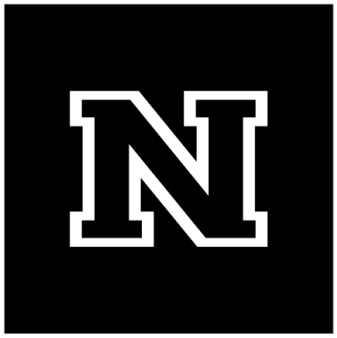
Department of Computer Science and Engineering
University of Nevada, Reno



Course No.: **CS 705**
Course Name: **Cryptography and Blockchain**

Assignment No.: **1**
Assignment Name: **Bitcoin**

Date of Submission: **April 16, 2020**

Student Name: **Farhan Sadique**
NSHE ID: **8001062452**

Part 1: Answer the following questions: 5pt (1+1+1+2)

Please read the bitcoin white paper https://bitcoin.org/bitcoin.pdf and respond to the following questions:

1. Double Spending: Please explain how Bitcoin addressed double spending.
   Answer: Bitcoin addresses double-spending by keeping a public record of all transactions using a peer-to-peer (P2P) network. Bitcoin ensures that records cannot be modified by an attacker using proof-of-work (PoW) that quickly becomes computationally impractical for the attacker to change if honest nodes control a majority of CPU power.

2. Proof of Work (Mining): In your own words please explain how Bitcoin deters denial of service attacks or other service abusers.
   Answer: Since bitcoin runs on a peer-to-peer distributed network, it is really difficult to take the network down using denial-of-service (DoS) attacks. Because to take down the bitcoin network, a malicious party must take down every single miner node. For a traditional network, the attacker must attack only one server or one gateway which is easier to do than attacking thousands of miners with varying network capacity.
   However, network DoS is not the only type of DoS attack on bitcoin network. Malicious users can slow down the network by posting a lot of small transactions (e.g. 1 cent) on the block. Since, bitcoin only allows 1 block to be mined every 10 minutes, this can deny legitimate users to complete their transactions.
   This second type of DoS attacks take care of themselves, because of the transaction fees of each transaction. The attacker will continuously lose his/her bitcoin as transaction fees because of making a lot of small transactions. As a result, he/she will eventually be left with no bitcoin to transact bringing an end to the DoS attack.

3. Incentives: How does Bitcoin incentivize nodes to mine on the network? What are some other effects of these incentives (positive, negative, neutral)?
   Answer: Bitcoin provides two types of incentives: transaction fee and mining reward. The mining reward is the first transaction of each block and is offered as a reward for mining a block. This is done to attract miners initially. However, there is only a limited number of coins to be mined (21 million for bitcoin), and after that the incentives are provided by transaction fees only.
   These incentives encourage miners to stay honest, because stealing bitcoins require a lot of CPU power to show PoW for all earlier blocks again. Rather than doing that, a node is better off mining bitcoins honestly using the same CPU power. Also, as he acquires more coins, he is more likely to use his CPU power to protect the system rather than undermine it.
   One of the negative effects of providing mining rewards is a lot of people are investing their CPU power just to mine blocks. This is essentially a waste of electricity and CPU power. Moreover, mining groups with large CPU power can perform hash attacks on the bitcoin block chain if they suddenly stop mining when the PoW burden is high.

4. Competing Chains: If there are two blockchains with divergent histories, which blockchain does Bitcoin protocol tell miners to mine on?

   Answer: Miners will mine on the longer chain which essentially has more proof of work (PoW). This ensures that so long as honest nodes control more than 50% of the total mining power, the network remains stable.

   This is the same reason for which bitcoin does not allow miners to use their mining rewards until their mined block is buried under at least 100 newer blocks. However, bitcoin design is conservative because, there is less than 1% probability of getting two divergent chains with 7 blocks or more in each.

