# Blockchain Lab Report

Xiaoxuan HEI

**Exercise 1. Where is the address to the previous block stored?**

It's stored in *Previous Hash*.

**Exercise 2. Why is it import to sort the keys?**

If we sort the keys, we can make sure the order of values is always the same. So, the blocks can be serialized in order. We can get the text with the same format every time.

```python
def __init__(self, index, previous_hash, timestamp, nonce):
    # Store internally
    self.index = index
    self.previous_hash = previous_hash
    self.timestamp = timestamp
    self.nonce = nonce

def to_dict(self):
    # Transform object into a dictionary for future transformation in JSON
    # The gave of the fields are the name of the variables
    head_dict = {}
    head_dict['index'] = self.index
    head_dict['previous_hash'] = self.previous_hash
    head_dict['timestamp'] = self.timestamp
    head_dict['nonce'] = self.nonce
    return head_dict

def to_json(self):
    # Transforms into a json string
    # use the option sort_key=True to make the representation unique
    return json.dumps(self.to_dict(), sort_keys=True)
```

**Exercise 3. Which of the header's fields is unnecessary and redundant?**

*Index*. Because a blockchain is a linked list of blocks. With the *Previous Hash*, we can easily know the position of every blocks. The index doesn't matter.

**Exercise 4. In the file *block reader*, Implement the method read header.**

```python
def read_header(header):
    # Implement these functions to help you
    # Takes a dictionary as an input
    return BlockHeader(header["index"], header["previous_hash"], header["timestamp"], header["nonce"])
```

**Exercise 5. Compare our header with the header of Bitcoin. Which fields are**

**missing? Describe briefly what they are used for.**

version: The block version number indicates which set of block validation rules to follow.

merkle root hash: The merkle root is derived from the hashes of all transactions included in this block, ensuring that none of those transactions can be modified without modifying the header.

nBits: An encoded version of the target threshold. This block's header hash must be less than or equal to nBits.

**Exercise 6. In the class *Transaction*, implement the functions *__init__* and *to_dict*.**

```
def __init__(self, index, sender, receiver, amount):
    # Store internally
    self.index = index
    self.sender = sender
    self.receiver = receiver
    self.amount = amount
    pass

def to_dict(self):
    # Transform object into a dictionary for future transformation in JSON
    # The names of the fields are the name of the variables
    tran_dict = {}
    tran_dict["index"] = self.index
    tran_dict["sender"] = self.sender
    tran_dict["receiver"] = self.receiver
    tran_dict["amount"] = self.amount
    return tran_dict
```

**Exercise 7. Which field is missing to make the transaction secured? Discuss a possible attack.**

The Digital Signature is missing. Without this signature, the data might be forged because we can't authenticate the source of messages.

**Exercise 8. How is a person generally identified on a blockchain? How does a person prove ownership?**

Putting the hash of the signature of a file's hash in the blockchain makes the owner of the private key associated with the signature the owner of the file in the blockchain. Only the people with the private key associated with the signature can prove they are the owner. So, the person can prove ownership by signing with private key.

**Exercise 9. In the file *block_reader*, implement the method *read_transaction*.**

```
def read_transaction(transaction):
    # Same above for transformation
    return Transaction(transaction["index"],transaction["sender"],transaction["receiver"],transaction["amount"])
```

**Exercise 10. In the file *merkle_tree* complete the function *create_merkle_tree*.**

```
def create_hashList(transactions):
    # Using the Merkle algorithm build the tree from a list of transactions in the block
    # transactions is list of Transaction
    hashList = []
    for t in transactions:
        hashList.append(hashlib.sha256(t).hexdigest())
    return hashList

def create_merkle_tree(hashList):
    if len(hashList) == 1:
        return hashList[0]

    newHashList = []

    for i in range(0, len(hashList) - 1, 2):
        newHashList.append(dhash256(hashList[i], hashList[i + 1]))

    if len(hashList) % 2 == 1:  # odd, hash last item twice
        newHashList.append(dhash256(hashList[-1], hashList[-1]))
    return create_merkle_tree(newHashList)

def dhash256(a, b):
    # due to big-endian / little-endian nonsense
    concat = a + b
    temp = hashlib.sha256(concat).digest()
    h = hashlib.sha256(temp).hexdigest()
    return h
```

**Exercise 11. What is the advantage of using the Merkle tree?**

The Merkle tree allows verifying that a transaction exists in the block without having the entire block, by following its Merkle branch.

