Visweswar Sirish Parupudi (2020AAPS0330H)
Konkala Rithvik (2020A8PS0517H)
Gautham Gutta (2020AAPS2204H)
Gokul Pradeep (2018B5A70785H)

THIS FILE WILL GIVE A BRIEF BUT EXHAUSTIVE DESCRIPTION OF THE IMPLEMENTED CODE
IMPORTANT NOTE:- PLEASE RUN THE GIVEN .ipynb FILE ON Google Colab OR ON Jupyter Notebook ONLINE

1) Libraries imported:-

- hashlib:- Contains all the functions and algorithms essential for calculating the hash of a transaction
- Merkletools:- contains the tools and functions for creating merkle trees
- Json :- helps us work with json data
- Time:- returns real time, helps us with functions which involve real-time
- Datetime:- contains functions which help us declare/define date objects, and also contains functions which return real time as a value.
- Random:- Contains a plethora of functions which help with random number generation
- Numpy:- The library which contains all the mathematical functions which will help us simplify operations on arrays

2) Defining a block:-

A block is supposed to contain the timestamp, merkle root, hash of the previous block, etc. Here, we intend to give the block all those “traits” by defining our block as a class, and implementing all of its features, by defining them as objects in our class

Here, since the class doesn’t have its own object yet, we declare a dummy object named ‘self’, in order to define the values of individual “traits”

We’ve declared a function `__init__()` in which we initialize the index of the block, the past transactions , hash of current block, hash of the previous block, merkle root value, and timestamp.

We’ve declared a function `display_block(self)`
in which we print out the values of aforementioned traits

Last but not least, we define a function `compute_hash(self)`
which computes the hash of the specified block

We use the json.dumps() function to return the block as a json string, and then we calculate the hash by passing this string into the sha256() function, and we also use the hexdigest() function, which converts the encoded hash value into a hexadecimal digest of fixed length in conjunction with the sha256()

3) Defining our blockchain:-
Here, we've declared our blockchain as a class, and defined its major attributes as objects

We initialize a list chainb[], which essentially is an array which contains the blockchain.

On this system, we receive a lot of transactions. However, we can only add the verified ones to our blockchain. So, we will need to store all those transactions at one place, until they're verified and ready to be added into the blockchain. Hence, we define an array, which will store the unconfirmed transactions. And, when a new transaction comes up, we append it to this array itself

We define a function `create_genesis_block(self):`
in which we define our genesis block and append it to the blockchain list variable

We also make our merkle tree, and try to implement its properties. We calculate the root value by using inbuilt functions of the merkletools library

We add a block to the blockchain, only if the hash of the previous block stored in the new block, actually matches with the hash of the previous block in the tree, hence we insert an if condition to check for this
We define a value NewBlock, which calls the object of the 'Block' class, and give it its input values, which are index, stored transactions, timestamp, previous hash and hash root value

We also define a function called `transactions()` which stores and returns the attributes like buyer name, seller name, property which the user intends to buy/sell, amount which the user would transfer, or basically the price of that property, and the timestamp of that transaction

We define a function mine(), with which we keep adding elements onto the blockchain

This way, we construct our blockchain and we print it as output

4) Defining our participant class:-

Here in this land management system, a participant is basically an end user who uses this system for buying or selling land
So, the user would have some details, like name, property which he/she intends to buy/sell, a specific id number for that user, amount which the user would transfer, or basically the price of that property.

Here, we declare a variable ‘coinage’ which is defined as the product of number of unused coins(declared a variable ‘StackedCoin’), multiplied with the amount of time for which they’ve been in the wallet

So, the PoS algorithm works as such. Various nodes have varying amounts of ‘Stake’ in the Blockchain system. The user with the highest stake has the highest probability of being able to add a block to the blockchain, with the probability being proportional to its stake.

Hence, here, we implement an algorithm, by defining a function
`minerSelection(participants):`
which selects a node which would be declared as the ‘Chosen node’, which has the capability of adding blocks to the blockchain. This node is selected randomly, but the node with higher stake has a higher probability of being selected, in this algorithm

5) The algorithm for inputting participants to make transactions and mine blocks

This section of code is highly menu driven. We let the user decide whether to add a block or not, depending on the user input. If they choose ‘1’, we ask for the required details, and then we append the node into the ‘participants’ list

We define a function named ‘pos’ in which the chosen node mines blocks

6) The algorithm asks for buyer and seller input and the amount of transaction which it then proceeds to add on block

Here, after we decide the miner node by invoking the ‘CoinAgeSelection()’ function, we proceed to ask for user details, such as buyer, seller, amount, property.

We verify for the validity of the property, by inserting an if condition, which cross checks the entered property, with given list of properties
We verify for the validity of the buyer, by inserting an if condition, which cross checks the entered buyer name, with given list of buyers

We change the amount on the buyers’ and sellers’ accounts, by subtracting(for buyer) the value of property from the present amount, and we add that amount, in case of the seller

In case the amount of transactions exceeds 6, we break out of the loop, display a message asking the user to stop, and print the blockchain up until that point

This function is terminated if

1) Buyer is invalid
2) Seller is invalid
3) Buyer has inadequate funds

Towards the end, we can input a property, and see its history of transactions, and check properties owned by each participants