CS 216: Introduction to Blockchain
Coding Assignment: Bitcoin Transaction & UTXO Simulator

Due: 3rd February 2025, 11:59 PM
Total Marks: 15

Important Information

Due Date:	3rd February 2025, 11:59 PM
Team Size:	4 students (mandatory)
Total Marks:	15
Submission:	GitHub Repository (Public)
Late Policy:	-20% per day

Contents

1 Introduction & Learning Objectives

1.1 Purpose of This Assignment

This assignment aims to help you understand the core mechanisms of Bitcoin's transaction system through hands-on implementation. You will build a simplified version of Bitcoin's transaction validation system, focusing on the UTXO model, double-spending prevention, and the transaction lifecycle.

1.2 Learning Outcomes

After completing this assignment, you should be able to:

- **Explain** the UTXO (Unspent Transaction Output) model and its advantages
- **Implement** Bitcoin's transaction validation rules
- **Demonstrate** how double-spending is prevented in Bitcoin
- **Simulate** the transaction lifecycle from creation to confirmation
- **Describe** miner incentives and fee economics
- **Compare** UTXO model with account-based systems

1.3 Important Note

This is a **local simulation**, not a distributed system. You don't need to implement networking, cryptography, or consensus algorithms. Focus on the logic of transaction validation and UTXO management.

2 Glossary of Key Terms

Students often get confused by these terms. Read this section carefully before starting!

2.1 Core Concepts

- **UTXO (Unspent Transaction Output):**
  - **What:** A piece of bitcoin that hasn't been spent yet
  - **Analogy:** Individual dollar bills in your wallet
  - **In our sim:** A tuple like `(tx_id, index, amount, owner)`
  - **Key point:** UTXOs are **created** in transaction outputs and **destroyed** when used as inputs
- **Transaction:**
  - **What:** A transfer of value from one address to another
  - **Analogy:** Taking some dollar bills from your wallet and giving them to someone

  - **Components:** Inputs (which UTXOs to spend) + Outputs (new UTXOs to create)
  - **Formula:** `Sum(inputs) = Sum(outputs) + fee`

- **Mempool (Memory Pool):**
  - **What:** A waiting area for unconfirmed transactions
  - **Analogy:** A post office sorting room before mail delivery
  - **Purpose:** Stores valid transactions waiting to be included in a block
  - **In our sim:** A list/dictionary of transactions with conflict detection

- **Double-Spending:**
  - **What:** Trying to spend the same bitcoin twice
  - **Analogy:** Writing two checks from the same bank account with insufficient funds
  - **How prevented:** UTXOs can only be spent once, checked by all nodes
  - **In our sim:** Reject transaction if UTXO already spent (in mempool or blockchain)

- **Mining:**
  - **What:** The process of confirming transactions and adding them to blockchain
  - **Analogy:** A postal worker collecting mail from sorting room and delivering it
  - **In our sim:** Selecting transactions from mempool and updating UTXO set permanently
  - **Reward:** Miner gets all transaction fees from block

- **Fee:**
  - **What:** Payment to miner for including transaction in block
  - **Formula:** `fee = Sum(inputs) - Sum(outputs)`
  - **Purpose:** Incentive for miners, prevents spam
  - **In our sim:** Any positive difference between inputs and outputs

- **Change Output:**
  - **What:** Sending leftover bitcoin back to yourself
  - **Analogy:** Paying $20 for a $15 item and getting $5 change
  - **Example:** Alice has 10 BTC UTXO, sends 3 BTC to Bob → creates outputs: 3 BTC to Bob + 6.999 BTC to Alice (0.001 BTC fee)

2.2 Common Confusions Clarified

Confusion	Clarification
"Balance" vs "UTXOs"	**Balance** is the sum of all UTXOs you own. Bitcoin doesn't store balances, only UTXOs.
Transaction "validation" vs "confirmation"	**Validation** = checking if transaction is valid (happens immediately). **Confirmation** = transaction included in block (happens after mining).
Mempool vs Blockchain	**Mempool** = unconfirmed transactions (temporary). **Blockchain** = confirmed transactions (permanent).
Inputs vs Outputs	**Inputs** = which UTXOs you're spending (destroying). **Outputs** = new UTXOs you're creating.
Double-spend in same tx vs different tx	**Same tx:** Two inputs reference same UTXO → easy to detect. **Different tx:** Two transactions spend same UTXO → need mempool/blockchain check.