**Exercise 12. In the class *Block*, implement the functions *__init__*, *to_dict*, *to_json*.**

```
def __init__(self, header, transactions):
    # Store everything internally
    # header is a BlockHeader and transactions is a list of Transaction
    # call create_merkle_tree function to store transactions in Merkle tree
    self.N_STARTING_ZEROS = 4
    self.header = header
    self.transactions = transactions
    create_merkle_tree(create_hashList(transactions))
    pass

def to_dict(self):
    # Turns the object into a dictionary
    # There are two fields: header and transactions
    # The values are obtained by using the to_dict methods
    block_dict = {}
    block_dict['header'] = self.header
    block_dict['transactions'] = self.transactions
    return block_dict

def to_json(self):
    # Transforms into a json string
    # use the option sort_key=True to make the representation unique
    return json.dumps(self.to_dict(), sort_key=True)
```

**Exercise 13. In the file *block_reader,* implement the method *read_block* and *read_block_json*.**

```python
def read_block(block):
    # Reads a block from a dictionary
    header = read_header(block["header"])
    transactions = []
    for transaction in block["transactions"]:
        transactions.append(read_transaction(transaction))
    return Block(header, transactions)

def read_block_json(block_json):
    # Reads a block in json format
    return read_block(json.loads(open(block_json, 'r').read()))
```

**Exercise 15. Why this is a bad choice for pricing function? Which of the properties of the pricing function, this function does not have?**

This function is amenable to amortizations. So, this is a bad choice for pricing function. And it's not easy to compute.

**Exercise 16. Why is *nonce* the only parameter we want to be able to modify?**

*Nonce* is a 32-bit field whose value is set so that the hash of the block will contain a run of leading zeros. For the proof of work, changing a little of data in the block will cause a huge change in hash, so *nonce* is used to find the hash value and define the difficulty of work. And also, the rest of parameters has its own meaning, we can't modify them.

**Exercise 17. In the class *BlockHeader*, write the function *get_hash*.**

```python
def get_hash(self):
    # Use hashlib to hash the block using sha256
    # Use hexdigest to get a string result
    return hashlib.sha256(self.to_json().encode("utf8")).hexdigest()
```

**Exercise 18. What is the advantage of hashing only the header and not the entire block?**

The header has contained all the information about the block because it has the field "Merkle Root" that gives the information about transactions. Hashing the header is enough to ensure the security. And instead of hashing the entire block, we can reduce much memory use and make the proof of work more efficient.

**Exercise 19. The structure of our header has a huge security problem. Can you guess it? Describe a possible attack.**

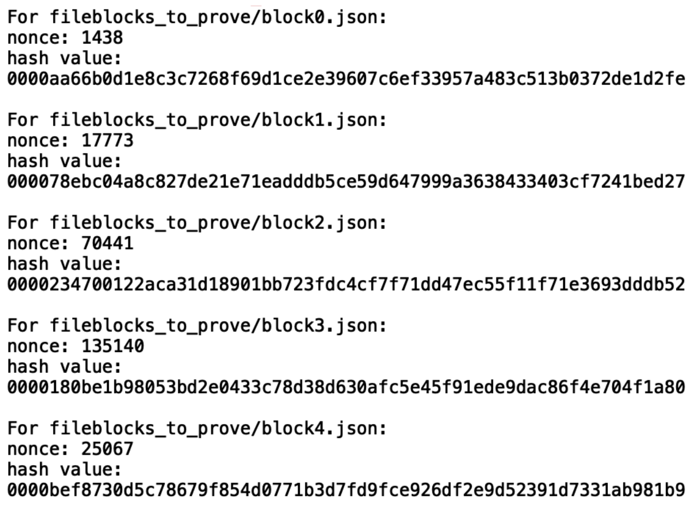
The header of our header is missing the field *merkle root*, which hashes all the transactions in the block. If the transaction is modified, the header of block won't change, and the hash value won't change. It means that the attacker can change the information of transactions without changing the header.

