**Practical No. 1: To Develop Naive Blockchain Construction**

**Aim:** To understand and simulate the working of a **naive blockchain** using an online blockchain demonstration tool.

**Apparatus / Platform Used:**

* **Platform:** https://andersbrownworth.com/blockchain/hash
* **Software Requirements:** Web browser with internet access
* **Programming Language:** Not required (visual simulation)

**Theory:** A **blockchain** is a chain of blocks, where each block contains data, its own hash, and the hash of the previous block. This structure ensures **immutability** - any change in one block invalidates all the following blocks.

**1. Components of a Block:**

1. **Block Number / Index:** Position of the block in the chain.
2. **Nonce:** A number that miners change to find a valid hash (Proof-of-Work).
3. **Data:** Transaction or record stored in the block.
4. **Hash:** Unique digital fingerprint of the block’s contents.
5. **Previous Hash:** Connects the block to the previous one, maintaining the chain.

**2. How Blockchain Ensures Security:**

1. Each block’s hash depends on its data and previous hash.
2. If any data changes, the hash changes instantly, breaking the chain.
3. To restore the chain, all subsequent hashes must be recalculated - practically infeasible, hence secure.

**Procedure:** Open the **Blockchain Demo** at https://andersbrownworth.com/blockchain/hash

1. Explore the **Hash** section first:
   * Type any text and observe the **SHA-256 hash** generated instantly.
   * Note that even a small change (adding a space) completely changes the hash - showing **hash sensitivity**.
2. Move to the **Block** section:
   * Observe fields like *Block Number*, *Nonce*, *Data*, *Hash*.
   * Try changing the *Data*; the *Hash* turns **red**, meaning the block is invalid until a valid hash is found again.
   * Click **“Mine”** - the tool changes the nonce until the hash begins with zeros (valid block found).
3. Move to the **Blockchain** section:
   * Multiple blocks are now linked using *Previous Hash*.
   * Try editing data in **Block 1** and see that **Block 2**, **Block 3**, etc., all turn red (invalid).
   * Press **Mine** on each block to repair the chain.

**Observation Table:**

| **Operation Performed** | **Action Taken** | **System Response / Observation** |
| --- | --- | --- |
| Typed text in Hash field | Changed “Hello” to “hello” | Hash changed completely |
| Changed block data | Hash turned red, block invalid | Indicates data tampering |
| Clicked “Mine” | Nonce changed repeatedly until hash started with zeros | Block became valid again |
| Changed data in first block of blockchain | All following blocks turned red | Shows linkage via Previous Hash |
| Mined all affected blocks | Chain became valid again | Restored integrity of blockchain |
| Edited data in one node of distributed blockchain | That node became invalid until consensus | Demonstrates distributed verification |

**Result:** The simulation successfully demonstrated how a naive blockchain operates:

1. Each block contains its data, nonce, hash, and previous hash.
2. Changing any data breaks the chain due to hash mismatch.
3. Mining revalidates a block by recalculating a valid hash.
4. The blockchain maintains data integrity and immutability through cryptographic linking.
5. Hence, the working of a basic (naive) blockchain has been **successfully simulated**.

**Output Screenshot:**

Insert screenshots of:

1. The **Hash** section (before and after text change)
2. The **Block** section showing mining
3. The **Blockchain** section showing invalid and valid chains

**{{{Do add your name and roll no in the Data Block}}}**

**Conclusion:** The practical provided a clear visual understanding of blockchain fundamentals. By observing how hashes link blocks and how mining restores validity, the concept of **block immutability, integrity, and proof-of-work** becomes clear without writing code.

**Practical No. 2: Program to Solve a Mining Puzzle using Blockchain**

**Aim:** To simulate and understand the **mining puzzle (Proof-of-Work)** process in blockchain by finding a valid hash through nonce computation using the online blockchain simulation tool.

**Apparatus / Platform Used**

* **Platform:** https://andersbrownworth.com/blockchain/block
* **Software Requirement:** Any web browser with internet access
* **Programming Language:** Not required (visual simulation)

**Theory:** In blockchain, **mining** is the process of solving a **cryptographic puzzle** known as **Proof-of-Work (PoW)**. Each block must have a hash that satisfies a predefined condition - typically the hash must start with a specific number of leading zeros.Because the hash depends on the **data** and a **nonce**, miners repeatedly change the nonce until the hash meets the required difficulty.

