Build a Blockchain! - Blockchain Design Exercises

HW1 - Build a Blockchain in Python

Getting Started

CS 198-077 Spring 2020 HW1-Build a Blockchain in Python

Created by Omkar Shanbhag, Updated by Simon Guo

This notebook is meant to be a short introduction to Blockchain implementations, aimed at helping us take the topics we learn about in fundamentals, and seeing how they translate to code.

In this notebook we will implement various different aspects of Blockchain technology that we understand including:

- The Blockchain Data Structure using OOP
- A proof of work simulation
- Understanding of the concept of difficulty of finding the next block hash
- A simulation of multiple miners with varying computational powers
- A bit of data analytics to see if what we've implemented makes sense

In [2]:
```
#Import statements
import hashlib as hasher
import random as rand
import time
import datetime as date
import ipyparallel as ipp
import numpy as np
import matplotlib.pyplot as plt
```

Blockchain Data Structure Design

In this section we will define the class "Block" and create an init function that creates a new block given some parameters, as well as a function hash_block, that computes the hash of this block based on its class variables.

The init function is implemented for you below and creates a block with the following specifications

- Index --> the index of the block on the chain (zero indexed)
- Timestamp --> time that the block was added on to the chain
- data --> The data the block contains (Usually points to root of merkel tree, but we can fill it with whatever for this)
- previous_hash --> The hash value of the previous block
- hash --> hash of this block computed using the hash_block function
- nonce --> the variable value that we change to alter the hash output (Default value = 0, irrelevant in this section)

QUESTION #1: Implement hash_block()

We need to implement the hash_block() which computes the hash value of this block based on how we initialize it. This function takes in no parameters.

Desired Functionality:

- Concatenate string representations of all the class variables
- Computes the SHA256 hash of this concatenation

-- HINTS --

- use the first import statement!!!
- initialize a new hasher
- Look into the following words
    - utf-8 encoding
    - .update() function
    - .hexdigest() function
    - In order for the test function below to work properly, create the string concatenation in the following order with no spaces or any other characters in between
        - index
        - timestamp
        - data
        - previous block hash
        - nonce
            - If you want to do it another order that incorporates everything thats totally fine, but test function wont work!

Type *Markdown* and LaTeX: α^2

In [31]:

```
```

Out[31]:

```
b'\xce\x06\t/\xb9H\xd9\xff\xac}\x1a7n@K&\xb7W[\xcc\x11\xee\x05\xa4a_\xefO\xec:0\x8b'
```

In [43]:

```
s = "kkko"
s.encode("utf-8")
```

Out[43]:

```
'kkko'
```

In [48]:

```
hasher?
```

In [53]:

```
class Block:
    def __init__(self, index, timestamp, data, previous_hash, nonce=0):
        self.index = index
        self.timestamp = timestamp
        self.data = data
        self.nonce = nonce #set to zero as default not applicable in first section
        self.previous_hash = previous_hash
        self.hash = self.hash_block()

    def hash_block(self):
        s = str(self.index)+str(self.timestamp)+str(self.data)+str(self.previous_hash)+str(self.nonce)
        enc = s.encode("utf-8")
        hsh = hasher.sha256()
        hsh.update(enc)
        return hsh.hexdigest()
```

TEST YOUR QUESTION 1 CODE HERE

Run the block of code below

In [54]:
```
def test_question_1(index, time, data, previous_hash):
    new_block = Block(index, time, data, previous_hash)
    check_string = '2def27922fc1c67254a9cdb0c660b91abf9b135ad38fc13c7c77007448b824a0'
    print_statement = "PASSED!!! Move on to next Question" if str(new_block.hash) == check_string else "FAIL
    print(print_statement)

time = '2019-10-17 00:37:35.256774'
data = 'Machine Learning Blockchain AI'
previous_hash = '6ffd1464f68ef4aeb385d399244efa19293ba5c842c464a82c02f8256ef71428'
index = 0

test_question_1(index, time, data, previous_hash)
```

```
PASSED!!! Move on to next Question
```

Functions for Building the Chain

Now that we have our class Block completed, lets figure out how to make a chain out of them. For a chain, we need to first have a function that creates a genesis block, which serves as the first block of our chain, and then create the function next_block() which builds a new block on top of a given block.

create_genesis_block() has been implemented for you. It creates a block with the following specifications.