**Exercise 20. In the class *Block*, implement the method *is_proof_ready* and *make_proof_ready*.**

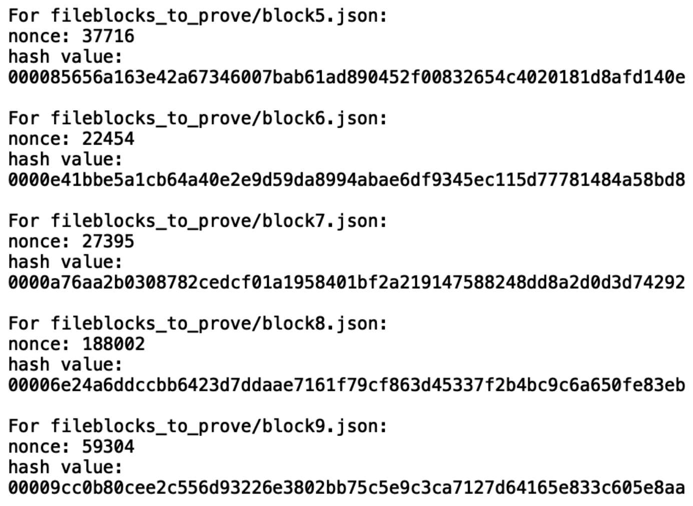
```
def is_proof_ready(self):
    # Check whether the block is proven
    # For that, make sure the hash begins by N_STARTING_ZEROS
    hash_value = self.header.get_hash()
    if hash_value[:self.N_STARTING_ZEROS] == "0000":
        return True
    else:
        return False

def make_proof_ready(self):
    # Transforms the block into a proven block
    nonce = 0
    while not self.is_proof_ready():
        nonce += 1
        self.header.set_nonce(nonce)
    print("nonce: " + str(nonce))
    print("hash value: " + self.header.get_hash())
```

**Exercise 21. For all blocks in the directory blocks to prove, prove it and give the nonce and the value of the hash function.**

```
for i in range (10):
    file = "blocks_to_prove/block" + str(i) + ".json"
    with open(file, 'r') as load_f:
        block = read_block(json.load(load_f))
        print("For file" + file +": ")
        block.make_proof_ready()
```



```
For fileblocks_to_prove/block0.json:
nonce: 1438
hash value:
0000aa66b0d1e8c3c7268f69d1ce2e39607c6ef33957a483c513b0372de1d2fe

For fileblocks_to_prove/block1.json:
nonce: 17773
hash value:
000078ebc04a8c827de21e71eadddb5ce59d647999a3638433403cf7241bed27

For fileblocks_to_prove/block2.json:
nonce: 70441
hash value:
0000234700122aca31d18901bb723fdc4cf7f71dd47ec55f11f71e3693dddb52

For fileblocks_to_prove/block3.json:
nonce: 135140
hash value:
0000180be1b98053bd2e0433c78d38d630afc5e45f91ede9dac86f4e704f1a80

For fileblocks_to_prove/block4.json:
nonce: 25067
hash value:
0000bef8730d5c78679f854d0771b3d7fd9fce926df2e9d52391d7331ab981b9
```

