# INDEX

| Sr. No | | Practical | Signature |
|---|---|---|---|
| 1 | A | Develop a secure messaging application where users can exchange messages securely using RSA encryption. Implement a mechanism for generating RSA key pairs and encrypting/decrypting messages. | |
| | B | Allow users to create multiple transactions and display them in an organised format. | |
| | C | Create a Python class named Transaction with attributes for sender, receiver, and amount. Implement a method within the class to transfer money from the sender's account to the receiver's account.. | |
| | D | Implement a function to add new blocks to the miner and dump the blockchain. | |
| | | | |
| 2 | A | Write a python program to demonstrate mining. | |
| | B | Demonstrate the use of the Bitcoin Core API to interact with a Bitcoin Core node. | |
| | C | Demonstrating the process of running a blockchain node on your local machine. | |
| | | | |
| 3 | A | Write a Solidity program that demonstrates various types of functions including regular functions, view functions, pure functions, and the fallback function. | |
| | B | Write a Solidity program that demonstrates function overloading, mathematical functions, and cryptographic functions. | |
| | C | Write a Solidity program that demonstrates various features including contracts, inheritance, constructors, abstract contracts, interfaces. | |
| | D | Write a Solidity program that demonstrates use of libraries, assembly, events, and error handling | |
| | | | |
| 4 | A | Install and demonstrate use of hyperledger-Irhoa. | |

## Practical 1

**A. Develop a secure messaging application where users can exchange messages securely using RSA encryption. Implement a mechanism for generating RSA key pairs and encrypting/decrypting messages.**

```
import hashlib
import random
import string
import json
import binascii
import numpy as np
import pandas as pd
import pylab as pl
import logging
import datetime
import collections
import Crypto
import Crypto.Random
from Crypto.Hash import SHA
from Crypto.PublicKey import RSA
from Crypto.Signature import PKCS1_v1_5
import binascii
class Client:
    def __init__(self):
        random = Crypto.Random.new().read
        self._private_key = RSA.generate(1024, random)
        self._public_key = self._private_key.publickey()
        self._signer = PKCS1_v1_5.new(self._private_key)

self.identity=binascii.hexlify(self._public_key.exportKey(format='DER')).decode('asc
ii')

class Transaction:
    def __init__(self, sender, recipient, value):
        self.sender = sender
        self.recipient = recipient
        self.value = value
        self.time = datetime.datetime.now()

    def to_dict(self):
        if self.sender == "Genesis":
            identity = "Genesis";
        else:
            identity = self.sender.identity
        return collections.OrderedDict({
            'sender': identity,
            'recipient': self.recipient,
            'value': self.value,
            'time': self.time})
```

```
    def sign_transaction(self):
        private_key = self.sender._private_key
        signer = PKCS1_v1_5.new(private_key)
        h = SHA.new(str(self.to_dict()).encode('utf8'))
        return binascii.hexlify(signer.sign(h)).decode('ascii')

client1 = Client()
client2 = Client()

print (client1.identity)
print("*******")
print (client2.identity)
```

**B. Allow users to create multiple transactions and display them in an organised format.**

**C. Create a Python class named Transaction with attributes for sender, receiver, and amount. Implement a method within the class to transfer money from the sender's account to the receiver's account..**

```
import hashlib
import random
import string
import json
import binascii
import numpy as np
import pandas as pd
import pylab as pl
import logging
import datetime
import collections
import Crypto
import Crypto.Random
from Crypto.Hash import SHA
from Crypto.PublicKey import RSA
from Crypto.Signature import PKCS1_v1_5
import binascii
class Client:
    def __init__(self):
        random = Crypto.Random.new().read
        self._private_key = RSA.generate(1024, random)
        self._public_key = self._private_key.publickey()
        self._signer = PKCS1_v1_5.new(self._private_key)

self.identity=binascii.hexlify(self._public_key.exportKey(format='DER')).decode('asc
ii')

class Transaction:
    def _init_(self, sender, recipient, value):
        self.sender = sender
        self.recipient = recipient
        self.value = value
        self.time = datetime.datetime.now()

    def to_dict(self):
        if self.sender == "LDS":
            identity = "LDS"
        else:
            identity = self.sender.identity
        return collections.OrderedDict({
            'sender': identity,
            'recipient': self.recipient,
            'value': self.value,
```

```
            'time': self.time})

    def sign_transaction(self):
        private_key = self.sender._private_key
        signer = PKCS1_v1_5.new(private_key)
        h = SHA.new(str(self.to_dict()).encode('utf8'))
        return binascii.hexlify(signer.sign(h)).decode('ascii')

Manisha = Client()
Karan = Client()
t = Transaction (sender=Manisha,recipient=Karan.identity,value=5.0)
signature = t.sign_transaction()
print (signature)

def display_transaction(transaction):
#for transaction in transactions:
    dict = transaction.to_dict()
    print ("sender:" + dict['sender'])
    print ('====================')
    print ("recipient:" + dict['recipient'])
    print ('====================')
    print ("value:" + str(dict['value']))
    print ('====================')
    print ("time:" + str(dict['time']))
    print ('====================')

transactions = []
Manisha = Client()
Karan = Client()
Seema = Client()
Priti = Client()
t1 = Transaction( Manisha, Karan.identity, 15.0 )
t1.sign_transaction()
transactions.append(t1)
t2 = Transaction( Manisha, Seema.identity, 6.0 )
t2.sign_transaction()
transactions.append(t2)
t3 = Transaction( Karan, Priti.identity, 2.0)
t3.sign_transaction()
transactions.append(t3)
t4 = Transaction( Seema, Karan.identity, 4.0 )
t4.sign_transaction()
transactions.append(t4)
t5 = Transaction( Priti, Seema.identity, 7.0 )
t5.sign_transaction()
transactions.append(t5)
t6 = Transaction( Karan, Seema.identity, 3.0 )
t6.sign_transaction()
transactions.append(t6)
t7 = Transaction( Seema, Manisha.identity, 8.0 )
```

```
t7.sign_transaction()
transactions.append(t7)
t8 = Transaction( Seema, Karan.identity, 1.0 )
t8.sign_transaction()
transactions.append(t8)
t9 = Transaction( Priti, Manisha.identity, 5.0 )
t9.sign_transaction()
transactions.append(t9)
t10 = Transaction( Priti, Karan.identity, 3.0 )
t10.sign_transaction()
transactions.append(t10)

for transaction in transactions:
    display_transaction (transaction)
print ('================================')
```

