DOS Project Report 4.1 : Bitcoin

Team Members:

- Anurag Bagalwadi (UFID: 4936 9125)
- Fatema Saifee (UFID: 1508 1278)

Instructions

Expected Input

- numOfWallets - the number of wallets in the Bitcoin network
- numOfTransactions - the number of random Transactions to be made between wallets.

Additional Features For **Bonus** are given at the end of the file.

Sample

Input

mix run lib/bitcoin.exs <numOfWallets> <numOfTransactions>

For input numberOfNodes = 10, numberOfRequests = 20

```
'mix run lib/bitcoin.exs 10 20'
```

Output

```
Network created...
Genesis transaction created...
Genesis block created...
Blockchain created with Genesis block...
Mining in progress...
Nonce: AgJQrau2p2
Bitcoin mined successfully!
| Wallet# 2     |  12.5 BTC |
Nonce: S5NaMdF0F0
Bitcoin mined successfully!
| Wallet# 1     |  12.5 BTC |
Nonce: MKQVg5QFFT
Bitcoin mined successfully!
| Wallet# 3     |  12.5 BTC |
Nonce: cU1PVSsoKT
Bitcoin mined successfully!
| Wallet# 4      |  12.5 BTC |
Nonce: MAVB6IQTyy
Bitcoin mined successfully!
| Wallet# 5      |  12.5 BTC |
Transaction 1:
   Sending 7.5 BTC from Wallet# 5 ---> Wallet# 2
All wallet balances after transaction 1
-----------------------------
| WALLET ID      |  BALANCE  |
| Wallet# 1      |  12.5 BTC |
| Wallet# 2      |  20.0 BTC |
| Wallet# 3      |  12.5 BTC |
| Wallet# 4      |  12.5 BTC |
| Wallet# 5      |  5.0 BTC  |
-----------------------------
Transaction 2:
   Sending 7.5 BTC from Wallet# 1 ---> Wallet# 3
All wallet balances after transaction 2
-----------------------------
| WALLET ID      |  BALANCE  |
| Wallet# 1      |  5.0 BTC  |
| Wallet# 2      |  20.0 BTC |
| Wallet# 3      |  20.0 BTC |
| Wallet# 4      |  12.5 BTC |
| Wallet# 5      |  5.0 BTC  |
-----------------------------
Transaction 3:
   Sending 7.5 BTC from Wallet# 3 ---> Wallet# 5
All wallet balances after transaction 3
-----------------------------
| WALLET ID      |  BALANCE  |
| Wallet# 1      |  5.0 BTC  |
| Wallet# 2      |  20.0 BTC |
| Wallet# 3      |  12.5 BTC |
| Wallet# 4      |  12.5 BTC |
| Wallet# 5      |  12.5 BTC |
-----------------------------
Transaction 4:
   Sending 7.5 BTC from Wallet# 3 ---> Wallet# 1
All wallet balances after transaction 4
-----------------------------
```

```
| WALLET ID      |  BALANCE  |
| Wallet# 1      |  12.5 BTC |
| Wallet# 2      |  20.0 BTC |
| Wallet# 3      |  5.0 BTC  |
| Wallet# 4      |  12.5 BTC |
| Wallet# 5      |  12.5 BTC |
-----------------------------
Transaction 5:
   Sending 7.5 BTC from Wallet# 2 ---> Wallet# 5
All wallet balances after transaction 5
-----------------------------
| WALLET ID      |  BALANCE  |
| Wallet# 1      |  12.5 BTC |
| Wallet# 2      |  12.5 BTC |
| Wallet# 3      |  5.0 BTC  |
| Wallet# 4      |  12.5 BTC |
| Wallet# 5      |  20.0 BTC |
-----------------------------
Transaction 6:
   Sending 7.5 BTC from Wallet# 1 ---> Wallet# 3
All wallet balances after transaction 6
-----------------------------
| WALLET ID      |  BALANCE  |
| Wallet# 1      |  5.0 BTC  |
| Wallet# 2      |  12.5 BTC |
| Wallet# 3      |  12.5 BTC |
| Wallet# 4      |  12.5 BTC |
| Wallet# 5      |  20.0 BTC |
-----------------------------
Transaction 7:
   Sending 7.5 BTC from Wallet# 4 ---> Wallet# 5
All wallet balances after transaction 7
-----------------------------
| WALLET ID      |  BALANCE  |
| Wallet# 1      |  5.0 BTC  |
| Wallet# 2      |  12.5 BTC |
| Wallet# 3      |  12.5 BTC |
| Wallet# 4      |  5.0 BTC  |
| Wallet# 5      |  27.5 BTC |
-----------------------------
Transaction 8:
   Sending 7.5 BTC from Wallet# 3 ---> Wallet# 5
All wallet balances after transaction 8
```

```
-----------------------------
| WALLET ID       |  BALANCE  |
| Wallet# 1       |  5.0 BTC  |
| Wallet# 2       |  12.5 BTC |
| Wallet# 3       |  5.0 BTC  |
| Wallet# 4       |  5.0 BTC  |
| Wallet# 5       |  35.0 BTC |
-----------------------------
Transaction 9:
   Sending 7.5 BTC from Wallet# 5 ---> Wallet# 3
All wallet balances after transaction 9
-----------------------------
| WALLET ID       |  BALANCE  |
| Wallet# 1       |  5.0 BTC  |
| Wallet# 2       |  12.5 BTC |
| Wallet# 3       |  12.5 BTC |
| Wallet# 4       |  5.0 BTC  |
| Wallet# 5       |  27.5 BTC |
-----------------------------
Transaction 10:
   Sending 7.5 BTC from Wallet# 1 ---> Wallet# 5
Oops! Not enough Bitcoins
All wallet balances after transaction 10
-----------------------------
| WALLET ID       |  BALANCE  |
| Wallet# 1       |  5.0 BTC  |
| Wallet# 2       |  12.5 BTC |
| Wallet# 3       |  12.5 BTC |
| Wallet# 4       |  5.0 BTC  |
| Wallet# 5       |  27.5 BTC |
-----------------------------
```