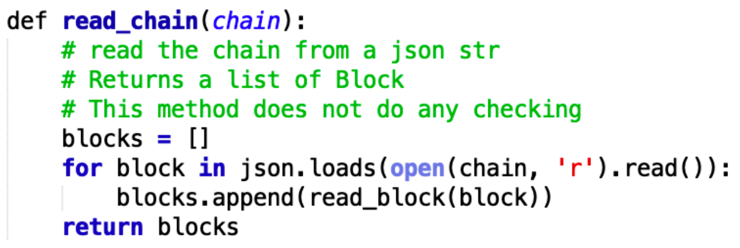


```
For fileblocks_to_prove/block5.json:
nonce: 37716
hash value:
000085656a163e42a67346007bab61ad890452f00832654c4020181d8afd140e

For fileblocks_to_prove/block6.json:
nonce: 22454
hash value:
0000e41bbe5a1cb64a40e2e9d59da8994abae6df9345ec115d77781484a58bd8

For fileblocks_to_prove/block7.json:
nonce: 27395
hash value:
0000a76aa2b0308782cedcf01a1958401bf2a219147588248dd8a2d0d3d74292

For fileblocks_to_prove/block8.json:
nonce: 188002
hash value:
00006e24a6ddccbb6423d7ddaae7161f79cf863d45337f2b4bc9c6a650fe83eb

For fileblocks_to_prove/block9.json:
nonce: 59304
hash value:
00009cc0b80cee2c556d93226e3802bb75c5e9c3ca7127d64165e833c605e8aa
```

**Exercise 22. Take one block from the directory blocks to prove and observe what happens when you increase the number of requested starting zeros in the proof. From which number of leading zeros does the computation of the proof takes more than one minute? How many leading zeros are required in the Bitcoin system?**

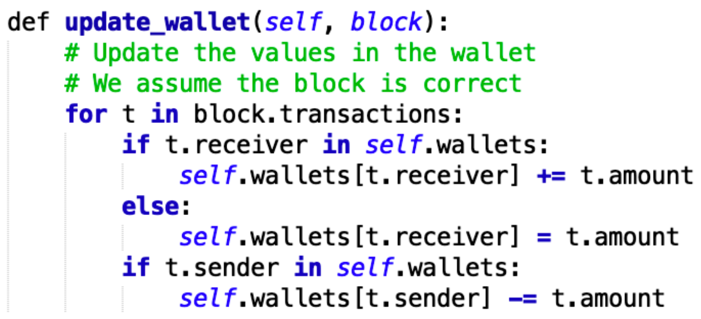
For "block0.json", when I increase the number of requested starting zeros, the running time will also increase. From 7 leading zeros, the computation of the proof takes more than 1 minute. In the year 2016 -2018, the required leading zeros was from 17 to 19.

**Exercise 23. In the file *block reader*, implement the method *read_chain*.**



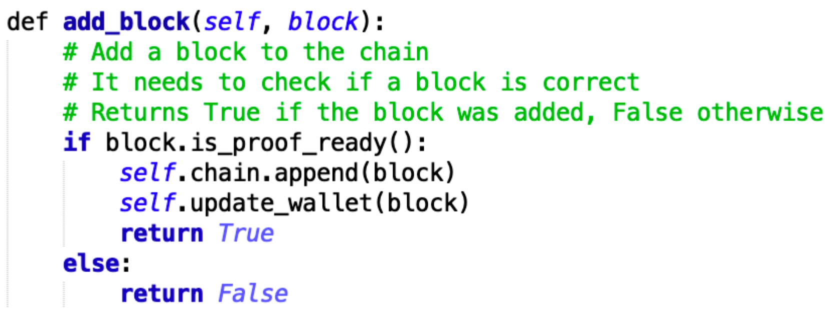
```
def read_chain(chain):
    # read the chain from a json str
    # Returns a list of Block
    # This method does not do any checking
    blocks = []
    for block in json.loads(open(chain, 'r').read()):
        blocks.append(read_block(block))
    return blocks
```