**D. Implement a function to add new blocks to the miner and dump the blockchain.**

```
import hashlib
import random
import string
import json
import binascii
import numpy as np
import pandas as pd
import pylab as pl
import logging
import datetime
import collections
import Crypto
import Crypto.Random
from Crypto.Hash import SHA
from Crypto.PublicKey import RSA
from Crypto.Signature import PKCS1_v1_5
import binascii

class Client:
    def __init__(self):
        random = Crypto.Random.new().read
        self._private_key = RSA.generate(1024, random)
        self._public_key = self._private_key.publickey()
        self._signer = PKCS1_v1_5.new(self._private_key)

self.identity=binascii.hexlify(self._public_key.exportKey(format='DER')).decode('ascii')

class Transaction:
    def __init__(self, sender, recipient, value):
        self.sender = sender
        self.recipient = recipient
        self.value = value
        self.time = datetime.datetime.now()

    def to_dict(self):
        if self.sender == "LDS":
            identity = "LDS"
        else:
            identity = self.sender.identity
        return collections.OrderedDict({
            'sender': identity,
            'recipient': self.recipient,
            'value': self.value,
            'time': self.time})

    def sign_transaction(self):
        private_key = self.sender._private_key
        signer = PKCS1_v1_5.new(private_key)
```

```
        h = SHA.new(str(self.to_dict()).encode('utf8'))
        return binascii.hexlify(signer.sign(h)).decode('ascii')

Manisha = Client()
Karan = Client()
t = Transaction (sender=Manisha,recipient=Karan.identity,value=5.0)
signature = t.sign_transaction()
print (signature)
def display_transaction(transaction):
    #for transaction in transactions:
    dict = transaction.to_dict()
    print ("sender:" + dict['sender'])
    print ('====================')
    print ("recipient:" + dict['recipient'])
    print ('====================')
    print ("value:" + str(dict['value']))
    print ('====================')
    print ("time:" + str(dict['time']))
    print ('====================')

transactions = []
Manisha = Client()
Karan = Client()
Seema = Client()
Priti = Client()
t1 = Transaction( Manisha, Karan.identity, 15.0 )
t1.sign_transaction()
transactions.append(t1)
t2 = Transaction( Manisha, Seema.identity, 6.0 )
t2.sign_transaction()
transactions.append(t2)
t3 = Transaction( Karan, Priti.identity, 2.0)
t3.sign_transaction()
transactions.append(t3)
t4 = Transaction( Seema, Karan.identity, 4.0 )
t4.sign_transaction()
transactions.append(t4)
t5 = Transaction( Priti, Seema.identity, 7.0 )
t5.sign_transaction()
transactions.append(t5)
t6 = Transaction( Karan, Seema.identity, 3.0 )
t6.sign_transaction()
transactions.append(t6)
t7 = Transaction( Seema, Manisha.identity, 8.0 )
t7.sign_transaction()
transactions.append(t7)
t8 = Transaction( Seema, Karan.identity, 1.0 )
t8.sign_transaction()
transactions.append(t8)
t9 = Transaction( Priti, Manisha.identity, 5.0 )
```

```
t9.sign_transaction()
transactions.append(t9)
t10 = Transaction( Priti, Karan.identity, 3.0 )
t10.sign_transaction()
transactions.append(t10)

for transaction in transactions:
    display_transaction (transaction)
print ('============================')

class Block:
    def __init__(self):
        self.verified_transactions = []
        self.previous_block_hash = ""
        self.Nonce = ""

last_block_hash = ""
Manisha = Client()
t0 = Transaction ( "LDS", Karan.identity, 500.0 )

block0 = Block()
block0.previous_block_hash = None
Nonce = None
block0.verified_transactions.append (t0)
digest = hash (block0)
last_block_hash = digest
TPCoins = []

def dump_blockchain (self):
    print ("Number of blocks in the chain:" + str(len (self)))
    for x in range (len(TPCoins)):
        block_temp = TPCoins[x]
        print ("block #" + str(x))
        for transaction in block_temp.verified_transactions:
            display_transaction (transaction)
            print ('==========================================')
        print ('==========================================')
TPCoins.append (block0)
dump_blockchain(TPCoins)
```

```
time:2025-06-06 10:36:39.923717
====================
sender:30819f300d06092a864886f70d010101050003818d0030818902818100e68bf736d3664542d08f40ca4ebcaa8aa5fb411d0a62f03f7c32a29d262b828f86fd136125be2ba35c32f14a78d7088eb96c473de93628f4678336cec7b
b42b2fbf497f118faedc5ccf170eb4124e054858ff8b9b7d59775a0fe24cece4dcc88baa290dd2144b083c917611c3178245ede709f48f8ab967b2da7f0b2d35b74ab0203010001
====================
recipient:30819f300d06092a864886f70d010101050003818d0030818902818100f3f58e3950bf07934149de06e06490907876fd21c9fb8228ffe5f6b5742d969b4ab38dc9b5c83bf576092a42e7cdab9a8fe99888f000626a6a6bef49
79b2b90751541cd1999c195131bb161a05093fc81a7330697d1ff576c8b8e80ffb559ba8484fc81e428d667f923a7b7a30fb5b2556074316d35cc69bf3100044af5365fb0203010001
====================
value:1.0
====================
time:2025-06-06 10:36:39.923717
====================
sender:30819f300d06092a864886f70d010101050003818d0030818902818100a93af40de2b8575c033293092ba151924538790d60fc83eb457724fff93a4bd766da27b8b17d25678ba94a2b9a057ee487a870fa0eb74f3b082038978b4
2ddb72ae9add17d718285ecb2b97e4e5fc0c260ea49e9cb61fdfdc6b5e98d3b328c93c314aab63fad19f2b37cfae3705cd084aee3d76bc0fbd66a785ed019a2cef0df0203010001
====================
recipient:30819f300d06092a864886f70d010101050003818d00308189028181009782f7c49db72935ad24527d9fe2106d1005abc8961c36cca053827cff759b495a2290fc349af5e2a54e48b3e11ab2089fa3c53881b70851b1e4a85
0185fef770e21cb95ca7372d152f612a9c67c49f5dc389dc21c4857cae5294c07c1fe7216d6ad222587b04dcd450b80e67e34538fa5b22d3994a5e3489983fc5bced95b70203010001
====================
value:5.0
====================
time:2025-06-06 10:36:39.923717
====================
sender:30819f300d06092a864886f70d010101050003818d0030818902818100a93af40de2b8575c033293092ba151924538790d60fc83eb457724fff93a4bd766da27b8b17d25678ba94a2b9a057ee487a870fa0eb74f3b082038978b4
2ddb72ae9add17d718285ecb2b97e4e5fc0c260ea49e9cb61fdfdc6b5e98d3b328c93c314aab63fad19f2b37cfae3705cd084aee3d76bc0fbd66a785ed019a2cef0df0203010001
====================
recipient:30819f300d06092a864886f70d010101050003818d0030818902818100f3f58e3950bf07934149de06e06490907876fd21c9fb8228ffe5f6b5742d969b4ab38dc9b5c83bf576092a42e7cdab9a8fe99888f000626a6a6bef49
79b2b90751541cd1999c195131bb161a05093fc81a7330697d1ff576c8b8e80ffb559ba8484fc81e428d667f923a7b7a30fb5b2556074316d35cc69bf3100044af5365fb0203010001
====================
value:3.0
====================
time:2025-06-06 10:36:39.923717
====================
============================
Number of blocks in the chain:1
block #0
sender:LDS
====================
recipient:30819f300d06092a864886f70d010101050003818d0030818902818100f3f58e3950bf07934149de06e06490907876fd21c9fb8228ffe5f6b5742d969b4ab38dc9b5c83bf576092a42e7cdab9a8fe99888f000626a6a6bef49
79b2b90751541cd1999c195131bb161a05093fc81a7330697d1ff576c8b8e80ffb559ba8484fc81e428d667f923a7b7a30fb5b2556074316d35cc69bf3100044af5365fb0203010001
====================
value:500.0
====================
time:2025-06-06 10:36:40.788674
====================
```

## Practical 2

### A. Write a python program to demonstrate mining.

```
import hashlib
def sha256(message):
    return hashlib.sha256(message.encode('ascii')).hexdigest()
def mine(message, difficulty=1):
   assert difficulty >= 1
   prefix = '1' * difficulty
   for i in range(1000):
      digest = sha256(str(hash(message)) + str(i))
      if digest.startswith(prefix):
         print ("after " + str(i) + " iterations found nonce: "+ digest)
         return digest
mine ("welcome", 2)
```

```
>>>
========== RESTART: C:/WORK/MSC IT/bloc
kchain/new2a.py ==========
after 258 iterations found nonce: 11ad95a2996e2c26c523bc7b298dd0de8ee83513b167943e5d07bce55e0767f9
```

**B. Demonstrate the use of the Bitcoin Core API to interact with a Bitcoin Core node.**

**C. Demonstrating the process of running a blockchain node on your local machine.**

**What Is A Full Node?**

A full node is a program that fully validates transactions and blocks. Almost all full nodes also help the network by accepting transactions and blocks from other full nodes, validating those transactions and blocks, and then relaying them to further full nodes. Most full nodes also serve lightweight clients by allowing them to transmit their transactions to the network and by notifying them when a transaction affects their wallet. If not enough nodes perform this function, clients won't be able to connect through the peer-to-peer network—they'll have to use centralized services instead.

Many people and organizations volunteer to run full nodes using spare computing and bandwidth resources—but more volunteers are needed to allow Bitcoin to continue to grow. This document describes how you can help and what helping will cost you.

**Costs And Warnings**

Running a Bitcoin full node comes with certain costs and can expose you to certain risks. This section will explain those costs and risks so you can decide whether you're able to help the network.

**Special Cases**

Miners, businesses, and privacy-conscious users rely on particular behavior from the full nodes they use, so they will often run their own full nodes and take special safety precautions. This document does not cover those precautions—it only describes running a full node to help support the Bitcoin network in general.

Please seek out assistance in the community if you need help setting up your full node correctly to handle high-value and privacy-sensitive tasks. Do your own diligence to ensure who you get help from is ethical, reputable and qualified to assist you.

**Secure Your Wallet**

It's possible and safe to run a full node to support the network and use its wallet to store your bitcoins, but you must take the same precautions you would when using any Bitcoin wallet. Please see the securing your wallet page for more information.

Minimum Requirements

Bitcoin Core full nodes have certain requirements. If you try running a node on weak hardware, it may work—but you'll likely spend more time dealing with issues. If you can meet the following requirements, you'll have an easy-to-use node.

- Desktop or laptop hardware running recent versions of Windows, Mac OS X, or Linux.
- 7 gigabytes of free disk space, accessible at a minimum read/write speed of 100 MB/s.
- 2 gigabytes of memory (RAM)
- A broadband Internet connection with upload speeds of at least 400 kilobits (50 kilobytes) per second
- An unmetered connection, a connection with high upload limits, or a connection you regularly monitor to ensure it doesn't exceed its upload limits. It's common for full nodes on high-speed connections to use 200 gigabytes upload or more a month. Download usage is around 20 gigabytes a month, plus around an additional 340 gigabytes the first time you start your node.
- 6 hours a day that your full node can be left running. (You can do other things with your computer while running a full node.) More hours would be better, and best of all would be if you can run your node continuously.

**Possible Problems**

- **Legal:** Bitcoin use is prohibited or restricted in some areas.
- **Bandwidth limits**: Some Internet plans will charge an additional amount for any excess upload bandwidth used that isn't included in the plan. Worse, some providers may terminate your connection without warning because of overuse. We advise that you check whether your Internet connection is subjected to such limitations and monitor your bandwidth use so that you can stop Bitcoin Core before you reach your upload limit.
- **Anti-virus:** Several people have placed parts of known computer viruses in the Bitcoin block chain. This block chain data can't infect your computer, but some anti-virus programs quarantine the data anyway, making it more difficult to run Bitcoin Core. This problem mostly affects computers running Windows.
- **Attack target:** Bitcoin Core powers the Bitcoin peer-to-peer network, so people who want to disrupt the network may attack Bitcoin Core users in ways that will affect other things you do with your computer, such as an attack that limits your available download bandwidth.

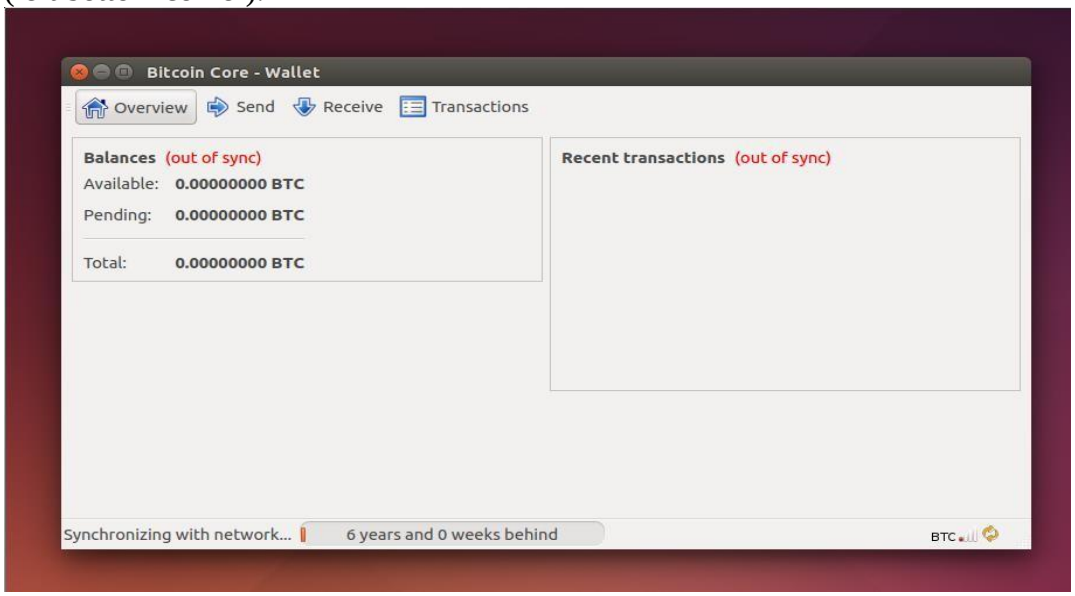
**Initial Block Download(IBD)**

Initial block download refers to the process where nodes synchronize themselves to the network by downloading blocks that are new to them. This will happen when a node is far behind the tip of the best block chain. In the process of IBD, a node does not accept incoming transactions nor request mempool transactions.