- index = 0
- timestamp = Now (whenever the function is being called)
- data = "Genesis Block"
- previous_hash = "0"

QUESTION #2: Implement next_block()

We need to implement the function next_block() which takes in 1 parameter:

- Last_block = an instance of class Block that is the block that we're building our next block on top of
- nonce = Dont do anything with this right now - just pass it in to the Block that you create using the default

We want to implement the function so that it returns a new instance of the class Block with the following specifications

- index = index of last_block + 1
- timestamp = Now (whenever the function is being called)
- data = "Hey! I'm block {index}" (for example block w/ index 5 would have data: "Hey! I'm block 5")

- previous_hash = hash of last_block

In [71]:
```
ates the first block with current time and generic data
create_genesis_block():
# Manually construct a block with
# index zero and arbitrary previous hash
return Block(0, date.datetime.now(), "Genesis Block", "0")

ction that creates the next block, given the last block on the chain you want to mine on
next_block(last_block, nonce=0):
#Your code for QUESTION 2 here

return Block(last_block.index+1, date.datetime.now(), "Hey! I'm block {}".format(last_block.index+1), last_b
```

TEST YOUR QUESTION 2 CODE HERE

Run the block of code below

In [73]:
```
def test_question_2(genesis_block):
    block_1 = next_block(genesis_block)
    if block_1.index == 1 and block_1.data == "Hey! I'm block 1" and block_1.previous_hash == genesis_block.
        print("PASSED!!! Move on to next part" )
    else:
        print("FAILED!!! Try again :(")


genesis_block = create_genesis_block()
test_question_2(genesis_block)
```

```
PASSED!!! Move on to next part
```

Spinning up a Chain

Now that we've created the data structure as well as the functions needed to create the chain, lets see how spinning up an actual instance of this would work.

Below we initialize three different variables

Below we initialize three different variables

- blockchain - this is a python list which we initialize with one block inside (the genesis block)
- previous_block - this points to our genesis block (since it references the first element in blockchain)
- num_blocks - this specifies the number of additional blocks we want to add to our chain

QUESTION 3 complete_chain()

We want to complete the implementation of the function complete_chain(). This function takes in three inputs, which correspond to the initializations that we made. It returns nothing, however by the time we are done running it, the list 'blockchain' that we initialized earlier has been turned into an array of length num_blocks + 1 in which each element is an instance of class Block and each element's self.previous_hash == the previous element's self.hash. Therefore we have created our own mini blockchain!!

The for loop and the print statements of complete_chain have been implemented for you, you need to add the statements that ***create a new block on top of previous_block, add it to the block chain, and edit previous block so that the for loop can continue correctly***

***HINT *** --> Literally just do all the things listed in the section immediately above in order

If it works out, you should get the same number of print statements as num_blocks

