MOHAMAD RABI -212753107

KAMEL BDER -211695960

1)

לפני שמתחילים ללמוד – מה זה מטבע דיגיטלי בכלל? מטבעות דיגיטליים כשמם כן הם, מטבעות וירטואליים שהשימוש בהם נעשה באמצעות אמצעים דיגיטליים. זה יכול להיות אינטרנט במחשב, בסמארטפון אוב מכשירים ייעודיים.

2)

<u>כסף אלקטרוני:</u> מוגדרת באופן רחב כחנות אלקטרונית בעלת ערך כספי במכשיר טכני שעשוי לשמש באופן נרחב לביצוע תשלומים לגופים שאינם מנפיק הכסף האלקטרוני. המכשיר פועל כמכשיר בעל תשלום מראש שאינו כרוך בהכרח חשבונות בנק בעסקאות

<u>כסף דיגיטלי:</u>

לכסף דיגיטלי אין צורה פיזית ומוחשית, כמו שטר דולר או מטבע, והוא מטופל ומועבר באמצעות מערכות מקוונות

3)

התשובה ב

4)

בעיית DOUBLE SPENDING כיוון שבניגוד להעברת כספים פיזיים, העברה של כסף דיגיטלי מגורם אחד לגורם אחר לא מסירה את המידע על אמצעי התשלום מהמחשב של המעביר, זה אומר שזה מאפשר למעביר להעביר את המטבע לגורם נוסף , מכאן נובע המושג "בזבוז כפול" או באנגלית DOUBLE SPENDING.

5)

המיינרים בעצם מוודאים (על ידי הקצאת כוחות מיחשוב) שהמטבעות הועברו במקביל לחשבון אחר. אחראים לבדוק ולוודא שטרנזקציה זהה לא התבצעה באף בלוק אחר ולהוסיף את הבלוק החדש לשרשרת הבלוקים.

6)

הכורה לאחר לאחר שביצוע וידוא של ההעברה מקבל עמלה מהעסקה ומיצירת מטבעות חדשים.

7)

התשובה D

8)

בעזרת ארנק דיגיטלי חומרתי המשתמש במפתח הפרטי.

10)

פסאודו אנונימיות הוא בעצם פופי-אנונימי, הכוונה היא שלא ניתן לדעת את זהותם של צדדי העסקה, אך כן ניתן לעקוב אחר כלל התנועות ולנתח אותן .

Block:

```
from dataclasses import dataclass
import dataclasses as dc
from datetime import datetime
import exceptions
import rsa
from utilities import get_fields_str
import hashlib
from Message import Message
from typing import Callable, List


@dataclass
class Block:
    prev_block_hash: str
    index:int
    messages:List[Message]
    current_block_hash:str = dc.field(init=False)
    time_added:datetime = dc.field(init=False)
    nonce:int = dc.field(init=False)

    TOKEN_PRIZE = 3

    def compute_block_header(self)->bytes:
        block_str = \
            get_fields_str(self.prev_block_hash,self.index,self.messages,self.
time_added,self.nonce)
        return block_str.encode()

    def compute_block_hash(self)->str:
        block_hash = hashlib.sha256(self.compute_block_header()).hexdigest()
        self.current_block_hash = block_hash
        return block_hash


    def block_validation(self):
        pass
```