If you are trying to set up a new node following the instructions below, you will go through the IBD process at the first run, and it may take a considerable amount of time since a new node has to download the entire block chain (which is roughly 340 gigabytes now). During the download, there could be a high usage for the network and CPU (since the node has to verify the blocks downloaded), and the client will take up an increasing amount of storage space (reduce storage provides more details on reducing storage).

Before the node finishes IBD, you will not be able to see a new transaction related to your account until the client has caught up to the block containing that transaction. So your wallet may not count new payments/spendings into the balance.

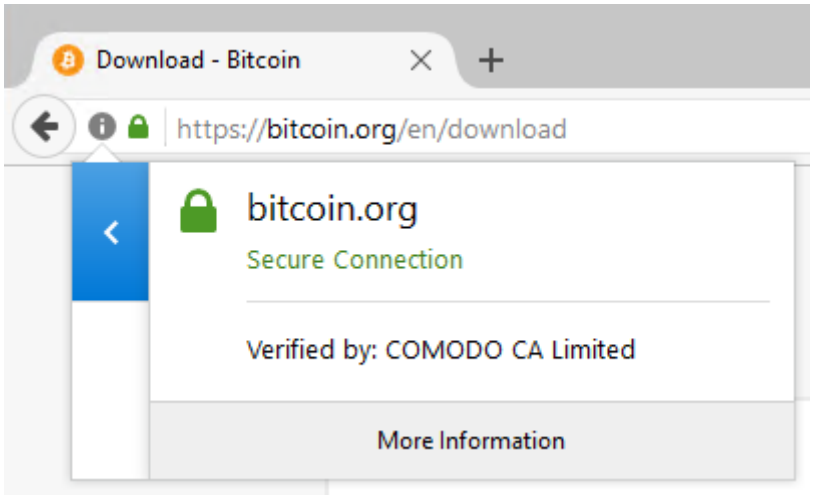
If you are using Bitcoin Core GUI, you can monitor the progress of IBD in the status bar (left bottom corner).



**Windows Instructions**

Windows 10

Go to the Bitcoin Core download page and verify you have made a secure connection to the server.




Click the large blue *Download Bitcoin Core* button to download the Bitcoin Core installer to your desktop.

If you know how to use PGP, you should also click the *Verify Release Signatures* link on the download page to download a signed list of SHA256 file hashes. The 0.11 and later releases are signed by Wladimir J. van der Laan's releases key with the fingerprint:

01EA 5486 DE18 A882 D4C2 6845 90C8 019E 36C2 E964

Earlier releases were signed by Wladimir J. van der Laan's regular key. That key's fingerprint is:

71A3 B167 3540 5025 D447 E8F2 7481 0B01 2346 C9A6

Even earlier releases were signed by Gavin Andresen's key. His primary key's fingerprint is:

2664 6D99 CBAE C9B8 1982 EF60 29D9 EE6B 1FC7 30C1

You should verify these keys belong to their owners using the web of trust or other trustworthy means. Then use PGP to verify the signature on the release signatures file. Finally, use PGP or another utility to compute the SHA256 hash of the archive you downloaded, and ensure the computed hash matches the hash listed in the verified release signatures file.

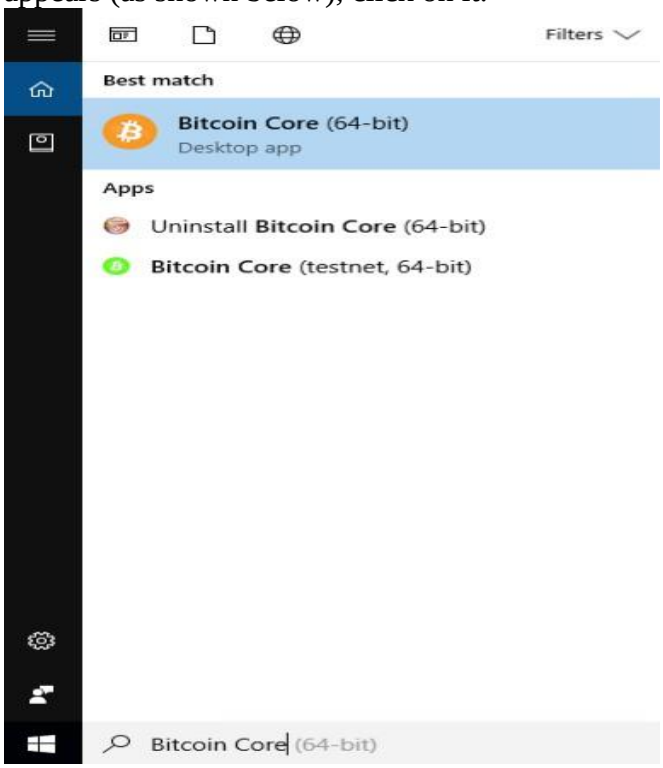
After downloading the file to your desktop or your Downloads folder (C:\Users\<YOUR USER NAME>\Downloads), run it by double-clicking its icon. Windows will ask you to confirm that you want to run it. Click Yes and the Bitcoin installer will start. It's a typical Windows installer, and it will guide you through the decisions you need to make about where to install Bitcoin Core.


*To continue, choose one of the following options*

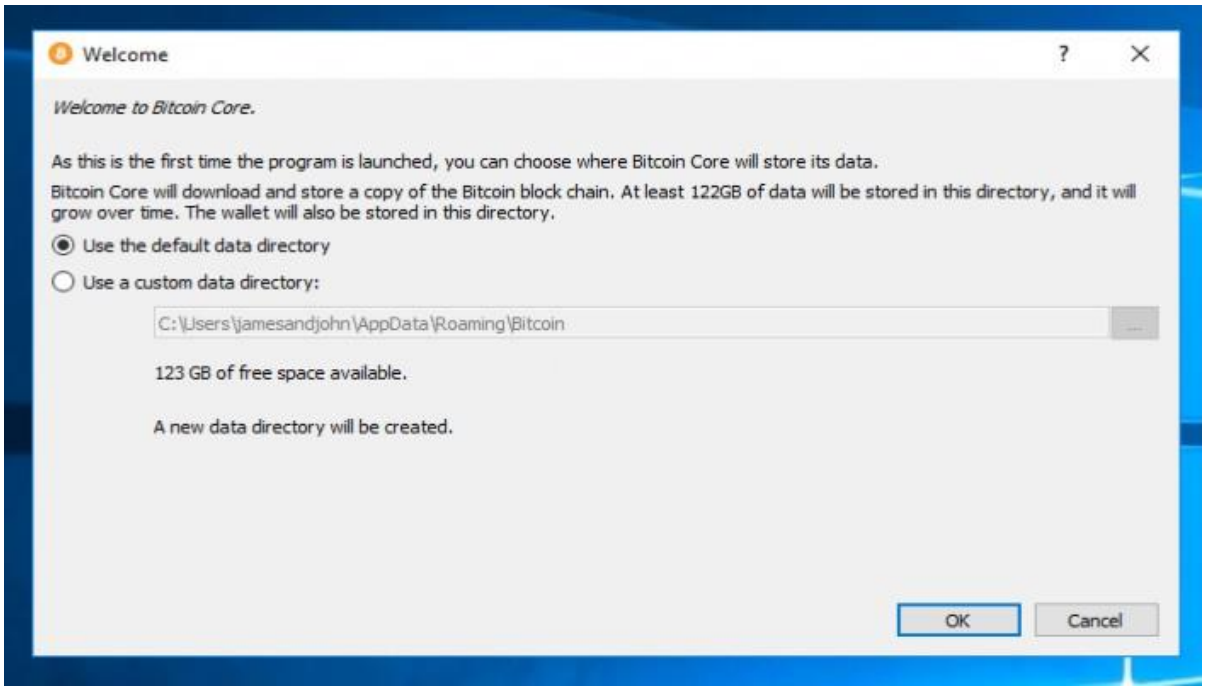
1. If you want to use the Bitcoin Core Graphical User Interface (GUI), proceed to the Bitcoin Core GUI section below.
2. If you want to use the Bitcoin Core daemon (bitcoind), which is useful for programmers and advanced users, proceed to the Bitcoin Core Daemon section below.
3. If you want to use both the GUI and the daemon, read both the GUI instructions and the daemon instructions. Note that you can't run both the GUI and the daemon at the same time using the same configuration directory.

Bitcoin Core GUI

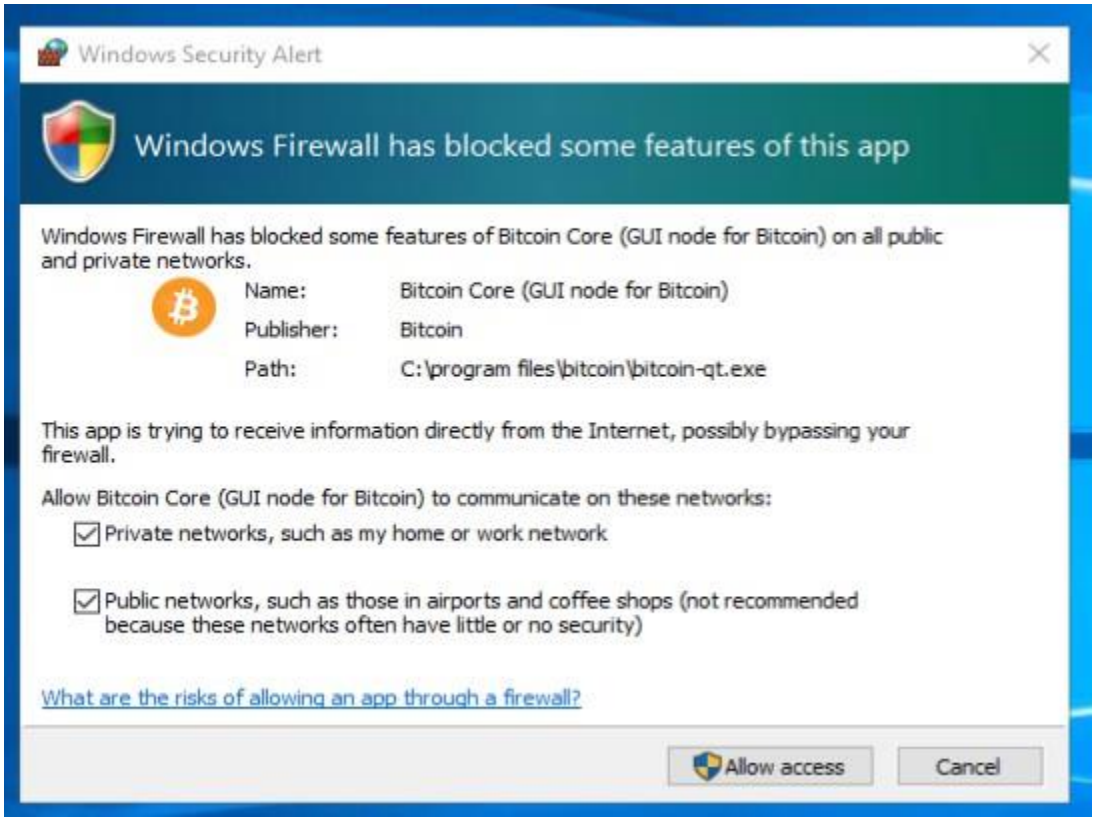
Press the Windows key (⊞ Win) and start typing "bitcoin". When the Bitcoin Core icon appears (as shown below), click on it.



