Blockchain Lab Practical Guided Exercises to Support the Coursework Assignment

Introduction

The goal of these practical sessions is to create an offline Blockchain application. This application can be developed in a language of your choice, although it is **highly recommended** to use C#. This brief operates on the assumption that C# is used and resources such as the template code are in C#. C# is an object-oriented language and is very similar to Java, and it should be relatively easy to pick up. Visual Studio is the recommended IDE for developing in C#, the latest version can be found here.

Along with this brief you should find a visual studio project which will provide the template code for the assignment. This code should be built upon during the practicals. The template makes use of Windows Forms for the UI. Windows Forms is simple and quick to use, meaning you can focus on the technical aspects more than the appearance. This template also includes ready-made classes that will be used to develop features.

Practical Guided Exercises and Assignment

It is expected that you will complete the first 5 parts of this brief during the practical sessions. The assignment tasks are based on the application you will have made in the first 5 parts. Marks for the assignment are allocated as follows:

PROVISIONAL Marking Scheme (to be confirmed by way of formal Assignment Sheet to be notified by 12/01/21).

	Evidence of implementation	**Report**	**Total**
Part 1 – Project Setup	3	2	**5**
Part 2 – Blocks and the Blockchain	5	5	**10**
Part 3 – Transactions and Digital Signatures	5	5	**10**
Part 4 – Consensus Algorithms (Proof-of-Work)	10	10	**20**
Part 5 - Validation	10	10	**20**
Part 6 – Assignment Tasks	15	20	**35**
	48	52	**(100)**

For each part in this brief, you are expected to provide evidence of implementation alongside a report which details your understanding of the topic.

In part 6 of the assignment, you are only expected to complete 3 out of 4 of the sections to achieve the maximum of 35 marks. However, if you choose to do all four you may be able to score more marks (**not** more than 35).

This brief will require students to develop a basic offline Blockchain application that includes the following:

- Blockchain made of cryptographically connected Blocks
- Transaction generation – including digital signature via asymmetric encryption
- A Proof-of-Work consensus algorithm – including hashing and threading
- Validation methods to ensure the Blockchain is valid
- A basic UI that can verify the implementation of the above features

The following instructions are built to ease you into programming with C# and Windows Forms and gradually get more complex throughout.

Part 1 – Project Setup

Setup

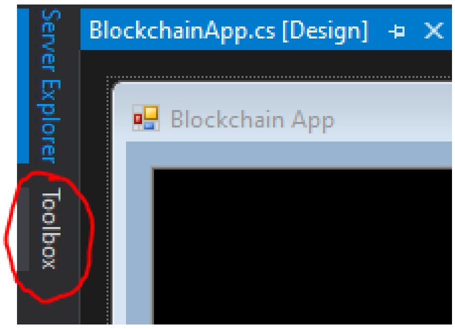
Download the BlockchainAssignment Project from Blackboard. Open Visual Studio click open project/solution and select *BlockchainAssignment.sln.* In the Solution explorer you will see the BlockchainAssignment.csproj. Expand this to find the components of the project, including the code, references and properties. For us, the important parts are the *BlockchainApp.cs*, *HashCode* folder, and *Wallet* folder.

Double click the *BlockchainApp.cs* to open a Windows Form, which shows the basic UI of the App. You are free to modify it during this assignment as you see fit but ensure that the results are clear (as you are required to provide screenshots in your report). Right click *BlockchainApp.cs* and click *view code* to see the underlying code behind the UI. Currently it is mostly empty.

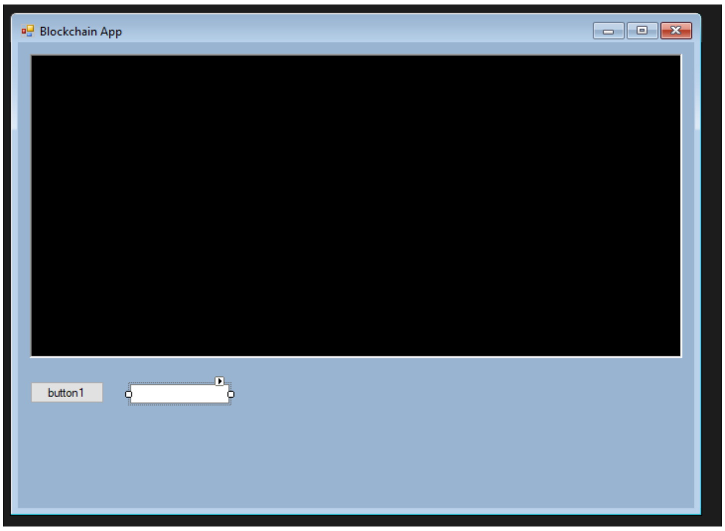
The *HashCode* and *Wallet* folders are host to ready-made code that will be applied later in this assignment. *Wallet* is responsible for the asymmetric encryption (public and private key pairs) and digital signatures in transaction. *HashCode* is responsible for combining hashes and converting hashes to and from byte arrays to strings both ways. This code is provided to help reduce the workload of the project, such that time can be evenly distributed across each of the components.

Customising the UI

To familiarise yourself with Windows Forms and C# briefly add some functionality to the UI. First add a button, on the left side of the screen a horizontal tab titled *Toolbox* should be visible.

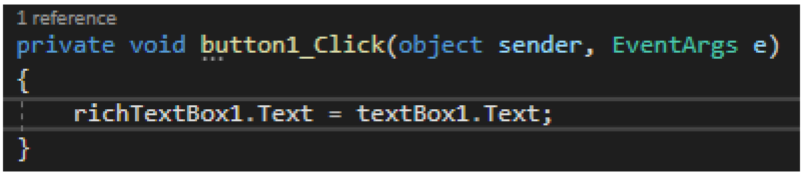


Click *Toolbox* and a window will expand, click *All Windows Forms* or *Common Controls* and click and drag *Button* onto the UI. Now search for a *TextBox* in the toolbox and drag it onto the UI. Your UI should now appear as the following:



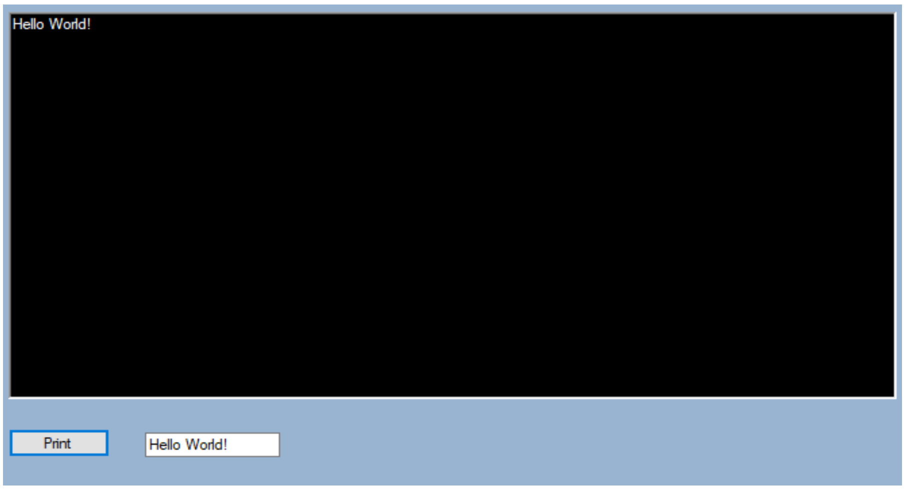
Click *button1* once to open its properties in the bottom left. Change its *text* property to say '*Print*' instead of '*button1'.* Now double click the button and you will be taken to the underlying code, a new function for clicking the button is created after double clicking it. Next you can write some code that takes the text from the *TextBox* we added and prints it to the larger textbox above, when we click the *Button.* To get the names of the Textboxes you can click them in the UI form, in this case the large black text box is *richTextBox1* and the smaller text box is *textBox1*.

Code:



```
1 reference
private void button1_Click(object sender, EventArgs e)
{
    richTextBox1.Text = textBox1.Text;
}
```

Output:



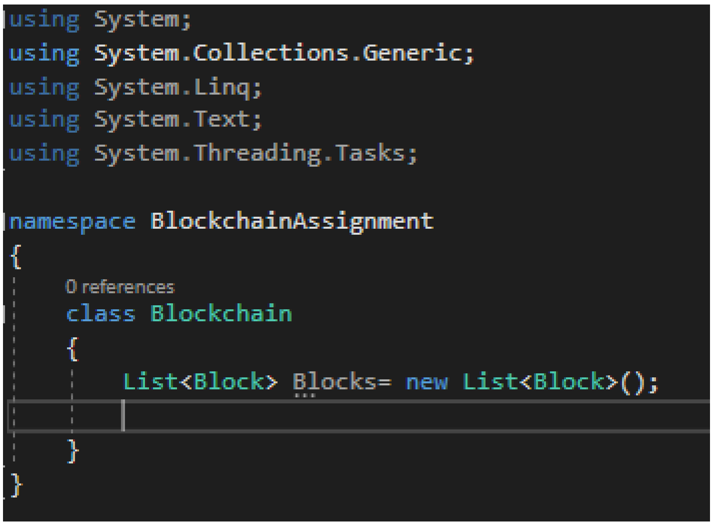
```
Hello World!
```

This textbox and button can be removed or re-purposed after this section as you wish.

Part 2 – Blocks and the Blockchain

Creating the classes and their contents

C# is an object-orientated language. Therefore, it makes sense to exploit this with the object-orientated nature of Blockchain. Right click *BlockchainAssignmentAnswer.csproj* and click *add* then *new item*. Add a new class called *Blockchain.cs* and another class called *Block.cs*. A Blockchain consists of a chain of blocks, so add a list variable into *Blockchain.cs* that holds Blocks.



