NAME : Palak Chanchlani
ROLL NO: 05
CLASS : D20A
BATCH : C

BLOCKCHAIN EXP 1

AIM: Write a Python program to understand SHA and Cryptography in Blockchain, Merkle root tree hash

THEORY:

Cryptographic Hash Function in Blockchain

A cryptographic hash function is a very important concept used in blockchain technology to provide security and data integrity. It is a mathematical algorithm that converts input data of any size into a fixed-length output called a **hash value**.
In blockchain systems, the most commonly used cryptographic hash function is **SHA-256 (Secure Hash Algorithm – 256 bits)**. No matter how small or large the input is, SHA-256 always generates a 256-bit hash value.

One of the main features of a cryptographic hash function is that it is **one-way** in nature. This means once the hash is generated, it is practically impossible to get back the original input data from the hash. This property ensures high security.

Cryptographic hash functions have the following important properties:

- **Deterministic**: The same input will always produce the same hash value.
- **Fast Computation**: Hash values can be generated quickly.
- **Avalanche Effect**: Even a small change in input results in a completely different hash.
- **Collision Resistance**: It is extremely difficult to find two different inputs that generate the same hash.
- **Pre-image Resistance**: It is computationally infeasible to determine the original input from a given hash.

In blockchain, cryptographic hash functions are used to secure transaction data, link blocks together, validate proof-of-work, and maintain immutability of the blockchain.

Merkle Tree
A Merkle Tree is a special data structure used in blockchain to store and verify large numbers of transactions efficiently. It is also known as a **hash tree** because it uses cryptographic hash functions at every level of the tree.
In a Merkle Tree, each transaction is first converted into a hash. These hashes form the leaf nodes of the tree. Then pairs of hashes are combined and hashed again to form higher-level nodes. This process continues until a single hash is produced at the top, known as the **Merkle Root**.

Merkle Trees help blockchain systems verify transaction data quickly without having to process every transaction individually. If even one transaction is altered, the hash of that transaction changes, which affects all the parent hashes and finally changes the Merkle Root.

Structure of Merkle Tree
The structure of a Merkle Tree consists of multiple hierarchical levels:

1. **Leaf Nodes**
   These are the bottom-most nodes of the tree. Each leaf node contains the hash of an individual transaction.
2. **Intermediate Nodes**
   These nodes are formed by combining two child node hashes and applying a hash function on the concatenated value.
3. **Root Node**
   The top-most node is called the Merkle Root. It represents all transactions present in the block.

If the number of transactions is odd, the last transaction hash is duplicated to maintain a balanced binary tree structure. This ensures proper tree formation.

Merkle Root
The Merkle Root is the final hash obtained after recursively hashing all transaction hashes. It acts as a unique digital fingerprint for all transactions in a block.
The Merkle Root is stored in the **block header** of the blockchain. Any change in even a single transaction will result in a different Merkle Root. Therefore, the Merkle Root plays a crucial role in ensuring transaction integrity and security.

Working of Merkle Tree

The working of a Merkle Tree can be explained in the following steps:

1. Each transaction in a block is hashed using a cryptographic hash function such as SHA-256.
2. Two transaction hashes are paired and concatenated.
3. The concatenated value is hashed again to form a parent node.
4. This pairing and hashing process continues level by level.
5. When only one hash remains, it becomes the Merkle Root.

This hierarchical hashing structure allows blockchain nodes to verify transactions efficiently and securely.

Benefits of Merkle Tree

Merkle Trees provide several advantages in blockchain technology:

- Efficient verification of large data sets
- Faster transaction validation
- Reduced storage requirements
- High level of data security
- Easy detection of data tampering
- Supports lightweight and mobile blockchain clients

Use of Merkle Tree in Blockchain

In blockchain systems, Merkle Trees are used to store and verify transaction data inside a block. They enable nodes to verify a transaction without downloading the entire block. Merkle Trees are also used in **Simplified Payment Verification (SPV)**, which allows lightweight clients to verify transactions using only block headers and Merkle proofs.