**1. Key Concepts:**

1. **Hash Function (SHA-256):** Converts block data into a fixed 256-bit string.
2. **Nonce:** A random number miners change to obtain a valid hash.
3. **Difficulty:** Number of leading zeros required in the hash.
4. **Proof-of-Work:** The computational proof that enough effort was spent to find a valid hash.

The process is intentionally **computationally expensive**, ensuring security and preventing tampering.

**Procedure:** Open the **Blockchain section** at https://andersbrownworth.com/blockchain/hash

1. Observe the block structure containing:
   * Block number
   * Nonce
   * Data field
   * Hash field
2. Type some text in the **Data** field and note the corresponding **Hash** value.
3. Change the data slightly - observe that the hash changes completely, and it is **not valid** (doesn’t start with zeros).
4. Click the **“Mine”** button.
   * The simulation automatically changes the nonce repeatedly.
   * After a few seconds, the hash starts with zeros, turning **green** (valid).
5. Record the **nonce value** and **hash** obtained after mining.
6. Try changing the data again - the block becomes **invalid** and must be mined again.
7. Optionally, increase the **difficulty** (in advanced settings) to require more leading zeros and observe that mining takes longer.

**Observation Table:**

| **Operation Performed** | **Difficulty** | **Action** | **Result / Observation** |
| --- | --- | --- | --- |
| Entered block data | 2 zeros | Initial hash shown | Hash invalid (no leading zeros) |
| Clicked “Mine” | 2 zeros | Nonce changed repeatedly | Valid hash found with 2 leading zeros |
| Changed data slightly | 2 zeros | Hash recomputed | Block invalid again |
| Clicked “Mine” again | 2 zeros | Nonce updated | New valid hash obtained |
| Increased difficulty | 4 zeros | Mining time observed | Mining took longer, required more nonce tries |

**Result:** The simulation successfully demonstrated the **Proof-of-Work mining puzzle**:

1. A valid block hash must start with a certain number of zeros.
2. Changing block data invalidates the hash, requiring re-mining.
3. As difficulty increases, mining takes more time and computation.

Thus, the concept of solving a mining puzzle using blockchain was **successfully simulated**.

**Output Screenshot:**

1. Initial invalid hash (red).
2. After clicking “Mine” - valid hash (green).
3. Increased difficulty showing slower mining.

**{{{Do add your name and roll no in the Data Block}}}**

**Conclusion:** The practical demonstrates the **core mechanism of blockchain mining** - solving the cryptographic puzzle through iterative nonce computation. It visually shows how Proof-of-Work ensures **data integrity, immutability, and security** within a blockchain network.

**Practical No. 3: Design, Build, and Deploy a Distributed Blockchain Application**

**Aim:** To simulate and understand the working of a **distributed blockchain network**, showing how multiple nodes maintain consensus and ensure data integrity across the system.

**Apparatus / Platform Used:**

* **Platform:** https://andersbrownworth.com/blockchain/distributed
* **Software Requirement:** Any web browser with internet access
* **Programming Language:** Not required (visual simulation)

**Theory:** A **distributed blockchain** is a decentralized network where multiple nodes maintain copies of the same ledger. Each node has an identical version of the blockchain, and updates are shared through **consensus mechanisms** to ensure all participants agree on the same valid state.

**1. Key Concepts:**

1. **Node:** An individual participant maintaining its own copy of the blockchain.
2. **Ledger:** A list of verified transactions (blocks) stored across nodes.
3. **Consensus:** The process by which all nodes agree on the valid version of the blockchain.
4. **Tampering:** Any data modification in one node breaks the chain’s hash link, invalidating that node’s copy.
5. **Synchronization:** During consensus, all nodes update to the longest valid chain, restoring consistency.

**2. Why Distribution Matters:**

* Eliminates a single point of failure.
* Increases transparency and trust among participants.
* Makes tampering nearly impossible without controlling the majority of nodes.

