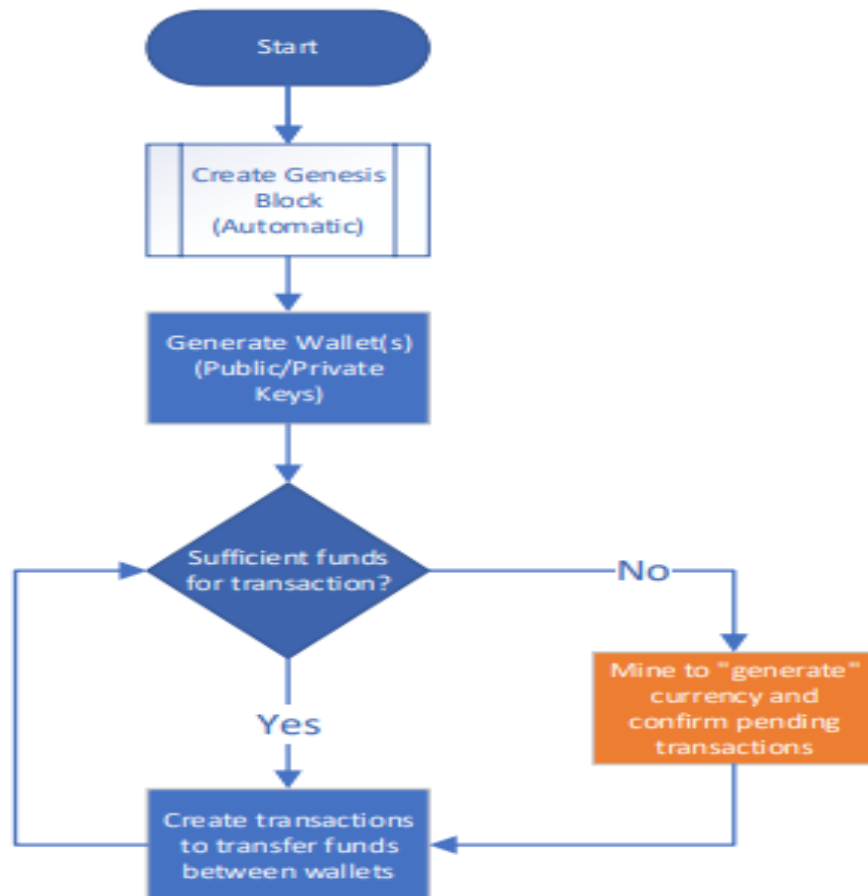


Blockchain Build

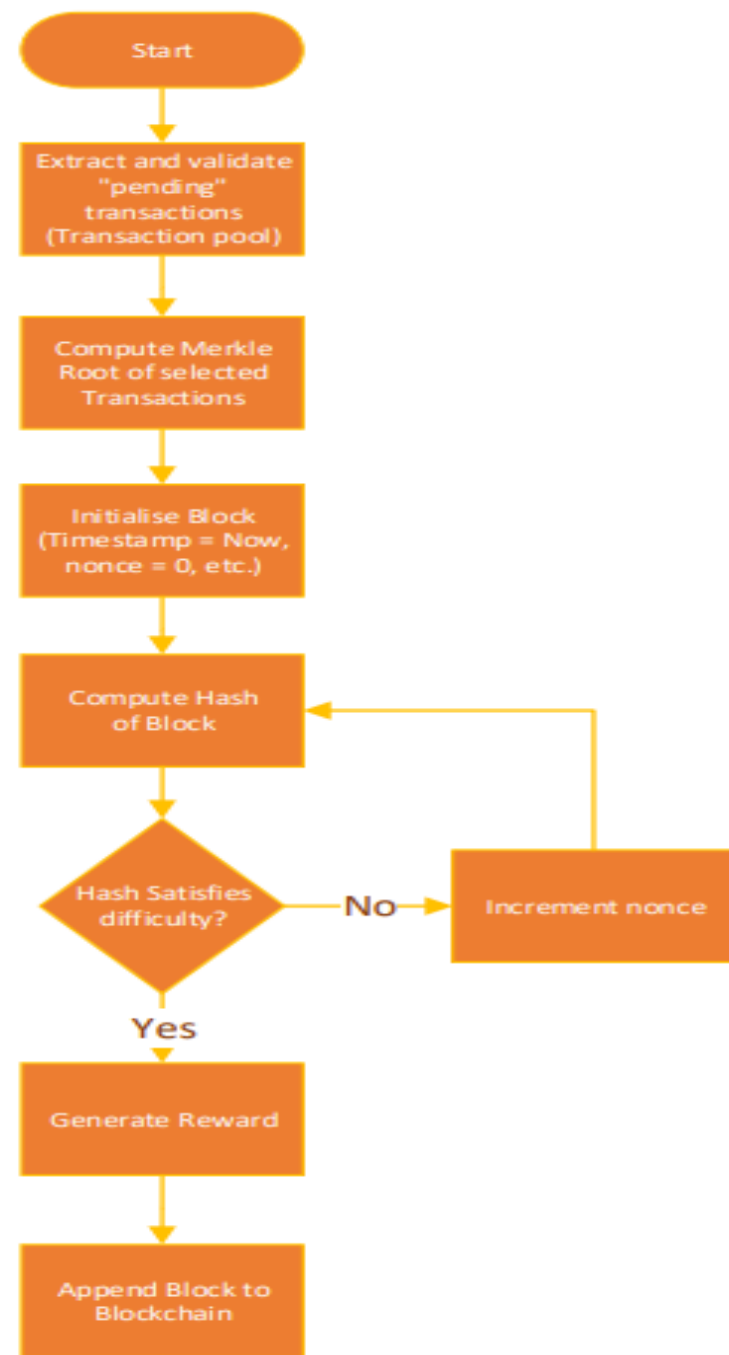
Flowchart and

Fully commented code

UI Interaction (Blockchain Usage)



Mining (Proof-of-Work)



Key

Part 1 – Project Setup

Part 2 – Blocks and the Blockchain

Part 3 – Transactions and Digital Signatures

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

Part 5 – Validation

Throughout these documents, the colours chosen highlight sections of the code that relate to different parts of the assignment tasks/lab script (Parts 1-5), this way it is easy to follow the steps taken to arrive at the final solution and it separates the different topics/themes/concepts for which students can then refer back to the tutorials for more information.

The code is organised by class (from the smallest to the largest element of the hierarchy i.e. Transaction->Block->Blockchain->Application) and each class is comprised of variables, constructors (initialising the variables) and then functions (operating on these values). I have indented these to make them stand out and easier to read.

```

/*****
** TRANSACTION.CS
** Class representing transactions
**/
using System;
using System.Collections.Generic;
using System.Linq;
using System.Security.Cryptography;
using System.Text;
using System.Threading.Tasks;

namespace BlockchainAssignment
{
    class Transaction
    {
        /* Transaction Variables */

        DateTime timestamp;           // Time of creation
        public String senderAddress,   // Participants public key addresses
            recipientAddress;
        public double amount, fee;     // Quantities transferred
        public String hash, signature; // Attributes for verification of validity
    }
}

```

/* Transaction Constructor */

```
public Transaction(String from, String to, double amount, double fee, String privateKey)
{
    this.timestamp = DateTime.Now;

    this.senderAddress = from;
    this.recipientAddress = to;

    this.amount = amount;
    this.fee = fee;

    this.hash = CreateHash();
    this.signature = Wallet.Wallet.CreateSignature(from, privateKey, hash);
}
```

// Hash the transaction attributes
// Sign the hash with the senders private key ensuring validity

/* Transaction Functions */

```
public String CreateHash()
{
    String hash = String.Empty;
    SHA256 hasher = SHA256Managed.Create();

    String input = timestamp + senderAddress + recipientAddress + amount + fee;
    Byte[] hashByte = hasher.ComputeHash(Encoding.UTF8.GetBytes(input));
    foreach (byte x in hashByte)
        hash += String.Format("{0:x2}", x);

    return hash;
}
```

// Hash the transaction attributes using SHA256

// Concatenate all transaction properties
// Apply the hash function to the "input" string

// Reformat to a string

```
public override string ToString()
{
    return "[TRANSACTION START]"
        + "\n Timestamp: " + timestamp
        + "\n -- Verification --"
        + "\n Hash: " + hash;
}
```

// Represent a transaction as a string for output to UI

```
+ "ln Signature: " + signature
+ "ln -- Quantities --"
+ "ln Transferred: " + amount + " Assignment Coin"
+ "ln Fee: " + fee
+ "ln -- Participants --"
+ "ln Sender: " + senderAddress
+ "ln Reciever: " + recipientAddress
+ "ln [TRANSACTION END]";
```

```

/*****
**/
/** BLOCK.CS
/** Class representing a Block instance
/*****
using System;
using System.Collections.Generic;
using System.Linq;
using System.Security.Cryptography;
using System.Text;
using System.Threading.Tasks;

namespace BlockchainAssignment
{
    class Block
    {
        /* Block Variables */
        private DateTime timestamp;           // Time of creation

        private int index,                    // Position of the block in the sequence of blocks
            difficulty = 4;                   // An arbitrary number of 0's to proceed a hash value

        public String prevHash,               // A reference pointer to the previous block
            hash,                             // The current blocks "identity"
            merkleRoot,                       // The merkle root of all transactions in the block
            minerAddress;                     // Public Key (Wallet Address) of the Miner

        public List<Transaction> transactionList; // List of transactions in this block

        // Proof-of-work
        public long nonce;                    // Number used once for Proof-of-Work and mining

        // Rewards
        public double reward;                // Simple fixed reward established by "Coinbase"

        /* Block constructors */

        public Block()                       // Genesis Block

```

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

```
/* Block Variables */
```

```
/* Block constructors */
```

```

{
    this.timestamp = DateTime.Now;
    this.index = 0;
    this.transactionList = new List<Transaction>();
    this.hash = Mine();
}

public Block(Block lastBlock, List<Transaction> transactions, String minerAddress) // Standard Block
{
    this.timestamp = DateTime.Now;

    this.index = lastBlock.index + 1;
    this.prevHash = lastBlock.hash;

    this.minerAddress = minerAddress; // The wallet to be credited the reward for the mining effort
    this.reward = 1.0; // Assign a simple fixed value reward

    transactions.Add(createRewardTransaction(transactions)); // Create and append the reward transaction
    this.transactionList = new List<Transaction>(transactions); // Assign provided transactions to the block

    this.merkleRoot = MerkleRoot(transactionList); // Calculate the merkle root of the blocks transactions
    this.hash = Mine(); // Conduct PoW to create a hash which meets the given difficulty requirement
}

```

/* Block Functions */

```

public String CreateHash() // Hashes the entire Block object
{
    String hash = String.Empty;
    SHA256 hasher = SHA256Managed.Create();

    String input = timestamp.ToString() + index + prevHash + nonce + merkleRoot; // Concatenate all of the blocks properties including nonce as to generate a new hash on each call
    Byte[] hashByte = hasher.ComputeHash(Encoding.UTF8.GetBytes(input)); // Apply the SHA hash function to the block as represented by the string "input"

    */
    foreach (byte x in hashByte)
        hash += String.Format("{0:x2}", x); // Reformat to a string
}