Part 2: Develop and write a report: 25pt (5+1+2+4+2+1+5+2+3)

Please review the python source codes from lecture 2 (Building smallest blockchain.txt) and lecture 3 (PoW-2.txt). The first application shows a chain of blocks, but it does not provide any proof of work. Second code shows how to build a simple proof of work, but it does not show any block chain. Using the knowledge of these two codes please build a simple local blockchain application where there will be 3-4 users (Alice, Bob, Charlie,….). They will exchange some values (like currency) to each other. Then these transactions will be gathered and stored in the blockchain. Before storing, the miners should validate the transactions and perform proof of work to create the block and store it in the chain. You can have as much as features in your application as you wish but not less than the below:

1. Make use of merkle tree to store the transaction hash. **5**
2. Predefined transaction pools between the users.**1**
3. You need be able to demonstrate creating genesis block and then adding following blocks in the chain. **2**
4. Demonstrate that miners are solving PoW puzzle and one get selected who solves it first. **4**
5. Demonstrate and explain how integrity and verifiability are ensured. **2**
6. The block should contain atleast the Hash=H(root hash of merkle tree, prev hash, nonce, timestamp) **1**
7. Also please make sure:
   a. Clear and all step by step explanation **5**
   b. Showing the full blockchain in output as well as in report. **2**
8. Explain the benefits of using merkle tree in this context. What would happen if you would not use the merkle tree to store all the transactions' hash? **3**

You can use any platform to build this application.

Prepare a report (like the Ethereum application manuals) and explain each and every possible function, provide screen shot of your code and output.

Please submit all answers in one pdf. This assignment carries 30 marks but later your marks will be converted to out of 10.

Code with Explanation:

1. First, I import necessary libraries.

```
# Code for Python 3.7.1

import random
from hashlib import sha256
from datetime import datetime as dt
```

2. The I define a function to generate `sha40` hash. `sha40` is chosen to make the output easier to read. For this simple demonstration `sha40` will be collision free.

```
def sha40(data):  # sha40 (40 bits) is sufficient for this example
  return sha256(data.encode('utf-8')).hexdigest()[:10]
```

3. Then I create the `User` class. A `User` is the person who sends or receives money. This assignment does not require us to maintain a `User` wallet and to verify transactions to show if sender has enough money. So, `User` only has a name attribute as his/her identity.

```
class User:
  def __init__(self, name):
    self.name = name              # User identity = name

  def __str__(self):
    return self.name
```

4. Next I create the `Miner` class. A `Miner` is like a `User` in that `Miner` can also send or receive coins. In addition, `Miner` can mine blocks to show PoW. So, a `Miner` has a name and another property called `cpu`. `cpu` denotes the CPU power or processing power of the `Miner`. If a `Miner` has more `cpu` he is more likely to mine a block.

```
class Miner(User):                # Miners can also be users (send/receive coin)
  def __init__(self, name, cpu):
    assert type(cpu) == int, "Miner cpu power must be integer"
    self.cpu = cpu                # cpu power of miner (integer > 1)
      super().__init__(name)

  def __str__(self):
    return '{} (CPU Power = {})'.format(self.name, self.cpu)
```

5. Next I create the `Transaction` class. A `Transaction` has a sender (`_from`) a receiver (`to`) and the amount of coins sent (`val`). It also has a hash calculated as the `sha40` of the string created by these values (example `'Alice -> Bob: 50'`). This hash is later used to calculate the **Merkle root** of a block. The __str__ method prints the transaction. Example:

```
'Bob -> Alice: 17.22'     # Bob = sender, Alice = receiver, 17.22 = Amount of coins sent
```

```
class Transaction:
  def __init__(self, _from, to, val):
    self._from = _from              # sender
    self.to = to                    # receiver
    self.val = val                  # amount of coins sent
    self.hash = sha40(str(self))  # transaction hash needed for merkle root

  def __str__(self):                # print format = 'Sender -> Recipient: amount'
    return '{} -> {}: {:.2f}'.format(self._from.name, self.to.name, self.val)
```

6. Next I create the `Block` class. A `Block` has 1 class attribute (in contrast to instance attributes) called difficulty. A class attribute is assigned when the class is defined whereas an instance attribute is assigned when the instance is initiated. Class attributes are usually used when the attributes value is same for all instances for this class. Here, I want the mining difficulty to be same for all instances of Block class. So, I define difficulty as a class attribute. To know more: https://docs.python.org/3/tutorial/classes.html#class-objects

   Furthermore, Block class has (1) time, (2) list of transactions, (3) nonce (used to show PoW), (4) index, (5) hash of previous block, (6) Merkle root (hash of all transactions in a Merkle tree) and (7) it's own hash. Please note the hash attribute is defined as a property. A property is a special type of attribute which always calls the getter function (here hash()) to get its value.