In [75]:

```python
# Create the blockchain and add the genesis block
blockchain = [create_genesis_block()]

#Create our initial reference to previous block which points to the genesis block
previous_block = blockchain[0]

# How many blocks should we add to the chain after the genesis block
num_blocks = 20

def complete_chain(num_blocks, blockchain, previous_block):
    # Add blocks to the chain
    for i in range(0, num_blocks):
        #Your code for QUESTION 3 Here
        block_to_add = next_block(previous_block)
        blockchain += [block_to_add]
        previous_block = block_to_add

        #Your code for QUESTION 3 ends Here
        # Tell everyone about it!
        print("Block #{} has been added to the blockchain!".format(block_to_add.index))
        print("Hash: {}\n".format(block_to_add.hash))

complete_chain(num_blocks, blockchain, previous_block)
```

```
Block #1 has been added to the blockchain!
Hash: 70363de0b1396bc1a711598dce80f5f76b8acf82a56df63aac7719e07a285647

Block #2 has been added to the blockchain!
Hash: d13553fa89fea7b25a480a52497ee437b2c75e11f8939cff439f2a3fda02c9ad

Block #3 has been added to the blockchain!
Hash: a62425f73e1aa56bcc9fdaed64f385858efeb8f7a633fe32379d73d57449d6ba

Block #4 has been added to the blockchain!
Hash: f6ee8a9cb6eb4d2d3c82c6ecc54f3d56c40b1cd9f76e50334bf6dc09b2505272

Block #5 has been added to the blockchain!
Hash: f9a78f2c882b279de47d2961d9a3fb6d153294dbcfed4d8c22819e348d587bd1

Block #6 has been added to the blockchain!
Hash: 8e4b7c733addb68719e4e6d89d0c182e2f27fffebd3cbf2ead7a66d8d8b0eaa4
```

```
Block #7 has been added to the blockchain!
Hash: dc47e751988548dccb9778556bd3f69bbd7f4d11ebd3b21541ff90f1022c631d

Block #8 has been added to the blockchain!
Hash: 5947fceb60d87ec9235c5634237d843cde5464d0c6fe9cba21dd6b51b6058981

Block #9 has been added to the blockchain!
Hash: 947c86d8d64d4774c1b90d6907b9be045b13c47d4cbfc22efc0b3955a9647540

Block #10 has been added to the blockchain!
Hash: 1689c889ab552be9e14b4cf7e1a4db380c26d6cd1a5d0116828d3e630cdf58d5

Block #11 has been added to the blockchain!
Hash: 0e334e71651efe246e0a8469569b7ec73de75c61a5abf75b07db7ecb0974c33b

Block #12 has been added to the blockchain!
Hash: 98232de13fb4ed0f840ccf1e81560f075366130a01e29df13240a50021bc80a4

Block #13 has been added to the blockchain!
Hash: c744389f47d59997d4d3da72e031ae3390869c86ebc843b8a10a90b932111f46

Block #14 has been added to the blockchain!
Hash: 0488252c5ddbe64d3eba6e2d8e9e0c11f9add9b88ca031f99f101765185dfd93

Block #15 has been added to the blockchain!
Hash: 46c36021632cbeb7c89db8871fb1ed5d44cfca42b06f173b416c583bbd0d5a0e

Block #16 has been added to the blockchain!
Hash: 7c140f4abefc95ec5cd83b67cbcd08063993bac3aee490aa9fc25757c7e48d63

Block #17 has been added to the blockchain!
Hash: 7b00d16980d287c466285da26cf0f6886823fd2c13e8521f3978062b1698de63

Block #18 has been added to the blockchain!
Hash: dab5ec7d2b904a54f06abbf23a0f367973d82557c407df4522123ae9dae85cc1

Block #19 has been added to the blockchain!
Hash: 6a53773227c1fa0d0652ec7406ad897e667be52a1771b03d5f2bdfa9ac7aaeac

Block #20 has been added to the blockchain!
Hash: 7a6822617f2d3ac6ad586b4f2eaa7cfd0585c3c7f6fb4930f21e462ca71a4f5f
```

TEST YOUR QUESTION 3 CODE HERE

Run the block of code below

In [76]:
```
def test_question_3(blockchain, num_blocks):
    correct = True
    if len(blockchain) != num_blocks + 1:
        correct = False
    for i in range(len(blockchain)-1):
        if blockchain[i + 1].previous_hash != blockchain[i].hash:
            correct = False
            break
    print_statement = "PASSED!!! Move on to the next Part" if correct else "FAILED!!! Try Again :("
    print(print_statement)

test_question_3(blockchain, num_blocks)
```

```
PASSED!!! Move on to the next Part
```

Proof of Work Simulation

In this section we will be doing a simulation of the proof-of-work consensus mechanism that the Bitcoin Blockchain (among others) uses. Let us define some of the concepts that we will be dealing with in this section.

The Nonce --> Randomly generated value that we add to our concatenation of our block to add variance to our hashes

Difficulty --> Specified by the network (in theory, here it is specified by us). Defines the number of hashes that are valid out of all possible values. Higher difficulty indicates a lower number of valid hashes.

QUESTION 4: generate_nonce(), generate_difficulty_bound(), find_next_block()

Part 1 - Description

The functions **generate_nonce()** and **generate_difficulty_bound()** have been implemented below for you. The first part of this question involves reading through, them understanding them completely, and writing a quick 1 - 2 line summary of what they are supposed to do and how they are implemented (write it in the space given, 2 cells below).