Use Cases of Merkle Tree

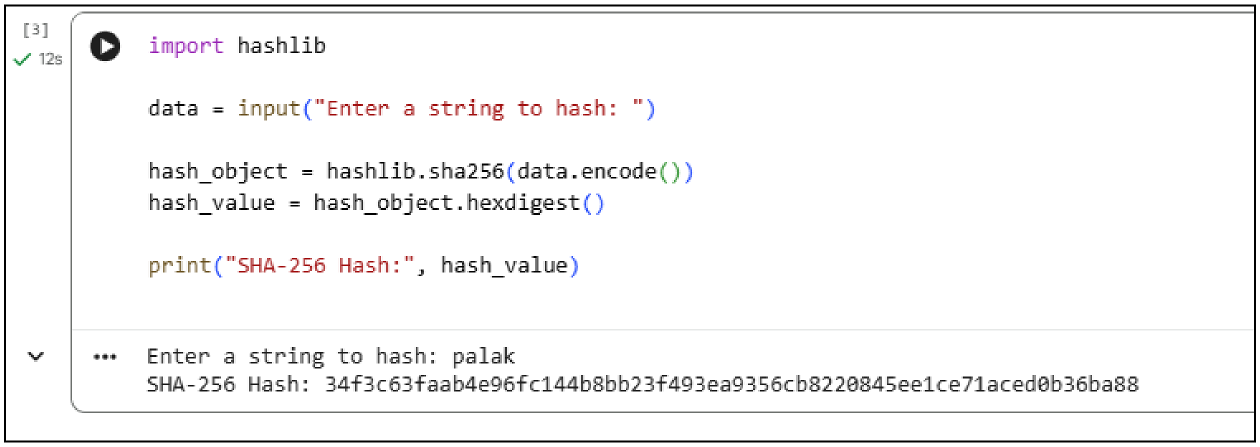
Merkle Trees are widely used in many applications such as:

- Blockchain and cryptocurrency networks
- Distributed and decentralized systems
- Secure data storage and retrieval
- Digital signatures and authentication systems
- Version control systems
- Data integrity verification mechanisms

CODE AND OUTPUT :

1. Hash Generation using SHA-256: Developed a Python program to compute a SHA-256 hash for any given input string using the hashlib library.

Objective:
To generate a SHA-256 hash for a given input string.



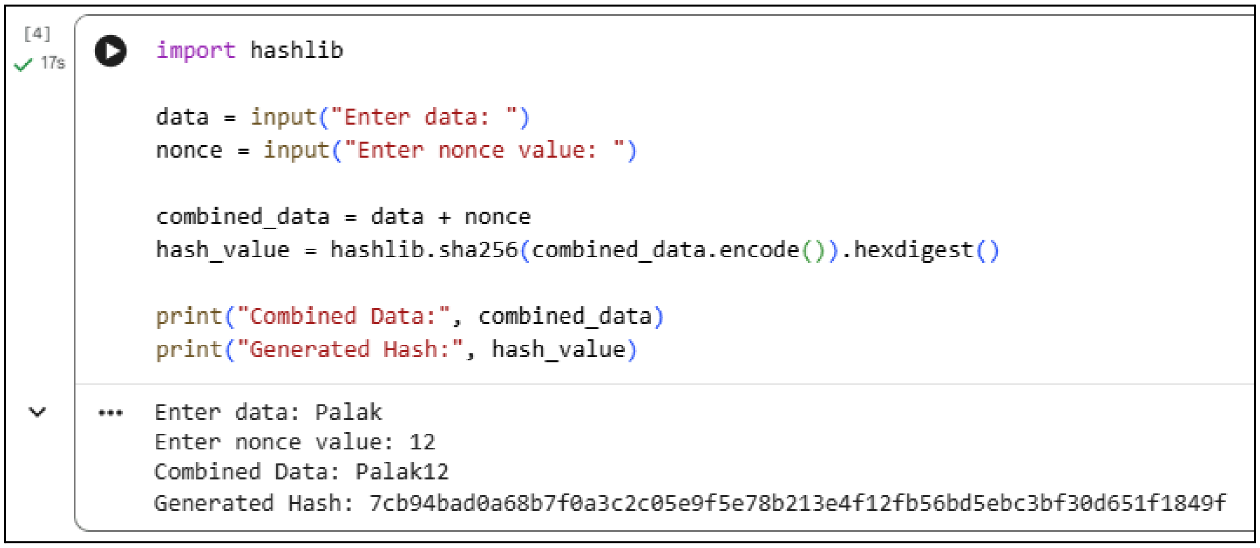
```
[3]
✓ 12s
```

```python
import hashlib

data = input("Enter a string to hash: ")

hash_object = hashlib.sha256(data.encode())
hash_value = hash_object.hexdigest()

print("SHA-256 Hash:", hash_value)
```

```
Enter a string to hash: palak
SHA-256 Hash: 34f3c63faab4e96fc144b8bb23f493ea9356cb8220845ee1ce71aced0b36ba88
```