```

```

    return hash;
}

```

```

public String Mine()                // Create a Hash which satisfies the difficulty level required for PoW
{
    nonce = 0;                      // Initialise the nonce
    String hash = CreateHash();      // Hash the block

    String re = new string('0', difficulty); // A string representing the "difficulty" for analysing the PoW requirement

    while(!hash.StartsWith(re))      // Check the resultant hash against the "re" string
    {
        nonce++;                    // Increment the nonce should the difficulty level not be satisfied
        hash = CreateHash();        // Rehash with the new nonce as to generate a different hash
    }

    return hash;                    // Return the hash meeting the difficulty requirement
}

```

```

public static String MerkleRoot(List<Transaction> transactionList) // Merkle Root Algorithm - Encodes transactions within a block into a single hash
{
    List<String> hashes = transactionList.Select(t => t.hash).ToList(); // Get a list of transaction hashes for "combining"

    // Handle Blocks with...
    if (hashes.Count == 0) // No transactions
    {
        return String.Empty;
    }
    if (hashes.Count == 1) // One transaction - hash with "self"
    {
        return HashCode.HashTools.combineHash(hashes[0], hashes[0]);
    }
    while (hashes.Count != 1) // Multiple transactions - Repeat until tree has been traversed
    {
        List<String> merkleLeaves = new List<String>(); // Keep track of current "level" of the tree

        for (int i=0; i<hashes.Count; i+=2) // Step over neighbouring pair combining each

```



```

    }
    if (i == hashes.Count - 1)
    {
        merkleLeaves.Add(HashCode.HashTools.combineHash(hashes[i], hashes[i])); // Handle an odd number of leaves
    }
    else
    {
        merkleLeaves.Add(HashCode.HashTools.combineHash(hashes[i], hashes[i + 1])); // Hash neighbours leaves
    }
}
hashes = merkleLeaves; // Update the working "layer"
return hashes[0]; // Return the root node
}

public Transaction createRewardTransaction(List<Transaction> transactions) // Create reward for incentivising the mining of block
{
    double fees = transactions.Aggregate(0.0, (acc, t) => acc + t.fee); // Sum all transaction fees
    return new Transaction("Mine Rewards", minerAddress, (reward + fees), 0, ""); // Issue reward as a transaction in the new block
}

public override string ToString() // Concatenate all properties to output to the UI
{
    return "[BLOCK START]"
        + "\nIndex: " + index
        + "\ntimestamp: " + timestamp
        + "\nPrevious Hash: " + prevHash
        + "\n-- PoW --"
        + "\nDifficulty Level: " + difficulty
        + "\nNonce: " + nonce
        + "\nHash: " + hash
        + "\n-- Rewards --"
        + "\nReward: " + reward
        + "\nMiners Address: " + minerAddress
        + "\n-- " + transactionList.Count + " Transactions --"
        + "\nMerkle Root: " + merkleRoot
        + "\n" + String.Join("\n", transactionList)
}

```

```

    + "\n[BLOCK END]";
}
}

/*****
/** BLOCKCHAIN.CS
/** Class representing the entire Blockchain
*****/

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

namespace BlockchainAssignment
{
    class Blockchain
    {
        /** Blockchain Attributes */

        public List<Block> blocks; // List of block objects forming the blockchain
        private int transactionsPerBlock = 5; // Maximum number of transactions per block
        public List<Transaction> transactionPool = new List<Transaction>(); // List of pending transactions to be mined

        /** Blockchain Constructor */
        public Blockchain() // Initialises the list of blocks and generates the genesis block
        {
            this.blocks = new List<Block>()
            {
                new Block() // Create and append the Genesis Block
            };
        }
    }
}

```

/* Blockchain Functions */

```
public String GetBlockAsString(int index)           // Prints the block at the specified index to the UI
{
    if (index >= 0 && index < blocks.Count)         // Check if referenced block exists

        return blocks[index].ToString();           // Return block as a string
    else
        return "No such block exists";
}

public Block GetLastBlock()                         // Retrieves the most recently appended block in the blockchain
{
    return blocks[blocks.Count - 1];
}

public List<Transaction> GetPendingTransactions()   // Retrieve pending transactions and remove from pool
{
    int n = Math.Min(transactionsPerBlock, transactionPool.Count); // Determine the number of transactions to retrieve dependent on the number of pending transactions and the limit specified

    List<Transaction> transactions = transactionPool.GetRange(0, n); // "Pull" transactions from the transaction list (modifying the original list)

    transactionPool.RemoveRange(0, n);

    return transactions;                             // Return the extracted transactions
}

public static bool ValidateHash(Block block)        // Check validity of a blocks hash by recomputing the hash and comparing with the mined value
{
    String rehash = block.CreateHash();
    return rehash.Equals(block.hash);
}

public static bool ValidateMerkleRoot(Block block) // Check validity of the merkle root by recalculating the root and comparing with the mined value
```

```

    {
        String reMerkle = Block.MerkleRoot(block.transactionList);
        return reMerkle.Equals(block.merkleRoot);
    }

    public double GetBalance(String address) // Check the balance associated with a wallet based on the public key
    {
        double balance = 0; // Accumulator value for current Wallet

        foreach(Block block in blocks) // Loop through all approved transactions in order to assess account balance
        {
            foreach(Transaction transaction in block.transactionList)
            {
                if (transaction.recipientAddress.Equals(address))
                {
                    balance += transaction.amount; // Credit funds received
                }
                if (transaction.senderAddress.Equals(address))
                {
                    balance -= (transaction.amount + transaction.fee); // Debit payments placed
                }
            }
        }

        return balance;
    }
}

```

```

    public override string ToString() // Output all blocks of the blockchain as a string
    {
        return String.Join("\n", blocks);
    }
}

```

```

/*****
/** BLOCKCHAINAPP.CS
/** Class for handling user interactions with the blockchain via the Windows Form **
*****/

using System;
using System.Collections.Generic;
using System.Windows.Forms;

namespace BlockchainAssignment
{
    public partial class BlockchainApp : Form
    {
        // Global blockchain object
        private Blockchain blockchain;

        // Default App Constructor
        public BlockchainApp()
        {
            // Initialise UI Components
            InitializeComponent();
            // Create a new blockchain
            this.blockchain = new Blockchain();
            // Update UI with an initialisation message
            UpdateText("New blockchain initialised!");
        }

        /* PRINTING */
        // Helper method to update the UI with a provided message
        private void UpdateText(String text)
        {
            output.Text = text;
        }

        // Print entire blockchain to UI
        private void ReadAll_Click(object sender, EventArgs e)
        {

```

```

        UpdateText(blockchain.ToString());
    }

    // Print Block N (based on user input)
    private void PrintBlock_Click(object sender, EventArgs e)
    {
        if (int32.TryParse(blockNo.Text, out int index))
            UpdateText(blockchain.GetBlockAsString(index));
        else
            UpdateText("Invalid Block No.");
    }

    // Print pending transactions from the transaction pool to the UI
    private void PrintPendingTransactions_Click(object sender, EventArgs e)
    {
        UpdateText(String.Join("\n", blockchain.transactionPool));
    }

    /* WALLETS */
    // Generate a new Wallet and fill the public and private key fields of the UI
    private void GenerateWallet_Click(object sender, EventArgs e)
    {
        Wallet.Wallet myNewWallet = new Wallet.Wallet(out string privKey);

        publicKey.Text = myNewWallet.publicID;
        privateKey.Text = privKey;
    }

    // Validate the keys loaded in the UI by comparing their mathematical relationship
    private void ValidateKeys_Click(object sender, EventArgs e)
    {
        if (Wallet.Wallet.ValidatePrivateKey(privateKey.Text, publicKey.Text))
            UpdateText("Keys are valid");
        else
            UpdateText("Keys are invalid");
    }

    // Check the balance of current user
    private void CheckBalance_Click(object sender, EventArgs e)
    {

```

```
UpdateText(blockchain.GetBalance(publicKey.Text).ToString() + " Assignment Coin");
```

/* TRANSACTION MANAGEMENT */

```
// Create a new pending transaction and add it to the transaction pool
```

```
private void CreateTransaction_Click(object sender, EventArgs e)
```

```
{  
    Transaction transaction = new Transaction(publicKey.Text, receiver.Text, Double.Parse(amount.Text), Double.Parse(fee.Text), privateKey.Text);  
    blockchain.transactionPool.Add(transaction);  
    UpdateText(transaction.ToString());  
}
```

/* BLOCK MANAGEMENT */

```
// Conduct Proof-of-work in order to mine transactions from the pool and submit a new block to the Blockchain
```

```
private void NewBlock_Click(object sender, EventArgs e)
```

```
{  
    // Retrieve pending transactions to be added to the newly generated Block
```

```
    List<Transaction> transactions = blockchain.GetPendingTransactions();
```

```
    // Create and append the new block - requires a reference to the previous block, a set of transactions and the miners public address (For the reward to be issued)
```

```
    Block newBlock = new Block(blockchain.GetLastBlock(), transactions, publicKey.Text);
```

```
    blockchain.blocks.Add(newBlock);
```

```
    UpdateText(blockchain.ToString());  
}
```

/* BLOCKCHAIN VALIDATION */

```
// Validate the integrity of the state of the Blockchain
```

```
private void Validate_Click(object sender, EventArgs e)
```

```
{  
    // CASE: Genesis Block - Check only hash as no transactions are currently present
```

```
    if (blockchain.blocks.Count == 1)
```

```
    {  
        if (!Blockchain.ValidateHash(blockchain.blocks[0])) // Recompute Hash to check validity
```

```
            UpdateText("Blockchain is invalid");
```

```
        else
```

```
            UpdateText("Blockchain is valid");
```

```
        return;
```

```

    }
    for (int i=1; i<blockchain.blocks.Count-1; i++)
    {
        if(
            blockchain.blocks[i].prevHash != blockchain.blocks[i - 1].hash || // Check hash "chain"
            !Blockchain.ValidateHash(blockchain.blocks[i]) || // Check each blocks hash
            !Blockchain.ValidateMerkleRoot(blockchain.blocks[i]) // Check transaction integrity using Merkle Root
        )
        {
            UpdateText("Blockchain is invalid");
            return;
        }
    }
    UpdateText("Blockchain is valid");
}
}
}

