Practical 3-Creating a cryptocurrency. Implement Byzantine Generals problem.

A)Creating a cryptocurrency:

→ A smart contract code is deployed on Remix-ethereum IDE for a token. The source code is given below.

```
+-----------------------------------------------------------------------------------------------------+
pragma solidity >=0.4.16 <0.7.0;
contract owned {
   address public owner;
constructor() public {
      owner = msg.sender;
   }
modifier onlyOwner {
      require(msg.sender == owner);
      _;
   }
function transferOwnership(address newOwner) onlyOwner public {
      owner = newOwner;
   }
}
interface tokenRecipient
{
```

```
    function receiveApproval(address _from, uint256 _value, address _token, bytes
calldata  _extraData) external;
}
contract TokenERC20 {
    // Public variables of the token
    string public name;
    string public symbol;
    uint8 public decimals = 18;
    // 18 decimals is the strongly suggested default, avoid changing it
    uint256 public totalSupply;
// This creates an array with all balances
    mapping (address => uint256) public balanceOf;
    mapping (address => mapping (address => uint256)) public allowance;
// This generates a public event on the blockchain that will notify clients
    event Transfer(address indexed from, address indexed to, uint256 value);

    // This generates a public event on the blockchain that will notify clients
    event Approval(address indexed _owner, address indexed _spender, uint256
_value);
// This notifies clients about the amount burnt
    event Burn(address indexed from, uint256 value);
/**
     * Constrctor function
     *
     * Initializes contract with initial supply tokens to the creator of the contract
```

```
    */
   constructor(
       uint256 initialSupply,
       string memory tokenName,
       string memory tokenSymbol
   ) public {
       totalSupply = initialSupply * 10 ** uint256(decimals);  // Update total supply with the decimal amount
       balanceOf[msg.sender] = totalSupply;                // Give the creator all initial tokens
       name = tokenName;                                   // Set the name for display purposes
       symbol = tokenSymbol;                               // Set the symbol for display purposes
   }
/**
    * Internal transfer, only can be called by this contract
    */
   function _transfer(address _from, address _to, uint _value) internal {
       // Prevent transfer to 0x0 address. Use burn() instead
       require(_to != address(0x0));
       // Check if the sender has enough
       require(balanceOf[_from] >= _value);
       // Check for overflows
       require(balanceOf[_to] + _value > balanceOf[_to]);
       // Save this for an assertion in the future
```

```
        uint previousBalances = balanceOf[_from] + balanceOf[_to];
        // Subtract from the sender
        balanceOf[_from] -= _value;
        // Add the same to the recipient
        balanceOf[_to] += _value;
        emit Transfer(_from, _to, _value);
        // Asserts are used to use static analysis to find bugs in your code. They
should never fail
        assert(balanceOf[_from] + balanceOf[_to] == previousBalances);
    }
/**
     * Transfer tokens
     *
     * Send `_value` tokens to `_to` from your account
     *
     * @param _to The address of the recipient
     * @param _value the amount to send
     */
    function transfer(address _to, uint256 _value) public returns (bool success) {
        _transfer(msg.sender, _to, _value);
        return true;
    }
/**
     * Transfer tokens from other address
     *
```

```
    * Send `_value` tokens to `_to` in behalf of `_from`
    *
    * @param _from The address of the sender
    * @param _to The address of the recipient
    * @param _value the amount to send
    */
   function transferFrom(address _from, address _to, uint256 _value) public
returns (bool success) {
       require(_value <= allowance[_from][msg.sender]);     // Check allowance
       allowance[_from][msg.sender] -= _value;
       _transfer(_from, _to, _value);
       return true;
   }
/**
    * Set allowance for other address
    *
    * Allows `_spender` to spend no more than `_value` tokens in your behalf
    *
    * @param _spender The address authorized to spend
    * @param _value the max amount they can spend
    */
   function approve(address _spender, uint256 _value) public
       returns (bool success) {
       allowance[msg.sender][_spender] = _value;
       emit Approval(msg.sender, _spender, _value);
```

```
        return true;
    }
/**
     * Set allowance for other address and notify
     *
     * Allows `_spender` to spend no more than `_value` tokens in your behalf, and
then ping the contract about it
     *
     * @param _spender The address authorized to spend
     * @param _value the max amount they can spend
     * @param _extraData some extra information to send to the approved
contract
     */
    function approveAndCall(address _spender, uint256 _value, bytes memory
_extraData)
        public
        returns (bool success) {
        tokenRecipient spender = tokenRecipient(_spender);