2. Target Hash Generation with Nonce: Created a program to generate a hash code by concatenating a user input string and a nonce value to simulate the mining process.

Objective:
To generate a hash by combining input data with a nonce value.



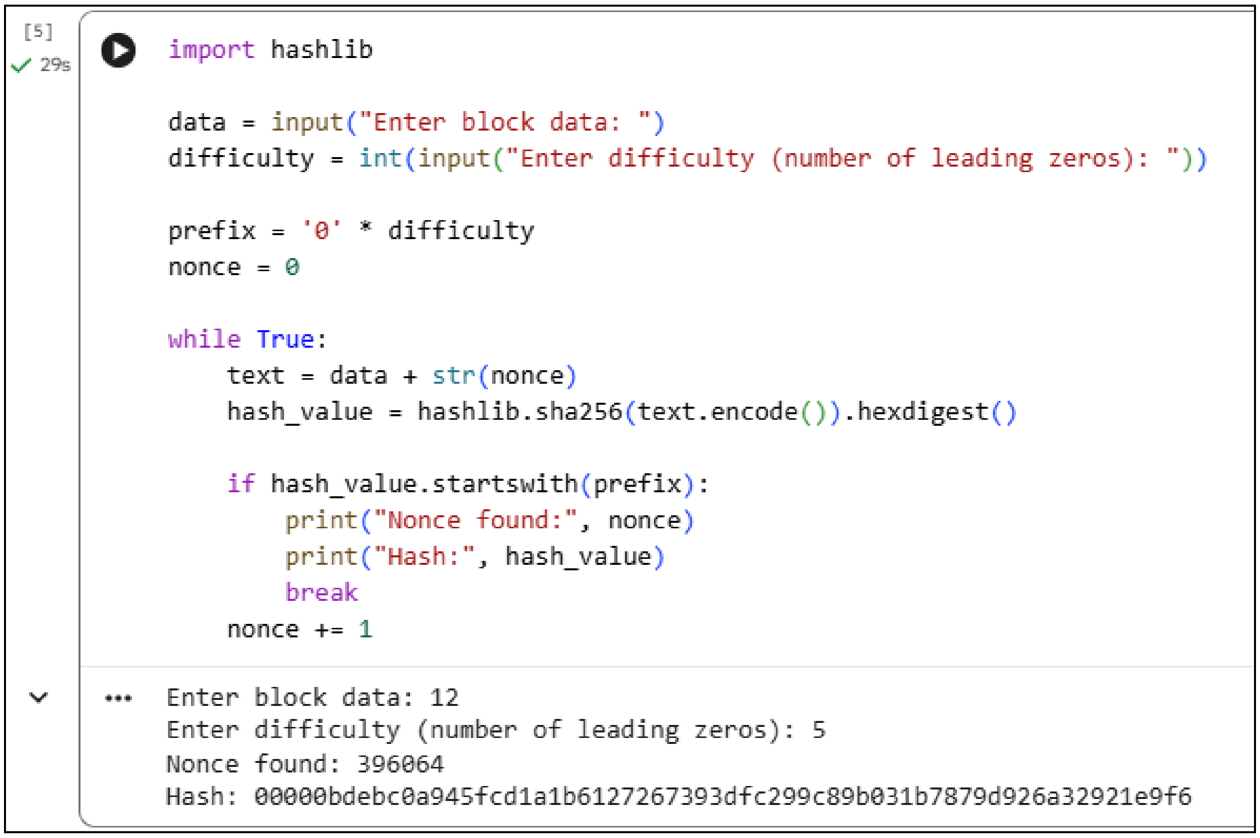
```
[4]
✓ 17s
```

```python
import hashlib

data = input("Enter data: ")
nonce = input("Enter nonce value: ")

combined_data = data + nonce
hash_value = hashlib.sha256(combined_data.encode()).hexdigest()

print("Combined Data:", combined_data)
print("Generated Hash:", hash_value)
```

```
Enter data: Palak
Enter nonce value: 12
Combined Data: Palak12
Generated Hash: 7cb94bad0a68b7f0a3c2c05e9f5e78b213e4f12fb56bd5ebc3bf30d651f1849f
```