**Exercise 24. In the class *Blockchain*, implement the method *update_wallet*.**



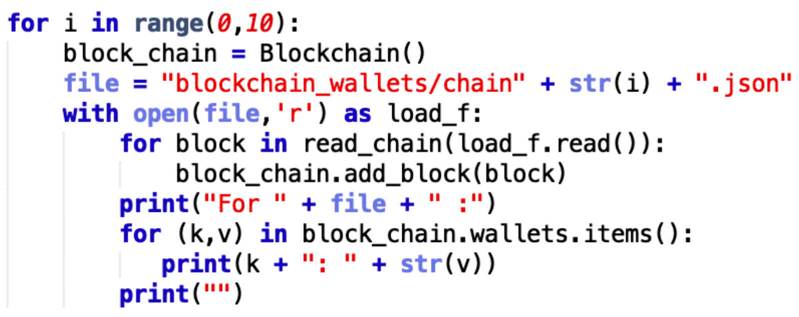
```
def update_wallet(self, block):
    # Update the values in the wallet
    # We assume the block is correct
    for t in block.transactions:
        if t.receiver in self.wallets:
            self.wallets[t.receiver] += t.amount
        else:
            self.wallets[t.receiver] = t.amount
        if t.sender in self.wallets:
            self.wallets[t.sender] -= t.amount
```

**Exercise 25. In the class *Blockchain*, implement the method *add_block*.**

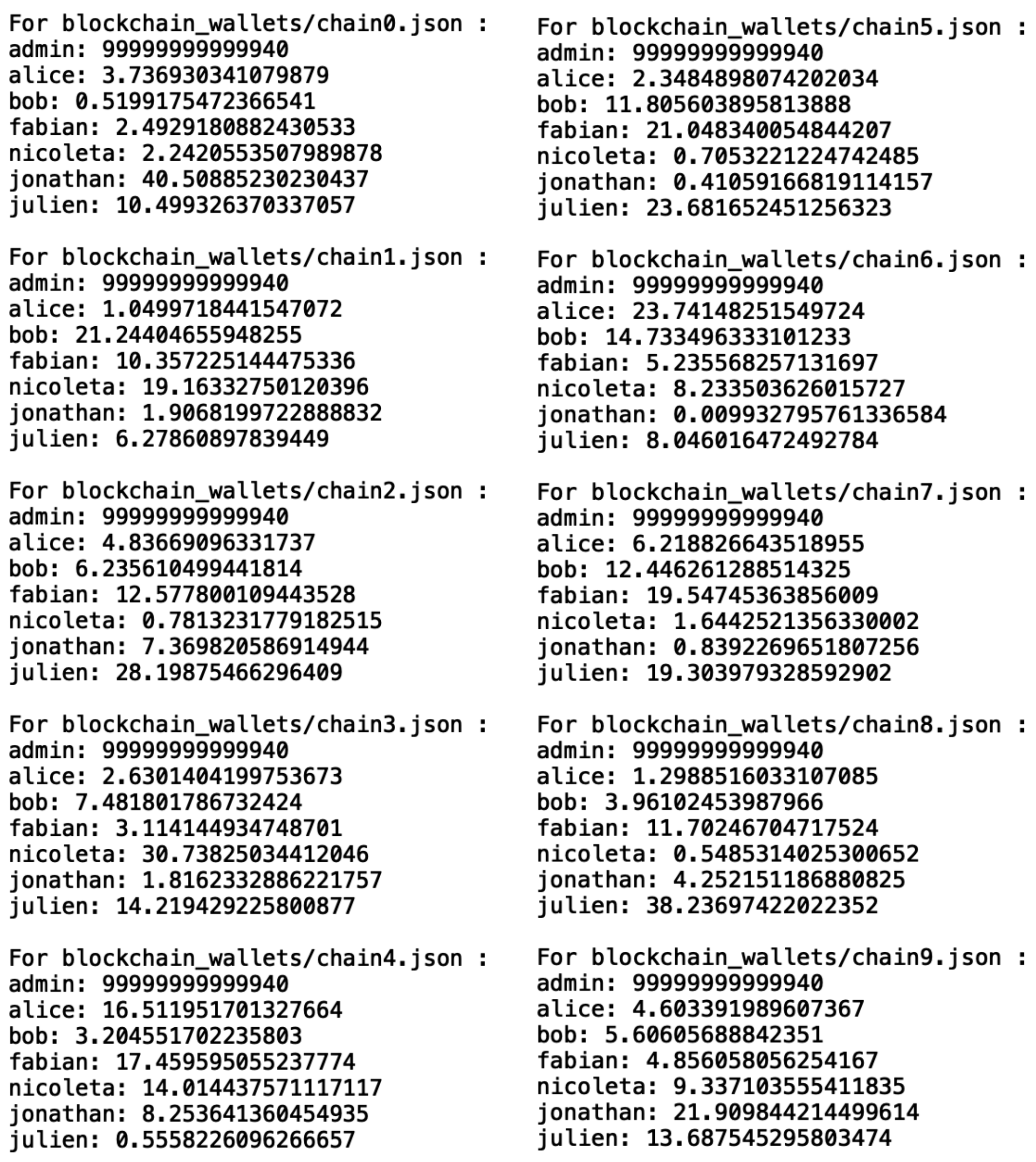


```
def add_block(self, block):
    # Add a block to the chain
    # It needs to check if a block is correct
    # Returns True if the block was added, False otherwise
    if block.is_proof_ready():
        self.chain.append(block)
        self.update_wallet(block)
        return True
    else:
        return False
```