```


**Next is How
Your Report
Will be Marked
and Feedback provided**

Aspect	Description	Potential Marks Rewarded		Notes
Student ID Number xxxxxxxx		Implementation	Documentation	
Practical 1 - Project Setup				
Customising the User-Interface	Addition of Buttons and Text Boxes	1	1	Windows Forms and User Interface design
Event Handlers	Implementation of Event Handler	1	1	Handling user input
		2	2	Sub-total for the above criteria
Practical 2 - Blocks and Blockchain				
Block and Blockchain Class Structure	Appropriate variables and data-types including list data-structure	1	1	Data-type justifications
Instantiation of a new Blockchain	Object definition and initialisation in Blockchain app	1	1	Class hierarchy description
Genesis Block Creation	Necessary constructor modifications	1	1	Special properties of Genesis blocks
Hashing	Hashing the entire block using the SHA256 algorithm	1	1	Description of hashing and hash properties
Printing Blocks	Outputting hashes as hexadecimal strings in the UI	1	0	Description missing
		5	4	Sub-total for the above criteria
Practical 3 - Transactions and Digital Signatures				
Wallet Creation	Asymmetric key generation and UI adaptation	1	1	Key usage and mathematical relationship/properties
Setting up Transactions	Transaction class implementation - Variables and constructor	1	0	Description missing
Digital Signature Creation	Signing the Hash using Senders Private Key	1	1	Use in authentication of transactions
Processing Transactions	Generate a transaction and printing the data	1	0	Description missing
Transaction Pools	Implemented as a list of "pending" transactions	1	1	Creating and managing Transaction Pools in Blockchains
		5	3	Sub-total for the above criteria
Practical 4 - Consensus Algorithms (Proof-of-Work)				
Generating new Blocks	Adding "Empty" Blocks to a Blockchain	2	2	Blockchain-Block relationship
Adding transactions into Blocks	Transaction Lists	2	2	Block composition
Proof-of-Work	Algorithm Implementation	2	2	Properties, Advantages and Disadvantages etc.
Nonce Generation	Random Number Generation	1	1	Requirement for nonce in Blocks
Difficulty Level	Value selection and checking	1	1	Justification of value selected
Rewards and Fees	Coinbase configuration	2	2	Mining and Incentives: Driving transactions
		10	10	Sub-total for the above criteria
Practical 5 – Validation				

Validating the Blockchain structure	Block Coherence and contiguity checks
Checking and Validating Balances	Ledger Tracing
Validating Blocks and Merkle Root	Implementing Merkle Root Algorithm - Combining Hashes
Validating Transactions	Checking digital signatures
Testing the Validation	Verification

2
2
2
2
2
10

Assignment Tasks	
Task 1 - Extending Proof-of-Work	
Multi-threading	Callbacks/delegates and threading
e-Nonce	Additional nonce generation
Sampling	Comparative study

3
1
1
5

Task 2 - Adjusting Difficulty Level for Proof-of-Work	
Block Time Measurement	Calculation of "Block Time"
Adaptive Difficulty	Adaptive Difficulty Algorithm Implementation
Block Time Selection	Adaptivity etc.

1
3
1
5

Task 3 - Implementing Alternative Mining Preference Settings	
Pool Adaptation	Use-cases
Greedy	Highest Fee
Altruistic	Longest Wait
Random	Random Selection
Address Preference	Owner

1
1
1
1
1
5

Task 4 - Other Extension or Modification	
Other	Alternate consensus algorithm implementation, networking, smart contracts etc.

5
5

2	How trustability is achieved
2	Double spend prevention
2	Merkle root properties and benefits
2	Authenticity and Integrity achieved as a result of usage
2	Incorporation of "rules"

10 Sub-total for the above criteria

3	Increasing the rate in which nodes mine blocks
1	Overcoming "Duplication of work" in parallelised systems
1	Performance comparisons

5 Sub-total for the above criteria

1	Using "Block Time" as a metric
3	Evidence of background reading
1	Justification of design and implementation

5 Sub-total for the above criteria

1	Potential use case for each preference
1	√
1	√
1	√
1	√

5 Sub-total For the above criteria

5	Design and Implementation justification
---	---

5 Sub-total for the above criteria

Total Assessment Mark

44

Percentage Mark

93.62%

Practical Exercises and Coursework Support

Blockchain Lab Practical Guided Exercises to Support the Coursework Assignment

Contents

Introduction	Error! Bookmark not defined.
Practical Guided Exercises and Assignment	Error! Bookmark not defined.
Marking Scheme	Error! Bookmark not defined.
Part 1 – Project Setup	Error! Bookmark not defined.
Setup.....	Error! Bookmark not defined.
Customising the UI	Error! Bookmark not defined.
Part 2 – Blocks and the Blockchain.....	Error! Bookmark not defined.
Creating the classes and their contents	Error! Bookmark not defined.
Hashing and creating the Genesis Block.....	Error! Bookmark not defined.
Printing a Block to the UI	Error! Bookmark not defined.
Part 3 – Transactions and Digital Signatures	Error! Bookmark not defined.
Wallets & Private and Public Key Generation.....	Error! Bookmark not defined.
Setting up Transactions	Error! Bookmark not defined.
Processing Transactions & Transaction Pools.....	Error! Bookmark not defined.
Part 4 – Consensus Algorithms (Proof-of-Work).....	Error! Bookmark not defined.
Generating new Blocks	Error! Bookmark not defined.
Adding transactions into Blocks	Error! Bookmark not defined.
Proof-of-Work	Error! Bookmark not defined.
Difficulty (Level)	Error! Bookmark not defined.
Rewards & Fees.....	Error! Bookmark not defined.
Part 5 – Validation	Error! Bookmark not defined.
Validating the Blockchain Structure	Error! Bookmark not defined.
Checking and Validating Balance	Error! Bookmark not defined.
Validating Blocks & Merkle Root.....	Error! Bookmark not defined.
Validating Transactions	Error! Bookmark not defined.
Testing the validation	Error! Bookmark not defined.
Part 6 – Assignment Tasks	Error! Bookmark not defined.
Threading during Proof-of-Work.....	Error! Bookmark not defined.
Adjusting the Difficulty Level in Proof-of-Work.....	Error! Bookmark not defined.
Mining Settings.....	Error! Bookmark not defined.
Your own idea	Error! Bookmark not defined.

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:

Table 1: Marking Scheme

	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-*, and *Wallet-* folders.

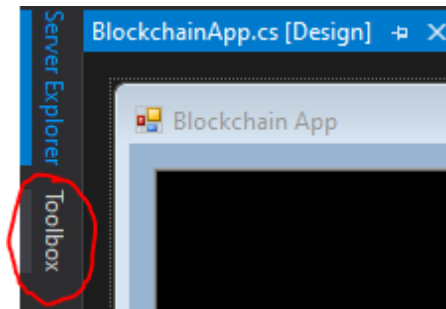
Double click the *BlockchainApp.cs* to open a Windows Form, which shows the basic UI of the App. You are free to modify it during this assignment as you see fit but ensure that the results are clear (as you are required to provide screenshots in your report). Right click *BlockchainApp.cs* and click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

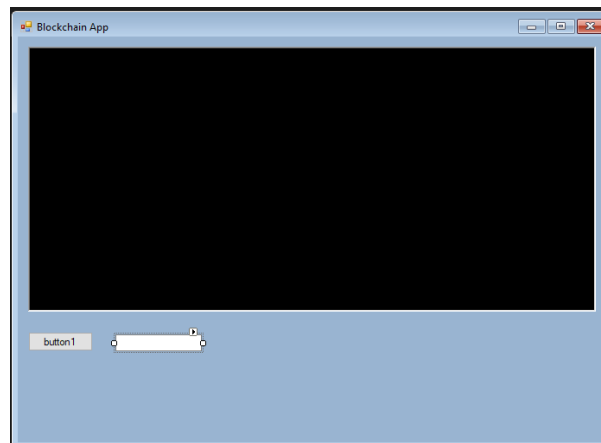
Wallet is responsible for the asymmetric encryption (public and private key pairs) and digital signatures in transaction. *HashCode* is responsible for combining hashes and converting hashes to and from byte arrays to strings both ways. This code is provided to help reduce the workload of the project, 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:

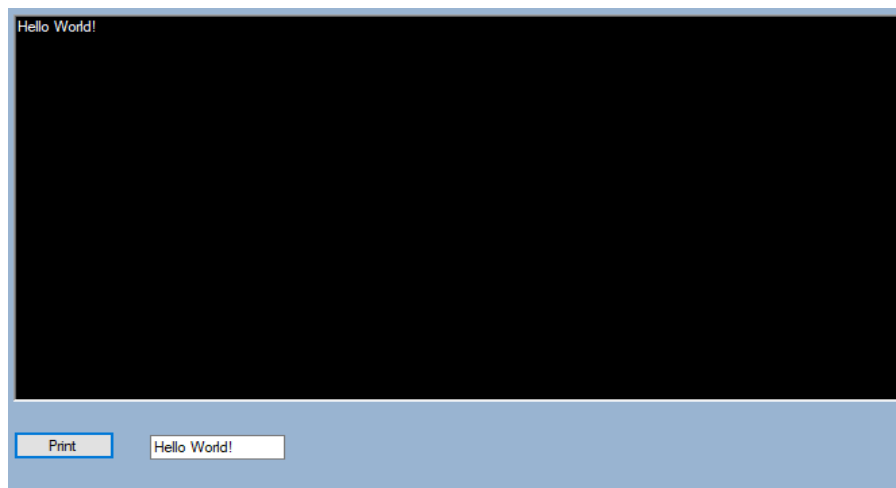


Click *button1* once to open its properties in the bottom left. Change its *text* property to say '*Print*' instead of '*button1*'. Now double click the button and you will be taken to the underlying code, a new function for clicking the button is created after double clicking it. Next you can write 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:



Note: 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
{
    0 references
    class Blockchain
    {
        List<Block> Blocks= new List<Block>();
    }
}
```

1: *Blockchain.cs* code

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

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

2: *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)
{
```

