**Reading from the blockchain**

This assignment has two parts. In the first, you will use the [get\_block()](https://web3py.readthedocs.io/en/stable/web3.eth.html" \l "web3.eth.Eth.get_block) function, and the [get\_transaction()](https://web3py.readthedocs.io/en/stable/web3.eth.html" \l "web3.eth.Eth.get_transaction) function to get data about blocks on the Ethereum blockchain

In order to use these functions, you will need to connect to an Ethereum node with an open RPC interface. We recommend using one of the free providers you signed up for in Module 1.

On Ethereum, as in almost all blockchains, each block is produced by single block producer, and that block producer has full authority to pack transactions into a block in any way they choose.

Historically, many block builders used a simple, greedy procedure in order to decide which transactions would eventually make it into the block.

Before EIP-1559, the common Ethereum clients like geth used the "[greedy](https://github.com/ethereum/go-ethereum/blob/release/1.8/core/types/transaction.go#L335)" algorithm to include transactions in [decreasing order of gasPrice](https://github.com/ethereum/go-ethereum/blob/release/1.8/miner/worker.go#L744). Now, it’s still a possible setting in block builders. The Flashbots builder has [three different block building strategies](https://github.com/flashbots/builder#usage), "mev-geth", “greedy” and "[greedy buckets](https://collective.flashbots.net/t/new-block-building-algorithms-for-flashbots-builder/2049)".

Just because it’s the default setting, does no mean that all miners follow this rule, however. We can’t peek into a miner’s brain to figure out what rule they’re using, but we can look at a block and see if the transactions were packed greedily, i.e., are the transactions ordered in decreasing order of gasPrice.

For this part of the assignment you’re going to use the Python web3 library to grab blocks from the Ethereum blockchain, and see if the transaction in those blocks are ordered in decreasing order of fee. If they’re not, that will indicate that the miner ordered the transactions according to some other mechanism (e.g. out of band payments).

In the second part you will be calling “read” functions on the “MerkleValidator” contract (found on the BNB testnet) to get the current state of the contract from the blockchain. You will accomplish this using the Python Web3 library, specifically the [contract functions](https://web3py.readthedocs.io/en/stable/web3.contract.html#contract-functions) section.

**Assignment**

The first two incomplete functions **connect\_to\_eth()** and **connect\_with\_middleware()** are exactly the same as what you submitted in week 1 and you should re-use that code here.

**Part 1: Getting priority fees from transactions (on Ethereum mainnet)**

Next you will need to write a function called **is\_ordered\_block(block\_num)** that takes a block number and returns a boolean value indicating whether *all* the transactions in the block were ordered in decreasing order of fee.

We have provided a template [reading\_the\_chain.py]. You should finish the function is\_ordered\_block(block\_num)

Your solution will have to call [get\_block()](https://web3py.readthedocs.io/en/stable/web3.eth.html" \l "web3.eth.Eth.get_block)to get all the transactions in the block, and then [get\_transaction()](https://web3py.readthedocs.io/en/stable/web3.eth.html" \l "web3.eth.Eth.get_transaction) for each transaction in the block.

**EIP1559**

EIP-1559 which, was introduced as part of the [London Hard Fork](https://ethereum.org/en/history/#london), changed the fee mechanism significantly. Before EIP-1559, a “greedy” algorithm would order transactions by the “gasPrice” field.

[After EIP-1559](https://docs.alchemy.com/docs/maxpriorityfeepergas-vs-maxfeepergas), however, there is now a new type of transaction (called a type 2 transaction), which includes the fields [maxPriorityFeePerGas and maxFeePerGas](https://web3py.readthedocs.io/en/stable/examples.html" \l "looking-up-transactions).

[It’s possible for a transaction to set both the gasPrice and maxPriorityFeePerGas](https://twitter.com/timbeiko/status/1413536260962938885). In this case, this is a type 2 transaction, and you should use maxPriorityFeePerGas and ignore gasPrice.

The **priority** fee paid on type 2 transactions is:

min( maxPriorityFeePerGas, maxFeePerGas-baseFeePerGas )

The **total** fee paid on type 2 transactions is

min( maxPriorityFeePerGas + baseFeePerGas, maxFeePerGas )

Many post EIP-1559 blocks contain a gasPrice field.  
In this case, the gasPrice field is set to be the **total** gas paid by the sender, i.e.,

min( maxPriorityFeePerGas + baseFeePerGas, maxFeePerGas )

Thus if you compare a type 2 transaction (that does not have gasPrice specified) to a type 0 transaction, you need to get the baseFeePerGas (this is included at the block level, not the transaction level).

If all the transactions in the block have gasPrice specified, then it’s safe to compare only gasPrice, because all transactions pay the same baseFeePerGas.

**Part 2: Reading from a contract (on BNB testnet)**

The final function you have to complete is **get\_contract\_values()** where you will read from the MerkleValidator contract that we have deployed on the BNB testnet.

This is the same contract you connected to in Module 1. We will interact with this contract in more detail in the “Merkle Validator” assignment in Module 4.

For this assignment, you don’t need to know much about how this contract works, but if you’re interested, here’s the brief overviow.  
At a high level, this contract contains the root for a Merkle Tree, whose leaves are the first 8192 primes, (2,3,5,7,…).  
The contract allows anyone to “claim” a prime, by providing a Merkle Proof of inclusion for that prime.  
When someone claims a prime, the address that claimed the prime as the “owner” of that prime.  
The contract also has various “administrators” that can change the state of the contract, and these administrative functions  
are managed by OpenZeppelin’s excellent [role-based access control](https://docs.openzeppelin.com/contracts/5.x/access-control) system.

For this assignment, you will need to check three things on the contract:

1. "merkleRoot": This function takes no arguments and returns the root of the Merkle tree held by the contract. This response should be returned in the variable “onchain\_root”
2. “hasRole” and "DEFAULT\_ADMIN\_ROLE": The MerkleValidator contract uses [OpenZeppelin’s role-based access control](https://docs.openzeppelin.com/contracts/5.x/access-control), which allows you to give certain roles to specific addresses. Use the “DEFAULT\_ADMIN\_ROLE” function to find the key associated with the default admin role, then use the hasRole function to determine whether the address stored in the argument “admin\_address” has been assigned this role. Return the result in the variable “has\_role”
3. "getPrimeByOwner": call this function with input with the “owner\_address” provided to you as a function argument, and return the result in the variable “prime”

You can see the available contract functions and recent transactions with this contract on the block explorer at <https://testnet.bscscan.com/address/0xaA7CAaDA823300D18D3c43f65569a47e78220073>

Mark as CompletedBack to dashboard