Part 2 - Implementation

The second part of this question is to complete the implementation of find_next_block(). This function's purpose is to try different blocks with the same data, index etc. but different nonces that satisfy the difficulty metric specified. The difficulty bound has already been generated for you and the first block has been created. You have to complete the implementation so that:

- You find a nonce such that the hash of the block is less than the difficulty bound.
- All data from new block has to be the same (including the timestamp, for simplicity purposes) and the only thing changing is the nonce
- For our learning purposes, increment the hashes_tried parameter appropriately everytime you try a hash so that we can see the number of hashes tried and how this correlates to our difficulty value.

HINTS

- How do you turn a hash string into its hex prepresentation that you can compare???
- if your implementation is working, you should notice something interesting about all the hashes of the blocks that you are getting when you run the code cell labeled "Create Proof of work Blockchain"**

you can test question 4 in the a few cells below this one, after running all the cells in between

In [108]:
```
import time

def generate_nonce(length=20):
    return ''.join([str(rand.randint(0, 9)) for i in range(length)])

def generate_difficulty_bound(difficulty=1):
    diff_str = ""
    for i in range(difficulty):
        diff_str += '0'
    for i in range(64 - difficulty):
        diff_str += 'F'
    diff_str = "0x" + diff_str  # "0x" needs to be added at the front to specify that it is a hex representa
    return(int(diff_str, 16))  # Specifies that we want to create an integer of base 16 (as opposed to the d

#Given a previous block and a difficulty metric, finds a nonce that results in a lower hash value
def find_next_block(last_block, difficulty, nonce_length):
    difficulty_bound = generate_difficulty_bound(difficulty)
    start = time.process_time()
    new_block = next_block(last_block)
    hashes_tried = 1
    #Your code for QUESTION 4 Starts here
    while (int(new_block.hash, 16) >= difficulty_bound):
        hashes_tried += 1
        new_nonce = generate_nonce(nonce_length)
        new_block.nonce = new_nonce
        new_block.hash = new_block.hash_block()

    #Your code for QUESTION 4 Ends here
    time_taken = time.process_time() - start
    return(time_taken, hashes_tried, new_block)
```

QUESTION 4 Description Section

Describe the following functions:

- generate_nonce(): generates a string length of 20 with random numbers 0-9.
- generate_difficulty_bound(): basically creates a 64 hex bit number with respect to difficulty which means the lower the difficulty the higher the bound will be so it will be less strict when we compare!

The cell below creates our proof of work blockchain in a similar way that we do in the earlier section.

Some initializations:

- blockchain_pow: Our new python list that signifies our proof of work blockchain, with the genesis block inside
- previous_block: The first block to use as previous block to build upon
- num_blocks: number of additional blocks to add to teh chain
- difficulty: difficulty of the network
- nonce_length: length of the randomly generated nonce