3. Proof-of-Work Puzzle Solving: Implemented a program to find the nonce that, when combined with a given input string, produces a hash starting with a specified number of leading zeros.

Objective:
To simulate blockchain mining by finding a nonce that produces a hash with leading zeros.



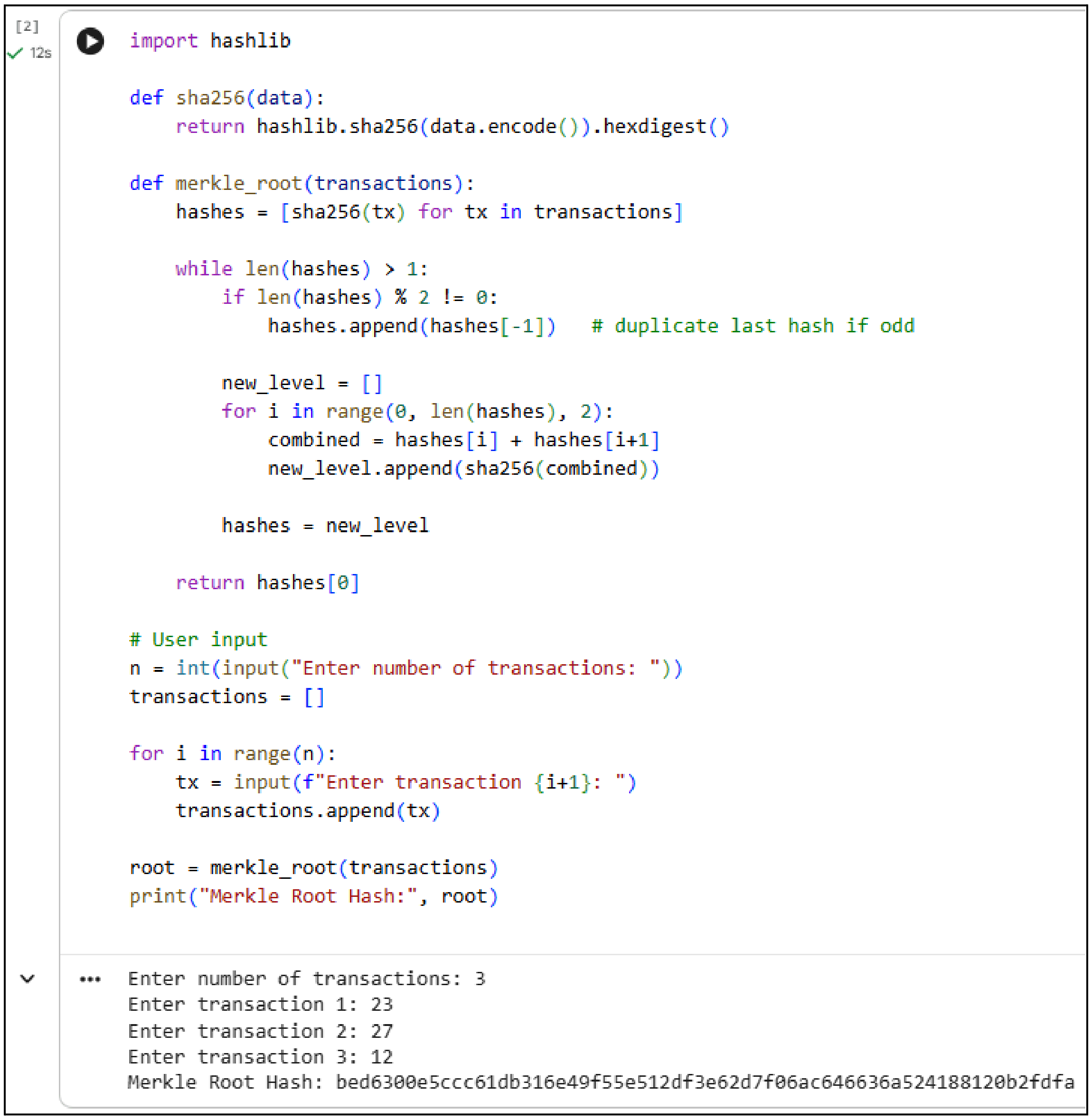
```python
import hashlib

data = input("Enter block data: ")
difficulty = int(input("Enter difficulty (number of leading zeros): "))

prefix = '0' * difficulty
nonce = 0

while True:
    text = data + str(nonce)
    hash_value = hashlib.sha256(text.encode()).hexdigest()

    if hash_value.startswith(prefix):
        print("Nonce found:", nonce)
        print("Hash:", hash_value)
        break
    nonce += 1
```

```
Enter block data: 12
Enter difficulty (number of leading zeros): 5
Nonce found: 396064
Hash: 00000bdebc0a945fcd1a1b6127267393dfc299c89b031b7879d926a32921e9f6
```