3: 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()
{
    ...
}
```

4: 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 = 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()`.

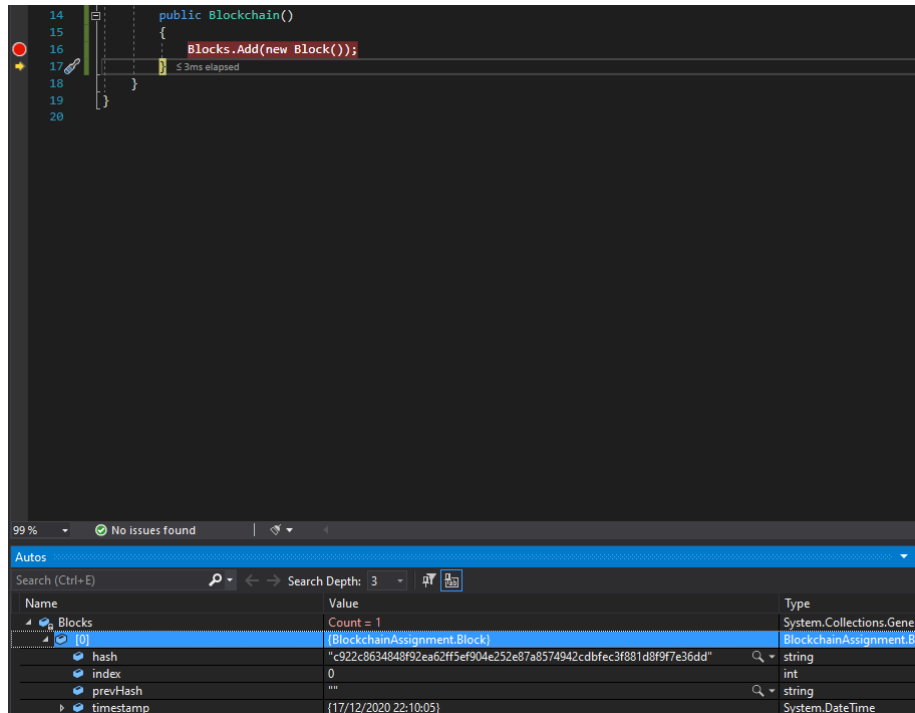
Now we have a constructor to create Blocks, but no methods that create them. As stated, the first Block in a Blockchain has no previous hash and is called the Genesis Block. To generate a Genesis Block we must create another constructor in *Block.cs*. This constructor will have no arguments, it will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



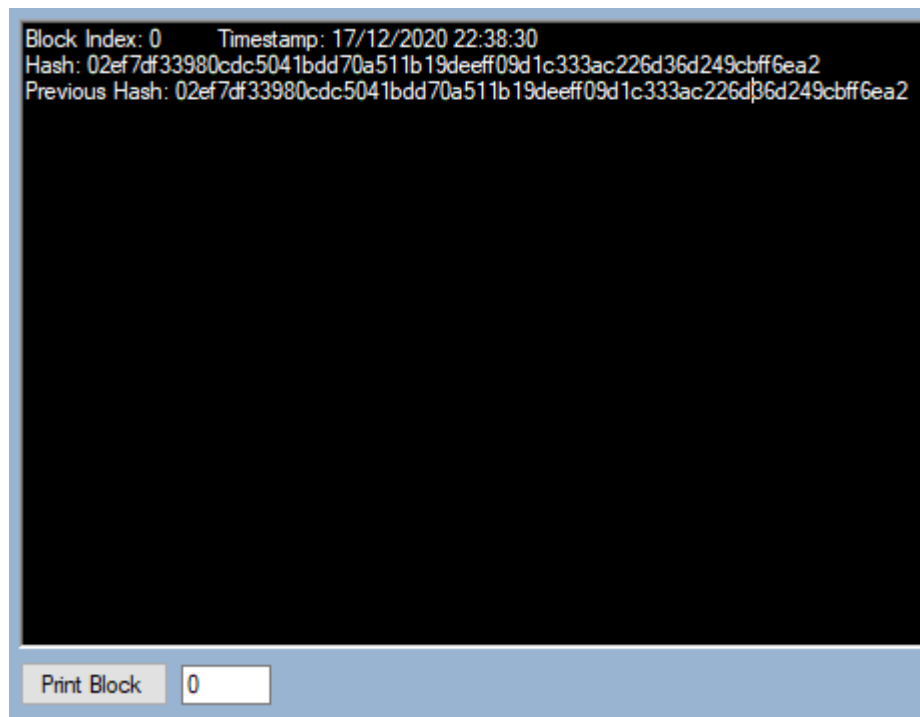
5: An example of debugging in C# that shows the contents of Blocks

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.



6: 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-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 can store 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 cryptocurrency 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.



7: 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 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.

8: 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: IffqdvV1MevQjortE+hinvT/MMILqz0sUDMdeSdcA1++5kciYsJBbrupMLzfaS6rzDhwbf2fp9N1Napn4oYQQ==
Timestamp: 18/12/2020 21:02:21
Transferred: 10 AssignmentCoin
Fees: 0.001
Sender Address: pz3JmTIUU5if6XZlkK4SeWiOajm5JSbXYqryCHHkhjDvHtyf2cxtrDvgfKvEaiWM8gNcfDYuKu6OfZix0+7NA==
Receiver Address: YdQlqpkvZHVk/767+j2U6birZhV80TMOAOPSerB3SRBmdJztOBsUwBvq2MWJ98oGaVGuyMa3xDfmYclogWIAQ==

Print Block
Public Key
Generate Wallet
Private Key
Validate Keys

Create Transaction
Amount
Fee
Receiver Key

9: 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'.

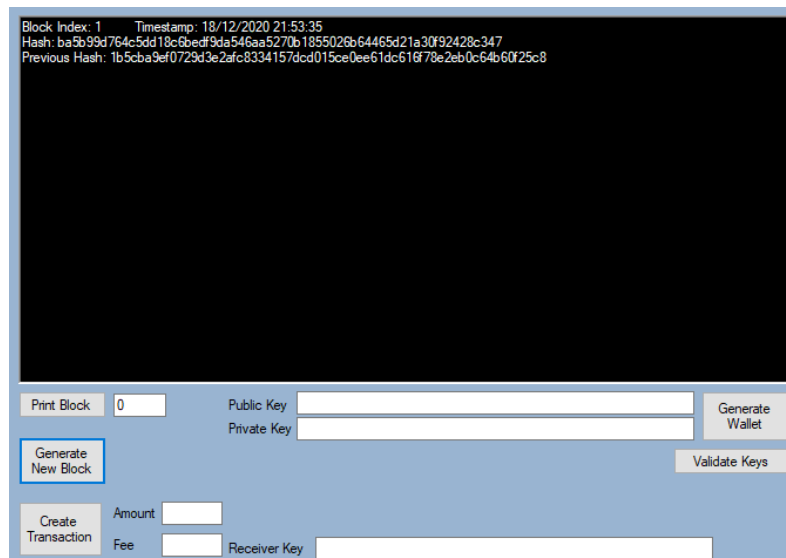
Now when we generate a transaction, we can add it to the transaction pool, so it is ready to be added to the Blockchain.

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

Generating new Blocks

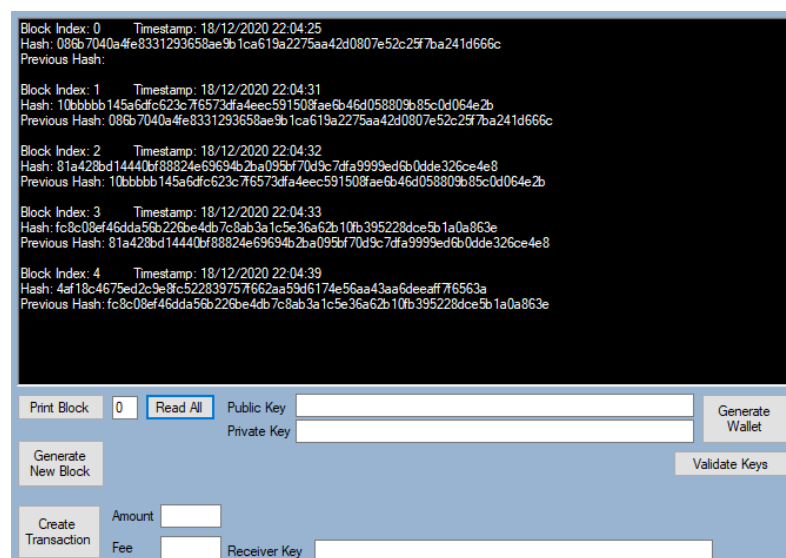
Your application should now be able to generate transactions and generate a genesis block. One Block on its own is not a Blockchain, therefore we need to generate some more blocks. We can add a button to the UI with the purpose of generating a new Blocks. When you click this button, it should call the 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.



10: 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.



11: 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

The screenshot shows a web application interface for a blockchain. At the top, a text area displays the details of a newly mined block and its two transactions. The block information includes its index (1), timestamp (21/12/2020 01:47:26), hash, previous hash, and the number of transactions (2). The first transaction details include its hash, digital signature, timestamp (21/12/2020 01:47:13), transferred amount (5 AssignmentCoin), fees (0.01), sender address, and receiver address. The second transaction details include its hash, digital signature, timestamp (21/12/2020 01:47:23), transferred amount (1 AssignmentCoin), fees (0.02), sender address, and receiver address. Below the text area, there is a control panel with several buttons: 'Print Block', 'Read All', 'Generate New Block', 'Read Pending Transactions', 'Create Transaction', 'Generate Wallet', and 'Validate Keys'. There are also input fields for 'Public Key', 'Private Key', 'Amount' (set to 1), 'Fee' (set to 0.02), and 'Receiver Key'.

working as

intended.

12: *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 Cryptocurrencies, 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

```

Print Block Read All
Public Key
Private Key
Generate Wallet

Generate New Block
Read Pending Transactions
Validate Keys

Create Transaction
Amount
Fee
Receiver Key

13: 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 cryptocurrencies the incentive comes in the form of a **reward** and **fees**.

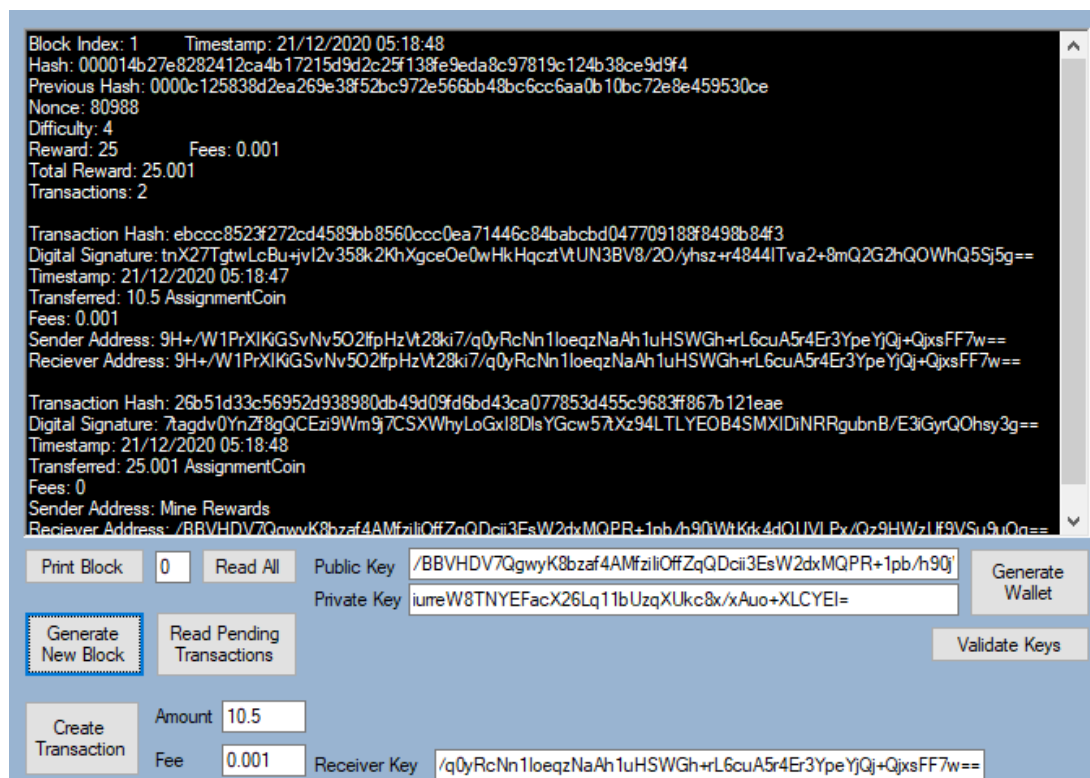
After each Block is mined in Bitcoin, a flat reward is given to the winning miner's address. This reward is given as a transaction; the receiver address is the mining node address and the sending node is technically from nowhere (as the currency is generated there and then). In Bitcoin the sender's address 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}.