**Procedure:** Open the **Blockchain Demo** at https://andersbrownworth.com/blockchain/ distributed

1. Observe the setup:
   * Three nodes (A, B, C), each with identical blockchains.
   * Each block contains **index**, **data**, **nonce**, and **hash** fields.
2. Note that initially, all nodes are valid (green hashes).
3. Modify data in one block of **Node A**.
   * The modified block and subsequent blocks in Node A turn **red**, showing invalid hashes.
   * Nodes B and C remain valid.
4. Click on the **“Resolve”** or **“Resync”** option.
   * The invalid node (A) compares its chain with others.
   * The **longest valid chain** from Nodes B and C replaces Node A’s invalid chain.
5. Observe that all nodes now display valid (green) chains again.
6. Repeat the process by editing a block in Node B or C and watch how consensus reestablishes a valid distributed ledger.

**Observation Table:**

| **Operation Performed** | **Node(s) Affected** | **System Response** | **Observation** |
| --- | --- | --- | --- |
| All nodes initialized | A, B, C | All chains valid | All hashes green |
| Edited data in Block 2 (Node A) | Node A | Hash turns red | Node A invalid |
| Compared Node A with others | Node A, B, C | Nodes B and C unchanged | Only Node A invalid |
| Clicked “Resolve” | Node A | Node A chain updated | Node A synchronized with valid chain |
| Edited data in Node B | Node B | Node B invalid | After resolve, all chains matched again |

**Result:** The simulation successfully demonstrated the working of a **distributed blockchain**:

* Each node maintains its own copy of the ledger.
* Tampering at one node causes hash mismatch and invalidation.
* The consensus mechanism ensures synchronization by restoring the valid longest chain.

Thus, a distributed blockchain system was **successfully simulated and analyzed**.

**Output Screenshot:**

1. Initial valid distributed ledger (all green).
2. After tampering with one node (hash turns red).
3. After resolving consensus (all chains valid again).

**{{{Do add your name and roll no in the Data Block}}}**

**Conclusion:** Through this simulation, we learned that in a distributed blockchain:

1. Every node stores a copy of the blockchain.
2. Tampering one node invalidates its version.
3. The consensus mechanism restores consistency by ensuring that all nodes agree on the longest valid chain.

**Practical No. 4: Create Cryptocurrency and Explore Open Research Challenges**

**Aim:** To simulate and understand the creation and working of a **cryptocurrency** using the **Coinbase concept** in blockchain and to identify open **research challenges** in cryptocurrency systems.

**Apparatus / Platform Used:**

* **Platform:** https://andersbrownworth.com/blockchain/coinbase
* **Software Requirement:** Any web browser with internet access
* **Programming Language:** Not required (visual simulation)

**Theory:** A **cryptocurrency** is a digital form of money that operates on a **decentralized blockchain network**. Every cryptocurrency system relies on a shared ledger that records all transactions transparently and securely through cryptographic hashing.

**1. Key Concepts:**

1. **Coinbase Transaction:** The first transaction in every block that rewards the miner with newly generated coins. It is how new cryptocurrency enters circulation.
2. **Transaction:** A record that transfers coins from one address (wallet) to another.
3. **Wallet / Address:** A unique identifier that holds a balance of coins.
4. **Mining:** The process of validating transactions and adding new blocks to the chain.
5. **Ledger:** A complete, chronological list of all transactions in the network.

**2. Research Challenges in Cryptocurrencies:**

* **Energy Consumption:** Proof-of-Work systems use high computational power.
* **Scalability:** Networks process limited transactions per second.
* **Security Threats:** Wallet thefts, 51 % attacks, and smart-contract vulnerabilities.
* **Privacy vs Transparency:** Balancing traceability with user confidentiality.
* **Regulation and Governance:** Absence of consistent global policies.

**Procedure:** Open the **Blockchain Demo** at https://andersbrownworth.com/blockchain/coinbase

1. Observe the interface showing a **list of wallets, balances, and blockchain blocks**.
2. Note the **Coinbase transaction** in the first block - it adds new coins to the system as a mining reward.
3. Create a **new transaction**:
   * Select a *sender* wallet and *receiver* wallet.
   * Enter an amount to transfer.
4. Click **“Add Transaction”** to include it in the block.
5. Press **“Mine”** to validate and finalize the block:
   * Mining finds a valid hash for the block.
   * The new transaction becomes part of the permanent blockchain.
6. Observe how wallet balances update automatically after mining.
7. Add more transactions to simulate continuous coin movement.
8. Modify data (optional) and observe that hashes turn red until re-mined - showing immutability.