   In addition I have a method called get_merkel_root() to recursively calculate the Merkle root of this block from the hashes of its transactions. This function takes the list of hashes of all transactions in this block as an argument. This function also generates the whole merkle tree and stores it in self.merkle_tree for verifiability in needed. The list of hashes of all transactions in this block is generated by the following line of code:

```
[t.hash for t in transactions]
```

   Finally, there is a mine() method to show proof of work (PoW) for this block. The PoW algorithm is the same as the given text file called PoW -2.txt. However, I have modified the difficulty to `0x00FFFFFFF` because I am using `sha40` not `sha256`.

   The __str__() method is called when I call the built-in str() function of Python. This function displays the block information in a very nice way in console. Example: (Note the 3 transactions)

```
Block # 2, Time: 2020-04-22 02:42:00.256769, Nonce: 5112, Miner: Wendy (CPU Power = 4)
Hash: 0008677ea7, Previous hash: 0004668974, Merkle root: 8341ab79d8,
3 Transactions: ['Bob -> Alice: 17.22', 'Bob -> Charlie: 72.00', 'Eve -> Charlie: 77.54']
```

```
class Block:
  difficulty = 0x00FFFFFFFF                 # default mining difficulty

  def __init__(self, transactions=['genesis'], prev_block=None):
    self.time = str(dt.now())
    self.transactions = transactions
    self.nonce = 0

    if transactions == ['genesis']:         # create genesis block
      self.miner, self.index = 'genesis', 0 # genesis block parameters
      self.prev_hash, self.merkle_root = '0000000000', bad4a79'
      return

    self.index = prev_block.index + 1       # this block index = previous index + 1
    self.prev_hash = prev_block.hash        # hash of previous block
    self.merkle_tree = []                   # store entire merkle tree for verifiability
    self.merkle_root = self.get_merkle_root([t.hash for t in transactions])

  @property
  def hash(self):                           # gets hash of this block
    return sha40(self.merkle_root + self.prev_hash + str(self.nonce) + self.time)

  def get_merkle_root(self, hash_list):     # merkle root of transaction hashes
    self.merkle_tree += hash_list           # update merkle tree of Block
    if len(hash_list) == 1: return hash_list[0]
    new_hash_list = []
    for i in range(0, len(hash_list)-1, 2):
        new_hash_list.append(sha40(hash_list[i] + hash_list[i+1]))
    if len(hash_list) % 2 == 1: # odd, hash last item twice
        new_hash_list.append(sha40(hash_list[-1] + hash_list[-1]))
    return self.get_merkle_root(new_hash_list)

  def mine(self, miner):                    # mine block for PoW = get Nonce
    while int(self.hash, 16) >= self.difficulty: self.nonce += 1
    self.miner = miner                      # miner who mined this block

  def __str__(self):                        # print block info in a nice manner
    return '''''Block # {}, Time: {}, Nonce: {}, Miner: {}
Hash: {}, Previous hash: {}, Merkle root: {},\n{} Transactions: {}'''\
      .format(self.index, self.time, self.nonce, str(self.miner),
      self.hash, f.prev_hash, self.merkle_root,
      len(self.transactions), [str(t) for t in self.transactions])
```

7. To simulate, first, I define some global variables. Transactions_queue is a list that holds all the transactions requested by users. Miners take transactions from this queue, add those to a block and mine that block to show PoW. Miners_cpu is a list of all miners but if a miner has cpu == 3, he/she shows up 3 times in this list. blockchain is the list of all blocks (lilnked list can also be used). Block.difficulty can be changed here if needed (this line is optional).

```
### ================================================================
### ========================== SIMULATION ==========================
### ================================================================

transactions_queue = []                 # global queue for requested transactions
miners_cpu = []                         # more cpu means more occurrence in this list
blockchain = []                         # global list for all blocks = blockchain
Block.difficulty = 0x000FFFFFFF         # mining (PoW) difficulty, smaller = more work
```

