Module Code: CS3BC  
Assignment report Title: Blockchain Coursework Assignment  
Student Number: 30002734  
Date (when the work was completed): 12/05/2025  
Actual hrs spent for the assignment:  
Which Artificial Intelligence tools used (if applicable):

Part 1

To start, I opened the provided BlockchainAssignment.sln in Visual Studio and double-clicked BlockchainApp.cs to launch the Windows Forms designer. From the Toolbox I dragged a single-line TextBox, a Button and a large RichTextBox onto the form. In the Properties panel I renamed the TextBox to inputText, the Button to printButton and set its Text property to “Print,” and renamed the RichTextBox to outputBox so the purpose of each control was immediately clear. I then switched to the code view, double-clicked the Print button to generate its click handler, and wrote a simple one-line assignment that takes whatever string the user has entered into inputText and places it into outputBox.Text. After adjusting the size and anchoring of the controls for a clean layout, I ran the application, typed “test” into the small box and clicked Print, confirming that the text appeared correctly in the large console area. This quick UI prototype verified that control placement, naming, and event wiring were all working before moving on to the blockchain implementation.

Part 2

I then moved on to building the core blockchain logic by creating two new classes, Block and Blockchain. In Block.cs I declared private fields for timestamp, index, previous hash and hash. I provided a parameterless constructor for the genesis block that sets the timestamp to now, index to zero, previous hash to the empty string and then calls CreateHash to compute its SHA-256 hash. The CreateHash method concatenates the index, timestamp and previous hash into a single string, runs it through the hasher and formats the resulting bytes as a hexadecimal string, which becomes the block’s hash. I also added overloaded constructors that accept the last block so that new blocks automatically inherit the correct index and link back to their predecessor before computing their own hash.

In Blockchain.cs I declared a list of block objects and in the constructor initialised it with a single new Block so that the chain always starts with the genesis block. I then added getBlockAsString, which returns the ToString representation of any block by index or an error message if the index is out of bounds. Back in the form’s constructor in BlockchainApp.cs I kept the same UI controls and, immediately after initialising the component, instantiated my new Blockchain object and replaced the console text with the string returned by getBlockAsString(0) to display the genesis block’s details. Running the application shows the index, timestamp, SHA-256 hash and previous hash (empty for block zero) in the large RichTextBox, confirming that block creation and chaining are working as intended.

The appendix includes two images. The first image shows the RichTextBox when the program is first initialised, with the “New blockchain initialised!” message. The second image shows the running application with the RichTextBox displaying the full details of block zero.

Part 3

In the next phase, full wallet support and transaction creation were added to enable secure transfers of Assignment Coins. The application leverages the existing Wallet class, which implements ECDSA key‐pair generation and validation. Two new UI buttons, Generate Keys and Validate Keys, were placed beside text boxes labelled Public Key and Private Key. Clicking Generate Keys calls Wallet’s constructor, fills the Public Key field with the newly created public identifier and writes the matching private key into the Private Key field. Clicking Validate Keys then invokes Wallet.ValidatePrivateKey to confirm that the two values form a valid pair and displays “Keys are valid” or “Keys are invalid” in the main RichTextBox output.

With key management in place, a Transaction class was introduced to encapsulate everything needed for a blockchain transfer. Each transaction records a timestamp, sender address, recipient address, amount and fee. Its constructor concatenates those fields, runs the result through SHA-256 to produce a hash, and then calls Wallet.CreateSignature to sign that hash with the sender’s private key. This signature binds the transaction cryptographically to the sender and prevents tampering.

A Create Transaction button was added to the UI so that, once a valid key pair, recipient address, amount and fee are entered, clicking it constructs a new Transaction and writes its full details into the RichTextBox. This end-to-end flow ensures that every transaction is both authenticated and tamper-evident, laying the groundwork for the transaction pool and balance checks that follow.

In the appendix, the first screenshot will show the application after keys are generated and validated, with “Keys are valid” displayed in the output area. The second screenshot will show a completed transaction in the console, illustrating the timestamp, sender address, recipient address, amount, fee, transaction hash and digital signature.

Part 4

Part four extends the blockchain’s consensus by adding proof-of-work mining, transaction inclusion and miner rewards in the Block class. A new Mine method performs repeated SHA-256 hashing of the block header, which now includes index, timestamp, previous hash, Merkle root and a changing nonce, until the hash meets the difficulty target of four leading zeros. Each time the nonce is incremented, the CreateHash routine combines the header fields and computes the digest. As soon as Mine finds a valid hash, it returns, ensuring every block carries a verifiable proof of work.