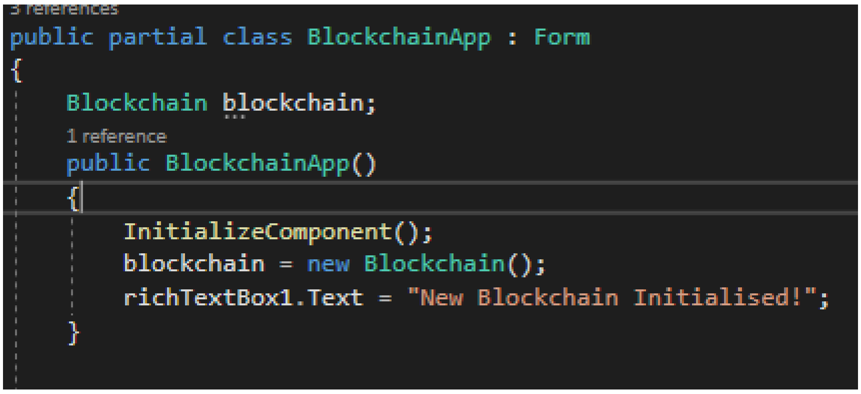
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace BlockchainAssignment
{
    class Blockchain
    {
        List<Block> Blocks= new List<Block>();

    }
}
```

***Blockchains.cs* code**

The Blockchain itself needs to be initialised in the code. You can initialise in it *BlockchainApp.cs* or initialise it in *Program.cs* (Main) and pass into the *BlockchainApp.cs*.



```
public partial class BlockchainApp : Form
{
    Blockchain blockchain;
    public BlockchainApp()
    {
        InitializeComponent();
        blockchain = new Blockchain();
        richTextBox1.Text = "New Blockchain Initialised!";
    }
```

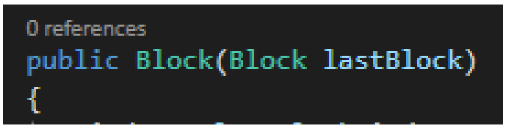
Initialising blockchain within the form

Now you can add variables into *Block.cs*. Typically, a Block within a Blockchain would have many variables. Many of these will added throughout the duration of the assignment, so write the code carefully such that it can be easily changed at a later point. For now, the following variables should be added to the Block class with appropriate types:

- A timestamp, set when the Block is made
- An index; the position of the Block within the Blockchain
- Hash of the Block

- Hash of the previous Block

Now create a constructor for a new Block to assign these variables. The constructor should have two arguments, the hash of the prior Block and the index of the prior Block. Alternatively, you can pass the whole previous Block through (In this case you will need to set the access modifiers for the variables to public). The timestamp should be set to the current time (current time can be retrieved using *Datetime.Now*), and the previous hash and current index should be set with respect to the given arguments. Assignment of the hash of the Block will be covered in the next section.



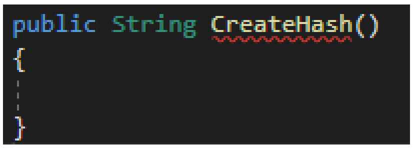
```
0 references
public Block(Block lastBlock)
{
```

A Constructor with one argument in C#

Hashing and creating the Genesis Block

A **genesis block** is the first block of a block chain. Modern versions of Bitcoin number it as **block 0**, though very early versions counted it as block 1

We now need to generate a hash to finish the assignment of the variables in *Block*.cs. First, we need to create a new method in *Block.cs* that will create the hash. This method will also be extended as w required during the course of the assignment.



```
public String CreateHash()
{

}
```

An empty method in C# that should return a String

The hash of a Block is normally the hash of all information within the Block. Therefore, we want to combine the index, timestamp and previous hash and hash the combination. The code displayed below uses the SHA256 to produce a hash. You can copy it into your CreateHash method.

SHA-256 and SHA-512 are novel hash functions computed with 32-bit and 64-bit words, respectively.

https://emn178.github.io/online-tools/sha256.html

```
SHA256 hasher;
hasher = SHA256Managed.Create();
String input = index.ToString() + timestamp.ToString() + prevHash;
Byte[] hashByte = hasher.ComputeHash(Encoding.UTF8.GetBytes((input)));

String hash = string.Empty;

foreach (byte x in hashByte)
{
       hash += String.Format("{0:x2}", x);
}
return hash;
```

Finally, now assign the value of the hash variable in the constructor to the output of CreateHash()

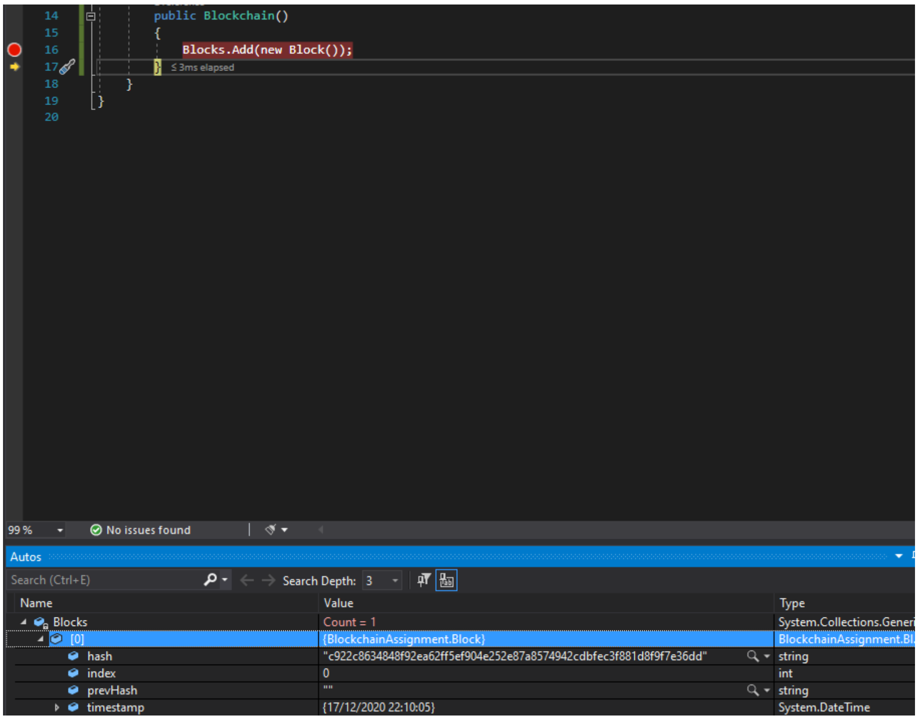
Now we have a constructor to create Blocks, but no methods that actually create them. As stated, the first Block in a Blockchain has no previous hash and is called the Genesis Block. To generate a Genesis Block we must create another constructor in *Block.cs*. This constructor will have no arguments, it will assign the previous hash to an empty string and the index to 0.

Now do the following:

- Create a constructor for *Blockchain.cs*
- This constructor calls the Genesis Block constructor in *Block.cs*.
- The Genesis Block is then added to the Blockchains list of Blocks.

Now a genesis Block should be created when the app is started.

Tip: Debugging in C# is very easy and very useful. You can place a breakpoint at any point in the code and view the state of variables and objects in real-time.



```
        public Blockchain()
        {
            Blocks.Add(new Block());
        } ≤3ms elapsed
    }
}
```

Name	Value	Type
Blocks	Count = 1	System.Collections.Generi
[0]	{BlockchainAssignment.Block}	BlockchainAssignment.Bl
hash	"c922c8634848f92ea62ff5ef904e252e87a8574942cdbfec3f881d8f9f7e36dd"	string
index	0	int
prevHash	""	string
timestamp	{17/12/2020 22:10:05}	System.DateTime

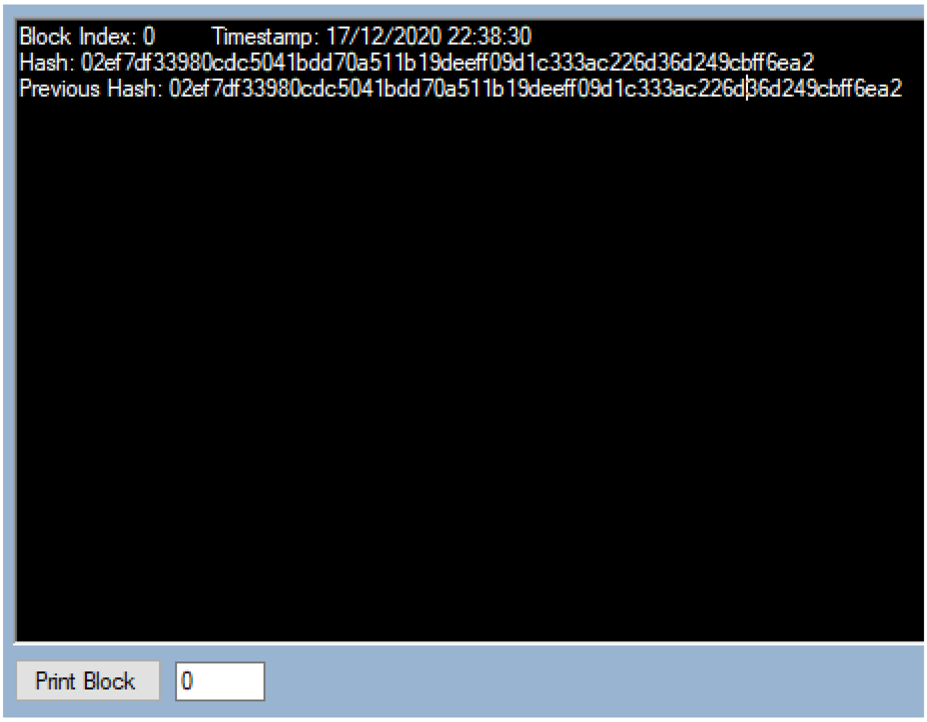
An example of debugging in C# that shows the contents of Blocks after the first Genesis Block has been added.

Printing a Block to the UI

Now that we have generated a Genesis Block we should verify this by displaying its variables to the UI. To achieve this the following should be done.

- Add a new method in *Block.cs* to return a string containing the Block index, previous hash, hash and timestamp.
- Add a new method in *Blockchain.cs* that takes a Block index as an argument, then calls the above method in the chosen block and returns the output.

Add a button in the UI that calls the *Blockchain.cs* method to read a given Block. Print this text to the large black text box. Tip: You can re-purpose the buttons you made earlier for this purpose. Also, you may want to make a method dedicated to changing the text of richTextBox1, as you may be changing it frequently.