8. Next I create some Users and Miners. Although, Miners can also be Users (that is send/receive coins), I keep them separate here for easier understanding of result.

```
### =============== Create Users ===============

Alice = User('Alice')
Bob = User('Bob')
Charlie = User('Charlie')
Eve = User('Eve')

Users = [Alice, Bob, Charlie, Eve]



### =============== Create Miners ===============

Oscar = Miner('Oscar', 1)              # Oscar has cpu power = 1
Trudy = Miner('Trudy', 2)              # Trudy has cpu power = 2
Victor = Miner('Victor', 3)            # ... and so on
Wendy = Miner('Wendy', 4)

miners = [Oscar, Trudy, Victor, Wendy]
```

9. The next piece of code populates the miners_cpu list in such a way that if a miner has cpu power = 1, he/she shows up 1 time in this list, if he/she has cpu power = 2, he/she shows up 2 times in this list and so on. This is done so that miners with more cpu power are proportionately more likely to be chosen if a random choice is made from this list. This is the same algorithm used in the given PoW -2.txt file.

```
### =============== Add to miners_cpu by CPU Power ===============

for m in miners: miners_cpu += [m]*m.cpu
```

10. I have a predefined_transaction_queue to test the code (commented below). This is used to test the code. Please note predefined transactions will not result in same block hash every time because block hash has timestamp in it. Moreover, I have also generated transactions randomly and added to queue. Number of transactions are randomly selected between 1 and 15 (can be modified). Sender, receiver and amount for each transaction are also randomly selected. This shows that the code is robust.

```
## predefined_transaction_queue = [Transaction(Alice, Bob, 10), Transaction(Bob, Charlie, 20,
Transaction(Charlie, Eve, 15.72), Trasaction(Eve, Alice, 7), Transaction(Bob, Alice, 5)]

### =============== Generate Random Transactions and Print =============

num_transactions = random.randint(1, 15)      # Random number of transactions 1 to 15

for i in range(num_transactions):
  sender, receiver = random.sample(Users, 2)  # Random sender, receiver
  amount = random.uniform(0.0, 100.0)         # Random amount in [0, 100]
  t = Transaction(sender, receiver, amount)   # create transactions
  transactions_queue.insert(0, t)
```

11. The requested (randomly generated) transactions are printed to console before mining to show that they were correctly mined later.

```
print("\n{} Transactions Requested:\n(Sender -> Receiver: Amount)"
      .format(num_transactions))
for t in transactions_queue:
  print(t)
```

12. The Block class is defined with default parameters for **genesis block**. So, to create the genesis block I need to instantiate Block class with no arguments. See results below for correctness.

```
### =============== Create Genesis Block ===============

genesis = Block()
blockchain.append(genesis)                    # Include genesis block in blockchain
```

13. I do not create fixed sized blocks. Rather create random sized blocks to show that the mining algorithm (mine() function in Block class) and Merkle tree (get_merkle_root() function in Block class are robust and general purpose. In other words, the mine() and get_merkle_root() functions work for any number of transactions in the block. Each mined block is added to the end of the blockchain list to create the blockchain. Finally, I print the entire blockchain as result.

```
### =============== Mine Transactions into Blocks ===============

while transactions_queue:
  miner = random.choice(miners_cpu)           # Miner with more cpu is more probable
  remaining = len(transactions_queue)         # remaining # of transactions in queue
  n = random.randint(1, remaining)            # number of transactions in this ck

  t = []                                      # transactions in this block
  for i in range(n):
    t.append(transactions_queue.pop())

  block = Block(t, blockchain[-1])            # create block with transactions
  block.mine(miner)                           # mine block to show PoW
  blockchain.append(block)                    # include block in blockchain


### =============== Print Entire Blockchain ===============

print('\n-------------------------------------\n{} transactions mined into {} cks'
      .format(num_transactions, len(blockchain)-1))

for b in blockchain:
  print('-------------------------------------')
  print(b)
```