        if (approve(_spender, _value)) {
            spender.receiveApproval(msg.sender, _value, address(this), _extraData);
            return true;
        }
    }
/**
     * Destroy tokens
```

```
    *
    * Remove `_value` tokens from the system irreversibly
    *
    * @param _value the amount of money to burn
    */
   function burn(uint256 _value) public returns (bool success) {
      require(balanceOf[msg.sender] >= _value);   // Check if the sender has
enough
      balanceOf[msg.sender] -= _value;          // Subtract from the sender
      totalSupply -= _value;                    // Updates totalSupply
      emit Burn(msg.sender, _value);
      return true;
   }
/**
    * Destroy tokens from other account
    *
    * Remove `_value` tokens from the system irreversibly on behalf of `_from`.
    *
    * @param _from the address of the sender
    * @param _value the amount of money to burn
    */
   function burnFrom(address _from, uint256 _value) public returns (bool success)
{
      require(balanceOf[_from] >= _value);              // Check if the targeted
balance is enough
```

```
        require(_value <= allowance[_from][msg.sender]);    // Check allowance
        balanceOf[_from] -= _value;                         // Subtract from the targeted balance
        allowance[_from][msg.sender] -= _value;             // Subtract from the sender's allowance
        totalSupply -= _value;                              // Update totalSupply
        emit Burn(_from, _value);
        return true;
    }
}
/******************************************/
/*       Change the name of the contract from customcrypto to your own    token name
*/
/******************************************/
contract customcrypto is owned, TokenERC20 {
uint256 public sellPrice;
    uint256 public buyPrice;
mapping (address => bool) public frozenAccount;
/* This generates a public event on the blockchain that will notify clients */
    event FrozenFunds(address target, bool frozen);
/* Initializes contract with initial supply tokens to the creator of the contract */
    constructor(
        uint256 initialSupply,
        string memory tokenName,
```

```
        string memory tokenSymbol
    ) TokenERC20(initialSupply, tokenName, tokenSymbol) public {}
/* Internal transfer, only can be called by this contract */
    function _transfer(address _from, address _to, uint _value) internal {
        require (_to != address(0x0));                    // Prevent transfer to 0x0
address. Use burn() instead
        require (balanceOf[_from] >= _value);             // Check if the sender has
enough
        require (balanceOf[_to] + _value >= balanceOf[_to]); // Check for overflows
        require(!frozenAccount[_from]);                   // Check if sender is frozen
        require(!frozenAccount[_to]);                     // Check if recipient is frozen
        balanceOf[_from] -= _value;                       // Subtract from the sender
        balanceOf[_to] += _value;                         // Add the same to the recipient
        emit Transfer(_from, _to, _value);
    }
/// @notice Create `mintedAmount` tokens and send it to `target`
    /// @param target Address to receive the tokens
    /// @param mintedAmount the amount of tokens it will receive
    function mintToken(address target, uint256 mintedAmount) onlyOwner public {
        balanceOf[target] += mintedAmount;
        totalSupply += mintedAmount;
        emit Transfer(address(0), address(this), mintedAmount);
        emit Transfer(address(this), target, mintedAmount);
    }
/// @notice `freeze? Prevent | Allow` `target` from sending & receiving tokens
```

```
    /// @param target Address to be frozen
    /// @param freeze either to freeze it or not
    function freezeAccount(address target, bool freeze) onlyOwner public {
        frozenAccount[target] = freeze;
        emit FrozenFunds(target, freeze);
    }
/// @notice Allow users to buy tokens for `newBuyPrice` eth and sell tokens for `newSellPrice` eth
    /// @param newSellPrice Price the users can sell to the contract
    /// @param newBuyPrice Price users can buy from the contract
    function setPrices(uint256 newSellPrice, uint256 newBuyPrice) onlyOwner public {
        sellPrice = newSellPrice;
        buyPrice = newBuyPrice;
    }
/// @notice Buy tokens from contract by sending ether
    function buy() payable public {
        uint amount = msg.value / buyPrice;              // calculates the amount
        _transfer(address(this), msg.sender, amount);              // makes the transfers
    }
/// @notice Sell `amount` tokens to contract
    /// @param amount amount of tokens to be sold
    function sell(uint256 amount) public {
        address myAddress = address(this);
```

```
        require(myAddress.balance >= amount * sellPrice);      // checks if the
contract has enough ether to buy
        _transfer(msg.sender, address(this), amount);              // makes the transfers
        msg.sender.transfer(amount * sellPrice);          // sends ether to the seller. It's
important to do this last to avoid recursion attacks
    }
}
```