3 What to Implement

3.1 Part 1: UTXO Manager (3 marks)

Create a class to manage UTXOs. Think of it as Bitcoin's "database" of spendable coins.

```python
class UTXOManager:
    def __init__(self):
        # Store UTXOs as dictionary: (tx_id, index) -> (amount, owner)
        self.utxo_set = {}

    def add_utxo(self, tx_id: str, index: int, amount: float, owner: str):
        """Add a new UTXO to the set."""
        pass

    def remove_utxo(self, tx_id: str, index: int):
        """Remove a UTXO (when spent)."""
        pass

    def get_balance(self, owner: str) -> float:
        """Calculate total balance for an address."""
        pass

    def exists(self, tx_id: str, index: int) -> bool:
        """Check if UTXO exists and is unspent."""
        pass

    def get_utxos_for_owner(self, owner: str) -> list:
        """Get all UTXOs owned by an address."""
```

```
        pass
```

Listing 1: UTXO Manager Interface

3.2 Part 2: Transaction Structure & Validation (4 marks)

Define what a transaction looks like and implement validation rules.

```
# Transaction format (you can use dict or class)
transaction = {
    "tx_id": "abc123",  # Unique identifier
    "inputs": [
        {
            "prev_tx": "previous_tx_id",  # Which transaction
            "index": 0,                   # Which output in that
transaction
            "owner": "Alice"              # Who owns this UTXO
        }
    ],
    "outputs": [
        {"amount": 1.5, "address": "Bob"},
        {"amount": 0.299, "address": "Alice"}  # Change
    ]
}
```

Listing 2: Transaction Structure

Validation Rules (MUST implement all):

1. **All inputs must exist in UTXO set**
2. **No double-spending in inputs** (same UTXO twice in same transaction)
3. **Sum(inputs) $\geq$ Sum(outputs)** (difference = fee)
4. **No negative amounts** in outputs
5. **No conflict with mempool** (UTXO not already spent in unconfirmed tx)

3.3 Part 3: Mempool Management (3 marks)

Create a mempool that stores unconfirmed transactions and prevents conflicts.

```
class Mempool:
    def __init__(self, max_size=50):
        self.transactions = []  # Store transactions
        self.spent_utxos = set()  # Track UTXOs spent in mempool
        self.max_size = max_size

    def add_transaction(self, tx, utxo_manager) -> (bool, str):
        """Validate and add transaction. Return (success, message)."""
        pass

    def remove_transaction(self, tx_id: str):
        """Remove transaction (when mined)."""
        pass

    def get_top_transactions(self, n: int) -> list:
```

```
        """Return top N transactions by fee (highest first)."""
        pass

    def clear(self):
        """Clear all transactions."""
        pass
```

Listing 3: Mempool Operations

3.4 Part 4: Mining Simulation (3 marks)

Simulate the mining process that confirms transactions.

```
def mine_block(miner_address: str, mempool: Mempool,
               utxo_manager: UTXOManager, num_txs=5):
    """
    Simulate mining a block.
    1. Select top transactions from mempool
    2. Update UTXO set (remove inputs, add outputs)
    3. Add miner fee as special UTXO
    4. Remove mined transactions from mempool
    """
    pass
```

Listing 4: Mining Function

3.5 Part 5: Double-Spending Prevention (2 marks)

Create test cases that demonstrate:

- Simple double-spend detection
- Mempool conflict prevention
- Race attack concept (first-seen rule)

4 Program Interface Requirements

Your program **MUST** provide this exact menu interface:

```
=== Bitcoin Transaction Simulator ===
Initial UTXOs (Genesis Block):
- Alice: 50.0 BTC
- Bob: 30.0 BTC
- Charlie: 20.0 BTC
- David: 10.0 BTC
- Eve: 5.0 BTC

Main Menu:
1. Create new transaction
2. View UTXO set
3. View mempool
4. Mine block
5. Run test scenarios
6. Exit