**Observation Table:**

| **Operation / Event** | **Action Taken** | **System Response / Observation** |
| --- | --- | --- |
| Initial blockchain | Opened Coinbase demo | Displayed existing wallets and balances |
| Viewed coinbase transaction | Checked first block | Showed mining reward created for miner |
| Added new transaction | Entered sender, receiver, amount | Transaction added to pending block |
| Clicked “Mine” | Block validated, hash updated | Balances changed according to transaction |
| Modified transaction data | Edited block details | Hash turned red → invalid block |
| Re-mined block | Clicked “Mine” again | Valid hash restored, chain consistent |

**Result:** The simulation successfully demonstrated:

1. How new coins are created through **Coinbase mining rewards**.
2. How transactions transfer coins between wallets.
3. How mining secures and validates each transaction.
4. How tampering changes the hash, proving **immutability** of blockchain data.

Thus, the **creation and functioning of a cryptocurrency** were effectively simulated using an online blockchain platform.

**Output Screenshots:**

1. Coinbase transaction showing new coin creation.
2. Added transaction and pending block before mining.
3. Valid block after mining with updated balances.

**{{{Do add your name and roll no in the Data Block}}}**

**Conclusion:** The Coinbase simulation clearly illustrated the **working of cryptocurrencies** - from coin generation to transaction validation and balance updates. It also highlighted the blockchain principles of **transparency, immutability, and decentralization**, along with the real-world challenges such as scalability and energy efficiency. Hence, the **creation and operational mechanism of a cryptocurrency** were successfully demonstrated through online simulation.

**Practical No. 5: Mathematical Modelling and Problem-Solving using Blockchain Data Structures**

**Aim:** To model and understand the internal **data structures and mathematical relationships** that form the foundation of blockchain technology and demonstrate how immutability and chain validation are achieved.

**Apparatus / Platform Used:**

* **Platform:** Any Python IDE / Online Compiler / Jupyter Notebook
* **Programming Language:** Python 3
* **Additional Tools:** SHA-256 hashing (via Python’s hashlib library)

**Theory:** A blockchain is a **mathematical and structural model** that links data blocks through cryptographic hashes. Each block contains data, its own hash, and the hash of the previous block - forming a **secure linked list**. Any modification in one block alters its hash, thereby invalidating all subsequent blocks. This ensures **immutability** and **data integrity**.

**1. Key Data Structures:**

1. **Linked List Structure:** Each block references the previous block, forming a sequential chain.
2. **Hash Function (SHA-256):** A one-way mathematical function
3. **Immutability Model:** If any changes, then block become invalid.

**Procedure:**

1. Open any **Python environment** (IDLE, Jupyter Notebook, or Online GDB).
2. Import the hashing library: import hashlib.
3. Define a simple **Block** class containing:
   * Index
   * Data
   * Previous hash
   * Own hash (computed by SHA-256)
4. Construct a **Blockchain** class that stores a list of blocks.
5. Create the **genesis block** (first block with previous hash = “0”).
6. Add subsequent blocks by providing data and automatically computing their hashes.
7. Display the complete chain (index, data, hash, previous hash).
8. Change the data of an earlier block and recompute hashes.
9. Observe that all subsequent blocks’ hashes become invalid - proving blockchain immutability.