4. Merkle Tree Construction: Built a Merkle Tree from a list of transactions by recursively hashing pairs of transaction hashes, doubling up last nodes if needed, and generated the Merkle Root hash for blockchain transaction integrity.

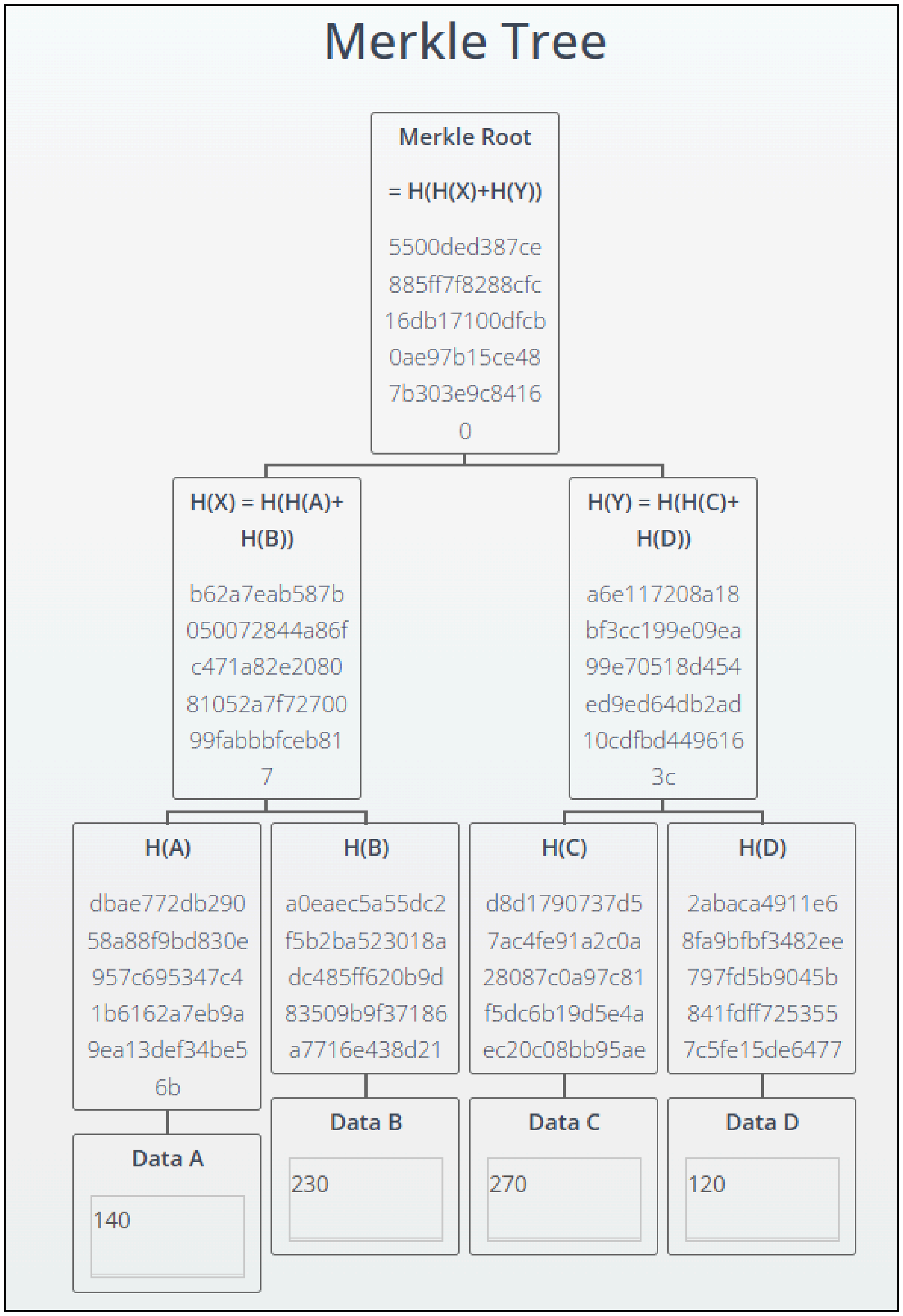
Objective

To construct a Merkle Tree from a list of transactions and generate the Merkle Root.



```python
import hashlib

def sha256(data):
    return hashlib.sha256(data.encode()).hexdigest()

def merkle_root(transactions):
    hashes = [sha256(tx) for tx in transactions]

    while len(hashes) > 1:
        if len(hashes) % 2 != 0:
            hashes.append(hashes[-1])   # duplicate last hash if odd

        new_level = []
        for i in range(0, len(hashes), 2):
            combined = hashes[i] + hashes[i+1]
            new_level.append(sha256(combined))

        hashes = new_level

    return hashes[0]

# User input
n = int(input("Enter number of transactions: "))
transactions = []

for i in range(n):
    tx = input(f"Enter transaction {i+1}: ")
    transactions.append(tx)

root = merkle_root(transactions)
print("Merkle Root Hash:", root)
```

```