Create Proof of Work Blockchain

In [109]:

```
# Create the blockchain and add the genesis block
blockchain_pow = [create_genesis_block()]

#Create our initial reference to previous block which points to the genesis block
previous_block = blockchain_pow[0]

# How many blocks should we add to the chain after genesis block
num_blocks = 20

#magnitude of difficulty of hash - number of zeroes that must be in the beginning of the hash
difficulty = 3

#length of nonce that will be generated and added
nonce_length = 20

# Add blocks to the chain based on difficulty with nonces of length nonce_length
def create_pow_blockchain(num_blocks, difficulty, blockchain_pow, previous_block, nonce_length, print_data=1
    hash_array = []
    time_array = []
    for i in range(0, num_blocks):
        time_taken, hashes_tried, block_to_add = find_next_block(previous_block, difficulty, nonce_length)
        blockchain_pow.append(block_to_add)
        previous_block = block_to_add
        hash_array.append(hashes_tried)
        time_array.append(time_taken)
        # Tell everyone about it!
        if print_data:
            print("Block #{} has been added to the blockchain!".format(block_to_add.index))
            print("{} Hashes Tried!".format(hashes_tried))
            print("Time taken to find block: {}".format(time_taken))
            print("Hash: {}\n".format(block_to_add.hash))
    return(hash_array, time_array)

hash_array, time_array = create_pow_blockchain(num_blocks, difficulty, blockchain_pow, previous_block, nonce
```

```
Block #1 has been added to the blockchain!
4130 Hashes Tried!
Time taken to find block: 0.109375
Hash: 0008c04c30251f8396c99660e2cac10c2170b7b716932315c733a3fd6782cf86

Block #2 has been added to the blockchain!
5137 Hashes Tried!
```

```
Time taken to find block: 0.15625
Hash: 0004a863399d83be74c908626b30fcef0c36e49e2151d2971a8b1736d8f546d9

Block #3 has been added to the blockchain!
564 Hashes Tried!
Time taken to find block: 0.015625
Hash: 00053a7dc0e31faa043f92ef9c88d299849c13db5c115f1273f3eb67eb309395

Block #4 has been added to the blockchain!
5636 Hashes Tried!
Time taken to find block: 0.15625
Hash: 00084f0e17b5ceda045cfd5e88ddbba8d9b5e3d6c9119f15a5f432a4b8253f1c

Block #5 has been added to the blockchain!
11507 Hashes Tried!
Time taken to find block: 0.296875
Hash: 000f049c6a9c8d82d0d23a93e43ed17982db9e8eb1d759761c50fc0e41f6ea9e

Block #6 has been added to the blockchain!
6699 Hashes Tried!
Time taken to find block: 0.203125
Hash: 000f8584bdb55f90f42f350f53900044f4454cc737247f850d7f2f35d45617fc

Block #7 has been added to the blockchain!
2796 Hashes Tried!
Time taken to find block: 0.09375
Hash: 00004deb701363b25ce08668a5eeae49e95a9dea7509f045d5d85e381fec713d

Block #8 has been added to the blockchain!
8669 Hashes Tried!
Time taken to find block: 0.234375
Hash: 0003b3e495825c2d69f664a915c3ac8ba262b5f1c67faed8f06404e4218d369b

Block #9 has been added to the blockchain!
1380 Hashes Tried!
Time taken to find block: 0.03125
Hash: 00025aeacbc286b12851d36e5816ab3b424ab99a59eac8906939e73fb198f197

Block #10 has been added to the blockchain!
269 Hashes Tried!
Time taken to find block: 0.015625
Hash: 00037714a82283da2afdac1e6b5d31eadd807238752a91586b9c6587727b06a0
```

```
Block #11 has been added to the blockchain!
4 Hashes Tried!
Time taken to find block: 0.0
Hash: 0008b9afa130e3b93df1c84d1ba1e0aa8bd24d61504f79f6f438181f28eacb24

Block #12 has been added to the blockchain!
1264 Hashes Tried!
Time taken to find block: 0.03125
Hash: 000ec497880b12013f4f8b5667db614e2c49e4cb7d13132e47d9c1a2cfb0d489

Block #13 has been added to the blockchain!
180 Hashes Tried!
Time taken to find block: 0.0
Hash: 0008ba8598050820b76b372d1efcf089e5fd022469a310ab8c9256ab6baa6703

Block #14 has been added to the blockchain!
6543 Hashes Tried!
Time taken to find block: 0.171875
Hash: 0007aaa2043919a6bc06712bd6cf529d57c7298bf67d6e9abd57f6a14ca9ea9f

Block #15 has been added to the blockchain!
3293 Hashes Tried!
Time taken to find block: 0.09375
Hash: 0001ef28cbd553dbe6706e2d642d206600cf93cd58f4311fb2b456069fdd7832

Block #16 has been added to the blockchain!
405 Hashes Tried!
Time taken to find block: 0.015625
Hash: 00013fbfb2942d1c0031537aedd0b1cc946d72269cd1db1ad4e5f905a3f785d7

Block #17 has been added to the blockchain!
2434 Hashes Tried!
Time taken to find block: 0.078125
Hash: 0005847a7f1711185a27ef5d506289ba6a1bf9e9ce4ae37a5927896fbd8b35fa

Block #18 has been added to the blockchain!
6112 Hashes Tried!
Time taken to find block: 0.171875
Hash: 000739b00046d482e52dd919a9b6e4ddedcdea722df650ccee72b1559743c6d1

Block #19 has been added to the blockchain!
10420 Hashes Tried!
Time taken to find block: 0.328125
```

```
Hash: 000c7c86237fa6211ee9257c3c4b724bc94fd9e4d487c48c4b3ba06678597094

Block #20 has been added to the blockchain!
13643 Hashes Tried!
Time taken to find block: 0.390625
Hash: 000db719bcd30fe9203cdad1d7ec255f872453d373bae8516a9db4656ad3467a
```