You will be prompted to choose a directory to store the Bitcoin block chain and your wallet. Unless you have a separate partition or drive you want to use, click Ok to use the default.

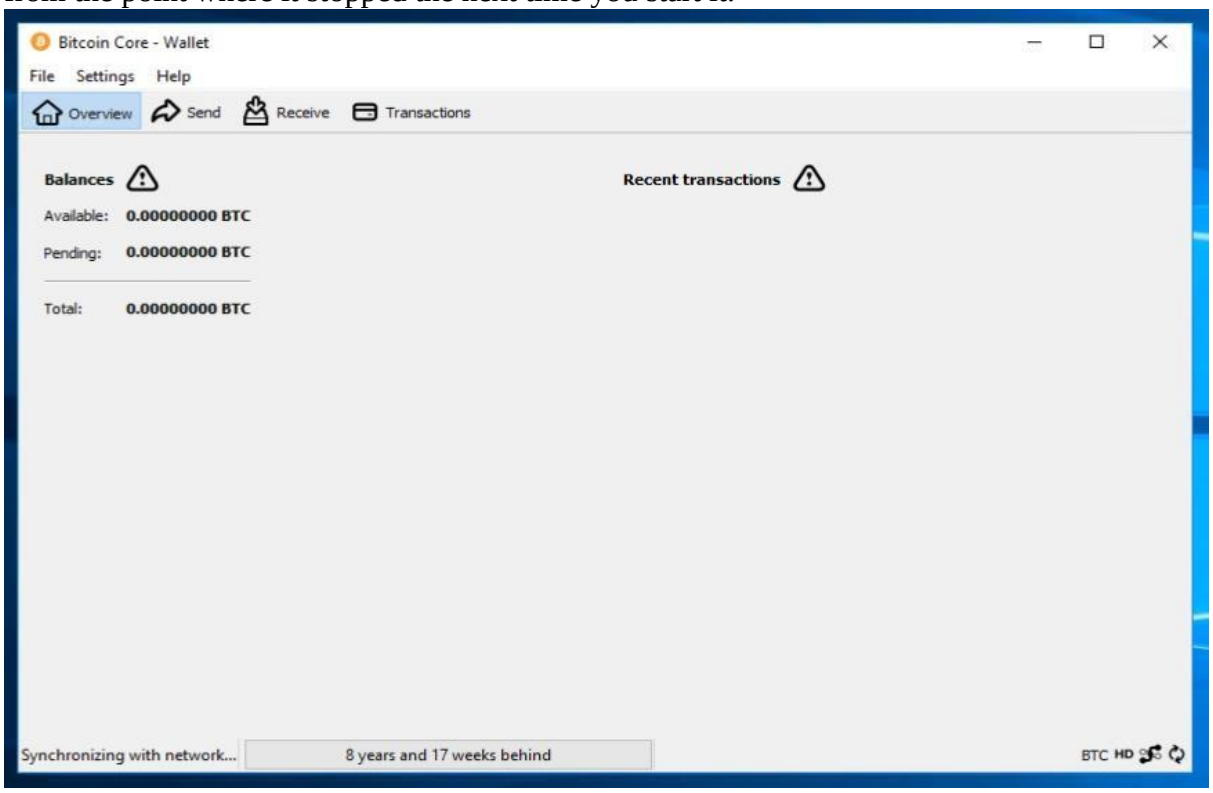



Your firewall may block Bitcoin Core from making outbound connections. It's safe to allow Bitcoin Core to use all networks.




Bitcoin Core GUI will begin to download the block chain. This step will take at least several days, and it may take much more time on a slow Internet connection or with a slow computer. During the download, Bitcoin Core will use a significant part of your

connection bandwidth. You can stop Bitcoin Core at any time by closing it; it will resume from the point where it stopped the next time you start it.

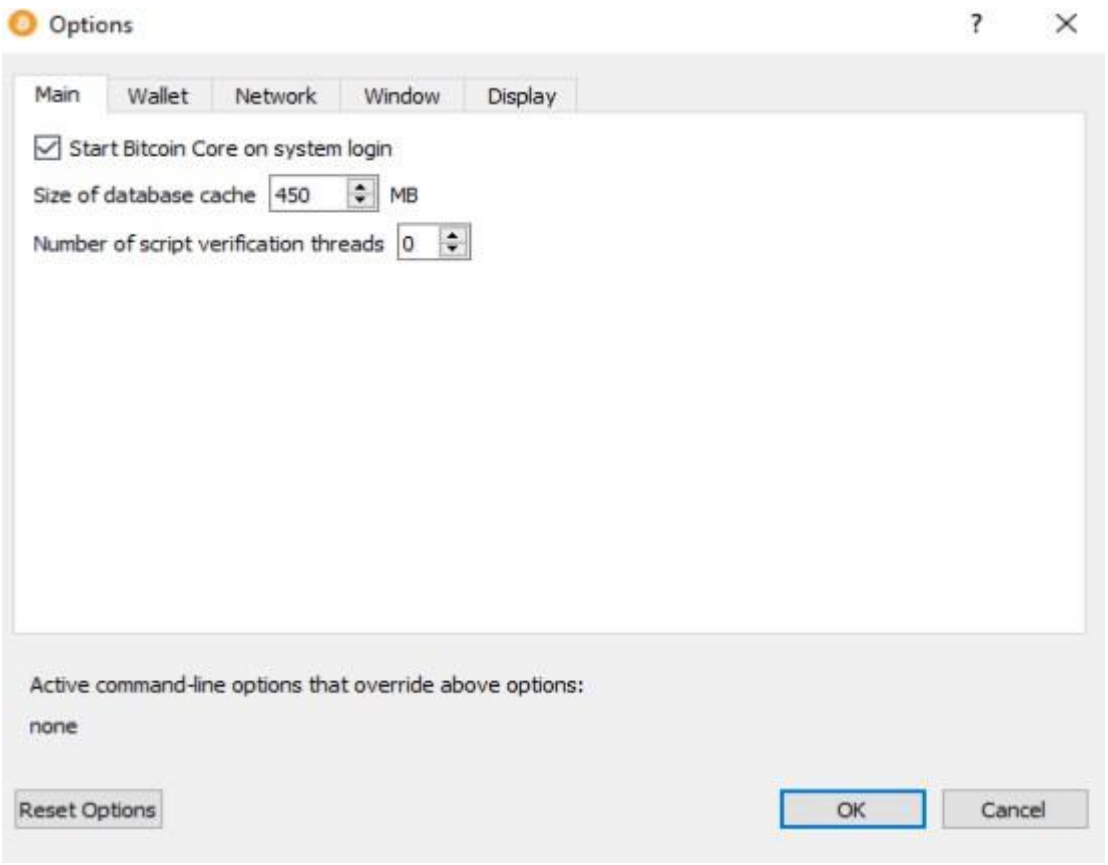



After download is complete, you may use Bitcoin Core as your wallet or you can just let it run to help support the Bitcoin network.

*Optional: Start Your Node At Login*

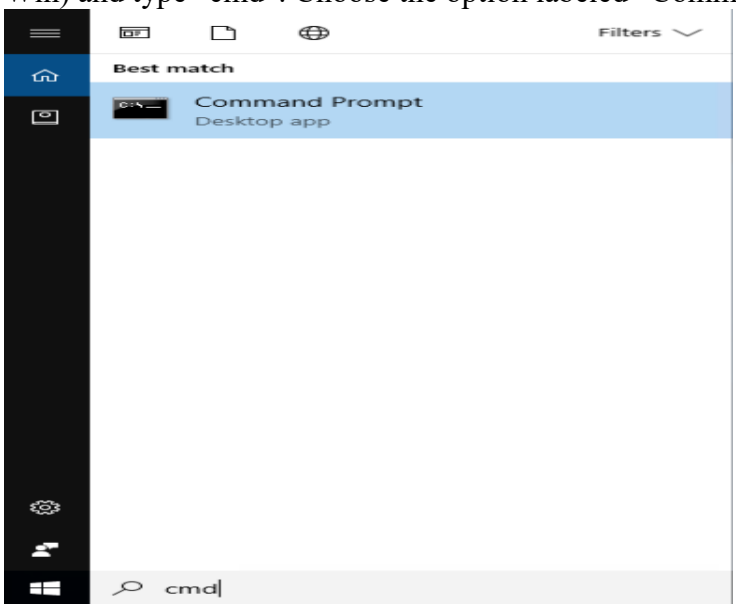
Starting your node automatically each time you login to your computer makes it easy for you to contribute to the network. The easiest way to do this is to tell Bitcoin Core GUI to start at login.

While running Bitcoin Core GUI, open the Settings menu and choose Options. On the Main tab, click *Start Bitcoin on system login*. Click the Ok button to save the new settings.




The next time you login to your desktop, Bitcoin Core GUI will be automatically started minimized in the task bar.

To start Bitcoin Core daemon, first open a command window: press the Windows key (⊞ Win) and type “cmd”. Choose the option labeled “Command Prompt”.




