Bitcoin and Cryptocurrency Technologies

Assignment 1: ScroogeCoin

In ScroogeCoin, the central authority Scrooge receives transactions from users. You will implement the logic used by Scrooge to process transactions and produce the ledger. In each discrete "epoch," Scrooge will receive a list of transactions, validate the transactions he receives, and publish a list of validated transactions.

Note that a transaction can reference another in the same epoch. As well, two/three/... transactions in the same epoch may represent an attempted double/triple/... spend. This means that choosing a subset of transactions that are together valid is a tricky problem.

You will be provided with a Transaction class that represents a ScroogeCoin transaction, and which has inner classes Transaction.Output and Transaction.Input.

A transaction output consists of a value and a public key to which it is being paid.

A transaction input consists of the hash of the transaction that contains the corresponding output, the index of this output in that transaction (indices are simply integers starting from 0), and a signature. For the input to be valid, the signature must be on an appropriate digest of the current transaction (see the `getRawDataToSign(int index)` method) with the private key that corresponds to the public key in the output that this input is claiming.

A transaction consists of a list of its inputs and outputs and a hash of the complete transaction (see the `getRawTx()` method), and contains methods to add and remove an input, add an output, compute digests to sign/hash, add a signature to an input (the computation of signatures is done outside the Transaction class by an entity that knows the appropriate private keys), and compute and store the hash of the transaction once all inputs/outputs/signatures have been added.

You will also be provided with a UTXO class that represents an unspent transaction output. A UTXO contains the hash of the transaction from which it originates as well as its index in that transaction. The `equals()`, `hashCode()`, and `compareTo()` methods in UTXO are overridden to provide for equality and comparison between two UTXOs based on their indices and the contents of their txHash arrays instead of their locations in memory.

Furthermore you will be provided with a UTXOPool class that represents the current set of outstanding UTXOs and contains a map from each UTXO to its corresponding transaction output. This class contains constructors to create a new empty UTXOPool or a defensive copy of a given UTXOPool, and methods to add and remove UTXOs from the pool, get the output corresponding to a given UTXO, check if a UTXO is in the pool, and get a list of all UTXOs in the pool. (Recall that HashMap looks up keys by using their hashCode and equals methods.)

Finally you will be provided with an rsa.jar file for using the RSAKey class. The public key in a transaction output and the private key used to create signatures in transaction inputs are both represented by an RSAKey, and an RSAKeyPair is a public/private key pair. APIs for RSAKey and RSAKeyPair are given on the COS 432 website.

You will be responsible for creating a file called TxHandler.java that implements the following API:

```
public class TxHandler {

    /* Creates a public ledger whose current UTXOPool (collection of unspent
     * transaction outputs) is utxoPool. This should make a defensive copy of
     * utxoPool by using the UTXOPool(UTXOPool uPool) constructor.
     */
    public TxHandler(UTXOPool utxoPool);

    /* Returns true if
     * (1) all outputs claimed by tx are in the current UTXO pool,
     * (2) the signatures on each input of tx are valid,
     * (3) no UTXO is claimed multiple times by tx,
     * (4) all of tx’s output values are non-negative, and
     * (5) the sum of tx’s input values is greater than or equal to the sum of
             its output values;
       and false otherwise.
     */
    public boolean isValidTx(Transaction tx);

    /* Handles each epoch by receiving an unordered array of proposed
     * transactions, checking each transaction for correctness,
     * returning a mutually valid array of accepted transactions,
     * and updating the current UTXO pool as appropriate.
     */
    public Transaction[] handleTxs(Transaction[] possibleTxs);

}
```

Your implementation of handleTxs should return a mutually valid transaction set of maximal size (one that can’t be enlarged simply by adding more transactions). It need not compute a set of maximum size (one for which there is no larger mutually valid transaction set). Based on the transactions it has chosen to accept, handleTxs should also update its internal UTXOPool to reflect the current set of unspent transaction outputs, so that future calls to handleTxs/isValidTx are able to correctly process/validate transactions that claim outputs from transactions that were accepted in a previous call to handleTxs.

Extra Credit: Create a second file called MaxFeeTxHandler.java with similar API whose handleTxs finds a set of transactions with maximum total transaction fees (i.e. maximize the sum over all transactions in the set of (sum of input values - sum of output values)). Finding such a set of transactions is NP hard and thus find only a heuristic which might increase the total transaction fees.