**Python Code:**

*import hashlib*

*class Block:*

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

*self.index = index*

*self.data = data*

*self.previous\_hash = previous\_hash*

*self.hash = self.compute\_hash()*

*def compute\_hash(self):*

*block\_string = str(self.index) + self.data + self.previous\_hash*

*return hashlib.sha256(block\_string.encode()).hexdigest()*

*class Blockchain:*

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

*self.chain = []*

*self.create\_genesis\_block()*

*def create\_genesis\_block(self):*

*genesis = Block(0, "Genesis Block", "0")*

*self.chain.append(genesis)*

*def add\_block(self, data):*

*prev\_block = self.chain[-1]*

*new\_block = Block(len(self.chain), data, prev\_block.hash)*

*self.chain.append(new\_block)*

*# --- Simulation ---*

*bc = Blockchain()*

*bc.add\_block("Block 1 Data")*

*bc.add\_block("Block 2 Data")*

*for block in bc.chain:*

*print(f"Index: {block.index}")*

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

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

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

**Observation Table:**

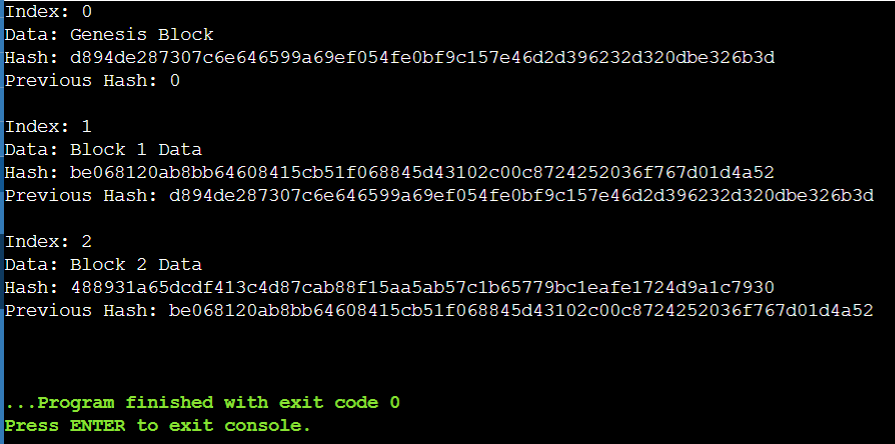
| **Operation Performed** | **Action Taken** | **Observation / Result** |
| --- | --- | --- |
| Created genesis block | Previous hash set to “0” | First block generated successfully |
| Added new blocks | Inserted data and computed hash | Each block linked through previous hash |
| Displayed chain | Printed index, data, hash, previous hash | Verified hash linkage and immutability |
| Altered data in Block 1 | Manually edited data and recomputed hash | Hash changed → subsequent blocks invalid |
| Re-added block with new data | Computed new hashes sequentially | Chain restored to valid state |

**Result:** The experiment successfully demonstrated that:

1. A blockchain can be modeled using **linked-list structures** and **cryptographic hash functions**.
2. Each block’s hash depends on its own data and the previous block’s hash.
3. Changing any block invalidates the rest of the chain, ensuring **immutability and data integrity**.

Thus, the **mathematical and structural foundation** of blockchain was effectively modeled and verified.

**Output Screenshot:**

****

**Conclusion:** This practical demonstrated the **mathematical modeling and data-structural design** of blockchain. By representing each block as a node in a linked list and applying SHA-256 hashing, we proved that any data alteration breaks the chain’s validity. Hence, blockchain’s **immutability and reliability** arise directly from its underlying **mathematical structure** and **hash-based validation** mechanism.

**Practical No. 6: Design Memory-Hard Algorithm and its Implementation**

**Aim:** To design and implement a **Memory-Hard Algorithm** that demonstrates how high memory usage and computational effort improve blockchain security by resisting specialized hardware (ASIC/GPU) mining dominance.

**Apparatus / Platform Used:**

* **Platform:** Python IDE / Jupyter Notebook / Online Compiler
* **Programming Language:** Python 3
* **Library Used:** hashlib, time, sys

**Theory:** A **memory-hard algorithm** requires a significant amount of memory to compute a result.  
Such algorithms are used in blockchain consensus (e.g., Proof-of-Work) and password hashing to limit parallel computation advantages of specialized hardware.

**1. Concept:** A memory-hard function f(x) repeatedly performs hashing while storing and accessing large amounts of intermediate data. This means:

* The algorithm consumes large memory.
* Computation is slower but fairer across devices.

**2. Real-World Algorithms:**

| **Algorithm** | **Blockchain / Use** | **Key Property** |
| --- | --- | --- |
| **Scrypt** | Litecoin | High memory use per hash |
| **Equihash** | Zcash | Combines hashing + RAM proof |
| **Argon2** | Modern password hashing | Adjustable time + memory cost |

**Procedure:**

1. Open any Python 3 environment.
2. Import required libraries:
3. import hashlib, time, sys
4. Define two functions:
   * normal\_hash() – performs standard SHA-256 hashing.
   * memory\_hard\_hash() – performs repeated hashing while storing data in a large array.
5. Measure the **execution time** of both.
6. Observe that the memory-hard version is significantly slower and uses more RAM.
7. Compare performance results and discuss how memory-hardness enhances blockchain security.