```
Block Index: 0        Timestamp: 17/12/2020 22:38:30
Hash: 02ef7df33980cdc5041bdd70a511b19deeff09d1c333ac226d36d249cbff6ea2
Previous Hash: 02ef7df33980cdc5041bdd70a511b19deeff09d1c333ac226d36d249cbff6ea2
```

***The output of the Genesis Block** – for now*

Note: You may notice that the hash is in hexadecimal meaning that each character in the hash can be one of 16 characters: 0-9 and A,B,C,D,E,F. This is more relevant later when we focus on mining and proof-of-work.

Part 3 – Transactions and Digital Signatures

Wallets & Private and Public Key Generation

Blockchain technology is capable of storing almost any type of data within the blocks; although typically Blockchains store a ledger of transactions. Transactions on a Blockchain are a receipt of the movement of digital crypto-currency from one wallet to another. This digital currency is generated through the consensus algorithm, which we will be discussed in detail in later section. In this section we are interested in the generation of the wallets. In Blockchain a wallet has two components, a private key and a public key. A public key is public to all entities in a Blockchain, while a private key must be kept secret, as it gives the owner control of the wallet funds.

Public and Private Key Pairs are mathematically related to each other, but a private key cannot be solved (deciphered, decoded) simply by exploiting this relationship with the public key. There exist many asymmetric encryption algorithms for the purpose of generating key pairs; in this assignment we will use the Elliptic Curve Digital Signature Algorithm (ECDSA). This is the same algorithm that Bitcoin and other Crypto-currencies use, therefore we can consider our key generation to be cryptographically strong.

Within the folder *Wallet*, *Wallet.cs* can be found. This class includes the code for key pair generation and validation. Feel free to look at this code to familiarise yourself with its structure. This code is provided for your convenience to free up your time to focus on and grasp the essential framework. process.

The following methods are included:

- Wallet(out String privateKey) – Constructor for a Wallet object. The Wallet object has one variable, public key. The private key is an output parameter. Example code of constructing a Wallet:

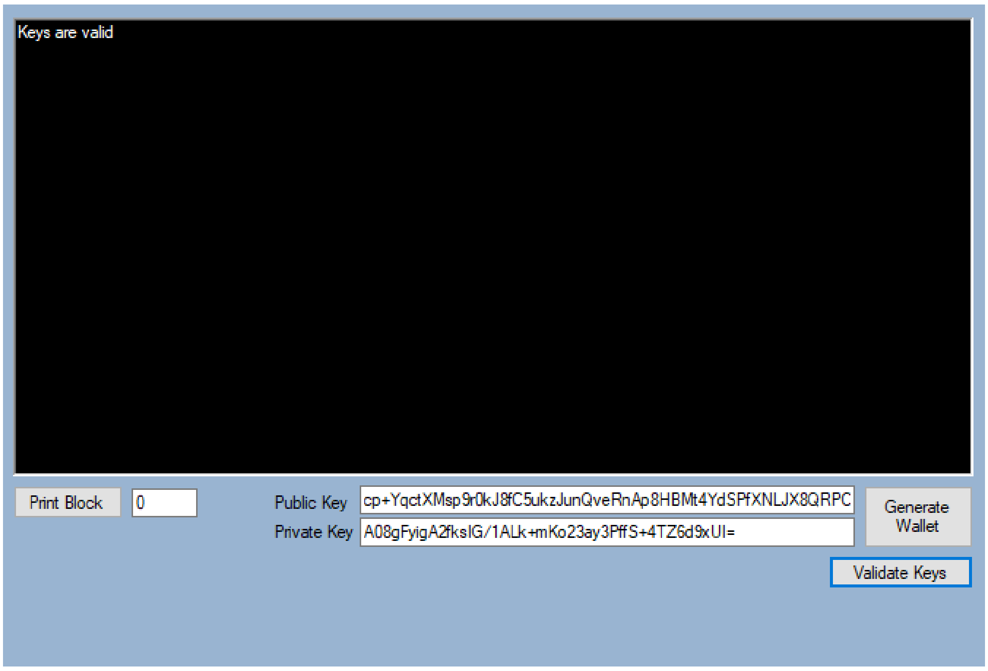
```
String privKey;
Wallet.Wallet myNewWallet = new Wallet.Wallet(out privKey);
String publicKey = myNewWallet.publicID;
Console.WriteLine(publicKey + "\n" + privKey);
```

- ValidatePrivateKey(privateKey, publicID) – Returns true if private and public key are mathematically related. Returns false otherwise.
- CreateSignature(publicID, privateKey, data) – Digitally signs the given data using the private key given. Returns the digital signature.
- ValidateSignature(publicID, data, digitalSig) - Confirms that a given signature has been signed by a private key related to the given public key. Returns true/false.

Most of these methods are static, meaning they can be called without creating an instance of the Wallet object. Next, we can make use of this code in our Blockchain application; so let us go ahead as follows:

- Add a new button to the UI that generates a new Wallet.
- Add two new text boxes to the UI to display the public and private keys after a wallet has been generated.
- Add *labels* to the UI to distinguish which key is public and private, respectively.
- Add a button to validate the public and private keys present in the textboxes(Call the ValidatePrivateKey method to do this)

Note: The public key is 64 characters, and the private key 32 characters.