TEST YOUR QUESTION 4 CODE HERE

Run the block of code below

In [110]:

```
def test_question_4(blockchain_pow, num_blocks):
    correct = True
    bound = generate_difficulty_bound(difficulty)
    if len(blockchain_pow) != num_blocks + 1:
        correct = False
    for i in range(len(blockchain_pow) - 1):
        if blockchain_pow[i + 1].previous_hash != blockchain_pow[i].hash:
            print("H")
            correct = False
            break
        if int(blockchain_pow[i + 1].hash, 16) > bound:
            correct = False
            print("B")
            break
    print_statement = "PASSED!!! Move on to the next Part" if correct else "FAILED!!! Try Again :("
    print(print_statement)

test_question_4(blockchain_pow, num_blocks)
```

```
PASSED!!! Move on to the next Part
```

Once you have passed the above test case, you can do things like play around with the difficulties and nonce lengths and seeing that happens

Note: The sections below are optional but super cool!

Distributed Network + Compute Power Simulation

In this section, instead of simulating a single node that carries out all the proof of work calculations, we will create a bunch of different nodes that will all compete to find different blocks. These nodes will all have different compute powers that we will be able to specify. Below we will define the MinerNodeNaive class which has an init function to create an instance, as well as a try_hash function that allows it to try a hash and see if it works.

In [11]:
```
#Naive miner class that races with other miners to see who can get a certain number of blocks first
class MinerNodeNaive:
    def __init__(self, name, compute):
        self.name = name
        self.compute = compute

    def try_hash(self, diff_value, chain):
        last_block = chain[-1]
        difficulty = generate_difficulty_bound(diff_value)
        date_now = date.datetime.now()
        this_index = last_block.index + 1
        this_timestamp = date_now
        this_data = "Hey! I'm block " + str(this_index)
        this_hash = last_block.hash
        new_block = Block(this_index, this_timestamp, this_data, this_hash)
        if int(new_block.hash, 16) < difficulty:
            chain.append(new_block)
            # Tell everyone about it!
            print("Block #{} has been added to the blockchain!".format(new_block.index))
            print("Block found by: {}".format(self.name))
            print("Hash: {}\n".format(new_block.hash))
```

Question #5: Describe the Following Cells

In the next 4 cells I try to simulate a bunch of different miners with different compute powers. However this isn't completely indicative of how a real system works, and is limited by what I had to work with. Read through the code, and run all of the cells and explain what is happening in a paragrpah. Also explain how this is different from real world vanilla proof of work systems and how this simulation isn't completely accurate of what's actually happening while nodes race to find the next block. There is a cell at the bottom where you can enter your description.

In [ ]:
```
#Initialize multiple miners on the network
berkeley_Miner = MinerNodeNaive("Berkeley Miner", 10)
stanford_Miner = MinerNodeNaive("Stanford Miner", 5)
MIT_Miner = MinerNodeNaive("MIT Miner", 2)
UCLA_Miner = MinerNodeNaive("UCLA Miner", 1)

miner_array = [berkeley_Miner, stanford_Miner, MIT_Miner, UCLA_Miner]
```

In [ ]:
```
def create_compute_simulation(miner_array):
    compute_array = []
    for miner in miner_array:
        for i in range(miner.compute):
            compute_array.append(miner.name)
    return(compute_array)

compute_simulation_array = create_compute_simulation(miner_array)
rand.shuffle(compute_simulation_array)
```

In [ ]:
```
chain_length = 20
blockchain_distributed = [create_genesis_block()]
genesis_block_dist = blockchain_distributed[0]
chain_difficulty = [rand.randint(2,4) for i in range(chain_length)]
```

In [ ]:
```
for i in range(len(chain_difficulty)):
    while len(blockchain_distributed) < i + 2:
        next_miner_str = rand.sample(compute_simulation_array, 1)[0]
        next_miner = berkeley_Miner #random default (go bears)
        for miner in miner_array:
            if next_miner_str == miner.name:
                next_miner = miner
        next_miner.try_hash(chain_difficulty[i], blockchain_distributed)
```

Question 5 Description Here

- Description: **#####Your description here####**

Blockchain Data Analytics

Here we will use data analytics to analyze the number of hashes we need to try before we find a valid hash, given a difficulty level. Look through the code briefly (in depth is not necessary), run all of the cells, and answer the question at the bottom. (difficulty level 3 takes a bit of time so dont be alarmed if this next cell takes a bit of time to run).