Implementation

Bitcoin is a Cryptocurrency for a decentralized network of wallets.
It allows the validation of any transaction in the system made by all the wallets, eliminating the interference of any central authority like Government.

Functionalities implemented:

- Create a Network to maintain Wallets, Transactions, and the Blockchain.
- Generate <numOfWallets> Wallets, who start Mining asynchronously.
- Mining involves performing Proof Of Work, creating a Coinbase Transaction, adding it to the new Block to be added to the Blockchain in case of successful Mining.
- Transfer BTC to and from random Wallets by creating Transactions and updating the respective Wallet states.
- Update and validate the Blockchain.

The System can be configured using two input parameters:

- numOfWallets - the number of wallets in the Bitcoin network
- numOfTransactions - the number of random Transactions to be made between wallets.

When all transactions are completed, the program will exit.

Function Prototypes

1. **generateNetwork()**

   Creates a Network Process. The Network Process state structure is as follows:

   ➔ blockchain: List of block IDs - blockchain
   ➔ transactions: List of unconfirmed transactions in the transaction pool
   ➔ walletIdPublicKeysMap: Map of all wallet ids and their public keys in the network

   Example

   iex> Wallet.generateWallet()

   Output

   #PID<0.99.0> // Network ID

2. **genesisTransaction()**
   Generates the first transaction in the Bitcoin System. This transaction can be later used in creating a genesis Block.
   Example

   iex> Transaction.genesisTransaction()

   Output

   #PID<0.100.0> // Transaction ID

3. **genesisBlock(**transactionId**)**
   Creates a new block with the genesis Transaction Id received in the input. Generates merkle root for the block and then calculates the final Hash for the block proceeding to update it in its state.

   Example

   iex> Block.genesisBlock(#PID<0.100.0>)

   Output

   #PID<0.101.0>

4. **generateTransaction()**
   Generates the first transaction in the Bitcoin System. This transaction can be later used in creating a genesis Block.
   The Transaction Process will have the State structure as:
   ➔ **Flag** - unused / used, indicating whether the transaction can be used as input to another transaction or not
   ➔ **Input** []
      - Hash of the previous transaction

- Signature of the wallet initializing the transaction
- Public key of the wallet initializing the transaction

➔ **Output []**
  - value: the amount of BTC transferred in the transaction
  - Public key of the receiver wallet

➔ **Lock tim**e: Timestamp when the transaction was created

➔ **Hash**: Generating a hash by using
```
generateHash(to_string(flag) <> inputsHash <> outputsHash
<> to_string(timestamp))
```

➔ **Amount**: Sum of all the values in the Output transaction list

Example

```
iex> Transaction.generateTransaction()
```

Output

```
#PID<0.100.0> // Transaction ID
```

5. **generateBlock(**oldBlockId, transactions**)**

Creates a new block with the transactions received in the input. Updates new index and prevHash of new block using the values oldBlockId. Calculates the hash of all transactions sent in the input and also generates Merkle Root. Finally after all calculations for the current block, it performs a hash of the current block and updates its hash value in its state and finally returns the new block id.

Example

```
iex>
Block.generateBlock(#PID<0.100.0>,[#PID<0.200.0>,#PID<0.201.0>,#P
ID<0.202.0>])
```

Output

```
#PID<0.101.0>
```

6. **generateWallet()**

Generates a wallet process with the following state:

➔ **Name** - Index of the wallet generated

➔ **Public key** - Public key generated using the Elliptic Curve Digital Signature Algorithm.

