This material is copyrighted. It must not be distributed without permission from Joonsang Baek and Jongkil Kim

Lab 4

Proof-of-Work and Digital Signature

You will use Ubuntu VM for this lab exercise.

[Proof-of-Work]

1. Test sha256 hash algorithm using python. Type *python3* in the terminal and execute the following commands (one-by-one):

   ```
   >>> import hashlib
   >>> out = hashlib.sha256(b"hello world").hexdigest()
   >>> print(out)
   ```

   Check the binary values of hash output using `bin` function. Try,
   `>>> bin(int(out,16))`

2. Download *Lab4_tr.py* file from Moodle. It has transaction records, which are written in a list variable *transaction*. You can add your codes in this file.

3. Make a genesis block containing the empty transaction (i.e., "") and save it in JSON file format using python. The block should have three data items: "Block number", "Hash" and "Transaction" as follows:

   ```
   {
      "Block number": 0,
      "Hash": Genesis,
      "Transaction": ""
   }
   ```

   You can use `json` package (https://docs.python.org/3/library/json.html) to store data in the JSON format.

   Particularly, you can use *json.dump* to export the json data. For example, execute the following and check the output:

   ```
   print(json.dumps({' Block number ': 0, ' Hash':
   "Genesis", 'Transaction': ""}, sort_keys=True,
   indent=4, separators=(',', ': ')))
   ```

   Note that the above genesis block does not contain the hash value of the previous block.

4. You need to create 10 json files, which are [Block number].json. (i.e, 1.json, 2.json, …, 10.json). From the first block, the followings are used to make up the blocks
   - "Hash" is a SHA256 hash value of the previous block in hexadecimal representation.

This material is copyrighted. It must not be distributed without permission from Joonsang Baek and Jongkil Kim

- "Transaction" is a transaction written in *Lab4_tr.py* in order. For example, the first block will have `transactions[0]` as the record for "Transaction".

[Note] You can create file by using a function `open` *(e.g. you can create "hello.txt" using the following codes):*

```
fw = open("hello.txt", "w+")
fw.write("hello! World!")
fw.close()
```

To read the file content you can use a function `open` *with an input* `"r"`*. Test the following codes for hints:*

```
fr = open("hello.txt", "r")
print(fr.read())
fr.close()
```

5. Change any record in the middle of *transactions* and check the change of the hash values.

6. Add a new data item "Nonce" in the block. Find a nonce value (in hexadecimal numbers, but not mandatory) which makes the hash output of the block starts with sixteen 0 bits (i.e., check whether the hash output is less than or equal to "0000ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff"). Find nonce values for all blocks, which satisfy "Sixteen 0 bits" condition. How long it takes to create whole blocks?

7. Set various mining difficulties by setting up the number of 0 bits that required at the beginning of "Hash".

8. *Verify the result:* You can verify the *i* th block using *(i+1)* th block, For example, 1) read the contents in 1.json as a string 2) compute the hash value of the string and 3) compare the computed value with the value in "Hash:" in 2.json file. You can use the `json.loads` to extract "Hash" value if the content is the Json format.

   The following codes, which read only record in a specific field, will be helpful:

```
data = json.loads('{"a": 1, "b": 2, "c": 3}')
print data["c"]
```

[Digital Signature]

Digital Signature Algorithm (DSA) is a widely used digital signature scheme. Multiple python packages are supporting DSA. **Pycryptodome** is one of them. In the following tasks, you will generate a key pair for DSA, sign a message and

This material is copyrighted. It must not be distributed without permission from Joonsang Baek and Jongkil Kim

verify the signature. The general instruction on how to use DSA can be found in https://pycryptodome.readthedocs.io/en/latest/src/public_key/dsa.html.

However, sometimes, we need to handle DSA at a lower level to satisfy various requirements. Particularly, in many blockchain systems including Bitcoin, signatures, hash values and all other necessary parameters in a ledger are written directly using hexadecimal numbers. The following tasks will help you to access those values and handle them in **Pycryptodome**.

For the following tasks, you need to import the following python packages:

```
from Crypto.PublicKey import DSA
from Crypto.Signature import DSS
from Crypto.Hash import SHA256
import binascii
```

9. (DSA key generation) You need to generate a public/private key pair for the DSA signature scheme. In order to let a verifier verify a signature in a digital signature algorithm, a signer must share some public parameters in addition to a public key. In DSA, a signer needs to share three parameters (g, p, q) and a public key (y). Also, a signer keeps its private key (x) to sign a message.

   In **Pycryptodome,** the following code will generate those parameters with a 1024 bit public key:

   ```
   key = DSA.generate(1024)
   ```

   After the key generation, you can check the actual values of `y`, `g`, `p` and `q` as decimal numbers by executing followings:

   ```
   tup = [key.y, key.g, key.p, key.q]
   print(tup)
   ```

10. (Signing a message) A hash algorithm is usually used before signing a message for the efficiency reason. The following code shows that message "Hello" is hashed using the SHA256 algorithm and, then, signed.

    ```
    message = b"Hello"
    hash_obj = SHA256.new(message)
    signer = DSS.new(key, 'fips-186-3')
    signature = signer.sign(hash_obj)
    ```

    Note that the DSS class is used instead of the DSA class to generate a signature. DSS is a standard for digital signature, which is defined with the recommended options such as key sizes and hash functions. Here, using DSS means that we will use the DSA algorithm in a way recommended in DSS (Digital Signature Standard), which is published by NIST, U.S. as 'fips-186-3'.

This material is copyrighted. It must not be distributed without permission from Joonsang Baek and Jongkil Kim

The hash outcome and the signature can be printed out by adding the following code:

```
print(hash_obj.hexdigest())
signature_hex = binascii.hexlify(signature)
print(signature_hex)
```

The outputs are printed using hexadecimal notation.

11. (Verifying a message) A signature can be verified by sharing the necessary information. First, the verifier must know which parameters are used for the DSA. Therefore, (g, p, q) are needed to prepare the same DSA setting for the verifier. Also, it needs a public key (y) for the verification. The DSA class in **Pycryptodome** allows the DSA public key can be set using this information. Let `tup` be the tuple defined in task 9. The DSA public key object can be created by using the following:

```
pub_key = DSA.construct(tup)
```

Note that the order of values in `tup` is important.

Once, it is set, the signature can be verified by using the following code:

```
hash_obj = SHA256.new(message)
verifier = DSS.new(pub_key, 'fips-186-3')

try:
    verifier.verify(hash_obj, signature)
    print("The message is authentic.")
except ValueError:
    print("The message is not authentic.")
```

[Note] If the signature is delivered as a hexadecimal number, which is encoded by `signature_hex = binascii.hexlify(signature)`. You need the following code to convert it to bytes :

```
signature = binascii.unhexlify(signature_hex)
```