14: Block with rewards implemented. Miner gets a reward of 25.001

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: 1hefsUJOYDEY40QuYxoxz4+VbLhOOZy/ZWMLv2qZgqdYmG2UBrAwkzuANZZ3U+5nH5BDmQTNZfAkMsJLmTtkA==
Balance: 15 AssignmentCoin

Transaction Hash: 290fe170b260f233bf5247e427ab7a14fd6b48da865f1a0dc652483f8c46b7
Digital Signature: J9Gz4QtxTqsoCF3qqsSAykn1T0sZRYweLSGC+loWb2kmQYqUxerUvJ2D1eT3eC5p4XMQIAWhvea9DcP9YC8Q==
Timestamp: 22/12/2020 02:15:05
Transferred: 15 AssignmentCoin
Fees: 0.5
Sender Address: bcWsuXNUP1cYh1ark18ZgHPjDpJCj2meUQvTKULCW28KN+7h3f9T0AhCbyZ25TG0wbSgEMr3SgGYwJK4bw==
Receiver Address: 1hefsUJOYDEY40QuYxoxz4+VbLhOOZy/ZWMLv2qZgqdYmG2UBrAwkzuANZZ3U+5nH5BDmQTNZfAkMsJLmTtkA==
```

Print Block 0 Read All Public Key 1ZgqdYmG2UBrAwkzuANZZ3U+5nH5BDmQTNZfAkMsJLmTtkA== Generate Wallet

Private Key 9hy/8+th0HhOEGIAWI/Jbz1dYZHn7gLrxzfB+ww8BvE=

Generate New Block Read Pending Transactions Check Balance Validate Keys

Create Transaction Amount 15 Fee 0.5 Receiver Key 1ZgqdYmG2UBrAwkzuANZZ3U+5nH5BDmQTNZfAkMsJLmTtkA== Full Blockchain Validation

15: 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 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.

Clarification re Assessment and Implementation

The only code to be added by students to the fully commented working program already provided is the additional code to be integrated within the **Mine()** routine in the program.

As the marking scheme below shows, by completing and documenting tasks 1-5, which is simply to run the program provided and presenting a report on it on the same basis as the Coursework Support Notes, a student can obtain (65%), and by completing **each of the last three tasks correctly** (requiring a few lines of code to be added) a student can obtain a further 35/3= **11.66 marks added to the 65% for Parts-1-5 if fully completed.**

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

Coursework Facts

Tasks 1-5 which cover the stages of building the mini-blockchain	65%
---	------------

Please see the sample assessment and feedback as provided to a student who took this course in the previous year. As you can see it is essential to show a comprehensive understanding of the code. Students are to submit a description of each of the steps including screenshots from their own running instance of the blockchain application to demonstrate their understanding.

Thus, students will get full marks allocated to these steps (**Yellow highlighted Parts in the marking scheme = 65%**) simply by showing their own screenshots arising from the execution of the code provided and outlining the steps, for example as illustrated by a flow chart.

All the code, including the constructors for each necessary object such as block, wallet and transaction, are integrated within the working code including the GUI using Windows Forms as provided in the practical folder to build a mini-blockchain. The document called Practical Exercises and Coursework Support and the Blockchain Cookbook PowerPoint that is based on it can be used for step-by-step creation of the mini-blockchain during laboratory sessions. The following are the normally expected screenshots to be

included in the report or in the Appendix to ensure compliance with the 7 pages limit of the main body of the report.

- 1. Screenshots of Gensis Block**
- 2. Screenshot of wallets showing key pairs**
- 3. Screenshot of a transaction with fee showing**
- 4. Screenshot of transaction with balance showing**
- 5. Screen shot of transaction Pool listing.**
- 6. Screenshot of Blocks with sufficient number of leading zeros satisfying a Degree of Difficulty of 4**
- 7. Screenshots for the results of each of Tasks 6.1, 6.2 and 6.3 as attempted as appropriate**

Part 6 (Tasks 1, 2, 3) the green highlighted tasks in the above Marking Scheme 35%

Some tutorial advice regarding the only additional coding necessary for implementing Part B tasks has been provided in lab/lecture sessions.

For Tasks 1, 2, 3 of Part 6, students can add some code of their own to the **Mine () routine** within the working code to experiment with alternative scenarios for Proof-of-Work which is the process of mining valid blocks as already implemented within the code.

The points at which this additional code could be introduced are dependent on the Tasks selected for Part 6 which are any three from those listed below:

1. Threading during Proof-of-Work
2. Adjusting the Difficulty Level in Proof-of-Work
3. Mining Settings
4. Your own idea

These are described below:

- i) **Task 1:** Multi-threading to carry out, in parallel, the block preparation (Proof-of-Work) process for different degrees of difficulty. Finding out as you increase the Degree-of-Difficulty, how high one needs to increase it, before multi-threading would give significant speed advantage in arriving at a valid block with the right number of leading zeros, noting that multi-threading requires a computational overhead in thread management, and such parallelised computation will lead to an efficiency gain only when the gain from the parallelism is over- compensated by the time taken for thread management.
- ii) **Task 2:** Develop an algorithm which periodically adjusts (re-sets) the “Degree-of-Difficulty” to match the Block-Update-Rate set for the next cycle – the time interval between successive block updates that has to allow enough time for the miners to arrive at a block hash with as many leading zeros as the number stated as the “Degree-of-Difficulty”.
- iii) **Task 3:** Develop an algorithm to be used by the miner to select the right transactions to be included in the next block as typically there could be more transactions in the pool than a miner could/would include in each next block to be submitted.

Part 6 - (Tasks 1, 2, 3) Preparatory steps

For task 6, students should first simply create a few more transactions by sending coins from one wallet to another, essentially establishing the mini-blockchain as a ledger – creating a number of accounts (wallets) and say 10-20 transactions within an hour time-period with varying amounts and fees so that the transactions will have sufficiently spread-out timestamps. Students would then be ready to attempt the additional Tasks for the last 35 marks; here are some more hints for tackling tasks 6.1, 6.2, 6.3.

Multi-threading tutorials are available for C# which is the language of the already completed program for the mini-blockchain build but for students who wish to translate the existing program to another language such as Python or Java, multi-threading tutorials have also been included for these languages with lots of code examples by way of additional support. Despite this, the additional code to be written for the additional Tasks is relatively minor, so it would probably be advisable to attempt to continue with C# even if one is more comfortable with some other language. Programming tutorial resources covering multi-threading are available on the module site are as follows:

C# Threads Tutorial

java Threads Tutorial

Everything Python Head-on – Coursework Support

Each thread needs to have its own nonce to do its mining in parallel trying to get to arrive at a string with the right number of zeros preceding the hash of the block, but each thread needs to use its own nonce to avoid duplication of results. There is such a thing as an “enonce” (extra nonce), and each thread could have its extra nonce assigned i.e. own distinct one-time-token (number) to keep incrementing to use in each cycle of mining - *enonce*(type *long*).

In this way as a distinct number shall be used for each enonce of each thread in each cycle of hash composition, no duplicated effort could occur.

This task involves the following considerations:

1. Using the full computational power available for multi-threading e.g. C# Threads.
2. Advanced parallel processing could be deployed e.g. GPU implementation (CUDA).
3. Must use e-nonce to prevent duplication of work.
4. Threads synchronization and control.
5. Performing a comparison study comparing mining times of single versus multi-threaded solution.
6. Varying the Degree-of-Difficulty (n) to examine the point at which parallel processing does make necessary to arrive at the required results (<n leading zero><hash>) faster.

Thus, to increase the chance of arriving at the above result fastest, for this task one can run several threads each doing the block validation/mining (Proof-of-Work) process in parallel. The first thread would use standard Nonce and subsequent threads will each use their own distinct extra-Nonces (e-Nonce). This is because Nonce can only be used with one of the threads, so to avoid duplication of computation with each new thread another e-Nonce should be used.

Thread synchronization and control has to be implemented to ensure that as soon as the first successful **Mine()** thread resulting in the block hash with the right number of leading zeros is arrived at as the final result, all threading computation is stopped and all threads are timed to enable benchmarking the efficiency of **single versus multi-threaded Mine()** which is the objective of this task.

Therefore, one would need to measure how long it takes with single or more threads and how many threads are needed to satisfy a workable level of Degree-of-Difficulty for a reasonable Block Update Time (time-to -compute-valid-block).

This just means using the existing code and adding some extensions to **Mine()** to multi-thread it and work out the time each thread takes to do its **Mine()**. This would mean setting up new variables for thread management and for recording the time taken by each thread until the process ends which is the time at which one of the threads has arrived at the block hash with required number of leading zeros. The

measure and record the individual thread latencies, one could make use of `system.Diagnostics.Stopwatch`).

One should be able to take multiple samples of mining times and compare them for different number of threads and for a few increasing “Degree-of-Difficulty” numbers (e.g. 4, 5, 6 etc.) and seeing for what Degree-of-Difficulty and what number of threads a most efficient [(shortest Mine()) time] is achievable.

By increasing the Degree-of-Difficulty one can examine the point at which the overheads of threading are exceeded by the efficiency gain arising from the parallelism provided by the threading. Say starting from 4 as the “Degree-of-Difficulty” and adapting the existing **Mine()** to a **multi-threaded Mine()** in the C# program provided and manually re-setting it for each run with a different number of threads (while increasing or decreasing the number of threads and timing the process to establish at what Degree-of-Difficulty, using how many threads, the **multi-threaded Mine()** will execute faster to deliver the valid block with requisite number of leading zeros, i.e. the optimal set-up at which the parallel execution of Mine() through multi-threading gets the job done fastest.

Part 6 - Task 2

35/3≈11.67 marks

Re-setting the Dynamic” or “Adaptive” Degree-of -Difficulty for Adaptively varying the required Proof-of-Work (APoW)

Clearly there is a trade-off between changing the Degree-of-Difficulty and time-to-submit of valid blocks which is the time it takes to complete the computation required to arrive at a submittable block with the requisite number of leading zeros that thus satisfying the "Degree- of-Difficulty".

In our current implementation, increasing the Degree-of-Difficulty by one would increase the amount of work by a factor of 16. This is not suitable for fine tuning a dynamically re-settable Degree-of-Difficulty. You are going to need to think of another approach, consider researching existing approaches.

In your report, please detail how you implemented your Dynamic Degree-of-Difficulty and why you chose that approach while also providing evidence that it works. Also mention the length of your ‘Block Time’ and justify why.

Note: Any ‘Block Time’ implementation is OK as long as you feel it has been fairly justified (and it is workable for the miners given a high-speed computational platform).

This task involves the following steps:

1. Decide your own ‘block time’ and implement an appropriate algorithm
2. Establish a target block time
3. Continuously call Mine()
4. Calculate the “Mine Rate” based on previous block timestamps
5. Adjust the difficulty based on the difference between target and actual/observed block time

The block update time (latency) usually does not change frequently for a specific Blockchain (e.g. In Ethereum the block update time 10-20 seconds, for Bitcoin it is presently 10 minutes, and the Degree-of-Difficulty maintained with this is 19, see below).

Example of valid hash of a block as required to be computed under Proof-of-Work

00000000000000000000578b16c88ffc7a2b59c8489b7cc442c5a8129c7a79c4b

(This string consists of the 19 leading zeros and the hash of data inside the block and the Nonce for this round (Number-used-only-once)

This block hash has 19 leading zeros and is in hexadecimal format, The chance of obtaining the above string for PoW is $1/16^{19}$.

10 thousand trillion hashes need, starting with the initial hash of the block with the Nonce, for the chance to obtain such a string as a block to be submitted for validation.

$$\text{Block time} = \text{timestamp for block } (n) - \text{timestamp for block } (n - 1)$$

For re-setting the “Degree-of-Difficulty” to vary the computational load of Dynamic Proof-of- Work for miners from time to time, one can assess the trend in the median value of Block Update Time and have an algorithm that dynamically increases the Degree-of-Difficulty responsive to the trend in the median value of the Block Update Time.

In this way, Blockchains could use the block update times of the last N blocks to re-adjust the Degree-of-Difficulty responsive to the trend in the value of the Block Update Time.

One can use an exponential function to have the Degree-of-Difficulty be re-set responsive to changes in the Block Update Time (median time it has taken for the miners to submit the last N blocks) or use alternative approaches for such Adaptive Degree-of-Difficulty responsive to the latency of the required Proof-of-Work and decide what Degree-of-Difficulty would be optimal.

However, the starting point could be to establish a target block time and apply an exponential decay function to converge to a “Degree-of-Difficulty” number incrementally best adjusted for the target block update latency desired.

1. start with a moderate level of difficulty say 4 or 5
2. Benchmark this i.e. measure how long it takes with or without threads and how many threads it takes to satisfy a workable level of difficulty and block time
3. This just means repeating the PoW using Mine() for different levels of difficulty with or without threads
4. Block time = timestamp for block (n) – timestamp for block (n-1)
5. Apply an exponential growth function. In the **exponential growth** of $f(x)$, the **function** doubles every time you add one to its input x . In the **exponential decay** of $g(x)$, the **function** shrinks in half every time you add one to its input x ... e.g.
6. Establish a target block time
7. Calibrate “Block-To-Block Stair-Stepping”
8. >Median block time of a batch of N recently-mined blocks
9. Exponential Decay Difficulty Adjustments
 1. delay/blocktime
10. Adjust the difficulty according to the formula 3
11. After a significant spike in hash-rate and difficulty followed by a huge drop in hash-rate and difficulty, working each new block gets easier until the difficulty stabilizes according to the true size of the network.

Variable Methods for Selection of Pending Transactions for Mining

A Miner may consider picking the transaction with lowest fees which tend to be also amongst the ones that may have been waiting to be mined the longest (Altruistic Approach) or picking the largest transactions with the largest fees first as most Miners tend to go for as the most profitable approach (Greedy Approach). You will be able to add buttons for each of the above approaches to select the transactions according to the Miner's Approach - you need to reflect on the side-effects, on the Blockchain as a whole, arising from each of the transaction selection approaches.

Using some additional coding to modify the way **Mine()** works, you can enable the selection of transactions from the transactions pool to take place based on any one of up to three approaches selected from the table below:

- **Greedy:** You can choose transactions with the higher fees with a cut-off of say six transactions to be chosen.
 - **Altruistic:** You can check the timestamp of transactions and choose the oldest ones with cut-off set on the oldest transaction to be selected. The algorithm could have a limit of no more than n transactions to be selected and would select them by prioritising transactions that are no older than say n-Minutes/Hours/Days and then add as many younger transactions as possible in the order of oldest to complete the number of n transactions to be selected.
 - **Unpredictable (Random):** You can run a **Rand()** function with a small range and choose the transaction which represents the closest match with say the transaction number or the hash of some parameter of the transaction - with a cut-off decided on the number n of transactions required to be selected.
 - **Address-based:** You can choose transactions based on the value of some other attribute of the transaction e.g., the highest wallet_ID addresses with a cut-off after the top six chosen.