```
Keys are valid
```

0

Public Key cp+YqctXMsp9r0kJ8fC5ukzJunQveRnAp8HBMt4YdSPfXNLJX8QRPC

Private Key A08gFyigA2fksIG/1ALk+mKo23ay3PffS+4TZ6d9xUI=

An example of key pair generation with validation message.

Setting up Transactions

Now that we have created wallets, we want to be able to send funds between them. To achieve this, as mentioned previously we need to add transactions to the Blockchain. First, we need to develop a *Transaction.cs* class. Next, we can add the appropriate variables you would expect to see in a Blockchain transaction. You can consider an example of a transaction from Bitcoin. From this example the following variables can be adopted (implement them in *Transaction.cs* with appropriate types):

- Hash – the hash of the transaction and its contents
- Signature – the hash of the transaction signed with the private key of the sender
- SenderAddress – the public key of the sender
- RecipientAddress – the public key of the receiver
- Timestamp – the time at which the transaction was made/sent
- Amount – the amount of currency being sent to the receiver
- Fee – the fee added to the transaction (this will be implemented later)

Now create a constructor for transaction, it should accept arguments for the above variables and also the sender's private key, although you should **exclude** hash, signature and timestamp as the method will generate these. The constructor should assign the arguments to their respective properties, then the timestamp should be set to current time.

The transaction should now have a sender, receiver, timestamp, amount, and fee. We can use this information to generate the hash. Take the code extract used in Part 2 to generate a SHA256 hash and adjust it to make a method to generate a transaction hash, then assign the hash with the result.

Finally, you need to sign the hash with the sender's private key (this was supplied in the argument of the constructor). You can generate a signature using the static method of *Wallet.cs* CreateSignature(), and provide the sender's address, their private key, and the hash of the transaction as arguments.

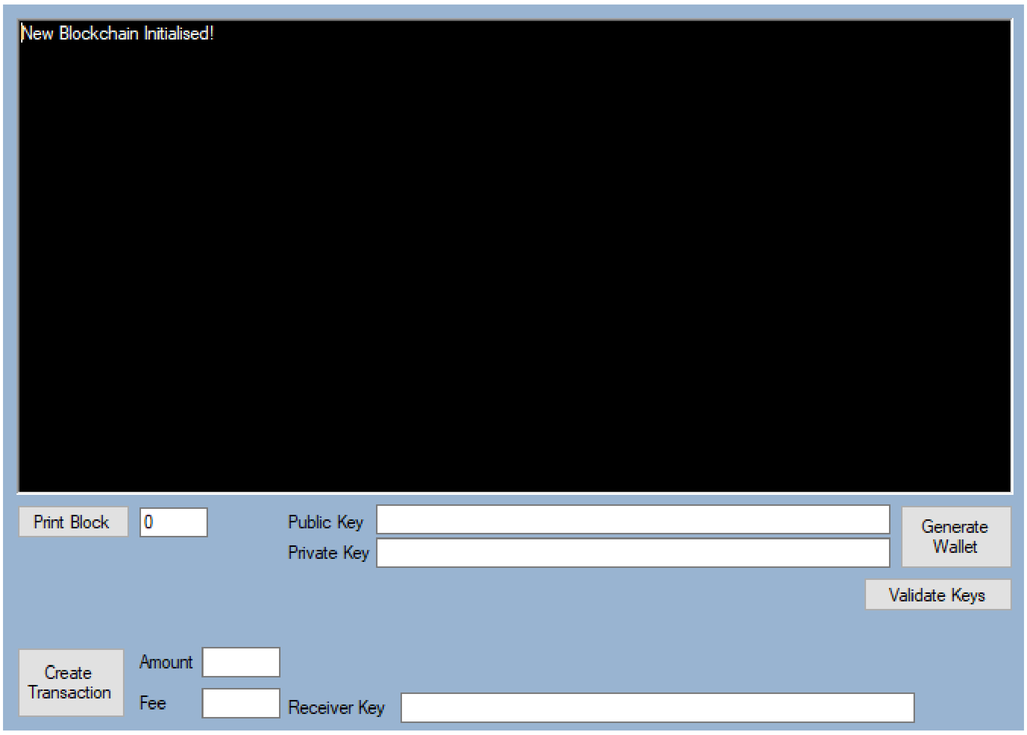
Note: The private key should not be left in the memory of the transaction instance, so do not save it as a variable.

Processing Transactions & Transaction Pools

Now that we have made the infrastructure for a transaction, we want to actually be able to generate and process transactions. For this we can first generate a transaction and then display it to the UI. This will require a new UI element; in particular:

- An 'Amount to send' label and textbox
- A 'Fee' label and textbox
- A 'receiver address' label and textbox
- A button to 'send the transaction', once the above fields have been completed

Note: You can re-use the private and public textboxes for this purpose.



New Blockchain Initialised!

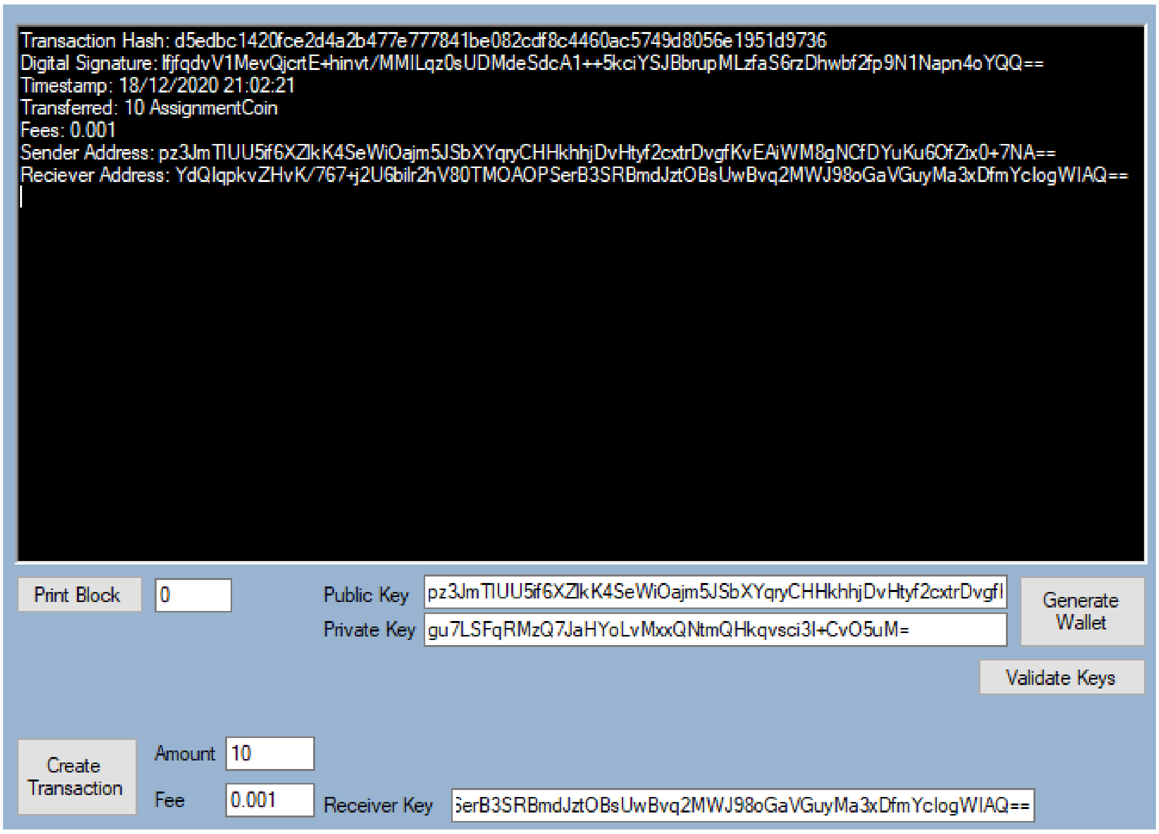
New UI for transactions

When the 'Create Transaction' button is clicked the following should occur:

- Amount, Fee, Public Key, Private Key, and Receiver Key fields are passed through to a new instance of Transaction
- The Transaction is generated
- The Transaction is printed to the UI large text box

Tip: It could be useful to make a method in *Transaction.cs* that returns the contents of a transaction as a string.

Note: Now is your chance to come up with the name of your own crypto-currency!



```
Transaction Hash: d5edbc1420fce2d4a2b477e777841be082cdf8c4460ac5749d8056e1951d9736
Digital Signature: lfjfqdvV1MevQjcrtE+hinvt/MMILqz0sUDMdeSdcA1++5kciYSJBbrupMLzfaS6rzDhwbf2fp9N1Napn4oYQQ==
Timestamp: 18/12/2020 21:02:21
Transferred: 10 AssignmentCoin
Fees: 0.001
Sender Address: pz3JmTIUU5if6XZlkK4SeWiOajm5JSbXYqryCHHkhhjDvHtyf2cxtrDvgfKvEAiWM8gNCfDYuKu6OfZix0+7NA==
Reciever Address: YdQIqpkvZHvK/767+j2U6bilr2hV80TMOAOPSerB3SRBmdJztOBsUwBvq2MWJ98oGaVGuyMa3xDfmYclogWIAQ==
```

0
Public Key: pz3JmTIUU5if6XZlkK4SeWiOajm5JSbXYqryCHHkhhjDvHtyf2cxtrDvgfI
Private Key: gu7LSFqRMzQ7JaHYoLvMxxQNtmQHkqvsci3I+CvO5uM=

Amount: 10
Fee: 0.001
Receiver Key: 5erB3SRBmdJztOBsUwBvq2MWJ98oGaVGuyMa3xDfmYclogWIAQ==

Sending some 'AssignmentCoins'

Note: You may notice there are no checks in place currently to verify if the information being provided is valid. For example, does the sender have the 10 AssignmentCoins to send? -safeguarding against double-spend!

We will add these checks in a later section, once it is possible to check the balance of a wallet.

We have successfully generated a transaction, but we are not yet able to process it so that the blockchain could show it recorded in the next update. In Bitcoin when a transaction is created it is then processed by being shared with the network and placed in a transaction pool which consists of transactions that are waiting to be added to the next block on the Blockchain.

A transaction is not confirmed or considered valid until it has been added to the Blockchain. Even then it is not always immediately accepted, some services that accept crypto-currency as payment require X number of confirmations. A transaction has one confirmation when it is first added to the Blockchain; when the next block is added it has two confirmations. Each additional block adds another confirmation. Transactions are not accepted until they reach a threshold of confirmations to reduce the impact of forks and attacks on the Blockchain. This is covered in more detail in the lectures.

Our Blockchain is offline, so we do not need to worry about sharing it to the network. However, we should still place it in a transaction pool with other transactions while it is waiting to be added to the Blockchain. Now we can create a place to store pending transactions by adding a list of transactions to *Blockchain.cs* class called 'transactionpool' or 'pendingtransactions'.

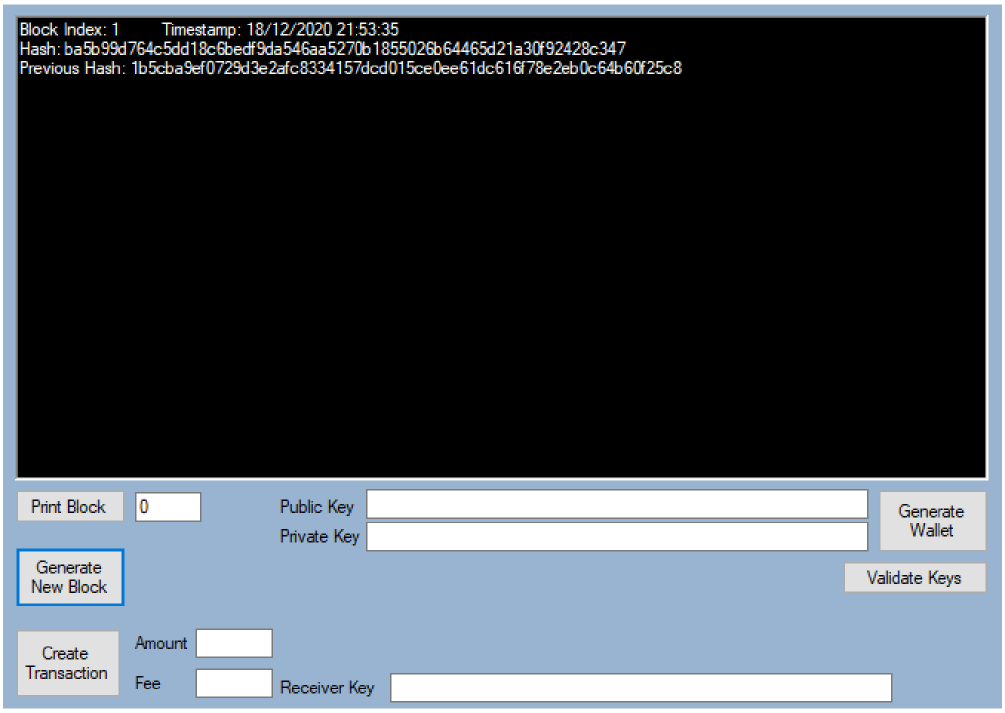
Now when we generate a transaction, we can add it to the transaction pool, so it is ready to be added to the Blockchain.

Part 4 – Consensus Algorithms (Proof-of-Work)

Generating new Blocks

Your application should now be able to generate transactions and generate a genesis block. One Block on its own is not a Blockchain, therefore we need to generate some more blocks. We can add a button to the UI with the purpose of generating a new Blocks. When you click this button, it should call the Block constructor and pass it the variables from the previous block. Once this new Block has been generated, you can add it to the Blockchain (the List<Blocks> variable in *Blockchain.cs)*. Now you can update the UI with the contents of the Block.

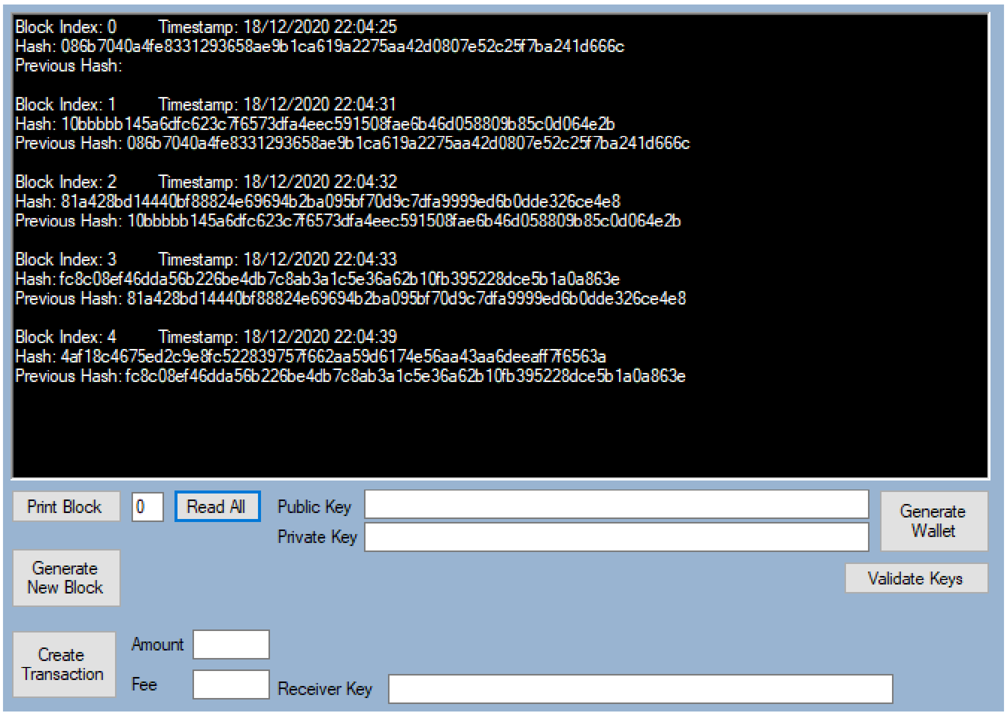
Tip: It may be useful to create a GetLastBlock method in *Blockchain.cs* to get the latest block on the chain. This method can be as simple as `return blocks[blocks.Count - 1];` Also ensure that the list of Blocks in *Blockchain.cs* is public or at least there is a public method to add new blocks to the list



