Q1 (10 points) [Hash Function].

1. Explain the two basic requirements for a cryptographic hash function and the two security properties: collision resistance and preimage resistance.

2. Consider the compression function $f : B^8 \rightarrow B^4$ given by

$f(b_0b_1b_2b_3b_4b_5b_6b_7) = e_0e_1e_2e_3$

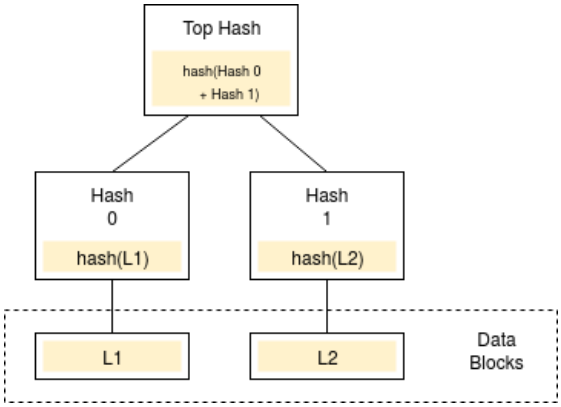
where $e_i = b_i \oplus b_{i+4}$

Let further x = 10111101010001010111.

- Apply the Merkle-Damgaard procedure to construct a hash function $h : B^* \rightarrow B^4$ from f and calculate $h(x)$. Explain each step in this calculation.
- Explain whether this hash function h is collision resistance and/or preimage resistance.

Q2 (5 points) [Merkle Tree]

Take a look at the Merkle tree in the following figure, which contains two data elements.



Extend this Merke tree with a single new element L3, so that the resulting Merkle tree has exactly the elements L1, L2, and L3.

Upload your solution as a PDF file.

Q3 (5 points) [Transactions in a Ledger].

Consider the following transactions in a ledger in the style of Bitcoin. Check if the transactions are valid. For each valid transaction, calculate the balances of each person at the end.

(1)

1	Input: ∅ Outputs: 25.0 → Alice	
2	Inputs: 1[0] Outputs: 5.0→Bob, 20.0→Alice	Signed by Alice
3	Inputs: 2[0] Outputs: 3.0→Mike, 2.0→Bob	Signed by Bob
4	Inputs: 2[1] Outputs: 5.0→David, 5.0 → Mike, 8.0→Alice	Signed by Alice
5	Inputs: 3[0], 4[1] Outputs: 2.0→David, 5.0 → Bob, 1.0→Mike	Signed by Mike

(2)

1	Input: ∅ Outputs: 25.0 → Alice	
2	Inputs: 1[0] Outputs: 5.0→Bob, 10.0 → Mike, 10.0→Alice	Signed by Alice
3	Inputs: 2[1] Outputs: 5.0→ David, 5.0→Alice	Signed by Alice
4	Inputs: 2[1] Outputs: 5.0→David, 2.0 → Bob, 3.0 → Mike	Signed by David

(3)

1	Input: ∅ Outputs: 25.0 → Alice	
2	Inputs: 1[0] Outputs: 5.0→Bob, 10.0 → Mike, 10.0→Alice	Signed by Alice
3	Inputs: 2[1] Outputs: 5.0→Bob, 4.0 → Mike	Signed by Mike
4	Inputs: 3[0] Outputs: 5.0→David, 3.0 → Mike, 2.0→Bob	Signed by Bob

Q4 (10 points) [Unlocked Bitcoin Script]

Given two transaction outputs as

(a) 2

5c1ac48bebc5a4b972c6aee74268dedf2adb6cfc46215ef72aa3c4d2b23277bc

0b15c915ec13e6b359c6bcd67225d3dd1cd2eeec37413ee83ab2d4d6a11296fa

9f1ac464eb4676b359c6bee76288dddf1ddbdfcc41719ef61ac3f4d4c23385ba

3 OP_CHECKMULTISIG

(b) OP_DUP OP_HASH160 3c738116c5d45e2e6cc5d57f4b54a72df615d4db OP_EQUALVERIFY OP_CHECKSIG

Explain the purpose of these two locking scripts, give the name of the kind of transaction, explain how the script works, and how to spend the output.