Result/Output: (Output will vary for each run, because block hash has timestamp)

```
15 Transactions Requested:
(Sender -> Receiver: Amount)
Bob -> Eve: 71.50
Charlie -> Bob: 81.63
Charlie -> Alice: 28.19
Bob -> Charlie: 28.45
Alice -> Bob: 30.49
Eve -> Charlie: 80.19
Eve -> Charlie: 77.54
Bob -> Charlie: 72.00
Bob -> Alice: 17.22
Alice -> Eve: 16.79
Bob -> Eve: 10.80
Bob -> Eve: 4.70
Bob -> Alice: 9.98
Eve -> Alice: 25.56
Charlie -> Bob: 34.89

-------------------------------------
15 transactions mined into 5 blocks
-------------------------------------
Block # 0, Time: 2020-04-22 02:42:00.224802, Nonce: 0, Miner: genesis
Hash: 5fb3efcd3f, Previous hash: 0000000000, Merkle root: aeebad4a79,
1 Transactions: ['genesis']
-------------------------------------
Block # 1, Time: 2020-04-22 02:42:00.224802, Nonce: 12157, Miner: Trudy (CPU Power = 2)
Hash: 0004668974, Previous hash: 5fb3efcd3f, Merkle root: 3c9c19dcc0,
6 Transactions: ['Charlie -> Bob: 34.89', 'Eve -> Alice: 25.56', 'Bob -> Alice: 9.98', 'Bob ->
Eve: 4.70', 'Bob -> Eve: 10.80', 'Alice -> Eve: 16.79']
-------------------------------------
Block # 2, Time: 2020-04-22 02:42:00.256769, Nonce: 5112, Miner: Wendy (CPU Power = 4)
Hash: 0008677ea7, Previous hash: 0004668974, Merkle root: 8341ab79d8,
3 Transactions: ['Bob -> Alice: 17.22', 'Bob -> Charlie: 72.00', 'Eve -> Charlie: 77.54']
-------------------------------------
Block # 3, Time: 2020-04-22 02:42:00.267769, Nonce: 6283, Miner: Wendy (CPU Power = 4)
Hash: 000e9a7af2, Previous hash: 0008677ea7, Merkle root: d84221641d,
1 Transactions: ['Eve -> Charlie: 80.19']
-------------------------------------
Block # 4, Time: 2020-04-22 02:42:00.281803, Nonce: 2427, Miner: Wendy (CPU Power = 4)
Hash: 0001d2dc30, Previous hash: 000e9a7af2, Merkle root: f7146e978f,
4 Transactions: ['Alice -> Bob: 30.49', 'Bob -> Charlie: 28.45', 'Charlie -> Alice: 28.19',
'Charlie -> Bob: 81.63']
-------------------------------------
Block # 5, Time: 2020-04-22 02:42:00.286803, Nonce: 4650, Miner: Victor (CPU Power = 3)
Hash: 0001a5e3fa, Previous hash: 0001d2dc30, Merkle root: dec6331b40,
1 Transactions: ['Bob -> Eve: 71.50']
```

Notes:

1. I have used Merkle tree to store the transaction hashes in a Block. The entire Merkle tree of a block is stored in the variable self.merkle_tree (see code above). The Merkle tree is generated by the function self.get_merkle_root(). This function also generates the Merkle root hash which is stored in the variable self.merkle_root and used to generate the hash of a block along with nonce, previous hash and timestamp.
2. I have used randomly predefined and randomly generated transactions to test the code.
3. I have created the genesis block and shown how to use the hash and index of previous block to create the next block.
4. The miners solve the PoW puzze using the mine() function in the Block class. The miner with more CPU power is more like to get selected because of how the mines_cpu list is created.
5. **Integrity** of each transaction is ensured because they are buried in a block. The block creates a hash of the merkle root of the transactions along with timestamp, nonce and previous block hash. This hash cannot be regenerated without doing the PoW again. As PoW is very CPU intensive this process ensures the integrity of transactions.

   Verifiability is ensured by maintain a Merkle tree attribute in the Blocks. The self.merkle_tree attribute has hashes of each transaction in it. So, if we want to verify a transaction we just have to proceed along the merkel tree to recalculate the Merkle root. If this is same as Merkle root stored in the block then we proceed the verify the blockchain after that block by recalculating the hashes of the next blocks. If all the blocks hashes are accurate the we can say that the transaction and the block both are verified.
6. A Block calculates the hash property by sha40(Merkle root, previous hash, nonce, timestamp)
7. The code and output result are given above with explanations.
8. Merkle tree allows us to verify every single transaction in this blockchain very fast. If we did not use the Merkle tree then it would be very slow to verify every single transaction. Because in that case we would have to recalculate the hashes of all transactions every single time. But since we are already storing the hashes in the self.merkle_tree variable, the process is less time consuming. We can simply request the hashes of the branches this transaction belongs to and verify it.