```
Block Index: 1        Timestamp: 18/12/2020 21:53:35
Hash: ba5b99d764c5dd18c6bedf9da546aa5270b1855026b64465d21a30f92428c347
Previous Hash: 1b5cba9ef0729d3e2afc8334157dcd015ce0ee61dc616f78e2eb0c64b60f25c8
```

The second block – connected to the prior Genesis Block through reference to its previous hash

If you have not already done so, you can now add a method or button to read all Blocks from the Blockchain and write them to the large text box.



```
Block Index: 0        Timestamp: 18/12/2020 22:04:25
Hash: 086b7040a4fe8331293658ae9b1ca619a2275aa42d0807e52c25f7ba241d666c
Previous Hash:

Block Index: 1        Timestamp: 18/12/2020 22:04:31
Hash: 10bbbbb145a6dfc623c7f6573dfa4eec591508fae6b46d058809b85c0d064e2b
Previous Hash: 086b7040a4fe8331293658ae9b1ca619a2275aa42d0807e52c25f7ba241d666c

Block Index: 2        Timestamp: 18/12/2020 22:04:32
Hash: 81a428bd14440bf88824e69694b2ba095bf70d9c7dfa9999ed6b0dde326ce4e8
Previous Hash: 10bbbbb145a6dfc623c7f6573dfa4eec591508fae6b46d058809b85c0d064e2b

Block Index: 3        Timestamp: 18/12/2020 22:04:33
Hash: fc8c08ef46dda56b226be4db7c8ab3a1c5e36a62b10fb395228dce5b1a0a863e
Previous Hash: 81a428bd14440bf88824e69694b2ba095bf70d9c7dfa9999ed6b0dde326ce4e8

Block Index: 4        Timestamp: 18/12/2020 22:04:39
Hash: 4af18c4675ed2c9e8fc522839757f662aa59d6174e56aa43aa6deeaff7f6563a
Previous Hash: fc8c08ef46dda56b226be4db7c8ab3a1c5e36a62b10fb395228dce5b1a0a863e
```

Read All button displaying the 4 blocks within the chain

Adding transactions into Blocks

We will now make use of the transaction pool that we created earlier. Blocks should carry transactions within them. We can examine this by first adding a new data field to *Block.cs* – transactionList, which should contain transactions. When a new Block is generated it should be accompanied with transactions from the transaction pool. You can write some code to add transactions from the transaction pool to newly generated Blocks (This should occur during the Block generation process – before the hash is generated). Please make sure that the constructor of the Block is changed to accept a list of transactions in the arguments. We will edit the hashing mechanism in the Block later to consider transactions (by adding the "Merkle root").

Note: Blocks cannot take an infinite number of transactions from the transaction pool, you can add a reasonable finite limit to the number of transactions that each block can take (such as 5).

Note: Ensure that transactions picked from the transaction pool are removed once the new block is added to the chain. You may find the following line helpful:

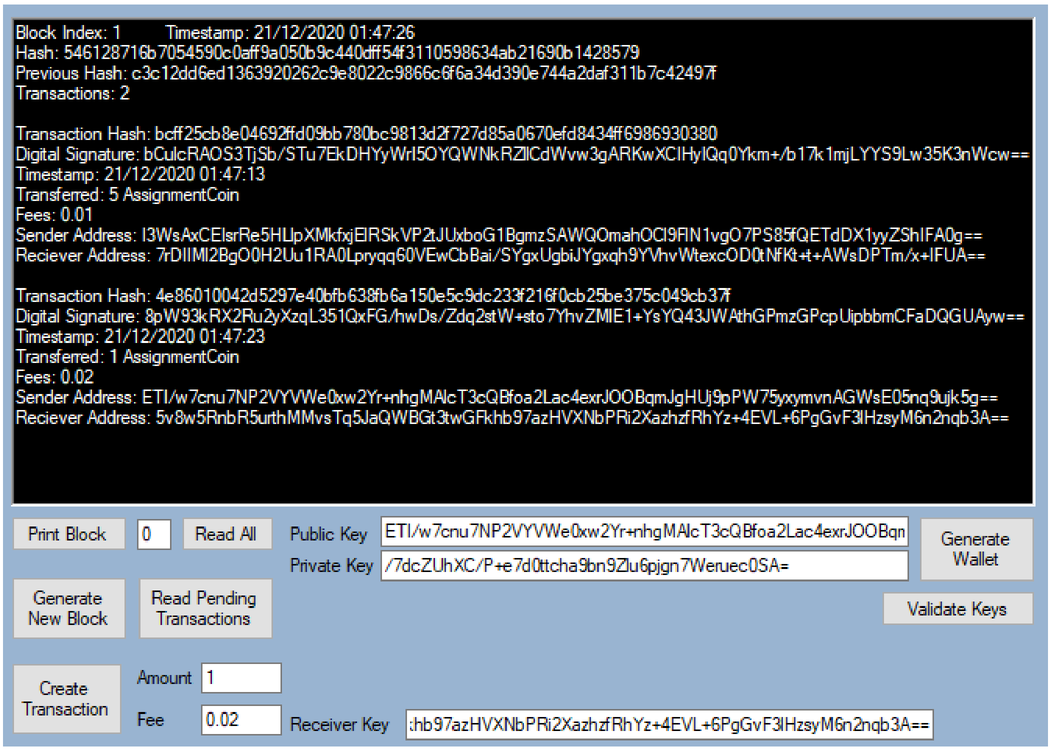
```
transactionPool = transactionPool.Except(chosenTransactions).ToList();
```

Now you can edit the Read function of Block to print out the contents of transactions within the Block.

Tip: Making use of a *foreach* loop in the Read code can make this task easier. See below.

```
foreach (Transaction t in transactions)
            {
                output += "\n" + t.ReadTransaction();
            }
```

Tip: It may be useful to add a button to read all transactions in the pool, to ensure that the code is working as intended.