➔ **Private key** - Private key generated using the Elliptic Curve Digital Signature Algorithm.

➔ **All Public keys** - List of all peer wallet's Public keys in the network

➔ **Unused Transactions** - List of all unused transactions of the wallet

➔ **All-Transactions** - List of all used and unused Transactions od the wallet

➔ **Blockchain** - List of all Blocks validated by the wallet

- ➔ **Target** - Leading number of Zeroes required by the wallet to mine
- ➔ **Network ID** - Process Id of the Bitcoin Network
- ➔ **Balance** - Current Balance of the Wallet (number of BTCs)

Example

```
iex> Wallet.generateWallet()
```

Output

```
#PID<0.109.0> // Wallet ID
```

7. **mining(**walletId**)**

Performs Bitcoin mining for the <walletId> sent in the input. Gets the latest blockchain from the network and updates the current version of the blockchain maintained by the wallet. Using the hash of the last block on the blockchain, a random **nonce** value is appended to the hash and this new string is hashed again. If we get a hash with the required target leading 0s, we have managed to mine a bitcoin successfully. In this case, we create a coinbase transaction, add it to a block and add this block to the blockchain and the new blockchain is updated in the network. If the hash does not contain the required target leading 0s, we call the same function recursively again for the same walletId.

Example

```
iex> mining(PID<0.100.0>)
```

8. **createWalletToWalletTx(**sender,receiver,amount**)**

Create a sender to receiver transaction. Main features of this transaction are:

- Get the wallet states of sender and receiver
- Select unusedTransactions from sender wallet sufficient for the amount to transfer
- Generate outputs depending on whether the total value of input transactions is greater than the <amount> or not.
- Update Network TransactionPool.
- Update sender's Wallet for UnusedTransactions
- Updatereceiver's Wallet for UnusedTransactions

The argument is as follows:

- Sender - PID of the wallet of the sender
- Receiver - PID of the wallet of the receiver
- Amount - Number of BTC to be transferred

Example

```
iex> Transaction.createWalletToWalletTx(#PID<0.100.0,
#PID<0.101.0, 20.00)
```

9. **isBlockValid(**blockId,lastBlockId**)**

   Receives the current Block Id and last Block Id

   Performs 3 basic validations:

   - Checks if current block index = previous block index +1
   - Checks if current block previous Hash value = hash value of the previous block
   - Recalculates hash of current node and verifies it against the hash value stored in it state

   Returns true if all 3 validations are successful. Else returns false
   Example

   ```
   iex> Bitcoin.isBlockValid(#PID<0.100.0>, #PID<0.101.0>)
   ```

   Output

   ```
   True
   ```

10. **calculateHash(**blockId**)**

    For the given block id received in the input, this function calculates the hash for the block. It gets the current state of the block, concatenates the index, timestamp, previous hash, Merkle root and performs a SHA256 hash on this concatenated string. This is the hash of the block and the value that is returned by the function.

    Example

    ```
    iex> Block.calculateHash(#PID<0.100.0>)
    ```

    Output

    ```
    74DD1E599E6C4EB4CE3837AD97E12C80796C78E7D2E2D034245861104668A770
    ```

11. **merkle_tree_hash(**[hashes]**)**

    While creating Blocks, each of its transaction are first hashed, and the hashes are then paired, hashed, paired again, and hashed again until a single hash remains, the merkle root of a merkle tree.
    Example

    ```
    iex> Commons.merkle_tree_hash([
    BF5D3AFFB73EFD2EC6C36AD3112DD933EFED63C4E1CBFFCFA88E2759C144F2D8,
    994EA33D94058425C90DDC4EFE6776AC692E91361E388C98134F0D0FC2A012D8,
    2DA088F23B65EAC349084E13BEC32E82804801B5C4C3B6BAEDB8811EA517334D
    ])
    ```

    Output

    ```
    07F7D10A67201FA91CED24C27EE59357BD183ED9BA1EB9676D42236D56E15884
    // Merkle Root
    ```

Testing

For testing the features of Bitcoin System we use Elixir’s built-in test framework called **ExUnit**.