If you installed Bitcoin Core into the default directory, type the following at the command prompt:

C:\Program Files\Bitcoin\daemon\bitcoind

Bitcoin Core daemon should start. To interact with Bitcoin Core daemon, you will use the command bitcoin-cli (Bitcoin command line interface). If you installed Bitcoin Core into the default location, type the following at the command prompt to see whether it works:

C:\Program Files\Bitcoin\daemon\bitcoin-cli getblockchaininfo
Note: it may take up to several minutes for Bitcoin Core to start, during which it will display the following message whenever you use bitcoin-cli:
error: {"code":-28,"message":"Verifying blocks..."}
After it starts, you may find the following commands useful for basic interaction with your node: getblockchaininfo, getnetworkinfo, getnettotals, getwalletinfo, stop, and help. For example, to safely stop your node, run the following command:
C:\Program Files\Bitcoin\daemon\bitcoin-cli stop
A complete list of commands is available in the Bitcoin.org developer reference.
When Bitcoin Core daemon first starts, it will begin to download the block chain. This step will take at least several days, and it may take much more time on a slow Internet connection or with a slow computer. During the download, Bitcoin Core will use a significant part of your connection bandwidth. You can stop Bitcoin Core at any time using the stop command; it will resume from the point where it stopped the next time you start it.
*Optional: Start Your Node At Boot*
Starting your node automatically each time your computer boots makes it easy for you to contribute to the network. The easiest way to do this is to start Bitcoin Core daemon when you login to your computer.
Start File Explorer and go to:
C:\ProgramData\Microsoft\Windows\Start Menu\Programs\StartUp
Right-click on the File Explorer window and choose New → Text file. Name the file start_bitcoind.bat. Then right-click on it and choose Open in Notepad (or whatever editor you prefer). Copy and paste the following line into the file.
C:\Program Files\Bitcoin\daemon\bitcoind
(If you installed Bitcoin Core in a non-default directory, use that directory path instead.)
Save the file. The next time you login to your computer, Bitcoin Core daemon will be automatically started.

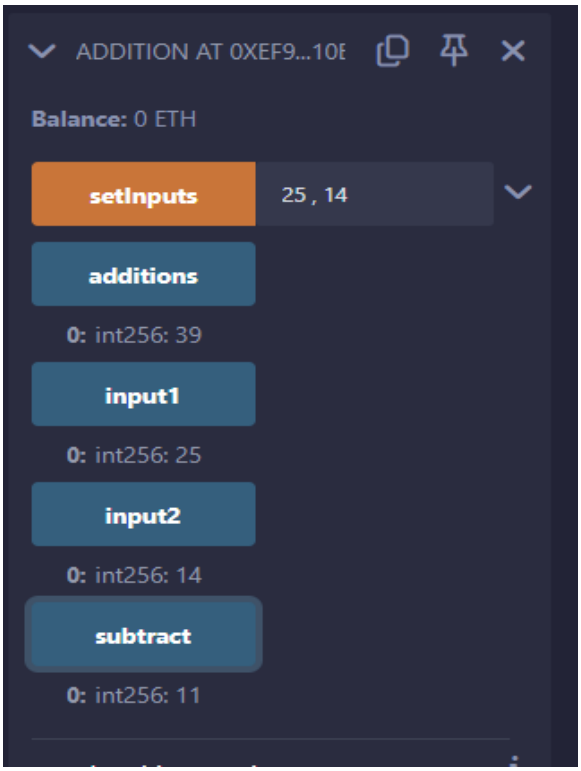
## Practical 3

**A. Write a Solidity program that demonstrates various types of functions including regular functions, view functions, pure functions, and the fallback function.**

- **regular functions**

**Code:**

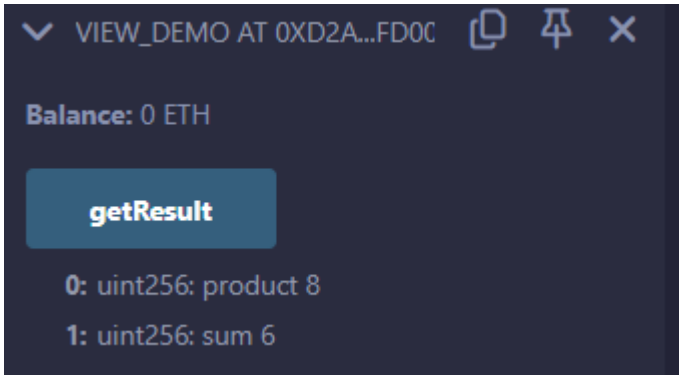
```
pragma solidity ^0.8.17;

contract Addition {

   int public input1;
   int public input2;

   function setInputs(int _input1, int _input2) public {
      input1 = _input1;
      input2 = _input2;
   }

   function additions() public view returns(int) {
      return input1 + input2;
   }

   function subtract() public view returns(int) {
      return input1 - input2;
   }
}
```



- **view functions**

**Code:**

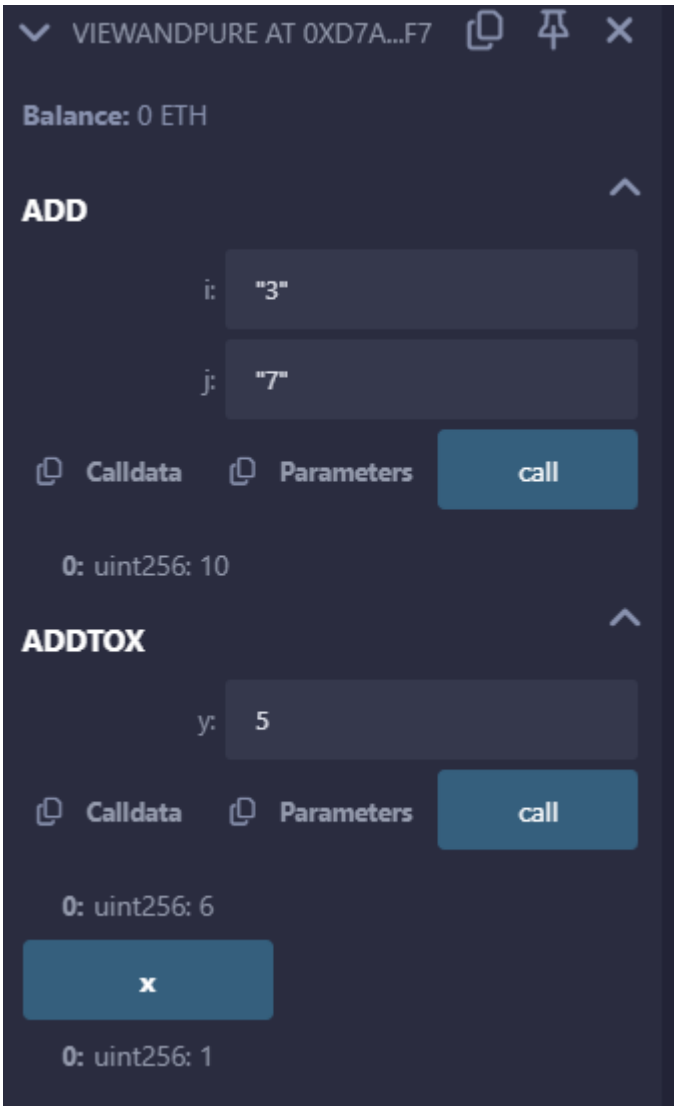
```
pragma solidity ^0.8.0;

contract view_demo
{
   uint256 num1 = 2;
   uint256 num2 = 4;

   function getResult() public view returns (uint256 product, uint256 sum) {
      product = num1 * num2;
      sum = num1 + num2;
   }
}
```




- **view and pure functions**

  **Code:**

```
pragma solidity ^0.8.3;

contract ViewAndPure {
   uint public x = 1;

   // Promise not to modify the state.
   function addToX(uint y) public view returns (uint) {
      return x + y;
   }

   // Promise not to modify or read from the state.
   function add(uint i, uint j) public pure returns (uint) {
      return i + j;
   }
}
```




- fallback function
  Code:

```
pragma solidity ^0.8.17;

contract fallbackfn {

    // Event to log details when fallback or receive function is called
    event Log(string func, address sender, uint value, bytes data);

    // Fallback function to handle calls to the contract with data or no matching function
    fallback() external payable {
        emit Log("fallback", msg.sender, msg.value, msg.data); // Emit log with details
    }

    // Receive function to handle plain ether transfers
    receive() external payable {
        emit Log("receive", msg.sender, msg.value, ""); // Emit log with details (msg.data is
empty)
```

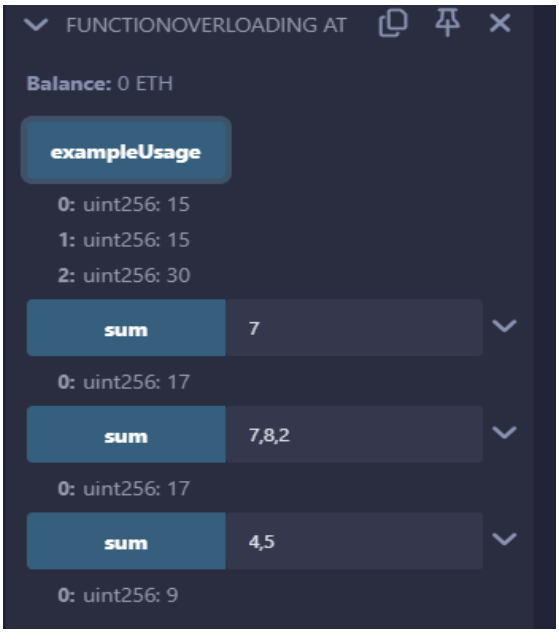
```
        //msg.data is empty hence no need to specify it and mark it as empty string
    }
}
```

**B. Write a Solidity program that demonstrates function overloading, mathematical functions, and cryptographic functions.**