**Exercise 26. For each blockchain in the directory *blockchain_wallets*, compute the value of the wallet of each user.**



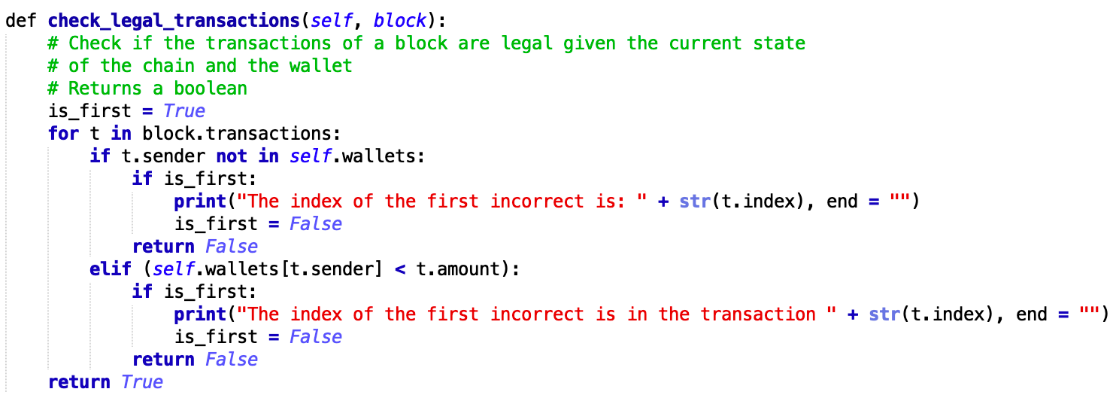
```
for i in range(0,10):
    block_chain = Blockchain()
    file = "blockchain_wallets/chain" + str(i) + ".json"
    with open(file,'r') as load_f:
        for block in read_chain(load_f.read()):
            block_chain.add_block(block)
        print("For " + file + " :")
        for (k,v) in block_chain.wallets.items():
           print(k + ": " + str(v))
        print("")
```