```

```
Enter choice:
```

Listing 5: Required Program Interface

4.1 Example Workflow

1. **Creating a transaction:**

```
Enter choice: 1
Enter sender: Alice
Available balance: 50.0 BTC
Enter recipient: Bob
Enter amount: 10.0 BTC

Creating transaction...
Transaction valid! Fee: 0.001 BTC
Transaction ID: tx_alice_bob_001
Transaction added to mempool.
Mempool now has 3 transactions.
```

2. **Mining a block:**

```
Enter choice: 4
Enter miner name: Miner1
Mining block...
Selected 3 transactions from mempool.
Total fees: 0.003 BTC
Miner Miner1 receives 0.003 BTC
Block mined successfully!
Removed 3 transactions from mempool.
```

3. **Double-spend attempt:**

```
Enter choice: 5
Select test scenario: 2 (Double-spend)
Running test...

Test: Alice tries to spend same UTXO twice
TX1: Alice -> Bob (10 BTC) - VALID
TX2: Alice -> Charlie (10 BTC) - REJECTED
Error: UTXO genesis:0 already spent by tx_alice_bob_001
```

5 Test Scenarios & Evaluation

5.1 Initial State (Genesis UTXOs)

The system starts with these UTXOs:

Owner	Amount (BTC)	UTXO Reference
Alice	50.0	(genesis, 0)
Bob	30.0	(genesis, 1)
Charlie	20.0	(genesis, 2)
David	10.0	(genesis, 3)
Eve	5.0	(genesis, 4)
Total	**115.0**	

5.2 Mandatory Test Cases

1. **Test 1: Basic Valid Transaction**
    - Alice sends 10 BTC to Bob
    - Must include change output back to Alice
    - Must calculate correct fee (0.001 BTC)
2. **Test 2: Multiple Inputs**
    - Alice spends two UTXOs (50 + 20 BTC) together
    - Sends 60 BTC to Bob
    - Tests input aggregation and fee calculation
3. **Test 3: Double-Spend in Same Transaction**
    - Transaction tries to spend same UTXO twice
    - Expected: REJECT with clear error message
4. **Test 4: Mempool Double-Spend**
    - TX1: Alice → Bob (spends UTXO)
    - TX2: Alice → Charlie (spends SAME UTXO)
    - Expected: TX1 accepted, TX2 rejected
5. **Test 5: Insufficient Funds**
    - Bob tries to send 35 BTC (has only 30 BTC)
    - Expected: REJECT with "Insufficient funds"
6. **Test 6: Negative Amount**
    - Transaction with negative output amount
    - Expected: REJECT immediately
7. **Test 7: Zero Fee Transaction**
    - Inputs = Outputs (fee = 0)
    - Expected: ACCEPTED (valid in Bitcoin)
8. **Test 8: Race Attack Simulation**

- Low-fee merchant TX arrives first
- High-fee attack TX arrives second
- Expected: First transaction wins (first-seen rule)

9. **Test 9: Complete Mining Flow**
   - Add multiple transactions to mempool
   - Mine a block
   - Check: UTXOs updated, miner gets fees, mempool cleared

10. **Test 10: Unconfirmed Chain**
    - Alice → Bob (TX1 creates new UTXO for Bob)
    - Bob tries to spend that UTXO before TX1 is mined
    - Test your design decision (accept/reject with explanation)

6 Technical Requirements & Guidelines

6.1 Allowed Technologies

- **Programming Language:** Any (Python recommended for simplicity)
- **Libraries:** Standard libraries only
- **Data Structures:** Lists, dictionaries, classes, sets
- **Interface:** Command-line/text-based (no GUI required)

6.2 Not Allowed

- **No external blockchain libraries** (bitcoinlib, pycoin, etc.)
- **No real cryptography** (simulate signatures with string comparison)
- **No networking/socket programming**
- **No cloud/distributed setup** (runs locally)

6.3 Implementation Tips

1. **Start with UTXO manager** - Get basic operations working first
2. **Test incrementally** - After each function, test with simple cases
3. **Use Python dictionaries** for O(1) UTXO lookup:

```python
# Efficient UTXO storage
utxo_set = {
    ("genesis", 0): {"amount": 50.0, "owner": "Alice"},
    ("genesis", 1): {"amount": 30.0, "owner": "Bob"},
    # ...
}

```

4. **Generate unique transaction IDs:**

```python
import time
import random

def generate_tx_id():
    return f"tx_{int(time.time())}_{random.randint(1000, 9999)}"

```

5. **Validate early** - Check basic rules before complex validation

6. **Keep it simple** - Don't over-engineer. Focus on core requirements.