- Start by using the Windows Form to create buttons for each of the above four types of Mining. Declare custom functions to select transactions from the transaction pool, according to each of the above criteria, thereafter, proceeding to run Mine() with this method to select the transactions to be selected for the new block generated – ready to be submitted.

Your own idea - Some Suggestions:

1. Implementation of a different consensus algorithm (e.g. Proof-of-Stake)
2. Creating multiple nodes running in a local network, automating the generation of transactions
3. Secure Smart Contracting for a specific application domain use-case

Basic Technical Reporting Hygiene

As you can see 52% of the marks are allocated to the reporting so it is important that the accompanying report submitted is complete, coherent, and well-presented as it is with the presentation that one can needlessly lose marks one could easily earn. Research and reference to other Blockchain implementations highly recommended for these sections.

Appendix: Blockchain Optimisation

Further support for Task 6.2: Dynamic Degree-of-Difficulty Optimisation

Block Time

Block time refers to the average time it takes to generate a new block in a blockchain network. This is the time between the addition of one block and the next. Different blockchain networks have varying block times. For example:

- **Bitcoin:** Approximately 10 minutes.
- **Ethereum:** Around 12-14 seconds.

Importance of Block Time

- **Transaction Speed:** Block time directly impacts how quickly transactions are confirmed. Shorter block times mean faster transaction processing, which is essential for user experience and the practical use of the blockchain.
- **Network Security:** Block time affects the security of the network. A shorter block time can lead to more frequent blocks, which may increase the probability of forks (split chains). Managing block time helps in balancing security and efficiency.
- **Scalability:** Block time influences the blockchain's ability to handle a large number of transactions. Shorter block times can help increase throughput, but they also require more computational power and efficient network propagation.
- **Energy Consumption:** The frequency of block creation affects the amount of computational work required. Shorter block times might lead to higher energy consumption, making it important to find a balance for sustainability.
- **Economic Incentives:** Miners are rewarded for solving complex cryptographic problems to create new blocks. The block time influences how often these rewards are distributed, impacting the economic dynamics of mining activities.

- **Finding the Balance**

Finding the optimal block time is about balancing transaction speed, network security, scalability, energy efficiency, and economic incentives. Too short a block time might lead to network instability and higher energy consumption, while too long a block time could result in slow transaction processing and user dissatisfaction.

Optimising the Block Time and Degree-of-Difficulty in the PoW as a Consensus Protocol for Blockchain

These items collectively contribute to optimizing the block time and PoW throughput/latency capacity in blockchain networks.

- **Mining Difficulty Adjustment:** Dynamically adjusting the mining difficulty to maintain a consistent block generation time. This ensures the blockchain doesn't grow too rapidly or too slowly.
- **Exponential Moving Average (EMA) & Simple Moving Average (SMA):** Both EMA and SMA are used to smooth out data and provide trends. These averages help in making informed adjustments to mining difficulty, directly influencing block time and PoW efficiency. EMA responds faster to recent changes, while SMA offers a balanced view over time. Utilizing EMA to give more weight to recent data, making the difficulty adjustment mechanism more responsive to abrupt changes in network activity.
- **Simple Moving Average (SMA):** Employing SMA for reporting purposes, providing a balanced view of network activity over a longer period.
- **Simple Moving Average (SMA):** Employing SMA for reporting purposes, providing a balanced view of network activity over a longer period.
- **Reinforcement Learning (RL):** Leveraging RL techniques to dynamically adapt the difficulty of PoW based on the computing power of miners. This helps in achieving fairness and efficiency in blockchain mining.
- **Optimal Block Size:** Analysing and determining the optimal block size to minimize latency and maximize throughput. This involves considering factors like transaction confirmation time and network congestion.
- **Latency:** The time taken for a transaction to be confirmed. Lower latency improves user experience.
- **Latency Reduction:** Implementing strategies to reduce end-to-end latency in PoW-based blockchain networks. This includes optimizing block generation rates and minimizing fork probabilities. Reducing latency improves the overall network efficiency and transaction speed, which is vital for

maintaining a stable throughput. Lower latency can also reduce the likelihood of forks, which can further stabilize the network.

- **Throughput Optimization:** Enhancing the number of transactions processed per second by optimizing the block size and block interval. Optimizing throughput ensures the network can handle a high volume of transactions, but it must be balanced with block size and mining difficulty to avoid excessive latency.
- **Scalability Improvements:** Ensuring that the blockchain network can handle an increasing number of users and transactions without performance degradation.
- **Energy Efficiency:** Developing more efficient consensus mechanisms to reduce energy consumption and operational costs. More efficient consensus mechanisms reduce energy consumption, which is crucial for sustainability. Energy efficiency can also influence mining difficulty, as less energy-intensive methods may require different difficulty adjustment strategies.

These items are interrelated, and they often influence each other in dynamic ways:

- **Mining Difficulty Adjustment:** Adjusting mining difficulty directly impacts throughput and latency. If difficulty increases, it can reduce throughput and increase latency because blocks take longer to mine. Conversely, lowering difficulty can increase throughput but may compromise security. Dynamically adjusting the mining difficulty to maintain a consistent block generation time. This ensures the blockchain doesn't grow too rapidly or too slowly.
- **Exponential Moving Average (EMA) & Simple Moving Average (SMA):** Both EMA and SMA are used to smooth out data and provide trends. These averages help in making informed adjustments to mining difficulty, directly influencing block time and PoW efficiency. EMA responds faster to recent changes, while SMA offers a balanced view over time.
- **Reinforcement Learning (RL):** Leveraging RL techniques to dynamically adapt the difficulty of PoW based on the computing power of miners. This helps in achieving fairness and efficiency in blockchain mining. RL can optimize the difficulty adjustment process by learning from past data and predicting future trends. This helps maintain an optimal balance between block time and throughput.
- **Optimal Block Size:** Analysing and determining the optimal block size to minimize latency and maximize throughput. This involves considering factors like transaction confirmation time and network congestion. The block size affects both throughput and latency. Larger blocks can process more transactions but may increase the time to propagate across the network, leading to higher latency. Finding the optimal block size is crucial for maintaining balance.
- **Latency Reduction:** Reducing latency improves the overall network efficiency and transaction speed, which is vital for maintaining a stable throughput. Lower latency can also reduce the likelihood of forks, which can further stabilize the network.
- **Throughput Optimization:** Optimizing throughput ensures the network can handle a high volume of transactions, but it must be balanced with block size and mining difficulty to avoid excessive latency.

- **Scalability Metrics:** The network's ability to handle increased demand. Metrics like TPS and latency under load are important indicators.
- **Scalability Improvements:** Ensuring that the blockchain network can handle an increasing number of users and transactions without performance degradation. Scalability ensures the network can grow without performance degradation. This includes optimizing mining difficulty, block size, and reducing latency to handle more users and transactions efficiently. The network's ability to handle increased demand. Metrics like TPS and latency under load are important indicators.
- **Energy Efficiency:** More efficient consensus mechanisms reduce energy consumption, which is crucial for sustainability. Energy efficiency can also influence mining difficulty, as less energy-intensive methods may require different difficulty adjustment strategies.

By understanding these interconnections, developers can create a more efficient and balanced blockchain system that optimizes for block time, throughput

For optimal blockchain performance, it is important to monitor a range of metrics that cover different aspects of the network. Here are some key metrics to keep an eye on:

Blockchain Metrics

1. **Block Time:** The average time taken to mine a new block. Consistent block time indicates network stability.
2. **Hash Rate:** The total computational power of the network. Higher hash rates generally improve network security.
3. **Difficulty:** The mining difficulty level, which adjusts to maintain consistent block times.
4. **Block Size:** The average size of blocks. Optimal block sizes help balance throughput and latency.
5. **Transaction Throughput (TPS):** The number of transactions processed per second. Higher TPS indicates better network performance.

Network Metrics

6. **Latency:** The time taken for a transaction to be confirmed. Lower latency improves user experience.
7. **Fork Rate:** The frequency of blockchain forks. Lower fork rates suggest a more stable network.
8. **Propagation Time:** The time taken for a new block to reach all nodes in the network. Faster propagation reduces the risk of forks or orphaned blocks, thereby ensuring better network synchronization and security.

Economic Metrics

9. **Transaction Fees:** The average fees paid for transactions. Lower fees can encourage more users to transact.
10. **Miner Revenue:** The total income of miners from block rewards and transaction fees. Higher revenue can incentivize more mining activity.

Security Metrics

11. **Attack Cost:** The cost to launch an attack on the network (e.g., 51% attack). Higher attack costs indicate better security.
12. **Double-Spend Rate:** The frequency of double-spend incidents. Lower rates suggest higher network integrity.

User Metrics

13. **Active Nodes:** The number of active nodes participating in the network. More nodes enhance decentralization and resilience.
14. **Wallet Activity:** The number of active wallets and transactions. High activity indicates robust user engagement.

Exponential Decay Function usage in implementation of adaptive dynamic optimisation of "Degree of Difficulty" and "Block Time" in Blockchain

Exponential Decay Function

Imagine you have a big, colourful balloon. Every minute, you let out a certain percentage of air from the balloon. At first, a lot of air escapes because the balloon is full. But as time goes on, less and less air comes out, because there's not much left in the balloon.

So, exponential decay means that something decreases at a rate proportional to its current value. The more you have, the faster it decreases. Over time, the amount gets smaller and smaller, but it never quite reaches zero.

In the context of blockchain, this concept can be used to make certain things, like rewards or difficulty levels, decrease smoothly over time.

Examples of how exponential decay functions can be applied in blockchain technology to optimize various parameters and ensure the network's stability and efficiency.

- **Exponential Decay of Sensitivity in Dynamic Optimization:** This paper explores the exponential decay of sensitivity in dynamic optimization problems, including model predictive control and moving horizon estimation. It provides insights into how perturbations propagate along the horizon and enable the development of approximation and solution schemes.
- **Controllability and Observability Imply Exponential Decay of Sensitivity in Dynamic Optimization:** This study delves into the property of exponential decay of sensitivity in dynamic optimization problems. It discusses how uniform controllability and observability imply exponential decay of sensitivity, providing insights into the development of approximation and solution schemes.

- **Gradient-Based Optimization Method Using Exponential Decay:** This article proposes a new gradient optimization method using exponential decay and adaptive learning rates. It discusses the application of this method in various optimization problems, including neural network training.
- **Difficulty Adjustment:** In blockchain networks like Bitcoin, the difficulty of mining new blocks is adjusted periodically to ensure that blocks are mined at a consistent rate. An exponential decay function can be used to smooth out these adjustments, preventing sudden spikes or drops in difficulty. This helps maintain a stable and predictable block generation time.
- **Block Reward Halving:** Many blockchain networks implement a block reward halving mechanism, where the reward for mining a new block is reduced by half at regular intervals. This can be modelled using an exponential decay function, where the reward decreases exponentially over time. This helps control the supply of new coins and can influence the long-term value of the cryptocurrency.
- **Fee Adjustment:** Transaction fees in blockchain networks can also be adjusted using exponential decay functions. For example, the minimum fee required to include a transaction in a block can be dynamically adjusted based on network congestion and demand. This ensures that the fee rate remains competitive and fair for users.
- **Token Economics:** In some blockchain-based token economies, the distribution of tokens or rewards can follow an exponential decay pattern. This can help incentivize early adopters and gradually reduce the rate of new token issuance over time, promoting scarcity and potentially increasing the token's value

References

1. [Adaptive Difficulty Adjustment in Blockchain: Harnessing the Power of Exponential Moving Average \(EMA\) - GRIDNET OS - Main - GRIDNET Community](#)
2. <https://vtechworks.lib.vt.edu/server/api/core/bitstreams/596fda75-96a6-4796-94e3-d6c9cae8d92a/content>
3. [\(PDF\) End-to-End Latency Analysis and Optimal Block Size of Proof-of-Work Blockchain Applications](#)
4. <https://arxiv.org/pdf/2202.01497>
5. [performance-optimization-strategies-for-blockchain-networks.pdf](#)
6. [\[2101.06350v4\] Exponential Decay of Sensitivity in Dynamic Optimization: A Graph-Theoretic Approach](#)
7. <https://arxiv.org/pdf/2101.06350.pdf>
8. <https://www.mdpi.com/2227-7080/12/9/154>

CSC Endchain Security Module

Mini Blockchain Build

Frequently Asked Questions & Answers

Question 1

Having completed the code for wallet creation, and the user interface buttons for the wallet private and public key, these do not appear on the designated text boxes for private and public key that I have created on the UI, what could be the problem?

Answer to Q1

I would check that the wallet constructor code has correctly output the private key and the UI textboxes for private and public key are correctly designated for displaying the respective keys and the validate button is correctly assigned to display them. Now if all code for the above is complete and you click on the Validate button, then the keys should be displayed in the designated textboxes, if this still does not occur, please use the debugging tools to step through the code as there must be something missing in the code- ensure that your code is identical to the fully commented code provided.

Question 2

When attempting to create and display a transaction, the sender's private and public keys are there ok but to send funds you have to provide the Recipient's Key where do I find the Recipient Key? where is this generated in the code?

Answer to Question 2

A typical blockchain would have a large number of participants/nodes/wallets and for even a mini blockchain as in this case we should use the Wallet Constructor to create many wallets so that we could create many transactions so that we could have a transaction pool to select transaction from to include in a block. Every time the **wallet class** is used a new unique wallet is generated. Therefore if you are already trying to create transactions you must have some wallets that have been generated, if not, then you could generate some and then you would be able to display their keys and copy and paste their keys on the notepad and select from those a wallet as your Recipient and copy and paste its key to send funds to when attempting to complete a transaction.

Question 3

For the blockchain coursework do we need to include the difficulty in the creation of the hash? Because in the cookbook it asks to include it in the creation. However, in the highlighted code provided and the video in session 4 it is not included. So wasn't sure if we should add it or not. See below

- 9) Set the difficulty to 4 as a float (we set it as a float to allow for the degree-of-difficulty to be changeable (i.e. a dynamic degree-of-difficulty) if this were to be required.
- 10) Include the difficulty variable in the hash composition used in `CreateHash()`.