```
Block Index: 1        Timestamp: 21/12/2020 01:47:26
Hash: 546128716b7054590c0aff9a050b9c440dff54f3110598634ab21690b1428579
Previous Hash: c3c12dd6ed1363920262c9e8022c9866c6f6a34d390e744a2daf311b7c42497f
Transactions: 2

Transaction Hash: bcff25cb8e04692ffd09bb780bc9813d2f727d85a0670efd8434ff6986930380
Digital Signature: bCulcRAOS3TjSb/STu7EkDHYyWrI5OYQWNkRZlICdWvw3gARKwXCIHylQq0Ykm+/b17k1mjLYYS9Lw35K3nWcw==
Timestamp: 21/12/2020 01:47:13
Transferred: 5 AssignmentCoin
Fees: 0.01
Sender Address: I3WsAxCElsrRe5HLlpXMkfxjElRSkVP2tJUxboG1BgmzSAWQOmahOCI9FIN1vgO7PS85fQETdDX1yyZShIFA0g==
Reciever Address: 7rDIIMI2BgO0H2Uu1RA0Lpryqq60VEwCbBai/SYgxUgbiJYgxqh9YVhvWtexcOD0tNfKt+t+AWsDPTm/x+IFUA==

Transaction Hash: 4e86010042d5297e40bfb638fb6a150e5c9dc233f216f0cb25be375c049cb37f
Digital Signature: 8pW93kRX2Ru2yXzqL351QxFG/hwDs/Zdq2stW+sto7YhvZMIE1+YsYQ43JWAthGPmzGPcpUipbbmCFaDQGUAyw==
Timestamp: 21/12/2020 01:47:23
Transferred: 1 AssignmentCoin
Fees: 0.02
Sender Address: ETI/w7cnu7NP2VYVWe0xw2Yr+nhgMAlcT3cQBfoa2Lac4exrJOOBqmJgHUj9pPW75yxymvnAGWsE05nq9ujk5g==
Reciever Address: 5v8w5RnbR5urthMMvsTq5JaQWBGt3twGFkhb97azHVXNbPRi2XazhzfRhYz+4EVL+6PgGvF3lHzsyM6n2nqb3A==
```

Public Key: ETI/w7cnu7NP2VYVWe0xw2Yr+nhgMAlcT3cQBfoa2Lac4exrJOOBqn

Private Key: /7dcZUhXC/P+e7d0ttcha9bn9Zlu6pjgn7Weruec0SA=

Amount: 1

Fee: 0.02

Receiver Key: :hb97azHVXNbPRi2XazhzfRhYz+4EVL+6PgGvF3lHzsyM6n2nqb3A==

Newly mined Block with two transactions

Proof-of-Work

In an online Blockchain which is distributed across nodes in a peer-2-peer network, the nodes must reach a consensus on how the next Block should be added to the chain. In Bitcoin and other existing Crypto-currencies, Proof-of-Work is used. Proof-of-Work dictates that for a new Block to be added to the chain, this Block must have a hash that satisfies a given difficulty threshold. In Proof-of-Work nodes compete to create a Block that satisfies this difficulty threshold in the shortest time Consider the example below where the difficulty threshold is set as 2.

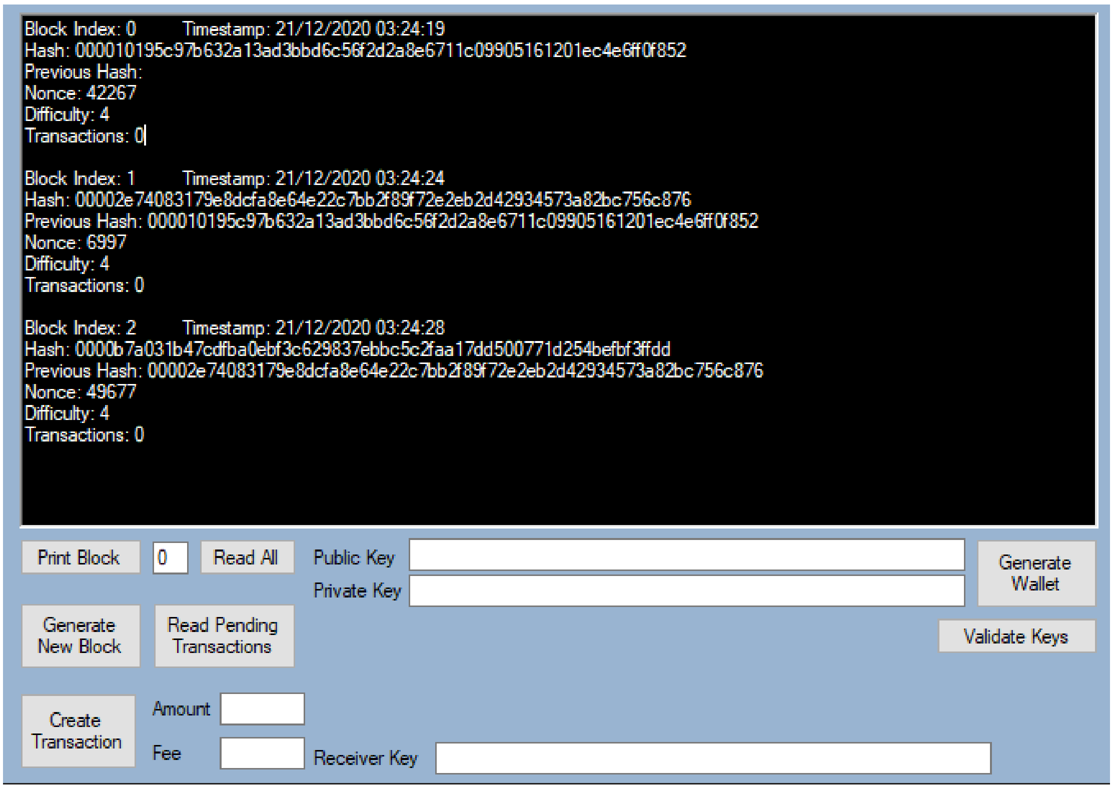
- A Blockchain node has generated a new Blockchain, however they cannot add it to the chain until the hash begins with a number of zeros corresponding to the difficulty threshold. (Difficulty = 2, therefore number of zeros required = 2).
- The node generates a hash of - *57169c0619650ff122a8d74776e74a5e6b6e8e517aea48579b2be0af19440488* for the block using the block attributes.
- This hash does not satisfy the difficulty threshold, a new hash must be generated based on this hash (re-hash). ; However hashing algorithms such as SHA256 ***always*** generate the same hash given the same input. To generate a non-identical hash the Block has an extra data property – *nonce* (number only used once). This nonce value is incremented after each hash attempt until the resulting hash satisfies the difficulty threshold.
- So, the Blockchain node increments the *nonce* and rehashes until the difficulty threshold is met. Finally, the following hash is generated that satisfies the threshold: *008a78f40bb59bf4c9da8cfccf6a9c8e1202b01933d2363eabb277ad867ba738*
- The winning node having achieved the difficulty in shortest time then goes on to shares the Block with the other nodes on the network, these nodes accept the new Block and add it to their identical copies of the Blockchain. In this way the blockchain as universal record of transactions is kept updated.

To implement Proof-of-Work within our example Blockchain we need to add a new data property to *Block.cs – nonce* (set it to 0 by default) and set a global difficulty threshold float that must be satisfied. We recommend selecting a difficulty threshold of either 4, or 5.

With the above complete, to create the Proof-of-Work algorithm the following must be implemented:

- Include the *nonce* in the CreateHash() method such that it is included in the process of the composition of the hash.
- Create a new method Mine() that executes the CreateHash() method within a while loop, which continues until the given hash starts with a number of zeros correlating to the difficulty.
- Change the *Block.cs* constructor to call Mine() instead of CreateHash().
- Increment the nonce after each failed hashing attempt. **Do not** increment the nonce after the process has already resulted in a hash that has satisfied the difficulty criteria.
- Change the Read() method in *Block.cs* to also return the last value of nonce and the difficulty level the Block was mined at.

Note: If you implemented this incorrectly your code could get stuck in a while loop. Making use of debug mode (and reducing the difficulty) may help you investigate this problem.



```
Block Index: 0        Timestamp: 21/12/2020 03:24:19
Hash: 000010195c97b632a13ad3bbd6c56f2d2a8e6711c09905161201ec4e6ff0f852
Previous Hash:
Nonce: 42267
Difficulty: 4
Transactions: 0

Block Index: 1        Timestamp: 21/12/2020 03:24:24
Hash: 00002e74083179e8dcfa8e64e22c7bb2f89f72e2eb2d42934573a82bc756c876
Previous Hash: 000010195c97b632a13ad3bbd6c56f2d2a8e6711c09905161201ec4e6ff0f852
Nonce: 6997
Difficulty: 4
Transactions: 0