6.4 Development Environment

- **OS:** Any (Windows, macOS, Linux)
- **Python:** 3.8+ recommended
- **Memory:** Minimal (program uses ¡ 100MB)
- **Storage:** ¡ 50MB
- **No internet required** after setup

7 Submission Instructions

7.1 GitHub Repository Requirements

1. Repository Name:

- Must include your team name
- Example: `CS216-TeamAlpha-UTXO-Simulator`

2. Repository Structure:

```
your-repository/
├── src/
│   ├── main.py    # Main program entry point
│   ├── utxo_manager.py    # UTXO handling class
│   ├── transaction.py    # Transaction class/structure
│   ├── mempool.py    # Mempool management
│   ├── validator.py    # Validation logic
│   └── block.py    # Block/mining logic
├── tests/
│   └── test_scenarios.py    # Your test cases
├── requirements.txt    # Dependencies (if any)
├── README.md    # Documentation
└── sample_output.txt    # Screenshot/text of demo run
```

3. README.md MUST Include:

- Team name and all members (names, roll numbers)

- Clear instructions to run the program
- Brief explanation of your design
- Mention if you attempted bonus question
- Any dependencies/installation steps

4. **Repository MUST be PUBLIC**

7.2 Submission Format on Moodle

Submit exactly this format on Moodle:

```
GitHub Repository: https://github.com/yourusername/your_repository_name
Team Name: [Your Team Name]
Team Members:
1. Name - Roll Number
2. Name - Roll Number
3. Name - Roll Number
4. Name - Roll Number
Bonus Attempted: [Yes/No] - [Describe bonus features]
```

7.3 Late Submission Policy

- **1 day late:** -20% (max 12/15)
- **2 days late:** -40% (max 9/15)
- **3+ days late:** Not accepted (0/15)
- **Weekend counts** as days

7.4 Git Collaboration Tips

1. **Create branches** for each feature:

```
git checkout -b feature-utxo-manager
git checkout -b feature-transaction-validation

```

2. **Commit frequently** with descriptive messages:

```
git commit -m "feat: implement UTXO manager with add/remove"
git commit -m "fix: correct balance calculation bug"

```

3. **Use pull requests** for code review
4. **Never commit** to main branch directly

8 Frequently Asked Questions

1. **Q: Do we need real cryptographic signatures?**
   - **A:** No. Simulate with string comparison. Example: if transaction says owner="Alice", accept it.
2. **Q: Can we use `bitcoinlib` or similar libraries?**
   - **A:** No. Use only standard libraries. The point is to understand the logic, not use pre-built solutions.
3. **Q: What if two transactions arrive at exactly the same time?**
   - **A:** In real Bitcoin, network latency determines order. In our sim, you decide (FIFO, random, or by fee). Document your choice.
4. **Q: How should we handle Test 10 (unconfirmed chain)?**
   - **A:** Two valid approaches:
     (a) **Reject:** "Cannot spend unconfirmed UTXO" (simpler)
     (b) **Accept with dependency:** Track parent-child relationships

     Choose one and implement consistently.
5. **Q: What happens when mempool reaches max size?**
   - **A:** Implement eviction policy (remove lowest-fee transaction). Document your approach.
6. **Q: Can we use SQLite/JSON files for storage?**
   - **A:** Yes, but in-memory is simpler and sufficient. Files add complexity without extra marks.
7. **Q: What if a team member doesn't contribute?**
   - **A:** Inform instructor early with evidence (git commits). Peer evaluation will be considered.

9 Getting Help

9.1 Resources

1. **Course Material:**
   - Lecture 3 slides (L3.pdf) - Pages 7-37
   - Bitcoin Whitepaper Sections 2, 5, 6
2. **Online Resources:**
   - UTXO Explained

- Bitcoin Transactions
- Python Tutorial

3. **Tools:**
   - GitHub for version control
   - Python IDEs: VS Code, PyCharm, Jupyter

Submission Deadline: 3rd February 2025, 11:59 PM