+----------+

→ We then set the name of token, initial supply and token symbol before deploying the contract.

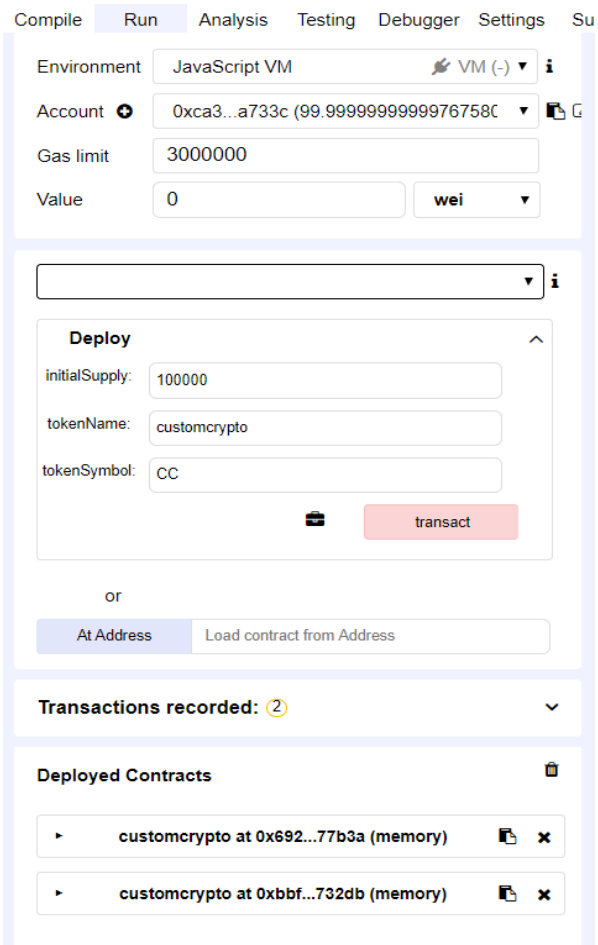


Figure 1: Setting up values before deploying the contract

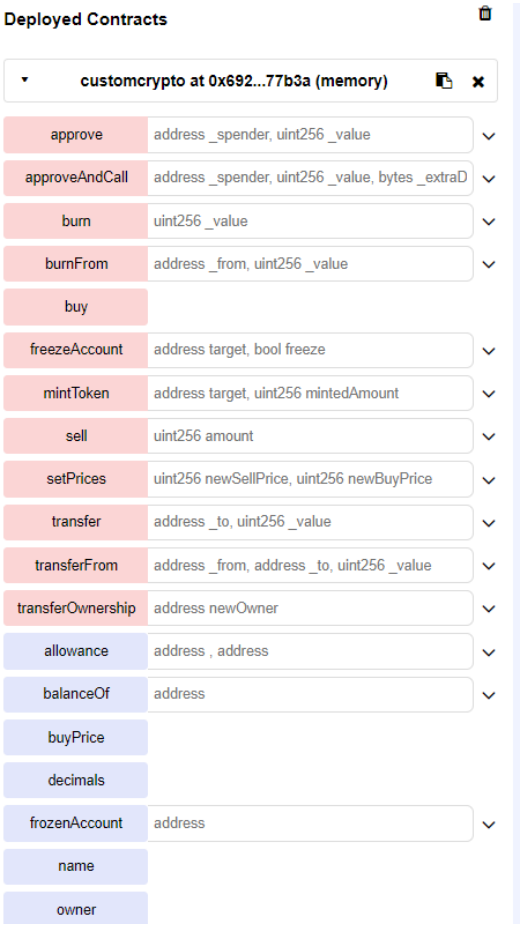


Figure 2: Functions available in the contract

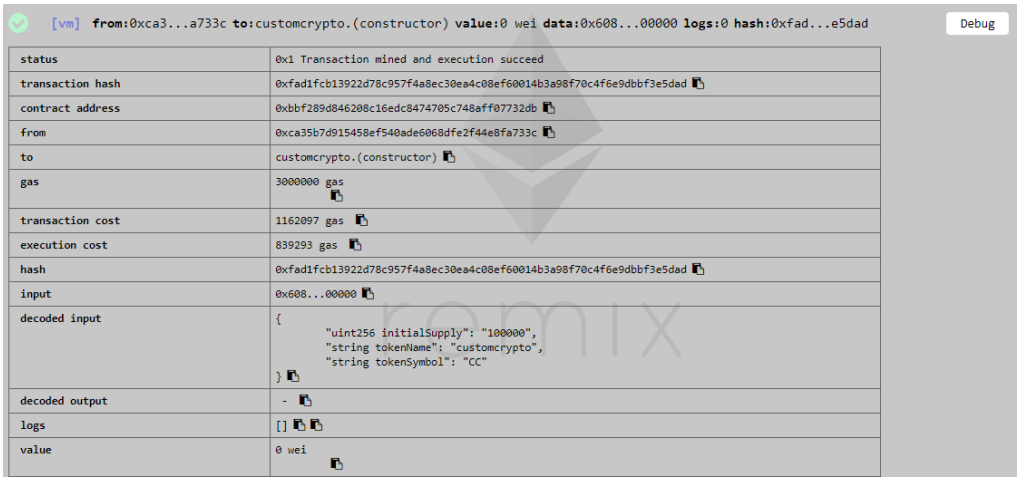


Figure 3: Transaction information

→ We now test this token on Rinkeby network which is basically a test