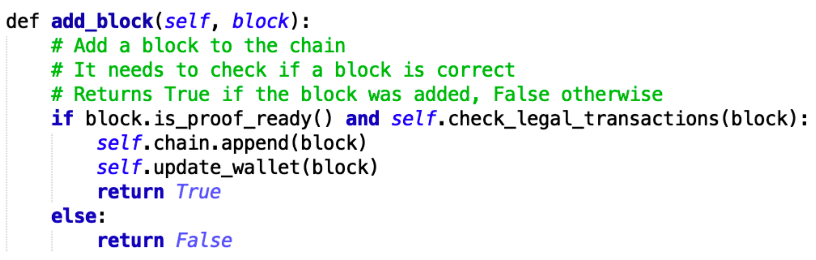
```
For blockchain_wallets/chain0.json :
admin: 99999999999940
alice: 3.736930341079879
bob: 0.5199175472366541
fabian: 2.4929180882430533
nicoleta: 2.2420553507989878
jonathan: 40.50885230230437
julien: 10.499326370337057

For blockchain_wallets/chain1.json :
admin: 99999999999940
alice: 1.0499718441547072
bob: 21.24404655948255
fabian: 10.357225144475336
nicoleta: 19.16332750120396
jonathan: 1.9068199722888832
julien: 6.27860897839449

For blockchain_wallets/chain2.json :
admin: 99999999999940
alice: 4.83669096331737
bob: 6.235610499441814
fabian: 12.577800109443528
nicoleta: 0.7813231779182515
jonathan: 7.369820586914944
julien: 28.19875466296409

For blockchain_wallets/chain3.json :
admin: 99999999999940
alice: 2.6301404199753673
bob: 7.481801786732424
fabian: 3.114144934748701
nicoleta: 30.73825034412046
jonathan: 1.8162332886221757
julien: 14.219429225800877

For blockchain_wallets/chain4.json :
admin: 99999999999940
alice: 16.511951701327664
bob: 3.204551702235803
fabian: 17.459595055237774
nicoleta: 14.014437571117117
jonathan: 8.253641360454935
julien: 0.5558226096266657
```

```
For blockchain_wallets/chain5.json :
admin: 99999999999940
alice: 2.3484898074202034
bob: 11.805603895813888
fabian: 21.048340054844207
nicoleta: 0.7053221224742485
jonathan: 0.41059166819114157
julien: 23.681652451256323

For blockchain_wallets/chain6.json :
admin: 99999999999940
alice: 23.74148251549724
bob: 14.733496333101233
fabian: 5.235568257131697
nicoleta: 8.233503626015727
jonathan: 0.009932795761336584
julien: 8.046016472492784

For blockchain_wallets/chain7.json :
admin: 99999999999940
alice: 6.218826643518955
bob: 12.446261288514325
fabian: 19.54745363856009
nicoleta: 1.6442521356330002
jonathan: 0.8392269651807256
julien: 19.303979328592902

For blockchain_wallets/chain8.json :
admin: 99999999999940
alice: 1.2988516033107085
bob: 3.96102453987966
fabian: 11.70246704717524
nicoleta: 0.5485314025300652
jonathan: 4.252151186880825
julien: 38.23697422022352

For blockchain_wallets/chain9.json :
admin: 99999999999940
alice: 4.603391989607367
bob: 5.60605688842351
fabian: 4.856058056254167
nicoleta: 9.337103555411835
jonathan: 21.909844214499614
julien: 13.687545295803474
```

**Exercise 27. In the class *Blockchain*, implement the method *check_legal_ transactions*. Add this new checking to the method add block so you do not add incorrect blocks.**

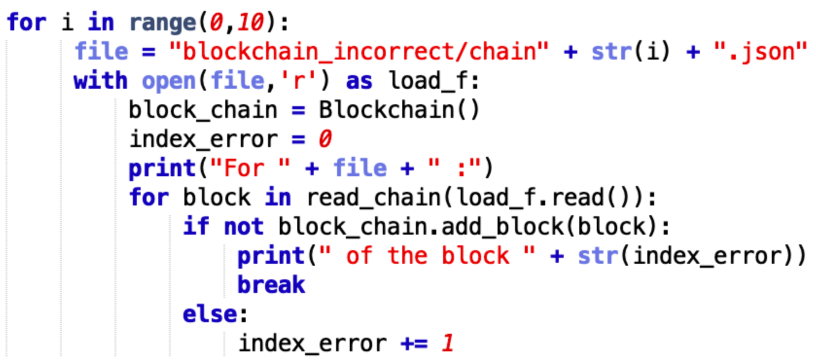


```python
def check_legal_transactions(self, block):
    # Check if the transactions of a block are legal given the current state
    # of the chain and the wallet
    # Returns a boolean
    is_first = True
    for t in block.transactions:
        if t.sender not in self.wallets:
            if is_first:
                print("The index of the first incorrect is: " + str(t.index), end = "")
                is_first = False
            return False
        elif (self.wallets[t.sender] < t.amount):
            if is_first:
                print("The index of the first incorrect is in the transaction " + str(t.index), end = "")
                is_first = False
            return False
    return True
```