- function overloading

**Code:**

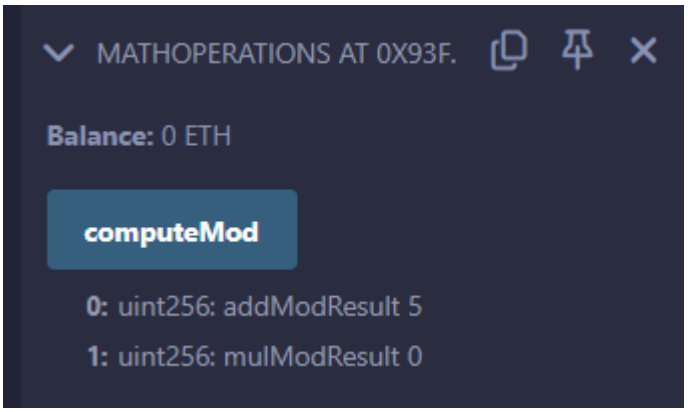
```
pragma solidity ^0.8.17;

contract FunctionOverloading {
    // Function with one parameter
    function sum(uint a) public pure returns (uint) {
        return a + 10;
    }

    // Overloaded function with two parameters
    function sum(uint a, uint b) public pure returns (uint) {
        return a + b;
    }
    // Overloaded function with three parameters
    function sum(uint a, uint b, uint c) public pure returns (uint) {
        return a + b + c;
    }
    // Examples of calling overloaded functions
    function exampleUsage() public pure returns (uint, uint, uint) {
        uint result1 = sum(5);           // Calls the first sum function
        uint result2 = sum(5, 10);       // Calls the second sum function
        uint result3 = sum(5, 10, 15);  // Calls the third sum function
        return (result1, result2, result3);
    }
}
```



- **mathematical functions**

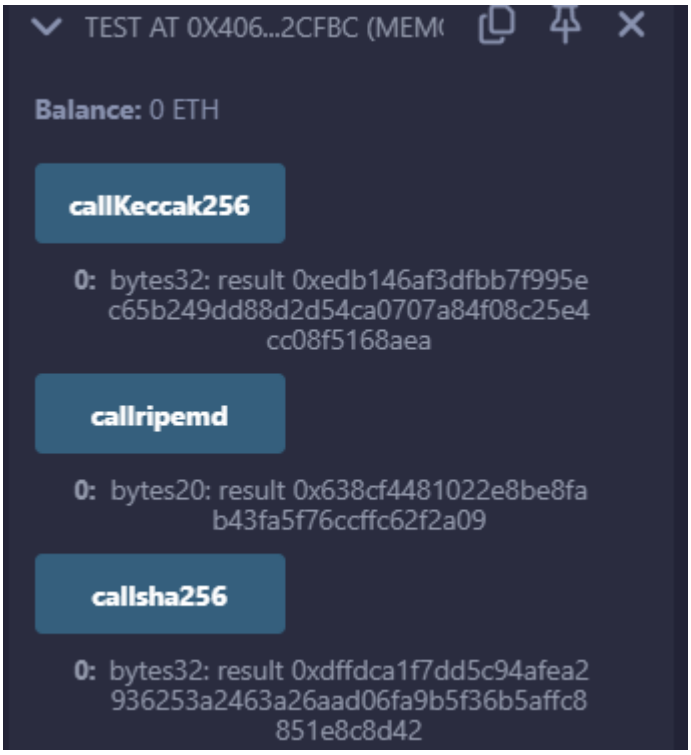
**Code:**

```
pragma solidity ^0.8.17;

contract MathOperations {
    // addMod computes (x + y) % k
    // mulMod computes (x * y) % k

    // Function to compute modular addition and multiplication
    // @return addModResult: Result of (x + y) % k
    // @return mulModResult: Result of (x * y) % k
    function computeMod() public pure returns (uint addModResult, uint mulModResult) {
        uint x = 3;
        uint y = 2;
        uint k = 6;
        addModResult = addmod(x, y, k); // Compute (x + y) % k
        mulModResult = mulmod(x, y, k); // Compute (x * y) % k
    }
}
```




- **cryptographic functions**
  **Code:**

```
pragma solidity ^0.8.17;
contract Test{
    function callKeccak256() public pure returns(bytes32 result){
        return keccak256("BLOCKCHAIN");
    }
    function callsha256() public pure returns(bytes32 result){
        return sha256("BLOCKCHAIN");
    }
    function callripemd() public pure returns (bytes20 result){
        return ripemd160("BLOCKCHAIN");
    }
}
```


TEST AT 0X406...2CFBC (MEM
Balance: 0 ETH
callKeccak256
0: bytes32: result 0xedb146af3dfbb7f995e
c65b249dd88d2d54ca0707a84f08c25e4
cc08f5168aea
callripemd
0: bytes20: result 0x638cf4481022e8be8fa
b43fa5f76ccffc62f2a09
callsha256
0: bytes32: result 0xdffdca1f7dd5c94afea2
936253a2463a26aad06fa9b5f36b5affc8
851e8c8d42

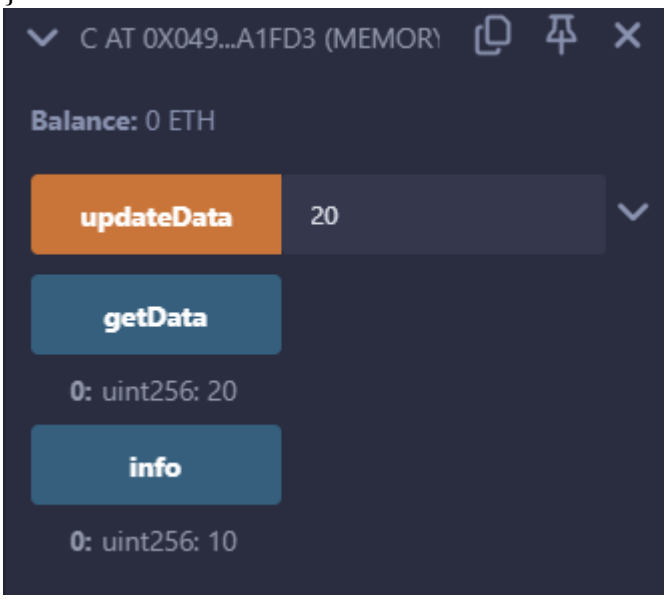
**C. Write a Solidity program that demonstrates various features including contracts, inheritance, constructors, abstract contracts, interfaces.**

- **Contracts, inheritance**

**Code:**

```
pragma solidity ^0.8.17;

contract C{

   uint private data;
   uint public info;

   constructor() {
      info = 10;
      }

      function increment(uint a) private pure returns(uint){
         return a + 1;
      }

      function updateData(uint a) public {
         data = a;
      }

      function getData() public view returns(uint) {
         return data;
      }
      function compute(uint a, uint b) internal pure returns (uint) {
         return a + b;
      }
}



contract D {

   function readData() public returns(uint) {
      C c = new C();
      c.updateData(7);
      return c.getData();
   }
}


contract E is C {

   uint private result;
   C private c;

   constructor() {
```

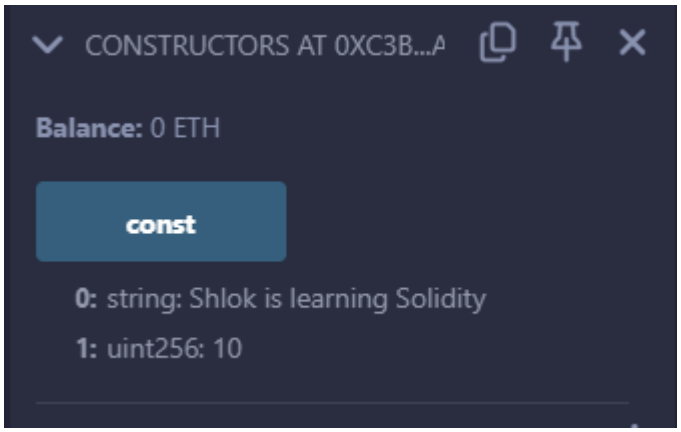
```
        c = new C();
    }

    function getComputedResult() public {
        result = compute(3, 6);
    }

    function getResult() public view returns(uint) {
        return result;
    }
}
```




- **Constructors**

**Code:**

```
pragma solidity ^0.8.17;

contract constructors{

    string str;
    uint amount;

    constructor(){
        str  = "Shlok is learning Solidity";
        amount = 10;
    }

    function const()public view returns(string memory,uint){
        return (str,amount);

    }

}
```




- **abstract contracts**

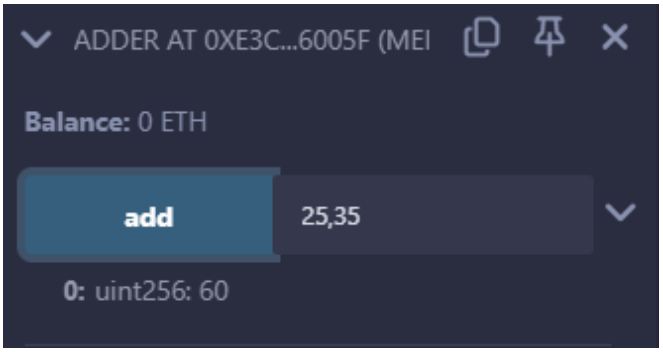
**Code:**

```
pragma solidity ^0.8.17;

abstract contract Main {
    // Define an abstract function that can be overridden
    function add(uint a, uint b) public virtual pure returns (uint);
}

contract Adder is Main {
    // Override the add function from the Main contract
    function add(uint a, uint b) public override pure returns (uint) {
        return a + b;
    }
}
```

- **interfaces**

**Code:**

```
pragma solidity ^0.8.17;

interface adder{
   function add(uint a, uint b)external pure returns(uint);
}

contract adderContract is adder{
   function add(uint a, uint b)external pure returns(uint){
      return a+b;
   }
}
```



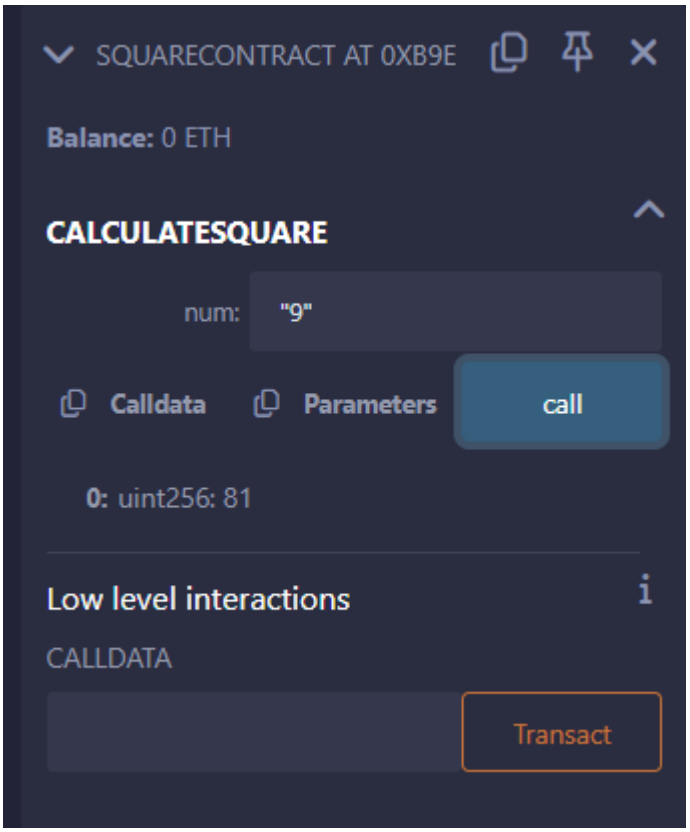
**D. Write a Solidity program that demonstrates use of libraries, assembly, events, and error handling.**

