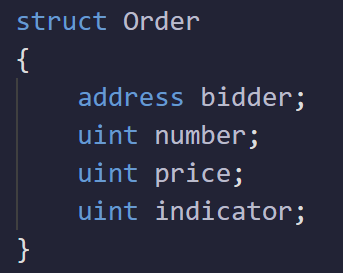
Design and implementation

# Data model

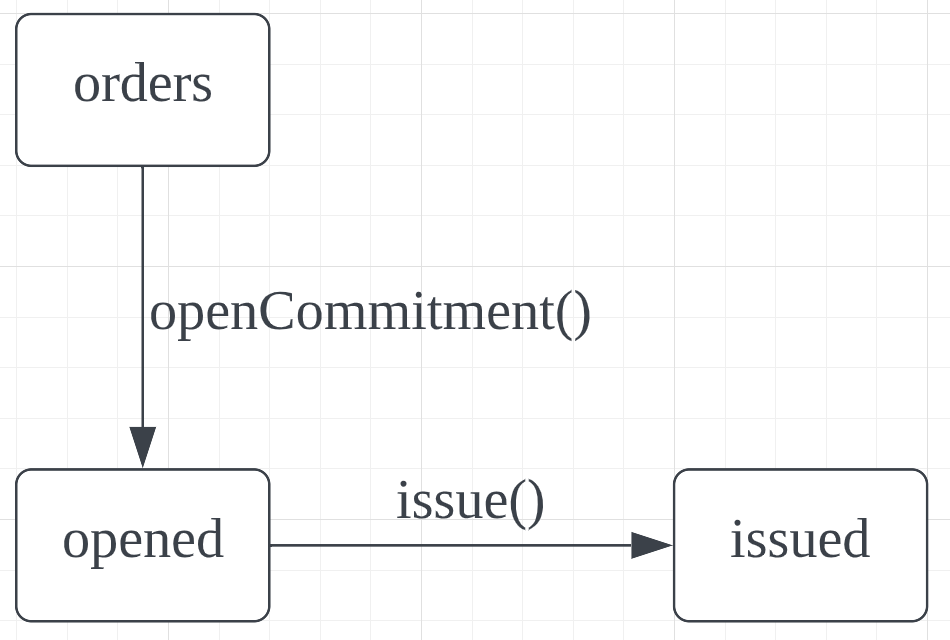
There is only one core data field: order. So I created a struct called Order defined as below:



There are four factors associated with an order: address of bidder, unit of shares purchased, price per share and an indicator.

The indicator is a nonce determined by bidder which act as an identifier of the order. It is used for hash generation because one bidder may upload two separate orders with the same number of purchase and bid price. To differentiate hash value of these two orders, another unique data field is needed.

Then three mapping variables are defined to keep track of order status which are: *orders*, *opened* and *issued*.



*Orders* is a mapping variable from byte32 to uint256 while *opened* and *issued* are mapping from byte32 to Boolean indicating whether or not a specific order is opened or issued.

The reason why the *orders* variable is mapped to uint256 is that we need to record the sequence that a order is submitted for sorting purpose. this will be discussed in detail in the section introducing sorting strategy.

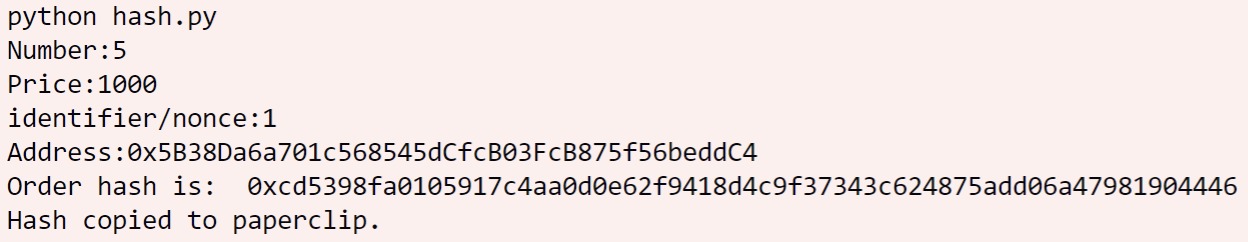
The byte32 value used in these three mapping variables is the unique order hash, which is calculated from *number*, *price*, *address*, *nonce* associated with the order. The reason why the system records the order hash only in the first round is that block information is public, we need to ensure that the information of the orderbook is not visible from any one until the orders are revealed. But we also need to ensure that an order can be recorded reliably in round one. So, we encode the critical values of an order using keccak256 hash function to achieve that. This hash function hash a satisfying collision resistance and is hard to be decoded.