When a block is constructed, it now accepts the previous block, the pending transactions list and the miner’s address. A reward transaction is generated that pays a base reward plus any collected fees, and all transactions are fed into a MerkleRoot method to produce the block’s Merkle root. Mine is then invoked to find the nonce and hash. Once mining completes, the block object holds its nonce, hash, Merkle root and the full sorted list of transactions so that every block records both its cryptographic proof and its contained transfers.

On the blockchain side, the transaction pool was introduced to hold pending transactions, and a GetPendingTransactions method was added to batch them in groups of five. A getLastBlock helper returns the current tip of the chain for linking new blocks. The Windows Forms interface gained three new buttons for Generate New Block, Print All and Print Pending Transactions. Generate New Block retrieves the next batch of transactions, calls the Block constructor to mine and link the new block, adds it to the chain and shows its details in the main output area. Print All lists every block in the chain, and Print Pending Transactions displays any transactions still waiting in the pool.

Part 5

Part five implements a complete validation layer to guarantee the blockchain’s integrity and the correctness of user balances. Two new static methods, ValidateHash and ValidateMerkleRoot, recompute each block’s SHA-256 hash (using its index, timestamp, prevHash, nonce and Merkle root) and rebuild the Merkle tree over its transactions to ensure nothing has been altered. In the UI a Full Blockchain Validation button now walks through every block from the second onward, checking that each block’s prevHash matches the actual hash of its predecessor, that its stored hash equals a fresh hash of its header, and that its stored Merkle root matches the recomputed root. If any of these checks fails, the app reports “Blockchain is invalid”; otherwise, it reports “Blockchain is valid.”

To tie balances back to the ledger, a GetBalance method was also added to the Blockchain class. It scans every confirmed transaction in every block, credits amounts where the given address is the recipient and debits the amount plus the fee where it is the sender and returns the net Assignment Coin balance. On the form you can enter a public key and click Check Balance to display the computed balance in the main console area.

Part 6

In Task 6.1, a multithreaded mining mode was introduced to exploit all available CPU cores and drive down average block-find time. Each thread probes a distinct slice of the nonce space by stepping its nonce counter by the total thread count on every iteration. A shared cancellation flag, guarded by a simple lock, ensures that as soon as one thread finds a valid hash that meets the leading-zero difficulty target, all others stop immediately. A stopwatch measures the elapsed time for each mining run, and those results are written to the console area to compare single-threaded performance with multithreaded runs. The appendix shows comparisons of three multithreaded and three single-threaded blocks and their mining time. The mean block time over the three multithreaded blocks using 8 threads was 6.938 seconds, whereas for single threading, the mean was 39.999 seconds.

Task 6.2 adds an adaptive difficulty algorithm so that block creation stays close to a chosen target interval. After each block is mined, the application checks whether the mining time was faster or slower than the target time. If blocks are being found faster than the target, the difficulty is incremented by one; if too slow, it is decremented, never dropping below one. The current difficulty level is displayed in the UI alongside the mining controls and is logged with every block print-out. As shown in the appendix, as the difficulty increases, so does the mining time until it is longer than the target time, then both the difficulty and mining time decrease again.

Task 6.3 implements transaction selection preferences to explore different miner behaviours. A set of radio buttons in the form lets the user choose between three strategies, including greedy (highest-fee-first), altruistic (oldest-pending-first) or random selection. When “Generate New Block” is clicked, the code sorts the pending-transaction list according to the chosen strategy and then selects the required number of transactions for inclusion in the new block. This makes it easy to observe how each policy affects total fees collected and average transaction wait time. The appendix shows how each strategy chooses 5 transactions from the transaction pool.

Appendix

Part 1

Application window with two textboxes and a button. The text is printed in the large text box from the small text box:

A screenshot of a computer

AI-generated content may be incorrect.