Block Chain :

```
import time
from datetime import timedelta, datetime
```

```python
BYTE_IN_BITS = 256
HEX_BASE_TO_NUMBER = 16
SECONDS_TO_EXPIRE = 20


GENESIS_BLOCK = Block(
    version=1,
    timestamp=1231476865,
    difficulty=1,
    nonce=1,
    previous_hash="000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8
ce26f",
    transactions=(Transaction("Satoshi Nakamoto", "Satoshi Nakamoto", 50),)
)


def calculate_difficulty_target(difficulty_bits: int) -> int:
    return 2 ** (BYTE_IN_BITS - difficulty_bits)


class Blockchain:
    VERSION = 1
    DIFFICULTY = 10
    MINUTES_TOLERANCE = 1

    def __init__(self):
        self.chain = [GENESIS_BLOCK]

    def get_last_block(self) -> Block:
        return self.chain[-1]

    def get_difficulty(self) -> int:
        return self.DIFFICULTY

    def add_block(self, block: Block) -> bool:
        is_valid = self._validate_block(block)
        if is_valid:
            self.chain.append(block)

        return is_valid

    def _validate_block(self, candidate: Block) -> bool:
        if candidate.version != self.VERSION:
            return False

        last_block = self.get_last_block()
        if candidate.previous_hash != encode_block(last_block):
```

```
            return False

        if candidate.difficulty != self.DIFFICULTY:
            return False

        min_allowed_time = (datetime.now() -
timedelta(minutes=self.MINUTES_TOLERANCE)).timestamp()
        if candidate.timestamp < min_allowed_time:
            return False

        candidate_hash = encode_block(candidate)
        candidate_decimal = int(candidate_hash, HEX_BASE_TO_NUMBER)

        target = calculate_difficulty_target(self.DIFFICULTY)
        is_block_valid = candidate_decimal < target

        return is_block_valid


blockchain = Blockchain()


def mine_proof_of_work(nonce: int, difficulty: int, prev_hash: str) ->
tuple[bool, Block]:
    block = create_block(nonce, difficulty, prev_hash)
    encoded_block = encode_block(block)
    block_encoded_as_number = int(encoded_block, HEX_BASE_TO_NUMBER)
    decimal_target = calculate_difficulty_target(difficulty)

    if block_encoded_as_number < decimal_target:
        return True, block

    return False, block


nonce = 0
start_time = time.time()
found = False
prev_hash = encode_block(blockchain.get_last_block())

while not found:
    found, block = mine_proof_of_work(nonce, blockchain.get_difficulty(),
prev_hash)

    if found:
        blockchain.add_block(block)
    else:
        print(f"❌ Nonce {nonce} didn't meet the difficulty target...")
        nonce += 1
```

```
    end_time = time.time()
    elapsed_time = end_time - start_time

    if elapsed_time > SECONDS_TO_EXPIRE:
        raise TimeoutError(
            f"Couldn't find a block with difficulty
{blockchain.get_difficulty()} fast enough"
        )

print(
    f"✅ Nonce {nonce} meet difficulty target, you mined the block in
{timedelta(seconds=elapsed_time)}!"
)
print("Here's the the blockchain: ⛓")
print(blockchain.chain)
```

Exceptions :

```
class TransactionException(Exception):
    def __init__(self,*args):
        Exception.__init__(self,*args)
```

Message :

```
from dataclasses import dataclass
```

```python
import dataclasses as dc
from datetime import datetime
import exceptions
import rsa
from utilities import get_fields_str
from rsa import VerificationError


@dataclass
class Message:
    amount:float
    sender_addr:rsa.PublicKey
    receiver_addr:rsa.PublicKey
    timestamp: datetime=dc.field(init=False)
    message_signature:bytes=dc.field(init=False)

    def __post_init__(self):
        self.timestamp = datetime.now()

    def message_as_bytes(self)->bytes:
        message_str =
get_fields_str(self.timestamp,self.amount,self.sender_addr,self.receiver_addr)
        return message_str.encode()

    def sign_message(self,sender_priv_key:rsa.PrivateKey,hash_algo:str)->None:
        known_hashes = ['MD5', 'SHA-1','SHA-224', 'SHA-256', 'SHA-384','SHA-
512']
        if hash_algo not in known_hashes:
            raise exceptions.TransactionException("Hash method is not valid")
        self.message_signature =
rsa.sign(self.message_as_bytes(),sender_priv_key,hash_algo)

        self.verify()


    def verify_message(self)->str:

       try:
           hash_method_name =
rsa.verify(self.message_as_bytes(),self.message_signature,self.sender_addr)
           return hash_method_name
       except VerificationError as ve:
           raise exceptions.TransactionException("Verification failed: " +
str(ve))
```