Implement the *Examcoin* system that is a simple version of Bitcoin and works according to the following strategies:

1. A block contains the following information

prev_hash	The hash of the previous block
merkle_root	The root of the Merkle-Tree of this block's transactions
timestamp	The approximate creation time of this block
nonce	The nonce, a counter used for the proof-of-work algorithm
difficulty	The proof-of-work algorithm difficulty target for this block
prev	The pointer points to the previous block
data	The set of transactions

2. There are two kinds of transactions

A. CoinsBase

When a miner mines a new block they will be awarded 20 coins. Here is an example transaction in json format (Python dictionary):

{"hash": "707e3e923d4e0b6c3c54ae6880659a9bb371cb7dd556",

"type": "CoinsBase",

"coins_created": [{"num": 0, "value": 20, "recipient": "[Miner]"}],

"Timestamp": "2020-08-24 14:24:46.986531"}

where [Miner] is the miner's public key (or the miner's alias).

B. PayCoins

A user can transfer his/her coins to another. The transaction includes a list of unspent coins that are owned by the user. The transaction creates new coins for the new user and may transfer back the rest of the coins to the old user. For example, Alice owns 5.5 unspent coins and she wants to transfer 4 coins to Bob. Alice's unspent coins that are used in this transaction are listed in the "coins_consumed" field.

{"hash": "242874663bcf7485099cb993ceb46e0b42594e20319e56",

"type": "PayCoins",

"coins_consumed": [

{"hash": "f1531ef8c83c9d37d68f2505b2a4809d416cfe1b101c27", "num": 0},

{"hash": "bbf6b700cf24734e2874663bcf52f47485099cb993ceb4", "num": 1}],

"coins_created": [

{"num": 0, "value": 4, "recipient": "[Bob]"},

{"num": 1, "value": 1.2, "recipient": "[Alice]"}],

"transaction fee" : 0.3

"signatures" : "signature of [Alice]",

"Timestamp": "2020-08-24 15:54:46.983531"}

2. When a user creates a new transaction, this transaction is stored in the pending pool that contains all pending transactions before they may be included to a block.

3. Miners mine a new block with a proof of work algorithm as in Bitcoin.

- The time to mine a new block is based on the difficulty. Assuming that the difficulty starts with 10 and it increases 20 % after 10 blocks created.

difficulty = 10

maxNonce = 2 ** 32

target = 2 ** (256 – difficulty)

- A miner collects transactions in the pending pool. Maximum 10 transactions would be included in one block. Lets us assume that there are three ways to collect the transactions for the block.

+ Miners collect transactions based on their transaction fees. Transactions with high transaction fees get selected first.

+ Miners collect 10 transactions based on the timestamp. Transactions with early timestamp get selected first.

+ Miners collect 10 transactions randomly.

- Miners build a Merkle tree of these selected transactions and include the Merkle tree root in the block.

- Before adding the new block to the block chain, a miner should validate the chain. The new block is supposed to be appended to the current head of the chain. However, if the head is invalid the miner is allowed to skip it.

- A miner receives 20 coins as reward for each new valid block created.

- A transaction is valid if

For CoinsBase:

+ the number of coins created is 20.

For PayCoins:

+ The coins spent plus the transaction fee is less or equal to the number consumed coins.

+ The consumed coins are unspent (no double spending).

+ The signature is valid.

4. The system should provide functions that allow users to validate transactions, blocks, and chains.

5. The system should provide a function call ***getBalance*** that allows users to check the number of coins that they own.

6. The system should provide a transfer function that transfers n coins from [A] to [B] with fee m ***transfer(n, m, [A], [B])***

where

- ***[A]*** and ***[B]*** are public keys (or alias)

- ***n*** is the amount of coins that ***[A]*** wants to transfer to ***[B]***

- ***m*** is the transaction fee

The results of the transfer function could be

- a proposal for a transaction that performs the transfer

{"hash": "...",

"type": "PayCoins",

"coins_consumed": [

{"hash": "...", "num": ...},

{"hash": "...", "num": ...},

…],

"coins_created": [

{"num": 0, "value": n, "recipient": "[B]"},

{"num": 1, "value": n', "recipient": "[A]"}],

"transaction fee" : m,

"signatures" : "signature of [A]",

"Timestamp": "…"}

This transfer function should search for unspent coins of [A] on the blockchain. They could be from one transaction or more such that the total amount of them is equal or greater than n + m

- Fail if the balance is too low.

Hint.
We suggest the following structure for Python code that implements the *examcoin* system. This way we can assess each step and give marks for partial solutions. If you use a different approach, you should structure into steps that can be assessed individually (if you want marks for a partial solution). In any case, meaningful comments are part of the solution.

1. The whole program could be structured into one main program *examcoin* and two subprograms as follows.

- The *crypto_key* program that implements all functions to generate private/public keys, sign a document, and verify a signature.

- The *merkletree* program that implements all functions to construct a Merkle tree.

2. The main *examcoin* program should contain three classes Transaction, Block, and Examcoin as follows.

- black text is Python code.