**a. Libraries.**

**Code:**

```
pragma solidity ^0.8.17;
library Search {
  function indexOf(uint[] storage self, uint value) internal view returns (uint) {
    for (uint i = 0; i < self.length; i++) {
      if (self[i] == value) {
        return i;
      }
    }
    return type(uint).max;
  }
}

contract Test {
  uint[] data;
  constructor() {
    data.push(1);
    data.push(2);
    data.push(3);
    data.push(4);
    data.push(5);
  }

  function isValuePresent() external view returns (uint) {
    uint value = 4;

    // Search if value is present in the array using Library function
    uint index = Search.indexOf(data, value);
    return index;
  }
}

library MathLibrary {
  function square(uint num) internal pure returns (uint) {
    return num * num;
  }
}
contract SquareContract {
  using MathLibrary for uint;
  function calculateSquare(uint num) external pure returns (uint) {
    return num.square();
  }
}
```

Output:

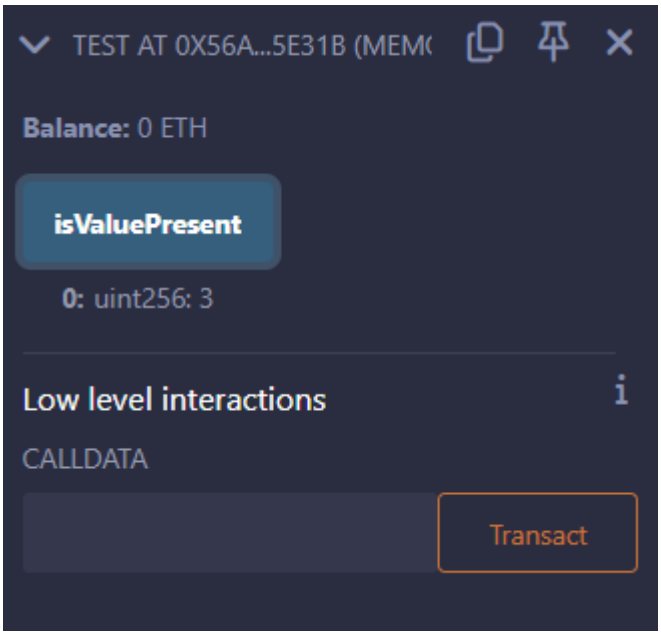



**b. Assembly**
**Code:**

```
pragma solidity ^0.8.17;

library Sum {
  function sumUsingInlineAssembly(uint[] memory _data) public pure returns (uint sum) {
    for (uint i = 0; i < _data.length; ++i) {
      assembly {
        // Load the value from memory at the current index
        let value := mload(add(add(_data, 0x20), mul(i, 0x20)))
        // Add the value to the sum
        sum := add(sum, value)
      }
    }
    // Return the calculated sum
    return sum;
  }
}
contract Test {
  uint[] data;
  constructor() {
    data.push(1);
    data.push(2);
    data.push(3);
    data.push(4);
    data.push(5);
  }
```

```
    function sum() external view returns (uint) {
        return Sum.sumUsingInlineAssembly(data);
    }
}
```

Output:

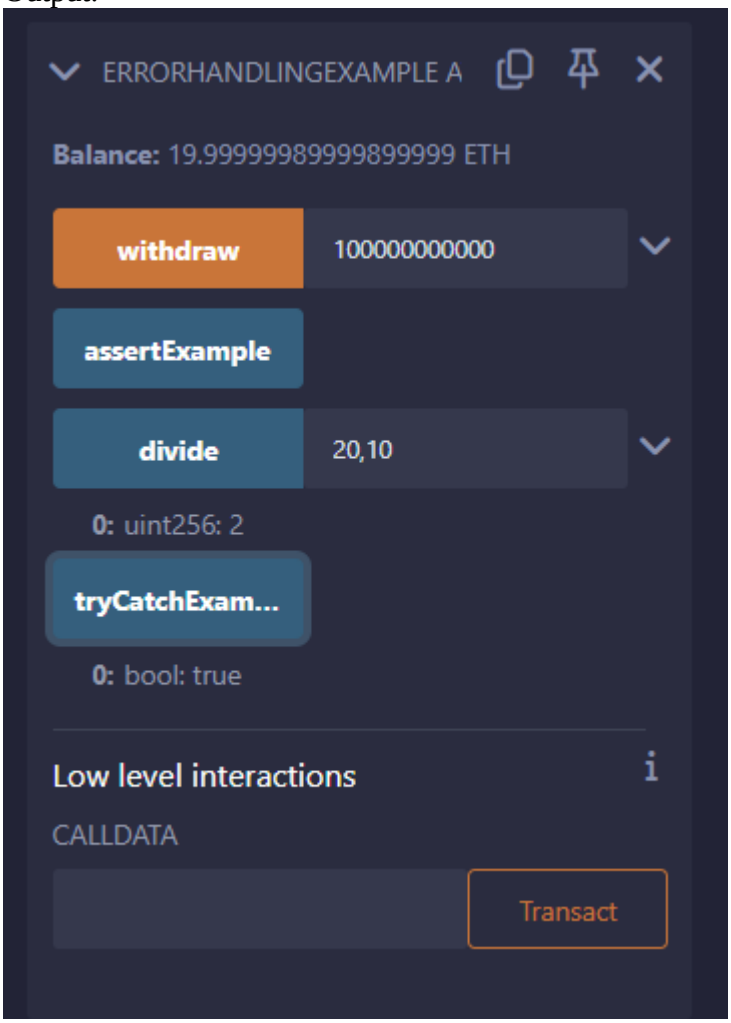



**c. Error handling.**
**Code:**

```
pragma solidity ^0.8.17;

contract ErrorHandlingExample {
    constructor() payable {
        // Allow the contract to receive Ether during deployment
    }

    function divide(uint256 numerator, uint256 denominator) external pure returns (uint256) {
        require(denominator != 0, "Division by zero is not allowed");
        return numerator / denominator;
    }

    function withdraw(uint256 amount) external {
        require(amount <= address(this).balance, "Insufficient balance");

        payable(msg.sender).transfer(amount);
    }

    function assertExample() external pure {
        uint256 x = 5;
        uint256 y = 10;
```

```
        assert(x < y);
    }

    function tryCatchExample() external view returns (bool) {
        try this.divide(10, 5) returns (uint256 result) {
            // Handle successful division
            return true;
        } catch Error(string memory errorMessage) {
            // Handle division error
            return false;
        }
    }
}
```

Output:



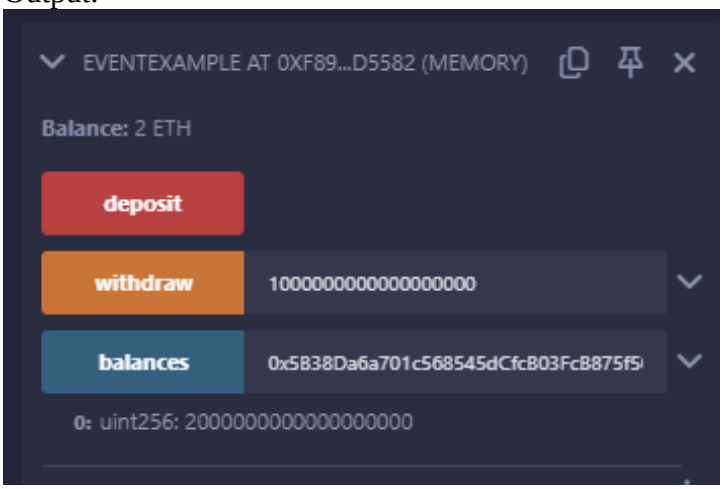

**d. Events.**

**Code:**

```
pragma solidity ^0.8.17;
contract EventExample {
    // Define an event
    event Deposit(address indexed from, uint256 amount);
    event Withdraw(address indexed to, uint256 amount);
    // Mapping to keep track of user balances
```

```
    mapping(address => uint256) public balances;
    // Function to deposit ether into the contract
    function deposit() public payable {
        require(msg.value > 0, "Must deposit more than 0 ether");
        // Update the balance
        balances[msg.sender] += msg.value;
        // Emit the Deposit event
        emit Deposit(msg.sender, msg.value);
    }
    // Function to withdraw ether from the contract
    function withdraw(uint256 amount) public {
        require(balances[msg.sender] >= amount, "Insufficient balance");
        // Update the balance
        balances[msg.sender] -= amount;
        // Transfer the ether
        payable(msg.sender).transfer(amount);
        // Emit the Withdraw event
        emit Withdraw(msg.sender, amount);
    }
}
```

Output:



## Practical 4

**Aim: Install and demonstrate use of hyperledger-Irhoa.**

Hyperledger is an open source collaborative effort created to advance blockchain technologies. The Hyperledger organization has a number of projects (10 right now) for various blockchain solutions, such as smart contract engines, permissioned networks, querying for information inside a ledger, etc.

Iroha is a blockchain platform implementation meant for the simple modeling of financial transactions. So basically, if you have some asset with a quantity (maybe 100 forks and 200 spoons), you can easily model that on the blockchain, transfer those assets, see how many of those assets a certain person has, etc. Of course, this can be easily extended for various assets and quantities.

**Iroha Terminology and Concepts**

There are a few terms we need to know to get a good grasp on Iroha.
To begin learning the terms and concepts, we'll setup an example scenario, which we're going to model on the blockchain.
*We own a farm, named "srcmakeFarm". We have some sheep, corn, a barn, and some other farm stuff on our land. We also have a few workers on our farm.*
With that in mind, let's see the Iroha terminology that we need to model our farm.