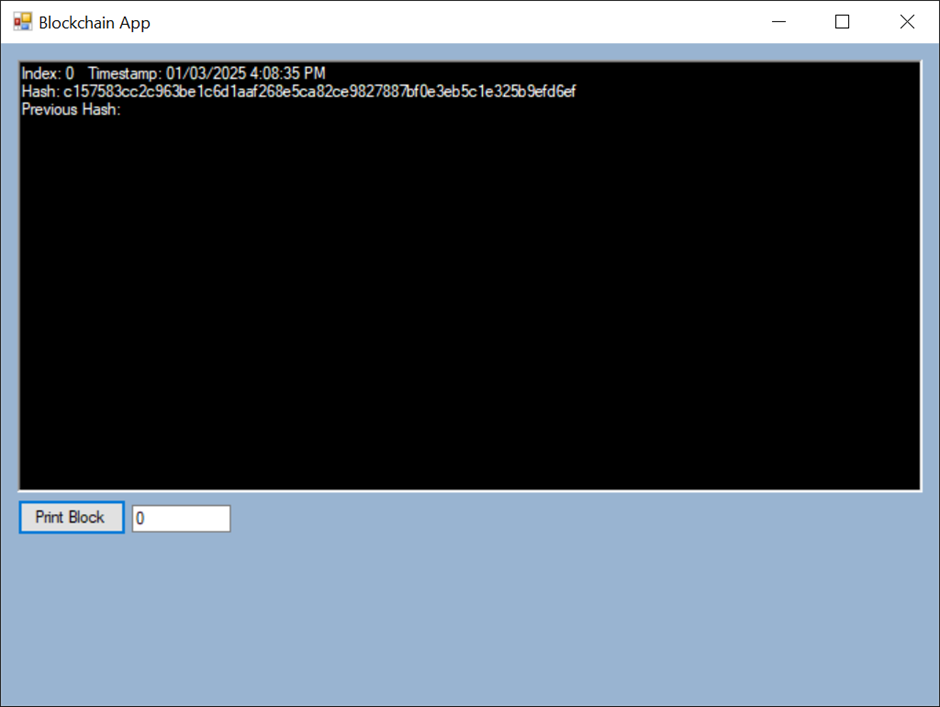
Part 2

New blockchain initialised message when starting the application:

A screenshot of a computer

AI-generated content may be incorrect.

Genesis block:



Part 3

Keys generated in the public and private key text boxes, then validated:

A screenshot of a computer

AI-generated content may be incorrect.

Transactions can be created, but they are not validated on the blockchain:A screenshot of a computer program

AI-generated content may be incorrect.

Part 4

New block and sent a transaction, which is pending:

A screenshot of a computer program

AI-generated content may be incorrect.

Both blocks are shown, with the transaction shown in the second block:A screenshot of a computer

AI-generated content may be incorrect.

Part 5

Keys generated with a transaction generated, giving the wallet tokens:

A screenshot of a computer

AI-generated content may be incorrect.

The transaction in the chain:

A screenshot of a computer

AI-generated content may be incorrect.

Blockchain being validated:

A screenshot of a computer

AI-generated content may be incorrect.

Part 6.1

Multithreaded block 1:

A screenshot of a computer

AI-generated content may be incorrect.

Multithreaded block 2:

A screenshot of a computer

AI-generated content may be incorrect.

Multithreaded block 3:

A screenshot of a computer

AI-generated content may be incorrect.

Single-threaded block 1:

A screenshot of a computer program

AI-generated content may be incorrect.

Single-threaded block 2:

A screenshot of a computer

AI-generated content may be incorrect.

Single-threaded block 3:

A screenshot of a computer

AI-generated content may be incorrect.

Part 6.2

|  |  |  |
| --- | --- | --- |
| Block Index | Difficulty | Mining Time |
| 0 | 4 | 0.211 |
| 1 | 5 | 6.487 |
| 2 | 6 | 24.768 |
| 3 | 5 | 14.148 |

Block 0:

A screenshot of a computer

AI-generated content may be incorrect.

Block 1:

A screenshot of a computer

AI-generated content may be incorrect.

Block 2:

A screenshot of a computer

AI-generated content may be incorrect.

Block 3:

A screenshot of a computer

AI-generated content may be incorrect.

Part 6.3

Three transactions in the pending list with fees from 1 to 3:

A screenshot of a computer

AI-generated content may be incorrect.

Four transactions in the pending list with fees from 4 to 7:

A screenshot of a computer

AI-generated content may be incorrect.

The next block was mined using greedy mining mode, containing the 5 highest fee transactions:

A screenshot of a computer

AI-generated content may be incorrect.

More transactions in the block:

A screenshot of a computer

AI-generated content may be incorrect.

End of block including mining reward transaction:

A screenshot of a computer

AI-generated content may be incorrect.

Using the same 7 transactions as before, we mine a new block using altruistic mode:

A screenshot of a computer

AI-generated content may be incorrect.

Next 3 transactions in the block:

A screenshot of a computer

AI-generated content may be incorrect.

End of block including the mining reward transaction:

A screenshot of a computer

AI-generated content may be incorrect.

Using the same transactions, now we use the random mode for transaction selection and mine a new block:

A screenshot of a computer

AI-generated content may be incorrect.

The next transactions (in random order of fees now):

A screenshot of a computer

AI-generated content may be incorrect.

End of block including mining reward transaction:

A screenshot of a computer

AI-generated content may be incorrect.