**Aim:** To design, implement and demonstrate a simple blockchain and a Proof-of-Work (PoW) mining program that solves mining puzzles (finds valid nonces), and to analyze its performance and behavior with varying difficulty levels.

**Apparatus / Tools**

* Python 3.8+ (recommended)
* Text editor / IDE (VS Code, PyCharm, Jupyter)
* Terminal / Command Prompt
* (Optional) Git, Docker

**Theory - Basics (start from fundamentals)**

**What is a blockchain?**

A **blockchain** is an ordered, append-only chain of blocks where each block contains a set of transactions (or arbitrary data), a timestamp, the hash of the previous block, and metadata (like nonce). Each block cryptographically links to the previous block via the previous-hash field. This makes tampering detectable.

**Block structure (simple)**

Typical fields in our simple block:

* index - block number in the chain
* timestamp - creation time
* data - payload (could be transactions or messages)
* previous hash - SHA-256 hash of previous block
* nonce - a number miners adjust to change block hash
* hash - SHA-256 hash of the block header (index, timestamp, data, previous\_hash, nonce)

**What is mining?**

**Mining** in PoW blockchains means searching for a nonce such that the hash of the block header satisfies a **difficulty target** (e.g., hash begins with a certain number of zero bits or leading hex zeros). The miner repeatedly changes the nonce and recomputes the hash until the condition is met.

**Proof of Work (PoW)**

Proof of Work is a consensus technique where the miner proves they expended computational effort. The difficulty parameter controls how hard finding a valid nonce is. Higher difficulty → more computational effort.

**Mining puzzle**

The puzzle: **Find a nonce** such that SHA256(block\_header) interpreted as hex starts with d leading zeros (or hash <= some target). This is hard to solve but easy to verify (one hash compute).

**Objectives**

1. Implement a basic blockchain data structure in Python.
2. Implement PoW mining that searches for a nonce meeting a difficulty value.
3. Demonstrate mining blocks, show how difficulty affects time to mine.
4. Validate blocks and verify chain integrity.
5. Provide analysis and observations.

**Design & Algorithm**

**Mining condition (example)**

We define difficulty = k meaning a valid hash must start with k leading hexadecimal zeros (i.e., hash\_hex.startswith('0'\*k)).

Note: Using hex-leading zeros is simple to demonstrate. Real Bitcoin uses a numeric target and counts leading zero **bits**, not hex digits.

**High-level algorithm (pseudocode)**

1. Initialize blockchain with genesis block (previous\_hash = '0')
2. For each new block:
   * Set block fields (index, timestamp, data, previous\_hash)
   * Initialize nonce = 0
   * While True:
     + Compute hash = SHA256(index | timestamp | data | previous\_hash | nonce)
     + If hash satisfies difficulty target: break (nonce found)
     + Else nonce += 1
3. Append block with found nonce and hash to chain
4. Validate chain by checking each block:
   * recompute block hash and compare with stored hash
   * check previous\_hash equals the previous block’s hash
   * check that stored hash satisfies difficulty

**Implementation (Python)**

Save this as simple\_blockchain\_miner.py. The code is written to be clear for a practical report and to be run as a script.

"""

simple\_blockchain\_miner.py

A simple blockchain implementation with proof-of-work mining (educational).

Author: Sukhdayal (edit as needed)

"""

import hashlib

import time

from dataclasses import dataclass, asdict

import json

@dataclass

class Block:

index: int

timestamp: float

data: str

previous\_hash: str

nonce: int = 0

hash: str = ""

def header(self) -> str:

"""

Create a deterministic string representing the block header (fields used for hashing).

"""

# Use json to ensure deterministic ordering

header\_dict = {

"index": self.index,

"timestamp": self.timestamp,

"data": self.data,

"previous\_hash": self.previous\_hash,

"nonce": self.nonce

}

return json.dumps(header\_dict, sort\_keys=True).encode()

def sha256\_hex(data: bytes) -> str:

return hashlib.sha256(data).hexdigest()

class SimpleBlockchain:

def \_\_init\_\_(self, difficulty: int = 4):

"""

difficulty: number of leading hex zeros required in hash (educational)

"""

self.chain = []

self.difficulty = difficulty

genesis = self.create\_genesis\_block()

self.chain.append(genesis)

def create\_genesis\_block(self) -> Block:

genesis = Block(index=0, timestamp=time.time(), data="Genesis Block", previous\_hash="0")

genesis.hash = sha256\_hex(genesis.header())

# For genesis we don't enforce difficulty (or could set specially)

return genesis

def last\_block(self) -> Block:

return self.chain[-1]

def mine\_block(self, data: str, max\_nonce: int = 2\*\*32-1) -> Block:

index = self.last\_block().index + 1

previous\_hash = self.last\_block().hash

block = Block(index=index, timestamp=time.time(), data=data, previous\_hash=previous\_hash)

target\_prefix = "0" \* self.difficulty

start = time.time()

nonce = 0

while nonce <= max\_nonce:

block.nonce = nonce

h = sha256\_hex(block.header())

if h.startswith(target\_prefix):

block.hash = h

elapsed = time.time() - start

print(f"[mined] index={block.index} nonce={block.nonce} hash={block.hash} time={elapsed:.4f}s")

return block

nonce += 1

raise RuntimeError("Failed to find valid nonce within max\_nonce")

def add\_block(self, block: Block):

if self.is\_valid\_new\_block(block, self.last\_block()):

self.chain.append(block)

return True

else:

return False

def is\_valid\_new\_block(self, block: Block, previous\_block: Block) -> bool:

# Check index continuity

if block.index != previous\_block.index + 1:

print("Invalid index")

return False

# Check previous hash

if block.previous\_hash != previous\_block.hash:

print("Invalid previous hash")

return False

# Check hash validity

recomputed\_hash = sha256\_hex(block.header())

if recomputed\_hash != block.hash:

print("Hash mismatch")

return False

# Check difficulty target

if not block.hash.startswith("0" \* self.difficulty):

print("Block does not satisfy difficulty")

return False

return True

def is\_chain\_valid(self) -> bool:

# Validate genesis

if len(self.chain) == 0:

return True

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

if not self.is\_valid\_new\_block(self.chain[i], self.chain[i - 1]):

return False

return True

def demonstration():

difficulty = int(input("Enter difficulty (leading hex zeros, e.g., 3 or 4): ").strip() or "4")

bc = SimpleBlockchain(difficulty=difficulty)

print(f"Genesis hash: {bc.chain[0].hash}")

num\_blocks = int(input("How many blocks to mine? (e.g., 3): ").strip() or "3")

for i in range(num\_blocks):

payload = f"Block payload #{i+1} - demo"

print(f"Mining block {i+1} with difficulty={difficulty} ...")

block = bc.mine\_block(data=payload)

added = bc.add\_block(block)

print("Block added to chain:", added)

print("-" \* 60)

print("Final chain length:", len(bc.chain))

print("Chain valid?", bc.is\_chain\_valid())

for blk in bc.chain:

print(f"Index: {blk.index} Nonce: {blk.nonce} Hash: {blk.hash[:24]}... Data: {blk.data}")

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

demonstration()

**How to run**

1. Ensure Python 3 is installed.
2. Save the script as simple\_blockchain\_miner.py.
3. Run from terminal:

python simple\_blockchain\_miner.py

1. Enter difficulty (e.g., 3) and number of blocks (e.g., 3) when prompted.

**Sample Output (example)**

Note: Exact values differ each run.

Enter difficulty (leading hex zeros, e.g., 3 or 4): 3

Genesis hash: 4f8b7a... (example)

How many blocks to mine? (e.g., 3): 2

Mining block 1 with difficulty=3 ...

[mined] index=1 nonce=15342 hash=000a1bcd3f... time=0.8763s

Block added to chain: True

------------------------------------------------------------

Mining block 2 with difficulty=3 ...

[mined] index=2 nonce=7403 hash=00078afe12... time=0.4129s

Block added to chain: True

------------------------------------------------------------

Final chain length: 3

Chain valid? True

Index: 0 Nonce: 0 Hash: 4f8b7a... Data: Genesis Block

Index: 1 Nonce: 15342 Hash: 000a1bcd... Data: Block payload #1 - demo

Index: 2 Nonce: 7403 Hash: 00078afe... Data: Block payload #2 - demo

**Observations and Analysis**

1. **Relationship of difficulty and time**: Mining time grows roughly exponentially with the number of required leading hex zeros. For hex digits, each extra leading hex zero increases difficulty by factor ~16 (because a hex digit is 4 bits). So difficulty change from 3 to 4 hex zeros → expected ~16× more attempts on average.
2. **Verification is cheap**: Checking a mined block requires one hash compute and simple comparisons - far cheaper than searching.
3. **Nonce space**: We used a 32-bit max\_nonce by default. Real systems use larger ranges and other header fields (timestamp, extra nonce) to expand the search space.
4. **Miner fairness and luck**: Mining is probabilistic. Different runs produce different nonce values and time-to-solution.
5. **Potential bottlenecks**: Python's single-threaded execution and interpreted nature limits hashing throughput compared to optimized C/C++ or GPU miners.

**Security and Limitations**

* **Educational only**: This is a simplified educational model. Real PoW blockchains use bit-level targets, block templates, and additional fields.
* **Variant attacks**: In real systems, 51% attacks, selfish mining, and time-jacking are concerns when an attacker controls a majority of hash power.
* **Energy**: PoW is energy intensive; alternatives like Proof of Stake (PoS) exist.
* **Collision risk**: SHA-256 collisions are computationally infeasible for current technology.

**Conclusion:** We implemented a simple blockchain with PoW mining. The miner successfully finds nonces satisfying difficulty targets; time to mine grows rapidly with difficulty. The implementation shows core PoW properties: easy verification, hard solution, and probabilistic mining. This practical illustrates how mining works and provides a base for further experiments (parallel mining, networked consensus, or alternative consensus mechanisms).