Q5 (10 points) [Bitcoin Script].

Alice and Bob want to play a virtual game of rock-paper-scissors. They do not trust each other, but do not want to be physically present either. It should not be possible to cheat, i.e. to influence the outcome of a player, and the winner is known to all. Create a scheme to implement this game with Bitcoin script.

Q6 (15 points} [PoW and PoS].

- How does bitcoin achieve consensus among distributed nodes? What is the problem behind this that requires an increase in mining difficulty after each 2016 blocks in bitcoin?
- How does Tezos achieve distributed consensus?
- In general, what is the problem in Proof of Work that is solved by Proof of Stake?

Q7 (20 points) [Randomness for Ethereum].

1. Research the internet to find out how to generate random numbers in a contract on the Ethereum blockchain. Name three different approaches to do that and explain their working and their security implications.
2. Give a code fragment in Solidity that implements your preferred approach.

Q8 (15 points) [Michelson Typing].
A charity contract that allows any users to withdraw 10 tez is implemented on Tezos using Michelson language as follows.

```
parameter address;
storage string;
code {
      # pair address string : []
      UNPAIR;
      CONTRACT unit;
      IF_SOME {} { FAILWITH; };
      PUSH mutez 10000000;
      UNIT;
      TRANSFER_TOKENS;
      NIL operation; SWAP; CONS; PAIR};
```

1. Give the type of the stack after each instruction.
2. If we change the type of the storage to Unit, explain whether the script is well-typed.
3. Explain two cases in which a contract will fail.

Q9 (50 points) [Solidity-Heap Data Structure].
Write a contract to implement a heap data structure on Ethereum using the Solidity language. A heap is a tree-based data structure, in which a node can have one parent node and may have child nodes. The top node that does not have a parent is called the root node. A heap satisfies the head property, which states that for any given node C, if P is a parent node of C, then the value of P is greater than or equal to the value of C. We consider a heap to be a "left-first" tree, i.e., if we insert a new node into the heap and it happens to be at a “leaf” node that has no children, it will be the left child of the node. The contract should provide the following functions:

- ***insert-elem***, which allows users to insert a new key.
- ***get-max***, which returns the value of the root node (the highest value).
- ***pop-max***, which removes the root node from the heap and then rebalances the heap.

Upload your solution as a Solidity file with extension .sol

Q10 (50 points) [Michelson - Deposit Contract].

Write a deposit contract for an on-chain wallet on Tezos using Michelson language. The contract will be created with an empty balance (zero coins) and will start in an open state where deposits are

accepted. Anyone can deposit, but only the contract owner can withdraw. The contract's storage contains its balance and the owner's address. There are two ways to interact with the contract: (1) deposit coins or withdraw an amount of coins. These interactions can be specified as two entrypoints in Michelson.

```
parameter (or (unit %deposit) (mutez %withdraw));
storage (pair mutez address);
```

Upload your solution as a Michelson file with extension .tz

Hints.

1. Deposit entrypoint
   - update the balance of the contract.
2. Withdraw entrypoint:
   - check if the caller is the contract owner.
   - check if the contract balance is sufficient.
   - update its balance.
   - transfer the coins.

Q11 (60 points) [*Recoin*]

Implement the *Recoin* system that works according to the following strategies:

- The blockchain contains a linked list of blocks tracked by its head, the root hash (the hash of the head), a value of reward for miners, which is currently 10 coins, a list of accounts, and a mempool containing transactions that were injected but not yet in a block on the blockchain.

head	The head of the chain
award	The award value to miners
account_list	The list of accounts on the blockchain
mempool	The list of pending transactions
root_hash	The root hash of the chain (the hash of the head)

- A block contains

blockNo	The block number
seed	The seed that is used to determine the mining right
timestamp	The approximate creation time of this block
prev_hash	The hash of the previous block
prev	The pointer points to the previous block
trans_list	The list of transactions

