Current layout: 1 Panel with tree

**1. Mining Blocks**

*1. Mining Blocks*

In this assignment, you will mint blocks for a notional blockchain by performing a simplified version of Proof of Work (PoW). You will write a brute-force algorithm to find a valid nonce that creates a block hash which, when converted to binary form, has a number of Least Significant Bits (LSB), i.e. trailing zeros, equal to or greater than the difficulty level set for the block.

If you’re generating nonces at random, and checking how many trailing zeros are in the resulting hash, you should expect, on average that it requires 2k nonces before you find one that produces a hash with k trailing zeros. Because the running time of your program will scale *exponentially* in k, our autograder will only query your function for k<20 (at higher values of k Codio will timeout). In the context of a real blockchain, the “difficulty” is *much* higher. Bitcoin’s initial difficulty required 32 leading zeros, and the [current difficulty](https://www.coinwarz.com/mining/bitcoin/difficulty-chart) requires over 70 leading zeros.

We will be using SHA256 hashes for our simple blockchain. You can compute SHA256 hashes using hashlib, which works like this.

import hashlib

word="Bitcoin"

hashlib.sha256(word.encode('utf-8')).hexdigest()

The function “hashlib.sha256” expects *bytes*, so we use the “encode” function to convert our string to bytes, before passing it to the function.

“hexdigest” gives you the hexadecimal output of the hash function.

Since SHA can process files of arbitrary length, it is sometimes useful to calculate the hash in smaller chunks. This is important when you’re hashing large files, but not necessary when the input to the hash fits easily in memory. The code below calculates the same hash one character at a time.

import hashlib

str = "Bitcoin"

m = hashlib.sha256()

for c in str:

m.update(c.encode('utf-8'))

m.hexdigest()

Using the template “findBlockNonce.py” write a function called "mine\_block()" that takes as input, k, where k is an integer representing the PoW difficulty, *prev\_hash*, where *prev\_hash* is the hash in bytes of the previous block, and *rand\_lines*, where *rand\_lines* is a list of strings representing the transactions in the block you are trying to mine. The function “mine\_block” should return one variable, *nonce*, such that that the SHA256(prev\_hash + rand\_lines + nonce) has zeros for the k LSBs.

The block hash should have zeros for the final k bits, not the final k hexadecimal digits.

Our transactions will be lines (as strings) from the bitcoin and ethereum whitepapers. A copy of the whitepapers in .pdf has been included for you to read while you wait for your blocks to be mined. A valid block hash in our simplified chain is the combined hash of the prev\_block hash followed by all the “transactions” in rand\_lines in-order followed by the nonce value that results in desired final hash (prev\_block, rand\_lines[0], rand\_lines[1], …, rand\_lines[n], nonce).

The return variable, nonce, should be encoded as *bytes*. You can either generate your nonces in bytes (the format that hashlib.sha256 expects) or, to encode a string as bytes in python:

str = "Hello World"

byte\_str = str.encode('utf-8')

For your testing we have provided you with the function “get\_random\_lines” to generate the input to your mine\_block(), but the grader will provide its own list of strings, prev\_hash, and values of k when it calls your mine\_block()

You can get feedback on your code by clicking “Run Test Cases”

When you are happy with it, click “Education->Mark as Completed” to submit.

Once you mark the unit as completed, you cannot re-submit, so make sure you’re happy with everything before you submit!

It is worth noting that our simplified PoW blocks have several differences from “real” PoW blocks. For example, we are looking at the LSBs where typically the MSBs are what is considered. Also, we aren’t including many of the block attributes that would “normally” be included in the block hash like gas fees and timestamps. There are resources available online that discuss what actually goes into PoW block hashes for different chains and we encourage you to read some of them if you want to know more about what goes on “under the hood” in building a reliable, fraud resistant blockchain.

Next