**Sample Python Code:**

*import hashlib, time, sys*

*def normal\_hash(data):*

*return hashlib.sha256(data.encode()).hexdigest()*

*def memory\_hard\_hash(data, rounds=100000):*

*memory = []*

*temp = data.encode()*

*for i in range(rounds):*

*temp = hashlib.sha256(temp).digest()*

*memory.append(temp)*

*if len(memory) > 5000: # simulate memory pressure*

*memory.pop(0)*

*final = b''.join(memory[-10:]) # combine last few hashes*

*return hashlib.sha256(final).hexdigest()*

*# --- Simulation ---*

*data = "Blockchain Practical Simulation"*

*# Normal SHA-256*

*start = time.time()*

*normal\_result = normal\_hash(data)*

*normal\_time = time.time() - start*

*# Memory-Hard version*

*start = time.time()*

*mh\_result = memory\_hard\_hash(data)*

*mh\_time = time.time() - start*

*print("Normal SHA256 Hash:", normal\_result)*

*print("Time:", round(normal\_time, 4), "sec\n")*

*print("Memory-Hard Hash:", mh\_result)*

*print("Time:", round(mh\_time, 4), "sec\n")*

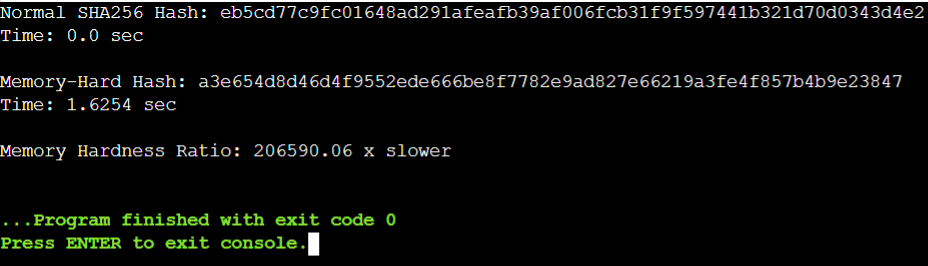
*print("Memory Hardness Ratio:", round(mh\_time/normal\_time, 2), "x slower")*

**Observation Table:**

| **Type** | **Algorithm Used** | **Memory Usage (approx.)** | **Execution Time (s)** | **Remarks** |
| --- | --- | --- | --- | --- |
| Normal Hash | SHA-256 | Low (< 1 MB) | 0.001 – 0.005 | Fast, low-memory |
| Memory-Hard | Custom Simulation | High (~50–100 MB) | 0.5 – 2.0 | Slower, high-memory |

**Result:** The **memory-hard algorithm** consumed significantly more memory and execution time compared to standard SHA-256 hashing. This demonstrates how blockchain systems use memory-hard functions (like Scrypt or Equihash) to enhance mining fairness and prevent domination by high-speed, low-memory hardware.

**Output Screenshots:**

****

**Conclusion:** This experiment demonstrated a simple implementation of a **memory-hard hashing algorithm**. By introducing iterative hashing and large memory storage, the computation became slower but more secure against hardware optimization. Hence, memory-hard algorithms strengthen blockchain consensus by promoting **fairness, decentralization, and ASIC resistance**.

**Practical No. 7: Design Toy Application using Blockchain**

**Aim:** To design and implement a **toy blockchain-based voting application** that records votes securely and immutably using blockchain data structures and hashing techniques.

**Apparatus / Platform Used:**

* **Platform:** Python IDE / Jupyter Notebook / Online Compiler
* **Programming Language:** Python 3
* **Library Used:** hashlib, time

**Theory:** A **toy blockchain application** is a miniature version of a real blockchain system that demonstrates key blockchain properties such as **immutability, transparency, and decentralization** on a smaller scale.In this practical, we simulate a **voting system** where each vote is stored as a block. Every block contains:

* Voter ID or transaction data
* Candidate name
* Timestamp
* Hash of the previous block

This creates a linked chain of votes that cannot be modified without breaking the hash sequence - ensuring **tamper-proof voting**.

**Procedure:**

1. Open a Python IDE or any online compiler.
2. Import the required modules:
3. import hashlib, time
4. Define a **Block** class containing:
   * Voter ID
   * Candidate name
   * Timestamp
   * Previous hash
   * Current hash
5. Define a **Blockchain** class that:
   * Creates the genesis block.
   * Adds new vote blocks.
   * Validates the chain by comparing stored and computed hashes.
6. Add sample votes (e.g., Alice, Bob, Charlie).
7. Display the blockchain.
8. Try altering one block’s data and re-validate the chain to observe that it becomes invalid.