- blue text is comments.

```
class Transaction:
	# the attribute _data is in a json format
	_data = None

	# 1. the constructor function
	def __init__(self, data):
		self._data = data
		self._data["Timestamp"] = str(datetime.datetime.now())
		self._data["hash"] = self.digest()

	# 2. there should be two different digest functions for hashing a transaction.
	# 2.1. this function returns a hash to be signed on. It is the hash of the fields
	"type", "coins_consumed", "coins_created" and "transaction fee" for "PayCoins" and
	"type" and "coins_created" for "CoinsBase"
	def digest_sign(self):

		# write your code here

	# 2.2. this function returns the hash of all fields
	def digest(self):

		# write your code here

	# 3. this function signs on a transaction by a given private key (or an alias that
	names to a private key file, which may require a password to open). Note that,
	after signing, the signature is appended to the field "signatures" of the
	attribute _data.
	def sign(self, private_key):
	# or def sign(self, alias, password):
```

```
            # write your code here

        # 4. this function verifies whether the signature of a transaction, which is in
        the "signatures" field is valid with a given public key (or an alias that names
        for a public key file)
        def verify_sig(self, public_key):
        # or def verify_sig(self, alias):

            # write your code here

        # 5. this function returns the total amount of coins created in a transaction,
        def get_sum_coin_created(self):

            # write your code here

        # 6. this function returns the value of a coin created
        def get_value(self, num):

            # write your code here

        # 7. there may be other functions.

        # write your code for other functions here


class Block:
        _prev_hash = bytearray(256)
        _merkle_root = bytearray(256)
        _timestamp = None
        _nonce = 0
        _difficulty = 0
        _prev = None
        _data = []

        # 1. the constructor function
        def __init__ (self, data, nonce, difficulty, prev = None):
            self._prev_hash = prev.digest() if prev is not None else
                              bytearray(256)
            self._merkle_root = construct_Merkletree(data)
            self._nonce = nonce
            self._difficulty = difficulty
            self._prev = prev
            self._data = data

         # 2. there should be two digest functions
        # 2.1. this function returns the hash of a block for the mining purpose. This hash
        is used to compare with the target in the proof of work algorithm. The data to be
        hashed includes _prev_hash, _nonce and _data.
        def mining_digest(self):

            # write your code here

        # 2.2. this function returns the hash of a block. The data to be hashed includes
        all the attributes
        def digest(self):

            # write your code here

        # 3. this function returns the value of a coin created
        def get_value_coin (self, trans_hash, num):

            # write the code here
```

```
	# 4. there may be other functions

	# write your code for other functions here

class examcoin:
	_head = None
	_root_hash = bytearray(256)
	_diff = 10
	_maxNonce = 2 ** 32
	_target = 2 ** (256 - _diff)
	_pending_pool = {}
	_level = 0

	# 1. the constructor function.
	def __init__(self):
		self._head = Block([], 0, 0, None)
		self._root_hash = self._head.digest()

	# 2. this function adds a new transaction to the pending pool.
	def add_transaction(self, trans):

		# write your code here.

	# 3. this function adds a new block to the chain. Note that this block need to
	contain a transaction that awards 20 coins to the minier. When a new block is
	added, the attribute _level should be increased by 1 and all the transactions that
	are included in this block should be removed from the pending pool.
	def __add_block(self, block, minier):
		json = {"hash": 0,
			"type": "CoinsBase",
			"coins_created": [
				{"num": 0, "value": 20, "recipient": minier}],
			"Timestamp": ""}
		award = Transaction(json)
		block._data.append(award)

		# write your code here.

	# 3.1. this function removes all selected transactions from the pending pool.
	def __filtering_pending_pool(self, selected_trans):

		# write your code here.

	# 4. this function checks whether a given transaction is valid.
	def verify_trans(self, trans):

		# write your code here.

	# there are some functions that are useful to validate a transaction.
	# 4. 1. this function returns the owner of a coin,
	def get_coin_owner(self, trans_hash, num):

		# write your code here.

	# 4.2. this functions checks whether the coins consumed in a transaction are
	double spending.
	def check_double_spending(self, coins_consumed):

		# write the code here.

	# 5. this function validates whether a given block is valid.
	def verify_block(self, block):
		# check the validation of the mining, the merkel tree root and all the
		transactions.
```

```
        # write your code here.

    # 6. this functions checks whether the chain is valid.
    def verify_chain(self):

        # write your code here.

    # 7. this function collects all valid transactions from the pending pool.
    def collect_valid_trans(self, pending_pool):

        # write your code here.

    # 8. given an array of valid transactions:
    # 8.1. this function returns an array of maximum 10 valid transactions sorted by
    transaction fee.
    def sort_trans_by_fee(self, valid_trans):

        # write your code here.

    # 8.2. this function returns an array of maximum 10 valid transactions sorted by
    timestamp.
    def sort_trans_by_timstamp(self, valid_trans):

        # write your code here.

    # 8.3. this function returns an array of maximum 10 valid transactions randomly.
    def sort_trans_by_random(self, valid_trans):

        # write your code here.

     # 9. this function mines a new block. The arguments are the public key (or
    alias), the ming method that could be "fee",  "timestamp" or "random" and the
    block that the new block want to append after (as default, it is the current head
    of the chain).
    def mine(self, alias, method = "fee", head = None):
    # the mining strategy is the proof of work algorithm as in Bitcoin. The nonce,
    difficulty (target) are used for mining. Note that the difficulty increases 20%
    after each cycle of 10 blocks created. The attribute _level could be used for this
    purpose.

        # write your code here.

    # 10. this function returns the balance of an account (public key or alias). The
    balance of an account is the value of its unspent coins.
    def get_balance(self, public_key):
        # or def get_balance(self, alias):

        # write your code here.

    # 11. this function performs a transfer that transfers the amount "amount" of
    coins from an account "from_addr" (private_key or alias) to another account
    "to_adrr" (public key or alias) with the transaction fee "fee".
    def transfer(self, amount, fee, from_addr, to_addr):
        # the function need to find the unspent coins of "from_addr" account, which
        could be one coin or more such that the sum of their values are equal or
        greater than "amount" plus "fee".
        # the transfer is fail if the balance of the account "from_addr" is too low.
        # if the transfer is successful, the transaction is added to the pending
        pool.

        # write your code here.

    # 12. there may be other functions.

    # write your code for other functions here.
```