Block Index: 2        Timestamp: 21/12/2020 03:24:28
Hash: 0000b7a031b47cdfba0ebf3c629837ebbc5c2faa17dd500771d254befbf3ffdd
Previous Hash: 00002e74083179e8dcfa8e64e22c7bb2f89f72e2eb2d42934573a82bc756c876
Nonce: 49677
Difficulty: 4
Transactions: 0
```

Three Blocks mined at a difficulty of 4

Difficulty (Level)

The difficulty level as set in Proof-of-Work is proportional to the number of zeros at the beginning of the hash that constitute a valid block. The more zeros required there, the longer it will take to mine. SHA256 represents characters in hex format; meaning each character can be 1 of 16 different characters. Therefore, each extra zero required to mine in Proof-of-Work increases the difficulty by a factor of 16. For now, we are using a fixed difficulty of 4 or 5 zeros (up to you) at the beginning of the hash. Later however we will discuss the topic of a dynamic difficulty (which is why the data type of difficulty is float).

For this section, the only action you need to take is to include the *difficulty* to the hash composition used in CreateHash().

Rewards & Fees

If we take a random Block from Bitcoin and look at the number of zeros required, we can see that **19** zeros were required. The chance of generating a hash that begins with 19 zeros is $1/16^{19}$. This requires a huge amount of work to be done to generate a Block. Nodes all over the world compete to be the next one to generate a Block FIRST. Specialist computation hardware and much electricity is required to even have a slim chance of being the next node to mine a Block. Right now, in our Blockchain there is no incentive for nodes to mine, other than to get their transaction on the chain. In Bitcoin and other Proof-of-Work crypto-currencies the incentive comes in the form of a **reward** and

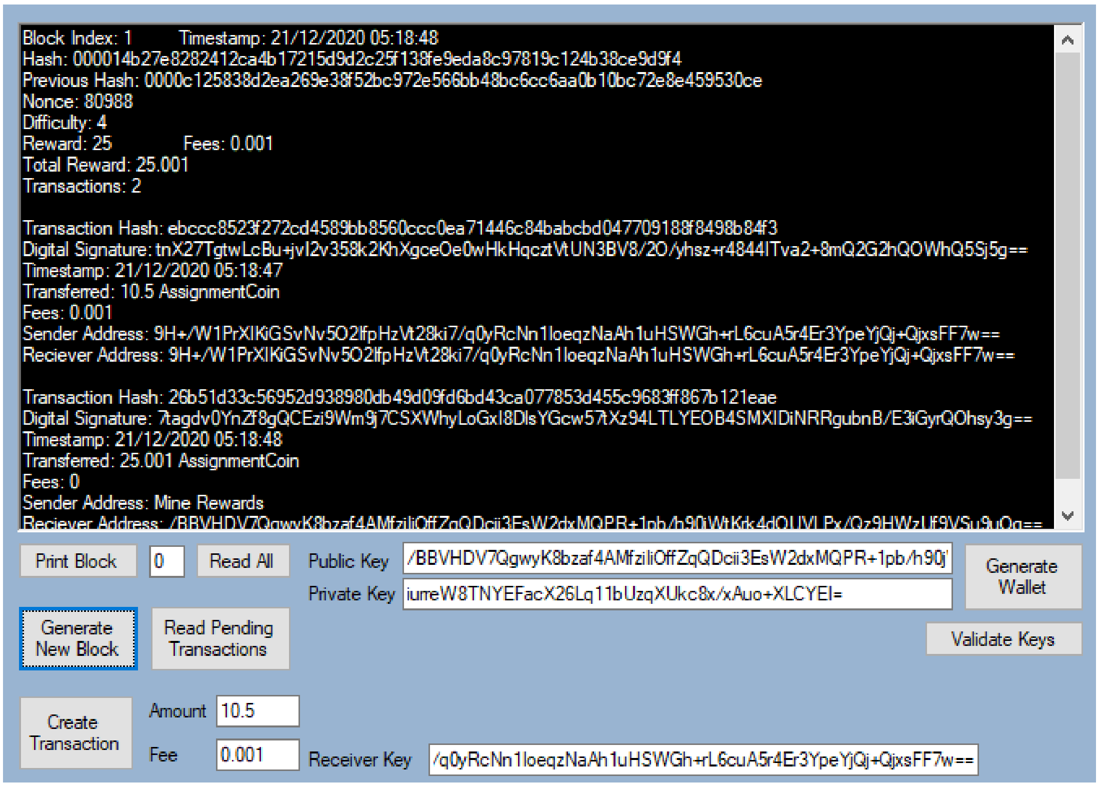
fees.

After each Block is mined in Bitcoin, a flat reward is given to the winning miner's address. This reward is given as a transaction; the receiver address is the mining node address and the sending node is technically from nowhere (as the currency is generated there and then). In Bitcoin the sender's address is 'Coinbase', which is simply the terminology to refer to currency generated as a reward. In our implementation the sender's address should be *'Mine Rewards'* (because *Wallet.cs* is configured for reward currency to be sent from '*Mine Rewards*'. You can change this if you reconfigure *Wallet.cs* too). You can set the reward amount to any number you wish; you can even add conditions to the way rewards are PAID if you wish. For example in Bitcoin, rewards are halved every 210,000 blocks; as a result only a finite amount in Bitcoins will ever exist.

Fees are a small sum added to each transaction to incentivise the mining node to choose their transaction in the next Block. The higher the fee, the more likely the node will pick it. When the node successfully mines a Block, they receive a flat reward **and** the accumulative fees from all transactions in the Block. As we have already implemented fees in our transactions, we are ready to implement rewards and fees to mining nodes.

To implemented fees and rewards we need to do the following:

- Add the logic for rewards; it can be fixed, variable, anything you decide.
- Update the UI code; such that the public address for the miner is sent through to the Block constructor.
- Before we start mining the Block, we need to add all new transaction to the transaction list (the pool) used by Block .t. In the constructor of the Block add a new method (before *Mine()* is called) this method will calculate the reward and total fees owed (from all the transactions in the Block) and create a transaction for it.
- Create this new transaction using the following line of code:
-
  ```
  Transaction transaction = new Transaction("Mine Rewards", minerAddress, (reward
  + fees), 0,"")
  ```
- *Wallet.cs* is configured to deal with 'Mine Rewards' so a private key does not need to be provided.
- Now add this new transaction to the transaction list in the Block.
- Add the *reward* to the hash composition used in CreateHash()
- Finally update the read method of *Block.cs* to print the reward, fees, and the miner's address {wallet).



```
Block Index: 1        Timestamp: 21/12/2020 05:18:48
Hash: 000014b27e8282412ca4b17215d9d2c25f138fe9eda8c97819c124b38ce9d9f4
Previous Hash: 0000c125838d2ea269e38f52bc972e566bb48bc6cc6aa0b10bc72e8e459530ce
Nonce: 80988
Difficulty: 4
Reward: 25            Fees: 0.001
Total Reward: 25.001
Transactions: 2

Transaction Hash: ebccc8523f272cd4589bb8560ccc0ea71446c84babcbd047709188f8498b84f3
Digital Signature: tnX27TgtwLcBu+jvI2v358k2KhXgceOe0wHkHqcztVtUN3BV8/2O/yhsz+r4844ITva2+8mQ2G2hQOWhQ5Sj5g==
Timestamp: 21/12/2020 05:18:47
Transferred: 10.5 AssignmentCoin
Fees: 0.001
Sender Address: 9H+/W1PrXIKiGSvNv5O2lfpHzVt28ki7/q0yRcNn1loeqzNaAh1uHSWGh+rL6cuA5r4Er3YpeYjQj+QjxsFF7w==
Reciever Address: 9H+/W1PrXIKiGSvNv5O2lfpHzVt28ki7/q0yRcNn1loeqzNaAh1uHSWGh+rL6cuA5r4Er3YpeYjQj+QjxsFF7w==

Transaction Hash: 26b51d33c56952d938980db49d09fd6bd43ca077853d455c9683ff867b121eae
Digital Signature: 7tagdv0YnZf8gQCEzi9Wm9j7CSXWhyLoGxI8DlsYGcw57tXz94LTLYEOB4SMXIDiNRRgubnB/E3iGyrQOhsy3g==
Timestamp: 21/12/2020 05:18:48
Transferred: 25.001 AssignmentCoin
Fees: 0
Sender Address: Mine Rewards
Reciever Address: /BBVHDV7QgwyK8bzaf4AMfziliOffZqQDcii3EsW2dxMQPR+1pb/h90iWtKrk4dOUVLPx/Qz9HWzUf9VSu9uOg==
```

Public Key /BBVHDV7QgwyK8bzaf4AMfziliOffZqQDcii3EsW2dxMQPR+1pb/h90j

Private Key iurreW8TNYEFacX26Lq11bUzqXUkc8x/xAuo+XLCYEI=

Amount 10.5

Fee 0.001

Receiver Key /q0yRcNn1loeqzNaAh1uHSWGh+rL6cuA5r4Er3YpeYjQj+QjxsFF7w==

Block with rewards implemented. **Miner gets a reward of 25.001; 25 from the reward and 0.001 from fees in the Blocks transactions.**

At this point we have successfully created an offline Blockchain. We have a consensus algorithm, a Blockchain, and can create transactions and get them published on the chain. However, we have not added any validation methods along the way. The next part covers validation and how we can prevent malicious actors from attacking the chain by utilising what we have already implemented.

Part 5 – Validation

A Blockchain is normally hosted by many nodes in a peer-2-peer network. By definition being a trustless network, it is impossible to trust any node in the system, as they could be malicious. Therefore, instead of trusting the other nodes, you must trust the system. So far, we have developed a system that can generate digital keys, generate transactions with signatures, and generate and add Blocks to the Blockchain. However, we have no/minimal checks along the way to validate the Blockchain. The following section will highlight the areas that need validation and checks to ensure that the Blockchain is operating as intended, with no incorrect or malicious activity.

Validating the Blockchain Structure

As we already know, Blockchain is a linear structure of connected Blocks. The component that ‘connects’ the Blocks is the reference to the hash of the previous Block. We should validate that the Blocks are properly connected, by checking that each Block correctly refer to the hash of the previous Block (Block Coherence, Contiguity)

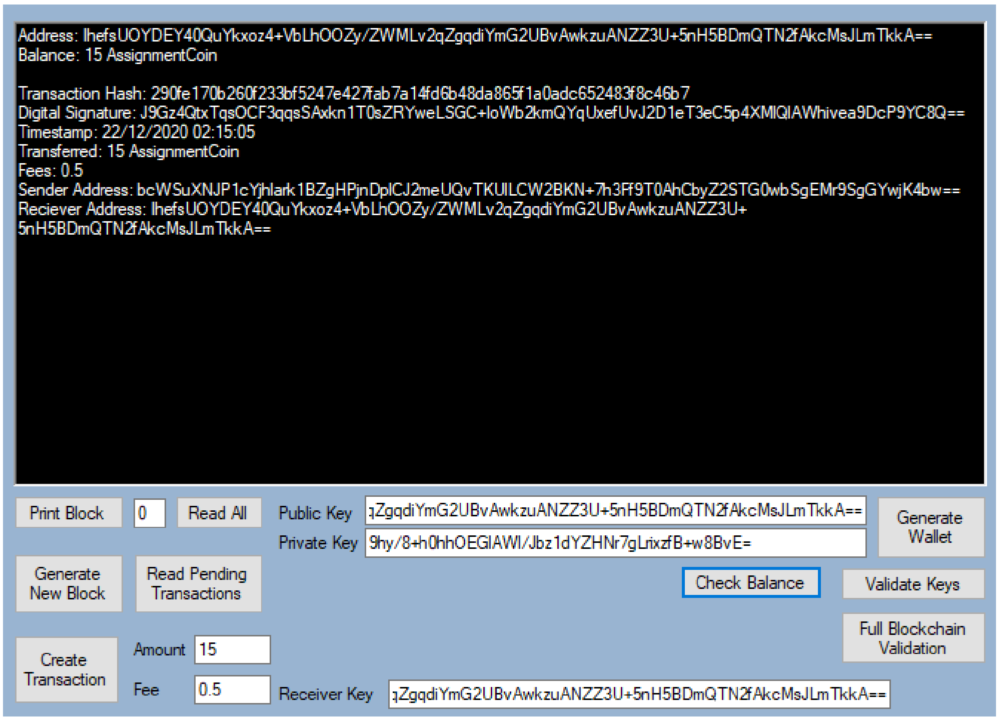
We can now include a new button within the UI that will perform full validation of the Blockchain. For now, you can make a method that iterates through the Blocks in the chain and checks the hashes of each block with its previous block to see if they match. This is to that each previous hash reference matches the hash of its previous block (the block that immediately preceded it; came just before it in the chain) If all Blocks pass (match their respective precursor block) , then this method must return a success message; otherwise return a failure message.

Checking and Validating Balance

So far, we have had no way to check the balance of a wallet, and we do not have checks before transactions are made to see if a wallet can afford to spend the funds needed to be sent to another wallet for given transaction.   We can include another button in the UI that checks the balance of a given wallet and include further checks within the transaction generation logic to ensure that the sender has sufficient funds before a transaction can be permitted to proceed.

(Remember currency is generated through mining so all wallets start with zero)

Tip: You will need to create code that goes through all Blocks and searches for transactions involving the wallet you are checking.