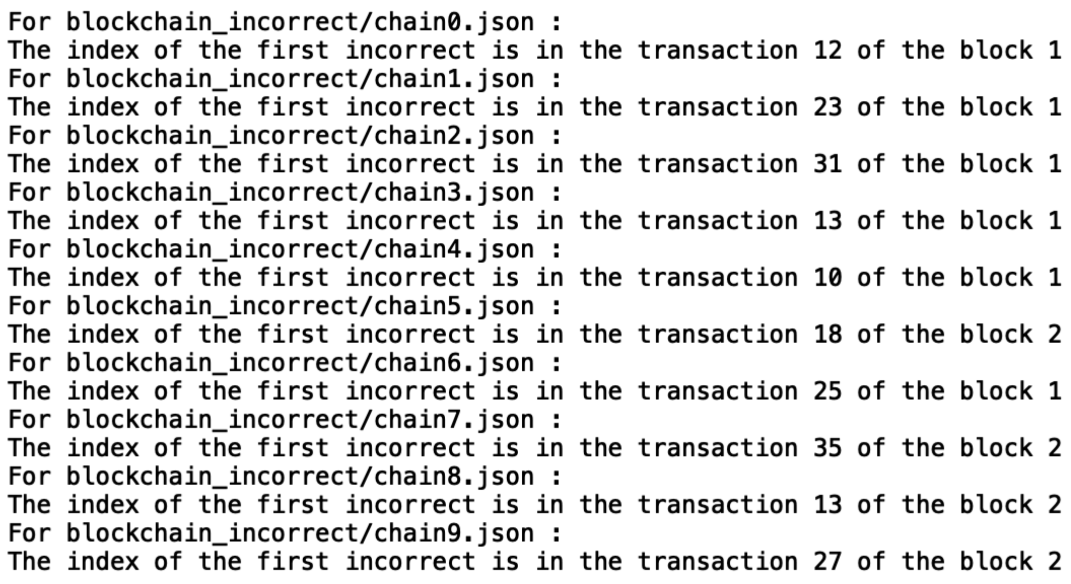


```python
def add_block(self, block):
    # Add a block to the chain
    # It needs to check if a block is correct
    # Returns True if the block was added, False otherwise
    if block.is_proof_ready() and self.check_legal_transactions(block):
        self.chain.append(block)
        self.update_wallet(block)
        return True
    else:
        return False
```

**Exercise 28. For each blockchain in the directory blockchain incorrect, check if it is correct. If a blockchain is incorrect, give the index of the first incorrect block and if necessary, the index of the first incorrect transaction.**



```python
for i in range(0,10):
    file = "blockchain_incorrect/chain" + str(i) + ".json"
    with open(file,'r') as load_f:
        block_chain = Blockchain()
        index_error = 0
        print("For " + file + " :")
        for block in read_chain(load_f.read()):
            if not block_chain.add_block(block):
                print(" of the block " + str(index_error))
                break
            else:
                index_error += 1
```



```
For blockchain_incorrect/chain0.json :
The index of the first incorrect is in the transaction 12 of the block 1
For blockchain_incorrect/chain1.json :
The index of the first incorrect is in the transaction 23 of the block 1
For blockchain_incorrect/chain2.json :
The index of the first incorrect is in the transaction 31 of the block 1
For blockchain_incorrect/chain3.json :
The index of the first incorrect is in the transaction 13 of the block 1
For blockchain_incorrect/chain4.json :
The index of the first incorrect is in the transaction 10 of the block 1
For blockchain_incorrect/chain5.json :
The index of the first incorrect is in the transaction 18 of the block 2
For blockchain_incorrect/chain6.json :
The index of the first incorrect is in the transaction 25 of the block 1
For blockchain_incorrect/chain7.json :
The index of the first incorrect is in the transaction 35 of the block 2
For blockchain_incorrect/chain8.json :
The index of the first incorrect is in the transaction 13 of the block 2
For blockchain_incorrect/chain9.json :
The index of the first incorrect is in the transaction 27 of the block 2
```