# Example usage of the blockchain
blockchain = Blockchain()  # Create a new blockchain with the genesis block
blockchain.add_block("Block 1 Data")  # Add a valid block with data "Block 1
Data"
blockchain.add_block("Block 2 Data")  # Add a second block
blockchain.add_block("Block 3 Data")  # Add a third block

# Attack: An attacker modifies the data of Block 1 and tries to manipulate
the chain
# This simulates a 51% attack where a majority of miners (malicious actors)
try to modify a block
blockchain.chain[1].data = "Malicious Block 1 Data"  # Modify the data of
Block 1
blockchain.chain[1].hash = blockchain.chain[1].compute_hash()  # Recompute
the hash for Block 1

# The attacker then recalculates all subsequent block hashes to make the
chain appear valid
for i in range(2, len(blockchain.chain)):  # Loop through the remaining
blocks
    blockchain.chain[i].previous_hash = blockchain.chain[i-1].hash  # Set the
previous hash of the current block
    blockchain.chain[i].hash = blockchain.chain[i].compute_hash()   #
Recompute the current block's hash

# Validate the blockchain after the attack
# This demonstrates that if enough blocks are controlled (as in a 51%
attack), an invalid chain could be made valid
print("Blockchain valid after attack:", blockchain.is_chain_valid())  #
Output whether the blockchain is still valid
```



Home Task 6: Simplified Blockchain Mining Reward System

Objective: Implement a simple reward system where a miner gets rewarded for successfully mining a block.

- **Step 1:** Define a reward that gets added to the block data.
- **Step 2:** Create a `miner` function that simulates mining and adds the reward to the miner's wallet.

```
class Blockchain:
    def __init__(self):
        self.chain = [self.create_genesis_block()]
        self.mining_reward = 50
        self.pending_rewards = {}

    def create_genesis_block(self):
        return Block(0, "Genesis Block", "0")

    def mine_block(self, miner_address, difficulty=2):
        # Add reward to miner's wallet for mining the block
        new_block = Block(len(self.chain), f"Reward to {miner_address}:
{self.mining_reward} coins", self.chain[-1].hash, difficulty)
        self.chain.append(new_block)
        self.pending_rewards[miner_address] =
self.pending_rewards.get(miner_address, 0) + self.mining_reward

    def print_rewards(self):
        for miner, reward in self.pending_rewards.items():
            print(f"Miner: {miner}, Reward: {reward} coins")

# Example usage
blockchain = Blockchain()
blockchain.mine_block("Miner1")
blockchain.mine_block("Miner2", difficulty=4)

blockchain.print_rewards()
```

Home Task 7: Basic Blockchain Peer-to-Peer (P2P) Network Simulation

Objective: Simulate a simplified blockchain peer-to-peer network where multiple nodes exchange blocks.

- **Step 1:** Define multiple nodes that maintain their own copy of the blockchain.
- **Step 2:** Implement a method to synchronize the blockchain among the nodes.

```
class Node:
    def __init__(self, name):
        self.name = name
        self.blockchain = Blockchain()

    def sync_with_node(self, other_node):
        if len(self.blockchain.chain) < len(other_node.blockchain.chain):
            self.blockchain.chain = other_node.blockchain.chain
            print(f"{self.name} synchronized with {other_node.name}'s
blockchain.")

# Example usage
node1 = Node("Node1")
```

```
node2 = Node("Node2")

# Node1 mines a block
node1.blockchain.mine_block("Miner1")
# Node2 synchronizes with Node1
node2.sync_with_node(node1)
node2.blockchain.print_blockchain()
```

These tasks introduce various blockchain concepts like block creation, proof of work, blockchain integrity validation, mining rewards, and P2P synchronization. Students can further build on these tasks by adding more complex features like transaction handling, consensus algorithms, or smart contracts.

Select one topic and prepare an implementation level report about the feasibility of following projects.

Blockchain project ideas

Here are a few project ideas for beginners looking to learn more about blockchain technology:

1. **Cryptocurrency Wallet:** Create a simple cryptocurrency wallet application that allows users to send and receive digital assets.
2. **Blockchain Explorer:** Develop a web-based application that allows users to view and search the transactions on a specific blockchain.
3. **Smart Contract:** Implement a simple smart contract on the Ethereum blockchain that can be used to manage a digital token or asset.
4. **Voting System**: Create a blockchain-based voting system that allows for secure and transparent voting while maintaining voter anonymity.
5. **Supply Chain Management:** Develop a blockchain-based system for tracking the movement of goods and services through a supply chain, providing greater transparency and traceability.
6. **Decentralized marketplace:** Create a decentralized marketplace using blockchain technology where the goods and services can be directly bought by the customers without any intermediary.
7. **Identity Management:** Create a decentralized digital identity management system that allows users to control their personal information and share it securely with others.