```
Address: IhefsUOYDEY40QuYkxoz4+VbLhOOZy/ZWMLv2qZgqdiYmG2UBvAwkzuANZZ3U+5nH5BDmQTN2fAkcMsJLmTkkA==
Balance: 15 AssignmentCoin

Transaction Hash: 290fe170b260f233bf5247e427fab7a14fd6b48da865f1a0adc652483f8c46b7
Digital Signature: J9Gz4QtxTqsOCF3qqsSAxkn1T0sZRYweLSGC+loWb2kmQYqUxefUvJ2D1eT3eC5p4XMIQIAWhivea9DcP9YC8Q==
Timestamp: 22/12/2020 02:15:05
Transferred: 15 AssignmentCoin
Fees: 0.5
Sender Address: bcWSuXNJP1cYjhlark1BZgHPjnDplCJ2meUQvTKUILCW2BKN+7h3Ff9T0AhCbyZ2STG0wbSgEMr9SgGYwjK4bw==
Reciever Address: IhefsUOYDEY40QuYkxoz4+VbLhOOZy/ZWMLv2qZgqdiYmG2UBvAwkzuANZZ3U+
5nH5BDmQTN2fAkcMsJLmTkkA==
```

Public Key: ıZgqdiYmG2UBvAwkzuANZZ3U+5nH5BDmQTN2fAkcMsJLmTkkA==

Private Key: 9hy/8+h0hhOEGIAWI/Jbz1dYZHNr7gLrixzfB+w8BvE=

Amount: 15

Fee: 0.5

Receiver Key: ıZgqdiYmG2UBvAwkzuANZZ3U+5nH5BDmQTN2fAkcMsJLmTkkA==

Check balance showing the balance of an address along with the transactions in which the address was involved.

Validating Blocks & Merkle Root

So far, we have had have no verification method in the Blocks to check if the transactions within it have been tampered with   Merkle root is a scalable and efficient method to encode data on the Blocks in an efficient manner. We can use the Merkle root algorithm to encode the transactions within a Block to a single hash. You can now include a new data field within *Block.cs*, as *merkleRoot. We can* include a new method within *Block.cs* to calculate the Merkle root too.

The Merkle root algorithm combines the hashes of multiple transactions iteratively, until only one resulting hash remains after the hash of the latest block has been included. Consider the following example for how Merkle root is implemented:

- A Block has five transactions within: T_1, T_2, T_3, T_4, and T_5.
- The hash of T_1 and T_2 are combined to make $H_{1,2}$
- The hash of T_3 and T_4 are combined to make $H_{3,4}$
- The hash of T_5 is left untouched for now to make H_5
- Now the hashes $H_{1,2}$ and $H_{3,4}$ are combined to make $H_{1,2,3,4}$
- Finally, the hashes $H_{1,2,3,4}$ and H_5 are combined to make $H_{1,2,3,4,5}$
- $H_{1,2,3,4,5}$ is the final hash, and therefore it is the Merkle root.

Now implement the Merkle root method in code. To help with this task you can make use of *HashTools.cs* made available for you in the *HashCode* folder. Use the method

combineHash(hash1,hash2) to combine the hashes of transactions.

Tip: It may be useful to make the Merkle root algorithm *static*, as it will be useful for verification.

Once the Merkle root algorithm and assignment are completed, Include the Merkle root validation with the validation method you made earlier. (Please do not forget to include the Merkle root within the Read() method in *Block.cs)*

We do not just perform comparison of the hash of the Blocks for the purpose of Proof-of-Work [i.e.to check if a miner has submitted a block that is i) carries the content of all previous blocks and ii) satisfies the difficulty level, iii) has included all the transactions] we also use it to ensure that none of the data in a Block has been tampered with. As hashing algorithms **always** create the same hash given the same input, we can rehash the contents of a Block and compare the result with which hash ? to check that the provided hash for the Block matches the combinatorial hash of all the previous blocks (their Merkle root. If the hashes do not match, then it is likely that some data in some Block(s) has been tampered with. If this happens the entire Block has to be rejected. So now we need to include this facility within our full Blockchain validation method, to check if the hash of a Block submitted by a miner does match the expected reference hash (who computed that, can they be trusted, who validates them?) indeed hash provided is correct.

Finally we can include the *merkleRoot* data property within the SHA256 hashing method that generates a Block hash. This means if any transactions have been tampered with, then the Merkle Root will be different from what? , and as a result the hash of the Block will be wrong.

Validating Transactions

Similarly to the last section, here we can include checks within the validation method to recalculate the hash of the transactions to arrive at the hash of a given block and compare it with a given hash of transaction within a submitted block. In addition to this we can also include logic to verify the signature provided in the transaction, to check to see that i the transaction has been signed off with the correct private key i.e. to authenticate the person making the transaction . For this purpose, *Wallet.cs* has a verification method that you can make use of: *ValidateSignature(publicKey, hash, signature).*

Testing the validation

You can test the validation methods that you have implemented in various ways. For example you can try some of the following steps

- Sending a transaction with address that does not have enough funds to support it
- Sending a transaction and providing an invalid private key

- Creating a button/method that generates invalid Blocks with incorrect features – e.g. wrong previous hash, wrong hash, invalid transactions within the block.

Part 6 – Assignment Tasks

This part will consist of **4 sections,** to earn full marks for **implementation** (15) in this part of the assignment you only need to include in your submission correct implementations of **three** out of **four** of these sections. T

This part carries 35 marks overall, 15 for implementation and 20 for report. You can earn the full 20 marks for the report section if you only implement **three** sections. However you may be able to pick up lost marks due to incomplete implementation of any three if you choose to implement all **four** (You cannot score more than 35 for this section)**.**

Research and reference to other Blockchain implementations is valued highly for this section, especially if you incorporate their logic into your solution.

Threading during Proof-of-Work

Currently when a node performs Proof-of-Work to find a hash to satisfy the difficulty, only one core in the CPU is working, the rest are idle.

1) Extend the Proof-of-Work algorithm to parallelize this task such that multiple threads are in use. For the report, document the implementation and prove that there is a performance efficiency gain when using multiple threads as opposed to one

Hint: Take multiple samples of mining times and compare them; make use of

system.Diagnostics.Stopwatch).

Do not forget to consider the fact that threads may repeat work (hashing the same data and creating hashes that have already been generated). Provide a solution to this problem and explain in the report how it prevents work from being repeated.

Tip: The following C# resources may help implementation: Callbacks/Delegates and Threading. If you have problems updating the UI using threads the following line may be useful:

```
textBox1.Invoke(new Action(() => textBox1.Text += message))
```

Adjusting the Difficulty Level in Proof-of-Work

As previously mentioned in this brief, the difficulty level is a static type right now and so does not change from Block to Block. In existing Blockchain (Proof-of-Work) implementations a dynamic difficulty level is used. In Proof-of-Work crypto-currencies 'Block Time' is considered the average amount of time required until the next Block is added to the chain. In Bitcoin this is 10 minutes and in Ethereum this is 10-20 seconds.

2) Decide your own 'Block time' (with justification in the report) and implement an adaptive difficulty level algorithm. Prove that the implementation works in the report and discuss how you developed your solution for your implementation.

Note: In our current implementation increasing the difficulty by one would increase the amount of work by a factor of 16. This is not suitable for dynamic difficulty level setting. You would need to consider another approach; you may wish to review the existing approaches.

In your report, please detail how you developed your implemented dynamic difficulty level and why you chose a particular approach while also providing evidence that it works. Also state what the duration of your 'Block Time' was and justify why.

Mining Settings

Currently, when Blocks are added to the chain, they pick transactions from the transaction pool arbitrarily. In reality, there are many ways that a node may wish to pick transactions from the pool. They may wish to be altruistic and pick the transactions that have been waiting the longest. They may wish to be greedy and pick the transactions with the largest fees. They may wish to be unpredictable and pick entirely randomly. They may have their own transactions pending and choose to pick up transactions involving their address first.

3) Implement a setting within the UI to enables the node to decide how it wishes to pick transactions from the pool. Include "Greedy" (highest fee first), "Altruistic" (longest wait first), "Random" and Address Preference.

In your report discuss what you believe would be the optimal setting to choose and why. Also discuss how to choose to implement the settings and provide evidence that each setting works.

Your own idea

4) You are welcome, within the scope of this work, to come up with your own extension/change to add to the application. This change cannot be trivial and should take a similar amount of implementation time to the other 3 sections. Some ideas, for example include:

- The implementation of a different consensus algorithm (Proof-of-Stake),
- Creating multiple nodes running in a local network, automating the generation of transactions,
- Smart contracts (if you are really ambitious).

You will need to justify your choice to add any extension to the application, document your implementation and provide proof that it works as intended.

****