In [12]:
```
blockchain = [create_genesis_block()]
previous_block = blockchain[0]
num_blocks = 10

#3 different types of difficulty to analyze
difficulty_0 = 1
difficulty_1 = 2
difficulty_2 = 3
difficulty_3 = 4

nonce_length = 20

hash_array_0, time_array_0 = create_pow_blockchain(num_blocks, difficulty_0, blockchain, previous_block, non
print("Difficulty Level: {} complete".format(difficulty_0))
hash_array_1, time_array_1 = create_pow_blockchain(num_blocks, difficulty_1, blockchain, previous_block, non
print("Difficulty Level: {} complete".format(difficulty_1))
hash_array_2, time_array_2 = create_pow_blockchain(num_blocks, difficulty_2, blockchain, previous_block, non
print("Difficulty Level: {} complete".format(difficulty_2))
hash_array_3, time_array_3 = create_pow_blockchain(num_blocks, difficulty_3, blockchain, previous_block, non
print("Difficulty Level: {} complete".format(difficulty_3))
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-12-13fc893880d5> in <module>
----> 1 blockchain = [create_genesis_block()]
      2 previous_block = blockchain[0]
      3 num_blocks = 10
      4 
      5 #3 different types of difficulty to analyze

NameError: name 'create_genesis_block' is not defined
```

In [13]:
```
mean_arr_hash = [np.mean(hash_array_0), np.mean(hash_array_1), np.mean(hash_array_2), np.mean(hash_array_3)]
mean_arr_time = [np.mean(time_array_0), np.mean(time_array_1), np.mean(time_array_2), np.mean(time_array_3)]
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-13-7ddcfa1ff361> in <module>
----> 1 mean_arr_hash = [np.mean(hash_array_0), np.mean(hash_array_1), np.mean(hash_array_2), np.mean(hash_
array_3)]
      2 mean_arr_time = [np.mean(time_array_0), np.mean(time_array_1), np.mean(time_array_2), np.mean(time_
array_3)]

NameError: name 'np' is not defined
```

In [ ]:
```
plt.plot(mean_arr_hash)
plt.show()
```

In [ ]:
```
plt.plot(mean_arr_time)
plt.show()
```

In [14]:
```
diff_factor_1 = np.mean(hash_array_1)/np.mean(hash_array_0)
diff_factor_2 = np.mean(hash_array_2)/np.mean(hash_array_1)
diff_factor_3 = np.mean(hash_array_3)/np.mean(hash_array_2)
print("Factor of difficulty increase from 1 to 2: {}".format(diff_factor_1))
print("Factor of difficulty increase from 2 to 3: {}".format(diff_factor_2))
print("Factor of difficulty increase from 3 to 4: {}".format(diff_factor_3))
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-14-767dd4854826> in <module>
----> 1 diff_factor_1 = np.mean(hash_array_1)/np.mean(hash_array_0)
      2 diff_factor_2 = np.mean(hash_array_2)/np.mean(hash_array_1)
      3 diff_factor_3 = np.mean(hash_array_3)/np.mean(hash_array_2)
      4 print("Factor of difficulty increase from 1 to 2: {}".format(diff_factor_1))
      5 print("Factor of difficulty increase from 2 to 3: {}".format(diff_factor_2))

NameError: name 'np' is not defined
```

Question 6: This one is simple

Look at the factor of difficulty increase from each level to the subsequent level, printed from the cell above.

What whould the factor of difficulty increase be for each level: **Your answer Here**

Check Off

If you have filled out all the skeleton code and put down your answers for the questions, let any of the instructor know and we will check you off!

If you cannot finish it this class, you have until next week's class to finish check off and earn points.