- A user can transfer his coins to another by creating a new transaction. When a valid transaction is injected, it is added into the mempool that contains all pending transactions before they may be included to a block. A transaction contains

sender	The sender's address
recipient	The receiver's address
counter	The counter's value of the sender
amount	The amount of coins

fee	The fee to a miner
timestamp	The time when the transaction is added into the mempool
sig	The sender’s signature

- Each participant on the blockchain is addressed as an account, identified by its alias (no two accounts have the same alias). An account stores its account balance and a counter whose form is a value-flag pair (n, b), where n is the counter's value and b is its flag. The is used to prevent replay attacks. The sender account can only inject a transaction if the counter value is correct and the counter's flag is unlocked (False) and if that transaction is added into the mempool, the flag is locked (True). Once a transaction is included in a block on the blockchain, the counter's value is incremented by 1 and the counter's flag is set to False. For a newly created account, the account balance is 0 coins, the counter's value is 0, and the counter's flag is False. Here is a sample account in a Json format (Python dictionary).

```
{ "alias": alice

  "balance": 25,

  "counter": {"num": 8, "flag": False} }
```

- A transaction can only be added into the mempool if it is valid. Otherwise, it is rejected. The transaction will stay in the mempool and wait to be included in a block by a miner. A transaction is valid if.

  - ✔ The sender’s address and the recipient's address are correct.
  - ✔ The counter is correct (it is the same as the current counter's value of the sender account).
  - ✔ The counter’s flag of the sender account is unlocked (False)
  - ✔ The coins spent plus the fee is less or equal to the sender account’s balance
  - ✔ The signature is valid.

- A new block is mined by a miner. Miners collect valid transactions in the pending pool based on their transaction fees. Transactions with high transaction fees get selected first. A maximum of 10 transactions would be included in a block. A miner can only mine a block if it has the mining right, which is determined by the seed in the current head of the chain. When a miner mines a new block, he randomly generates a seed (a natural number) in the range of the account list size. This seed value determines the index of an account in the account list, and only that account has the mining right to mine the next block. For mining the block after the generic block, everyone gets to mine. A miner receives 10 coins as a reward for each new block created.

- When a new block is added to the blockchain, the mempool is refreshed such that all transactions that are included in the new block are removed and also for the timeout transactions that have been in the mempool for more than 60 minutes.
- The system provides a function to verify whether the current account list is valid. It is valid if the balance of each account is correct. The chain, block and transaction should be validated as well. Some other query functions should be implemented to obverse the blockchain.