Enter number of transactions: 3
Enter transaction 1: 23
Enter transaction 2: 27
Enter transaction 3: 12
Merkle Root Hash: bed6300e5ccc61db316e49f55e512df3e62d7f06ac646636a524188120b2fdfa
```


Merkle Tree
Merkle Root
= H(H(X)+H(Y))
5500ded387ce885ff7f8288cfc16db17100dfcb0ae97b15ce487b303e9c84160
H(X) = H(H(A)+H(B))
b62a7eab587b050072844a86fc471a82e208081052a7f7270099fabbbfceb817
H(Y) = H(H(C)+H(D))
a6e117208a18bf3cc199e09ea99e70518d454ed9ed64db2ad10cdfbd4496163c
H(A)
dbae772db29058a88f9bd830e957c695347c41b6162a7eb9a9ea13def34be56b
H(B)
a0eaec5a55dc2f5b2ba523018adc485ff620b9d83509b9f37186a7716e438d21
H(C)
d8d1790737d57ac4fe91a2c0a28087c0a97c81f5dc6b19d5e4aec20c08bb95ae
H(D)
2abaca4911e68fa9bfbf3482ee797fd5b9045b841fdff7253557c5fe15de6477
Data A
140
Data B
230
Data C
270
Data D
120

Conclusion:

In this experiment, we studied cryptographic hash functions and their importance in blockchain security using SHA-256. We also understood the concept and working of Merkle Trees for efficient transaction verification. The Merkle Root ensures data integrity by detecting any changes in transactions, making blockchain systems secure and trustworthy.