1. **Asset** - Any countable commodity or value. It could be currency, like $US dollars$, Euros, bitcoins. It could be sheep, corn, windows. It can be anything that can be counted. We could have an asset named "sheep" on our blockchain.
2. **Account** - An entity that's able to perform a specified set of actions. For example, maybe one of our farmboys who feed the sheep has an account on our blockchain.
3. **Domain** - Assets and accounts are labeled by a domain, which is just a way of grouping them. For example, we could have a domain named "srcmakeFarm" that our "sheep" assets and "farmboy" account belong too.
4. **Permission** - Does an account have the privilege do perform a certain command? For example, to transfer sheeps from "srcmakeFarm" to another might require the "can_transfer" permission.
5. **Role** - A group of permissions. Perhaps we want the farmboy to be able to view how many sheep we have, but not be able to transfer the sheep to anyone else. In that case, we'd create a role for the farmboy with to view our asset count, but not to transfer them.
6. **Transaction** - A set of commands that are applied to the ledger. For example: 1) Add 20 "sheep" assets and 2) Remove 40 "sheep food" assets, could be some commands that form a transaction.
7. **Query** - Checking the state of data in the system. For example, a query to check how much sheep food "srcmakeFarm" currently has is valid.

**Creating An Iroha Network**

We're going to create a basic Iroha network.
Any Unix style terminal will work, but I've personally used Linux (Ubuntu 16.04).
The commands to install Docker on Ubuntu from the terminal are:

```
sudo apt-get install curl
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -
sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu $(lsb_relea
sudo apt-get update
apt-cache policy docker-ce
sudo apt-get install -y docker-ce
```

Next, we need to create a docker network. We'll name it "srcmake-iroha-network".

```
sudo docker network create srcmake-iroha-network
```

Next, we're going to add PostgreSQL to our network.

```
sudo docker run --name some-postgres \
-e POSTGRES_USER=postgres \
-e POSTGRES_PASSWORD=mysecretpassword \
-p 5432:5432 \
--network=srcmake-iroha-network \
-d postgres:9.5
```

Next, we'll create a volume of persistant storage named "blockstore" to store the blocks for our blockchain.

```
sudo docker volume create blockstore
```

Now, we need to configure Iroha on the network. Download the Iroha code from github. (And install git if you don't already have it.)

```
sudo apt-get install git
git clone -b develop https://github.com/hyperledger/iroh
```

We're going to use the Iroha configuration that's been prepared as an example to run the Iroha docker container.

```
sudo docker run -it --name iroha \
-p 50051:50051 \
-v $(pwd)/iroha/example:/opt/iroha_data \
-v blockstore:/tmp/block_store \
--network=srcmake-iroha-network \
--entrypoint=/bin/bash \
hyperledger/iroha:x86_64-develop-latest
```

Now we're going to actually run Iroha.

```
irohad --config config.docker --genesis_block genesis.bl
```

And now our Iroha blockchain is running in our terminal, so don't close it!

**Interacting with our Iroha Network**

So we have our Iroha network running, so let's make some transactions. If you look at the code for our genesis block, you can see that some commands were invoked to help get us started, so we're going to use those.

To interact with Iroha, we're going to use the command line tool.

Open a new terminal (don't close the one with our Iroha network!) and attach the docker container to our terminal.

```
sudo docker exec -it iroha /bin/bash
```

We should be inside the docker container's shell. Launch the iroha-cli tool and login as admin@test.

```
iroha-cli -account_name admin@test
```

```
root@f33e705c9721:/opt/iroha_data# iroha-cli -account_name admin@test
Welcome to Iroha-Cli.
Choose what to do:
1. New transaction (tx)
2. New query (qry)
3. New transaction status request (st)
> :
```

Type 1 and press enter to start a new transaction.

```
1
```

```
> : 1
Forming a new transactions, choose command to add:
1. Detach role from account (detach)
2. Add new role to account (apnd_role)
3. Create new role (crt_role)
4. Set account key/value detail (set_acc_kv)
5. Transfer Assets (tran_ast)
6. Grant permission over your account (grant_perm)
7. Subtract Assets Quantity from Account (sub_ast_qty)
8. Set Account Quorum (set_qrm)
9. Remove Signatory (rem_sign)
10. Create Domain (crt_dmn)
11. Revoke permission from account (revoke_perm)
12. Create Account (crt_acc)
13. Add Signatory to Account (add_sign)
14. Create Asset (crt_ast)
15. Add Peer to Iroha Network (add_peer)
16. Add Asset Quantity (add_ast_qty)
0. Back (b)
> :
```

There are a lot of commands that we can do. Let's try to model our farm a bit. Let's first create the domain.
Type 10 and create an domain with the id "srcmakeFarm". The default role name is role (list of permissions) the domain has, and we'll give it a "user" role that was created for us as part of this example.

```
10
srcmakeFarm
user
```

```
> : 10
Domain Id: srcmakeFarm
Default Role name: user
Command is formed. Choose what to do:
1. Add one more command to the transaction (add)
2. Send to Iroha peer (send)
3. Go back and start a new transaction (b)
4. Save as json file (save)
> : 1
```

Next, let's add some sheep to our farm. Type 1 to add one more command to the transaction. Then type 14 to create the sheep asset. It belongs to the domain srcmakeFarm, and the precision is 0 (meaning we don't allow decimals. Sheeps only come in whole numbers.)

```
1
14
sheep
srcmakeFarm
0
2
```

```
> : 14
Asset name: sheep
Domain Id: srcmakeFarm
Asset precision: 0
Command is formed. Choose what to do:
1. Add one more command to the transaction (add)
2. Send to Iroha peer (send)
3. Go back and start a new transaction (b)
4. Save as json file (save)
> : 2
```

Our farm looks pretty good, so let's send this to Iroha peer (the network) by typing 2. The peer address is "localhost" and it's on port "50051".

```
localhost
50051
```

```
> : 2
Peer address (0.0.0.0): localhost
Peer port (50051): 50051
[2018-04-25 03:53:24.674649529][th:41][info] TransactionResponseHandler Transact
ion successfully sent
Congratulation, your transaction was accepted for processing.
Its hash is 0fc3abceb23dce55f6756ccea788f81b47d0b96d7abb7a81984ae9f1c9154b33
--------------------
Choose what to do:
1. New transaction (tx)
2. New query (qry)
3. New transaction status request (st)
> :
```

```
root@a645aef9ea62: /opt/iroha_data
[2018-04-25 03:53:24.666927875][th:57][info] TxProcessor handle transaction
[2018-04-25 03:53:24.666985842][th:57][info] PCS propagate tx
[2018-04-25 03:53:24.667006281][th:57][info] OrderingGate propagate tx, account_
id:  account_id: admin@test
[2018-04-25 03:53:24.667025256][th:57][info] OrderingGate Propagate tx (on trans
port)
[2018-04-25 03:53:24.673036508][th:20][info] OrderingServiceTransportGrpc Orderi
ngServiceTransportGrpc::onTransaction
[2018-04-25 03:53:24.673704479][th:20][info] OrderingServiceImpl Queue size is 1
[2018-04-25 03:53:28.465926418][th:191][info] OrderingServiceImpl Start proposal
```

You can see we get a hash for the transaction, and back in our Iroha network terminal, stuff happens (because we pushed a transaction onto the blockchain). so we have our srcmakeFarm and created an asset named sheep. Let's add 50 sheep (and give it to admin@test since we haven't made our own account).

Type 1 to make another transaction. Type 16 to add some quantity to an asset. The account is "admin@test", the asset is "sheep#srcmakeFarm" (notice the #domain), there are 50 sheep, and the precision is 0. Type 2 and send it to the localhost at 50051.

```
1
16
admin@test
sheep#srcmakeFarm
50
0
2
localhost
50051
```

```
> : 16
Account Id: admin@test
Asset Id: sheep#srcmakeFarm
Amount to add (integer part): 50
Amount to add (precision): 0
50 0
Command is formed. Choose what to do:
1. Add one more command to the transaction (add)
2. Send to Iroha peer (send)
3. Go back and start a new transaction (b)
4. Save as json file (save)
> : 2
Peer address (0.0.0.0): localhost
Peer port (50051): 50051
[2018-04-25 04:03:08.653226959][th:41][info] TransactionResponseHandler Transact
ion successfully sent
Congratulation, your transaction was accepted for processing.
Its hash is 2ddd8a4fd0af0507572a2b3fb8b0682ce5f01f775cfbf70c69d6cf101437be0b
--------------------
Choose what to do:
1. New transaction (tx)
2. New query (qry)
3. New transaction status request (st)
> :
```

**Making A Query**
Let's check how many sheep we have. Type 2 to make a new query this time.

```
2
```

```
Choose what to do:
1. New transaction (tx)
2. New query (qry)
3. New transaction status request (st)
> : 2
Choose query:
1. Get all permissions related to role (get_role_perm)
2. Get Transactions by transactions' hashes (get_tx)
3. Get information about asset (get_ast_info)
4. Get Account's Transactions (get_acc_tx)
5. Get all current roles in the system (get_roles)
6. Get Account's Signatories (get_acc_sign)
7. Get Account's Assets (get_acc_ast)
8. Get Account Information (get_acc)
0. Back (b)
> :
```

There are lots of query types. We're going to query a particular account's assets, so type 7.

The account that owns the sheep is "admin@test", the asset we need is "sheep#srcmakeFarm". Send the query request to "localhost" at port "50051".

```
7
admin@test
sheep#srcmakeFarm
1
localhost
50051
```

```
> : 7
Requested account Id: admin@test
Requested asset Id: sheep#srcmakeFarm
Query is formed. Choose what to do:
1. Send to Iroha peer (send)
2. Save as json file (save)
0. Back (b)
> : 1
Peer address (0.0.0.0): localhost
Peer port (50051): 50051
[2018-04-25 04:16:12.946124963][th:41][info] QueryResponseHandler [Account Assets]
[2018-04-25 04:16:12.946210132][th:41][info] QueryResponseHandler -Account Id:- admin@test
[2018-04-25 04:16:12.946229238][th:41][info] QueryResponseHandler -Asset Id- sheep#srcmakeFarm
[2018-04-25 04:16:12.946251209][th:41][info] QueryResponseHandler -Balance- 50
--------------------
```