We suggest the following structure for Python code that implements the *recoin* system. This way we can assess each step and give marks for partial solutions. If you use a different approach, you should structure it 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 *recoin* and one subprogram as follows.
   - The *crypto_key* program that implements all functions to generate private/public keys, sign a document, and verify a signature (this program is provided).
2. The main *recoin* program should contain three classes Transaction, Block, and Recoin as follows.
   - black text is Python code.
   - blue text is comments.

```
import hashlib
import time
import random
from crypto_key import generate_key
from crypto_key import sign
from crypto_key import verify_sig

class Transaction:

    # create a new transaction
    def __init__(self, sender, recipient, counter, amount, fee):
        self._sender = sender
        self._recipient = recipient
        self._counter = counter
        self._amount = amount
        self._fee = fee
        self._timestamp = time.time()
        self._sig = ""

    # calculate the transaction hash
    def digest(self):
        m = hashlib.sha256()
```

```
        m.update(
        str(self._sender).encode('utf-8') +
        str(self._recipient).encode('utf-8') +
        str(self._counter).encode('utf-8') +
        str(self._amount).encode('utf-8') +
        str(self._fee).encode('utf-8') +
        str(self._timestamp).encode('utf-8'))
        return m.hexdigest()

    # sign a transaction
    def sign(self, alias):
        self._sig = sign(self.digest(), alias)

    # show the transaction content
    def print_trans(self):

        # write your code here

class Block:

     # create a new block
     def __init__ (self, trans_list, seed = None, prev = None):
         self._blockNo = prev._blockNo + 1 if prev else 0
         self._seed = seed
         self._trans_list = trans_list
         self._prev = prev
         self._prev_hash = prev.digest() if prev is not None else
     bytearray(256)
         self._timestamp = time.time()

     # calculate the block hash
     def digest(self):
         m = hashlib.sha256()
         m.update(
                str(self._blockNo).encode('utf-8') +
                str(self._seed).encode('utf-8') +
                str(self._trans_list).encode('utf-8') +
               str(self._prev).encode('utf-8') +
                str(self._prev_hash).encode('utf-8') +
                str(self._timestamp).encode('utf-8')
           )
         return m.hexdigest()

     # verify blocks
     def verify(self, root_hash):
         my_hash = self.digest()
         if (root_hash != my_hash):
             return False
         return (root_hash == my_hash and (not self._prev or
     self._prev.verify (self._prev_hash)))

     # show the block content
     def print_block(self):

         # write your code here
```

```
class Recoin:

      # create a new blockchain
      def __init__(self):
          self._head = Block([], None, None)
          self.__award = 10
          self.__account = []
          self.__mempool = []
          self.__root_hash = self._head.digest()

       #--------- 1. create a new account on the blockchain (5 points)
         # conditions:
         # (1) check if the alias already exists
         # notes:
         # (1) generate a pair of keys for the alias
         # (2) Json format
          # {"alias": alias,
         #  "balance": 0,
         #  "counter":
          #           {"num": 0, "flag": False}}
         def create_account(self, alias):

              # write your code here

         #--------- 2. verify a transaction (5 points)
         # 2.1. verify a transaction before adding it into the mempool
         # conditions:
         # (1) check if the sender account exists
         # (2) check if the recipient account exists
         # (3) verify the signature
         # (4) check if the sender's balance is sufficient
         # (5) check if the counter value is correct
         # (6) verify the sender counter's flag
          def verify_trans_add(self, trans):

              # write your code here

         # 2.2. verify a transaction in the mempool or in a block
         # conditions:
         # (1) check if the sender account exists
         # (2) check if the recipient account exists
         # (3) verify the signature
         # (4) check if the sender's balance is sufficient
         # (5) check if the counter value is correct
         def verify_trans_include(self, trans):

              # write your code here

         #--------- 3. add a transaction to the mempool (5 points)
         # conditions:
         # (1) check if the transaction is valid
         # notes:
         # (1) switch the counter's flag of the sender to False
          def add_transaction(self, trans):
```

```
        # write your code here

    # suggestions:
    # 3.1. update the counter's flag
    def __update_counter_flag(self, alias):

        # write your code here

    #--------- 4. verify a new block created (5 points)
    # conditions:
    # (1) check if the blockNo is correct
    # (2) check if the prev points to the current head
    # (3) check if the prev_hash is the hash of the current head
    # (4) check if the seed is not empty
    # (5) check if all transactions in the block are valid
    def verify_block(self, block):

        # write your code here

    # suggestions:
    # 4.1. get the balance of an account
    def get_balance(self, alias):