Instructions

Input

```
iex> mix test
```

Output

```
Testing Bitcoin...
Nonce: iMh6Jci9vX
Bitcoin mined successfully!
| Wallet# 1      |  12.5 BTC |
Nonce: AyVluLUMMq
Bitcoin mined successfully!
| Wallet# 2      |  12.5 BTC |
Creating wallet to wallet transaction
   Sending 10 BTC from Wallet# 1 ---> Wallet# 2
senderBalanceBefore - amount = senderBalanceAfter?
true
receiverBalanceBefore + amount = receiverBalanceAfter?
true
.Testing Bitcoin...
Nonce: 5gsn3QgF0L
Bitcoin mined successfully!
| Wallet# 1      |  12.5 BTC |
Nonce: AyVluLUMMq
Bitcoin mined successfully!
| Wallet# 2      |  12.5 BTC |


C19BD9C5BC47F08DAF9B28A9AAE107A12710FB1C5A8EC1EBB2ACF4BC90BB2E0
6
C19BD9C5BC47F08DAF9B28A9AAE107A12710FB1C5A8EC1EBB2ACF4BC90BB2E0
6
hash of previous block == previous hash of current block?
.Testing Bitcoin...
```

```
Nonce: BvNyozOAsp
Bitcoin mined successfully!
| Wallet# 1      |  12.5 BTC |
Nonce: AyVluLUMMq
Bitcoin mined successfully!
| Wallet# 2      |  12.5 BTC |


Checking balance of wallet 1 after mining is greater than 0
true
Checking balance of wallet 2 after mining is greater than 0
true
.
Is hash length 64?
true
..
Is network alive?
true
..

Finished in 15.5 seconds
7 tests, 0 failures

Randomized with seed 638000
```

Test cases

BitcoinTest

1. test "Mining"

    "Checking balance of wallet 1 after mining is greater than 0"

    ```
    assert senderBalanceBefore > 0
    ```

    "Checking balance of wallet 2 after mining is greater than 0"

    ```
    assert receiverBalanceBefore > 0
    ```

2. test "Wallet to Wallet transaction"

    "senderBalanceBefore - amount = senderBalanceAfter?"

    ```
    assert 12.5 - amount == senderBalanceAfter
    ```

    "receiverBalanceBefore + amount = receiverBalanceAfter?"

    ```
    assert 12.5 + amount == receiverBalanceAfter
    ```

3. test "validate block added to blockhain"

    "hash of previous block == previous hash of current block?"

    ```
    assert prevHash == hash
    ```

CommonsTest

1. test "Hash Length Valid"

    “Length of hash value returned == 64?”

    ```
    assert String.length(Commons.generateHash("a")) == 64
    ```

2. test "Merkle Root Valid"

    “When input hash list length == 1”

    ```
    assert Commons.merkle_tree_hash([Commons.generateHash("a")]) ==
    Commons.generateHash("a")
    ```

    “When input hash list length == even”

assert Commons.merkle_tree_hash([Commons.generateHash("a") , Commons.generateHash("b")]) == Commons.generateHash(Commons.generateHash("a")<>Commons.generateHash("b"))

"When input hash list length == odd"

assert Commons.merkle_tree_hash([Commons.generateHash("a") , Commons.generateHash("b"),Commons.generateHash("c")]) == Commons.generateHash(Commons.generateHash(Commons.generateHash("a")<>Commons.generateHash("b"))<>Commons.generateHash(Commons.generateHash("c")<>Commons.generateHash("c")))

GenerateRandomStringTest

1. test "Random string nonce Length Valid"

   "Length of random nonce string generated is valid"

   assert String.length(GenerateRandomString.randomizer(5)) == 5

NetworkTest

1. test "Network is Alive"

   "Network exists and network id is not nil"

   assert context[:networkId] != nil

<u>Bonus Features</u>

1. **Genesis Transaction**
   Generates the first transaction in the Bitcoin System. This transaction is later used in creating a genesis Block.
   The genesis transaction does not contain any input transactions.
   It contains one output field whose value is 50 BTC and public key to which this is sent is “`1A1zP1eP5QGefi2DMPTfTL5SLmv7DivfNa`”. It is said to be the public key address of creator of Bitcoin, Satoshi Nakamoto. It is never used for any further transactions.

2. **Genesis Block**
   Genesis Block is generated which contains the Genesis Transaction and its Merkle root. This Block is used to initialize the Blockchain in the Network.

3. **Block validation while adding to blockchain**
   Block is validated before adding it to the blockchain by the wallet by checking the following
   - The Index of the new Block = 1 + Index of the last block of Blockchain
   - Hash of the last block of Blockchain == PrevHash of the new Block
   - Recalculate the Hash of the new block and check if it matches.

4. **Merkle root generation**
   While creating Blocks, each of its transaction are first hashed, and the hashes are then paired, hashed, paired again, and hashed again until a single hash remains, the Merkle root of a Merkle tree.

5. **Wallet Balance calculation**
   Wallet Balance is calculated by adding all the amounts in the input List of the Unused Transactions in the wallet, whose public keys match with the Wallet’s Public Key.

6. **Transaction signature generation**
   Transaction Signature is generated with the help of Private key of the wallet initiating the transaction and message describing the transaction. This was done using **Elliptic Curve Digital Signature Algorithm**.