A array of struct Order called *openOrders* is defined to record the information of all opened orders. This array will be sorted for issuing.

# Off-chain computation

## Order hash

An investor must compute the order hash locally to ensure that others don’t know the exact information about his order. So, we provide a hashing function called *hash.py*. Investors can run this script by simply calling python hash.py in terminal and enter the information following the prompt:



Crypto.Hash, eth\_abi, web3 and pyperclip are required as support package. Users can install these packages by the following commands:

*pip install eth\_abi*

*pip install web3*

*pip install pyperclip*

*pip install pycryptodome*

# Requirement analysis

## A total of 10,000 shares are to be sold.

uint constant numberShares = 10000;       // Maximum number of shares

A constant numberShares is defined in the contract to explicitly limit the number of shares. This variable is used in constructing the NeverPayShares contract and issuing the shares. This is the pseudo code of issuing process.

uint remainingShares = numberShares;

for each order of the bidder:

{

    Issue shares to bidder;

    Settle and refund;

    decrement remainingShares;

}

Mark bidder issued;

## The minimum price per share is 1 Ether, any bid to buy shares for less than this amount will not be issued shares.

Price is checked when opening the orders, if a price that is lower than 1 Ether is passed to the function openCommitment(), the transaction will revert and an error message will be returned. Relevant code fragment shown below:

uint minPrice = 1000000000000000000; // Valued by Wei: 1ETH = 10^18 Wei

require(

    price >= minPrice,

    "The minimum price per share is 1 Ether."

);

## Investors should be identified by means of an anonymous Ethereum address.

Address is used to identify the bidders in the order struct.

## First round

A function called *submitCommitment* is defined to implement the functionality of round 1.

function submitCommitment(bytes32 newOrder) public {

    require(

//Timestamp for 20-04-2022-00:00:00-GMT

        block.timestamp < 1650412800,

        "The first round has ended."

    );

    if (orders[newOrder] != 0)

        return;

    numOrders++;

    orders[newOrder] = numOrders;

}

Firstly, we check that the order is submitted before April 20, 2022 by checking the timestamp of current block.

And then we check that whether this order has already been submitted. If it is a new order, *numOrders* is incremented and the sequence number of this order is set to the new value of *numOrders*.

Notice that all investors are provided a script to generate the order hash, so the parameter newOrder has already incorporates the price and amount the bidder is willing to offer.

### 4.1 blinded commitment

## Second round

A function called *openCommitment* is defined to implement the functionality of round 2.

function openCommitment(bytes32 orderHash, uint number, uint price,

uint nonce) payable public {

require(

        block.timestamp >= 1650412800,

        "The second round hasn't started.");

require(

//Timestamp for 27-04-2022-00:00:00-GMT

        block.timestamp < 1651017600,

        "The second round has ended.");

Øther checks on order detail.

        openOrders.push(Order(msg.sender, number, price, price+orders[orderHash]));

        opened[orderHash] = true;

        quickSort(openOrders, 0, int(openOrders.length)-1);

    }

In this function, time stamp of current block is check at first to ensure that the orders can be opened in appropriate time interval. Then a series of value check of order information is checked. If they are all passed, struct *Order* will be constructed and pushed to *openOrders* array and set the status of current order *opened*. Then perform quick sort once to ensure that *openOrders* is correctly sorted.

Explain functions

Analysis running costs

Reflectively discuss

Solidity security

security considerations