        # write your code here

    # 4.2. get the counter's value of an account
    def get_counter_num(self, alias):

        # write your code here

    # 4.3 get the counter's flag of an account
    def get_counter_flag(self, alias):

        # write your code here

    # 4.4. check whether an account exists
    def check_account(self, alias):

        # write your code here


    #--------- 5. add a block to the blockchain (10 points)
    # conditions:
    # (1) check if the account (alias) has the mining right
    # (2) check if the block created is valid
    # notes:
    # (1) update the sender account's balance and counter
    # (2) update the recipient account's balance
    # (3) pay the award to the miner
    # (4) refresh the mempool
    def __add_block(self, block, alias):

        # write your code here
```

```
        # suggestions:
        # 5.1. pay the award to the miner
        def __pay_award(self, alias):

            # write your code here

        # 5.2. update the sender account's balance and counter
        def __update_sender(self, trans):

           # write your code here

        # 5.3. update recipient's balance
        def __update_receiver(self, trans):

           # write your code here

        # 5.4. refresh the mempool
        # notes:
         # (1) remove the transactions that have included in the block
         # (2) remove the transactions that have been in the mempool more
than 60 minutes
         def __refresh_mempool(self, block):

             # write your code here

        #--------- 6. mine a block (10 points)
        # notes:
        # (1) collects maximum 10 valid transactions sorted by
transaction fee
        # (2) generate a random seed that is in the range of the account
list's size and not the miner account's index
        def mine(self, alias):

            # write your code here

        # suggestions:
        # 6.1 collects all valid transactions from the pending pool.
         def collect_valid_trans(self):

             # write your code here

         # 6.2 returns an array of maximum 10 valid transactions sorted
by transaction fee.
         def sort_trans_by_fee(self, valid_trans):

             # write your code here

        # 6.3 get the index of an account in the account list
         def get_account_index(self, alias):

            # write your code here

        #--------- 7. verify the validation of the account list (10
points)
        # conditions:
```

```
        # check if the balance of each account is consistent with the
    coins that have spent and the coins that have received
        def verify_account_list(self):

            # write your code here

        # suggestions:
        # 7.1. get all coins spent of an account
        def coins_spent(self, alias):

            # write your code here

        # 7.2. get all coins received of an account
        def coins_recieved(self, alias):

            # write your code here

        #--------- 8. some queries (5 points)
         # verify the chain
        def verify (self):
            return not self._head or
    self._head.verify(self.__root_hash)

        # list all accounts
        def show_account_list(self):

            # write your code here

        # show the mempool
        def show_mempool(self):

            # write your code here

        # show the root hash
        def show_root_hash(self):

            # write your code here

        # show the head block
        def show_head(self):

            # write your code here

        # show an account
        def show_account(self, alias):

            # write your code here

        # there may be some more function
        # write your code here

# 9. run and test (5 points)
# create a new coin system
recoin = Recoin();
# create accounts
```

```
recoin.create_account("alice")
recoin.create_account("bob")

# create an existing account
recoin.create_account("bob")
# show all accounts
recoin. show_account_list()
# mine a block
recoin.mine("bob")
# see bob account
recoin.show_account("bob")
# transfer coins from bob to alice
trans = Transaction("bob", "alice", 0, 5, 1)
trans.sign("bob")
recoin.add_transaction(trans)
# transfer coins from bob to alice
trans = Transaction("bob", "alice", 0, 2, 1)
trans.sign("bob")
recoin.add_transaction(trans)
# transfer coins from alice to bob
trans = Transaction("alice", "bob", 0, 1, 1)
trans.sign("alice")
recoin.add_transaction(trans)
# show the mempool
recoin.show_mempool()
# mine a block
recoin.mine("bob")
# show the head block
recoin.show_head()
# mine a block
recoin.mine("alice")
# transfer coins from bob to alice
trans = Transaction("bob", "alice", 0, 2, 1)
trans.sign("bob")
recoin.add_transaction(trans)
# transfer coins from bob to alice
trans = Transaction("bob", "alice", 1, 2, 1)
trans.sign("bob")
recoin.add_transaction(trans)
# transfer coins from alice to bob
trans = Transaction("alice", "bob", 0, 1, 1)
trans.sign("alice")
recoin.add_transaction(trans)
# show the mempool
recoin.show_mempool()
# create an accounts for mike
recoin.create_account("mike")
# transfer coins from alice to mike signed by mike
trans = Transaction("alice", "mike", 1, 2, 1)
trans.sign("mike")
recoin.add_transaction(trans)
# mine a block
recoin.mine("mike")
# mine a new block
recoin.mine("alice")
```

```
# mine a new block
recoin.mine("bob")
# show show_account_list
recoin.show_account_list()
# show the balance of bob
recoin.show_account("bob")
# verify the chain
recoin.verify()
# verify account list
recoin.verify_account_list()
```