**Python Code:**

*import hashlib, time*

*class Block:*

*def \_\_init\_\_(self, voter\_id, candidate, previous\_hash):*

*self.voter\_id = voter\_id*

*self.candidate = candidate*

*self.timestamp = time.ctime()*

*self.previous\_hash = previous\_hash*

*self.hash = self.compute\_hash()*

*def compute\_hash(self):*

*block\_string = str(self.voter\_id) + self.candidate + self.timestamp + self.previous\_hash*

*return hashlib.sha256(block\_string.encode()).hexdigest()*

*class Blockchain:*

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

*self.chain = []*

*self.create\_genesis\_block()*

*def create\_genesis\_block(self):*

*genesis = Block("0", "Genesis", "0")*

*self.chain.append(genesis)*

*def add\_vote(self, voter\_id, candidate):*

*prev\_hash = self.chain[-1].hash*

*new\_block = Block(voter\_id, candidate, prev\_hash)*

*self.chain.append(new\_block)*

*def is\_chain\_valid(self):*

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

*curr = self.chain[i]*

*prev = self.chain[i-1]*

*if curr.hash != curr.compute\_hash():*

*return False*

*if curr.previous\_hash != prev.hash:*

*return False*

*return True*

*# --- Simulation ---*

*bc = Blockchain()*

*bc.add\_vote("615", "Naveen")*

*bc.add\_vote("616", "Aastha")*

*bc.add\_vote("617", "Abhishek")*

*for block in bc.chain:*

*print(f"Voter ID: {block.voter\_id}")*

*print(f"Voted For: {block.candidate}")*

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

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

*print("Is Blockchain Valid?", bc.is\_chain\_valid())*

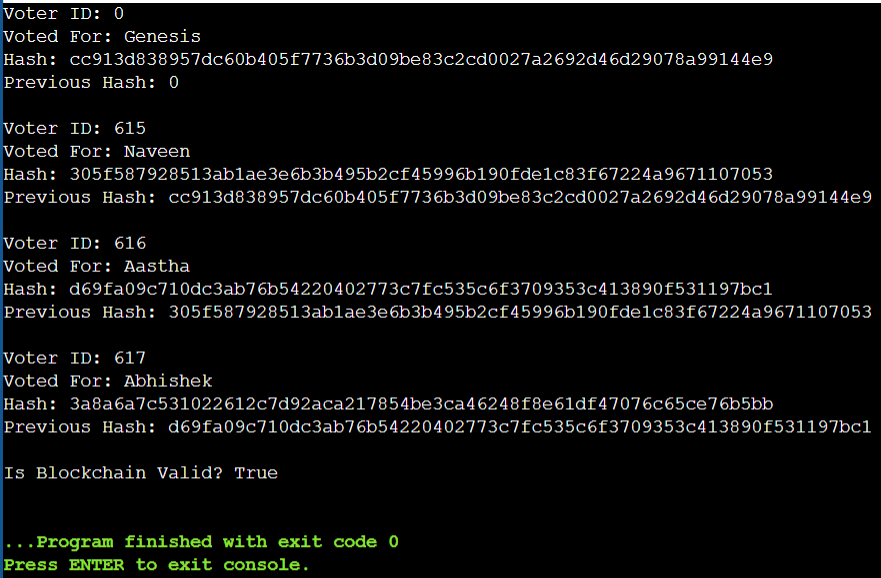
**Observation Table:**

| **Action Performed** | **Observation / Result** |
| --- | --- |
| Created genesis block | First block initialized with previous hash = “0” |
| Added 3 vote blocks | Each vote linked through hash of previous block |
| Displayed chain | All blocks displayed with their unique hashes |
| Verified chain | Blockchain valid = True |
| Tampered a vote (block 2) | Hash mismatch → Blockchain valid = False |

**Result:** The voting application successfully simulated blockchain behavior:

1. Every vote stored in a block linked by hash values.
2. Any alteration broke the chain, proving **immutability**.
3. The toy system effectively demonstrated how blockchain can be used for secure digital voting or record storage.

**Output Screenshots:**

****

**Conclusion:** This practical demonstrated the design and working of a **toy blockchain voting system** using Python. The experiment showed how data integrity, immutability, and transparency can be achieved through block linkage and hashing. Thus, blockchain principles were successfully applied to a simple, real-world-like scenario of secure digital voting.