```

public String CreateHash() // Hashes the entire Blc
{
    String hash = String.Empty;
    SHA256 hasher = SHA256Managed.Create();

    String input = timestamp.ToString() + index + prevHash + nonce + merkleRoot; // Concatenate all of the
    Byte[] hashByte = hasher.ComputeHash(Encoding.UTF8.GetBytes(input)); // Apply the SHA hash

    /*
    foreach (byte x in hashByte)
        hash += String.Format("{0:x2}", x); // Reformat to a string

    return hash;

```

Answer to Question 3

Semantic confusion:

by stating: "...include the degree-of-difficulty variable in the hash" it is meant not that one must hash the actual value of the degree-of-difficulty number together with block hash.

This would not make any sense because the degree-of-difficulty is essentially to serve as a counter to check after each hashing cycle of the program whether the target number of the leading zeros have indeed been achieved in the resulting hash after each cycle and if not to continue with the next hashing cycle and so on until the target number of leading zeros are achieved.

As has been stated previously, it is good programming practice not to hard-code the value of a variable into any program but to include it in the program as a variable whose value could be changeable as a dynamic value.

The variable "degree-of-difficulty" happens to be set at the value of 4 for Parts 1-5 of the coursework but for task 6.1 and 6.2 you would need to run instances of the hashing program with varying degrees-of-difficulty when running **multi-threaded Mine()** and/or when attempting to run experiments for optimisation of the degree of difficulty and block update latency.

Therefore, including the hash as a variable in the program makes sense as you can then run a program that passes a different value, as may be needed for the degree-of-difficulty, to the hashing program each time thereby ensuring it continues to go through the hashing loop until the number of leading zeros in the resulting hash of a final cycle has reached the particular degree-of-difficulty selected for that run.

I hope it is now clear that "include the degree-of-difficulty variable in the hash..." means include it as a variable not hard-code it and certainly not hash it together with the block hash.

Question 4

Hi for the blockchain module which, can I create my own implementation of a PoW chain in python as that is my most proficient language?

I was thinking to make a mini-bitcoin chain including
Proof of Work

2 nodes are enough for p2p (using 2 threads)

1 minute block time adjusting after x blocks

roll only the nonce for mining

4 wallets, add a transaction every 1 minute automatically and random (2 miners and 2 users)

Have a mempool (unconfirmed transactions in queue)

Accepted blocks are broadcasted to all nodes (validation)
Would this be enough ?

Answer to Question 4

Looks like you are re-negotiating the coursework. Language of implementation is optional but we cannot change the requirements of the coursework to be addressed by the Implementation and the Report - to be fair to all the students.

I would agree that a python implementation would be OK as long as all of the same blockchain features are replicated. For example wallets with Public/Private keys, transactions with rewards and fees for mining (Proof of Work) and Merkle Root algorithm implementation. This should be sufficient to show their understanding of the concepts of Blockchain.

However, despite a student being more proficient in python this might be more work than if they used the C# code given that C# should be very easy to pick up.

Question 5

I'm writing as I'm a little confused about the current assignment, particularly steps 1-5.
My problem is that I've interpreted the brief as requiring independently programming steps 1-5 but I've noticed that there is complete sample code available on Blackboard.

What are the requirements of this assignment - are we meant to use this code for part 6 and not program steps 1-5?

The brief suggests that part 6 requires implementation of our own code written for steps 1-5.

Answer to Question 5

The sample code has been provided as a guide to help direct those less experienced with programming - whether that be in general or specifically in C# - so that these individuals are not disadvantaged in anyway.

The assignment aims to give students the opportunity to display their knowledge of the fundamental components of a Blockchain and we do not put too much emphasis on the specifics of the implementation when marking - as long as the application is functional and logically correct. Instead, we focus on the understanding displayed by design justifications made in the code comments and accompanying report and evidenced by completed build evidenced by screens shots as indicated in the guide notes.

A good understanding can be developed by either following the script and producing your own code (step-by-step) or by studying the sample code, running the application. Students would not be penalised for building upon the sample code for the assignment as long as a good understanding of the principles of a Blockchain is evident by the description of steps and screen shots.

Question 6

Firstly, regarding the code you provided, assuming proper documentation we can just provide your code for these parts as the "Evidence of Implementation".

Would you be able to clarify whether we must submit our own code for parts 1 - 5 for the coursework and then can proceed to use the code you provided for part 6 (additional tasks) or would we be able to use your code for all parts, provided sufficient documentation?

Secondly, regarding the report, a page limit of 7 pages is mentioned, could you clarify that this is excluding images, screenshots and code snippets, as otherwise I believe it would be hard to write in detail for each of the required parts.

Answer to Question 6

1. The supplied code can be used for Parts 1-5 of the assignment provided it is well documented in the report supported by your own description and screenshots - This code, or your own equivalent if you prefer to implement your own code for Parts 1-5 (for example if coding in Python), you can then built upon the code for Parts 1-5 by extending Mine() as you would need to do for the additional assignment tasks - Part 6.

2. The report can make use of appendices, which does not count towards the page limit. The Appendix can be used for screenshots, tables and code snippets as you deem appropriate.

Question 7

I was trying to submit my work right now however it won't let me push my files into GitLab. I have attached a screenshot of the error, is there any way to fix this ? I will be submitting my report with all the screenshots of my code running, however my Gitlab link won't be in there as it won't allow me to upload my files. Do let me know if there's any other way I can upload my files.

Answer to Question 7

.There is a presentation on how to use the repository
It can be quite difficult to debug issues with Git as it does not always provide the most helpful or clear error messages. The general process is to: (From the BlockchainAssignment directory)

1. **git init** (Initialise the local repository)
2. **git add -A** (Mark the files to include - all in this case)
3. **git commit -m '...'** (Bundle the changes for upload)
4. **git remote add origin [url]** (Link the remote repository)
5. **git push -f origin main** (Overwrite remote with local commits)

If any of the above stages were missed when you attempted to do upload, then I would suggest trying again, or if not, I would suggest uploading the solution (BlockchainAssignment folder) to a shared drive or zipping it all and including it in the submission on the Blackboard.

Question 8

I was following the tutorial about processing my transactions but I couldn't see what you had done and now have errors from line 30 to line 39. Can you help me please?

Answer to Question 8

A common issue may arise due to the `getPendingTransactions()` function being incomplete and not returning a List of transactions as the message signature requires.

The complete function should "pop" (get and remove) the first "n" transactions from the transaction pool where n is the lowest value of the two: the maximum number of transactions per block or the total number of pending transactions should this be lower. These transactions are then to be returned by the function to be used when creating a new block.

The function should be as follows:

```
public List<Transaction> GetPendingTransactions()
{
    // Determine the number of transactions to retrieve dependent on the number of pending
    transactions and the limit specified
    int n = Math.Min(transactionsPerBlock, transactionPool.Count);

    // "Pull" transactions from the transaction list (modifying the original list)
    List<Transaction> transactions = transactionPool.GetRange(0, n);
    transactionPool.RemoveRange(0, n);

    // Return the extracted transactions
    return transactions;
}
```

I hope this helps resolve your problem.

Question 9

I have created the merkle root and recieved a final singular hash, I have added it into the `CreateHash()` method within Block class and added it to the `PrintBlock()` method, is there anywhere else I need to use the merkle root to validate anything for checks anywhere?

Answer to Question 9

By including the Merkle root into the Block hash and subsequently the signature any change to a single transaction will result in a significantly different hash which will be easy to detect during the validation process. This is one of the main advantages of hashing.

Note that during validation, the Merkle root is to be recalculated and compared with that of the submitted block and this should be sufficient to prove the validity of the contents of the transactions included within the Block.

I hope this clarifies the use of the Merkle root in the validation process.

Question 10

I am not sure what points are most important in my description in my report to demonstrate my understanding of what I have done, could you please explain?

Answer to Question 10

You need to ensure that you include:

1) Evidence of completion and execution of the code for Parts 1-5 and any Part 6 tasks (T6.1, 6.2, 6.3) that you have implemented on top of the code for Parts 1-5 and integrated with it (mainly by way of Mine() extensions as required] - all supported by resulting screenshots.

2) Description of the steps for the above, in particular the main blockchain concepts for example the function and purpose of the following elements of Blockchain.

I. Hashing for tamper-proofing of block contents – an important function and its application is described (combining data to form a unique hash)

II. Asymmetric cryptography

- a. Public and Private Keys for pseudo-anonymous identities and authentication.
- b. Digital signatures for secure transactions and non-repudiation of transactions.

III. Proof-of-Work as a consensus algorithm in a decentralised network (trust mechanism).

IV. Significance of the confirmation process in preventing attacks through irreversible transactions.

Question 11

I have completed up until the "Processing Transactions & Transaction Pools" section in the week 3 Blockchain module practical.

I am having trouble with the function associated with the button "create transaction" as I am unsure of where "sender address" and "receiver address" would come from in order to give to the transaction constructor. Also, I am unsure on what receiver key means and where it comes from. I tried to review the video demonstration however this did not show it.

Answer to Question 11

The sender and receiver address will be the public keys (aka IDs) belonging to the two parties involved in the transaction. In terms of our simulation parties or individuals are represented as wallets containing a public-private key pair and these are generated by pressing the "Generate Keys" button. This creates a new Wallet object and populates the two fields in the UI with the corresponding labels. Each time this button is pressed a new public-private key pair is generated and so for testing purposes, it is necessary to create and make a note of a few test "wallets" and then to create transactions between them.

I hope this clarifies the source of the "address" fields in the transaction class.

Question 12

So far in the assignment, I have managed to get everything working successfully up until generating the wallets and printing if the keys are valid. It is the create transactions part that I can't get to work. I have been struggling to work out how to complete the setting up transactions part of the assignment. I attach my code and hope you could explain what is wrong.

Answer to Question 12

The Transaction.cs class was developed has been developed as well as the UI elements for generating and validating Wallets (i.e. public-private keys) as presented in the colour coded fully commented code. The UI elements can be sued to create transactions which will be placed in a transaction pool where pending transactions await being mined.

In the fully commented code, the following variables and functions have been added for this:

1. Transaction.cs (All)
2. BlockchainApp.cs (Functions)
 - a. GenerateWallet_Click()
 - b. ValidateKeys_Click()
 - c. CreateTransaction_Click()
3. Blockchain.cs (Variables)
 - a. transactionsPerBlock
 - b. transactionPool

Please check that your code for the above is identical to the correct and complete code provided for the above elements. Then it must and will work.

Having checked your code, it is clear that you are very close to the solution. The only thing missing is the "public" access specifier for the Block attribute in Blockchain.cs (or alternatively, a public method in Block.cs to add a new block). This was preventing the application from being built as by default this variable is not visible or accessible outside of its class. By adding the public specifier:

```
public List<Block> Blocks = new List<Block>();
```

this means the list will be modifiable within the Blockchain class.

Mac user's Questions

Question 13

I have got a Mac how can I set it up to get the coursework done

Answer to Question 13

Jaydon Scarpa kindly provide this for the Mac User's Club

<https://www.youtube.com/watch?v=uis3pvt4wBU>

Any issues Jaydon and his colleagues can advise

Address: They are usually to be found at the row opposite the printer in G56

Question 14

I am using MacOS. I keep getting an error on visual studio "mono64 quit unexpectedly" when trying to run the Blockchain. I have tried looking up solutions but have found none. What should I do from here?

Answer to Question 14

This appears to be an issue with MacOS's "mono64" not supporting Windows Forms (see: <https://www.mono-project.com/docs/gui/winforms/getting-started-guide/>). The suggested solution is to switch to "Mono" (32-bit) in Visual Studio which can be found here:

"Run > Run with > Custom Configuration > Advanced > Mono runtime settings > Runtime > Architecture > 32bit"

I hope this helps resolve your issue and allows you to execute the code on your Mac.

Question 15

I am unable to open up the form design viewer in Visual Studio to modify the UI. I think this may be because I am using Mac OS so I've tried installing Xamarin instead but I am still unable to view/edit the form.

Answer to Question 15

Running the Blockchain application on Mac can be a bit challenging as the UI which is based on Windows Forms, relies on elements built into the Windows operating system. "Mono" on Mac attempts to bridge this and bring the same functionality to macOS however not everything is fully supported at this time. As a result, the best solution would be to install a Windows Virtual Machine (using Parallels, VMWare, Bootcamp, or VBox, etc.) on the Mac to host the Windows OS. This typically requires an x64-bit Intel processor however some providers are starting to support Apple chips (M1) too.

(See: <https://blogs.vmware.com/teamfusion/2021/09/fusion-for-m1-public-tech-preview-now-available.html>)

Should none of these options be successful, alternative solutions could be to translate the UI elements in (BlockchainApp.cs) to Xamarin Forms (see: <https://docs.microsoft.com/en-us/xamarin/cross-platform/desktop/controls/> for guidance) or to work in another programming language if confident in

doing so. Finally, it is also possible to access a Windows machine in the lab or using a cloud provider (SaaS/IaaS) e.g. Windows Azure Virtual Machines (<https://docs.microsoft.com/en-us/azure/virtual-machines/windows/using-visual-studio-vm>).

Other relevant sources on the web

Preview: Bringing macOS to Xamarin.Forms - Xamarin Blog

<https://devblogs.microsoft.com/xamarin/preview-bringing-macos-to-...>

Developing for **macOS** is a **Mac** only feature, so the same rules apply as when you are doing a **Xamarin.Mac** application. There is an excellent **Xamarin.Mac** guide here that covers the pre-requisites. When the time comes for you to integrate your Xamarin.Forms **macOS** project into a continuous integration system.....

[See more on devblogs.microsoft.com](https://devblogs.microsoft.com/xamarin/preview-bringing-macos-to-...)

EXPLORING FURTHER

2

Xamarin.Forms targeting macOS - Stack Overflow	stackoverflow.com
Creating a MAC Application Using Xamarin.Forms	www.c-sharpcorner.com
How to test your Xamarin.Forms iOS apps without a Mac ...	www.xamarinexpert.it
Look iOS Developer, No Mac Required - Build an iOS ...	nicksnettravels.builttoroam.com
Mac Platform Setup - Xamarin Microsoft Docs	docs.microsoft.com

• **c#** - How to run a Windows Forms application on macOS ...

<https://stackoverflow.com/questions/35566777>

Comments

I've used Xamarin and GTK#, as well as Xamarin for code-behind and XCode for forms design. They both work pretty decently with C#

Wine is capable of running some winforms-based Windows applications on Mac OSX. When it works as intended, all you have to do is install Wine.

You might be able to cross-compile using Mono. <http://www.mono-project.com/docs/gui/winforms/>.

[c# - How to run a Windows Forms application on macOS](#)

stackoverflow.com/questions/35566777/how-to-run-a-win...

Mac Platform Setup - Xamarin | Microsoft Docs

<https://docs.microsoft.com/en-us/xamarin/xamarin-forms/platform/other/mac>

Follow these instructions to add a Mac app that will run on macOS Sierra and macOS El Capitan: 1. In Visual Studio for Mac, right-click on the existing Xamarin.Forms solution and **choose Add > Add New Project...** 2. In the New Project window **choose Mac > App > Cocoa App** and press Next. 3. Type an App Name (and optionally choose a different name for the Dock Item), then press Ne...

[New Xamarin.Forms XAML IntelliSense in Visual Studio ...](#)

<https://devblogs.microsoft.com/xamarin/xaml-intellisense-vs4m-xamarin-forms>

[How To: Setting up macOS X Sierra on ... - Auri's Blog](https://devblogs.microsoft.com/xamarin/xaml-intellisense-vs4m-xamarin-forms)

<https://auri.net/2016/11/20/how-to-setting-up-macos-x-sierra-on...>

First, open Safari – unless you installed something else on the **Mac** already – and download **Xamarin** Studio for **Mac**. This is simple – go to **Xamarin.com**, and download the installer. Open the installer on your **Mac** from the Downloads folder, and click Open when it warns you that it's an application downloaded from the Internet.

[System Requirements - Xamarin | Microsoft Docs](https://devblogs.microsoft.com/xamarin/xaml-intellisense-vs4m-xamarin-forms)

<https://docs.microsoft.com/en-us/xamarin/cross-platform/get-started/...>