network to test contracts before publishing them on to a main network.
→ So, to deploy on Rinkeby network, we need to pay incentives in form of ether to deploy our contract.
→ First, we create a wallet using Metamask which is added as a chrome extension and then add funds to our wallet.

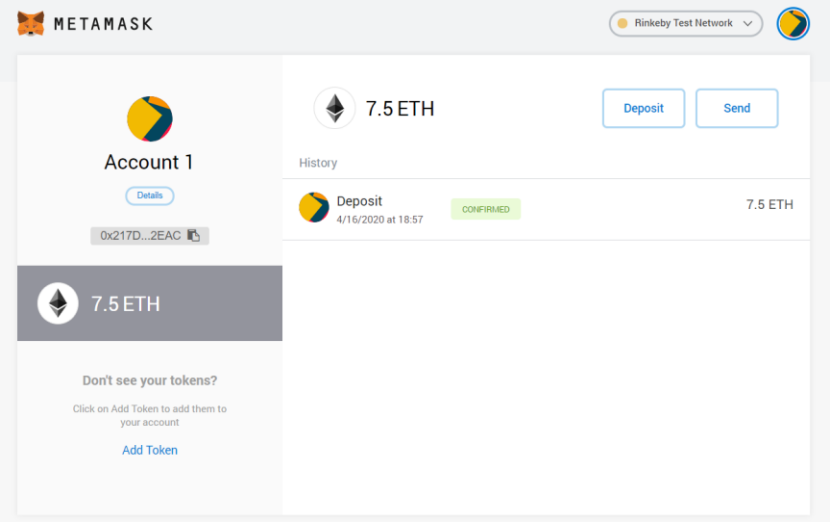


Figure 4: Metamask wallet

→ Change the environment to “Injected Web3” in remix browser.
→ Then deploy the smart contract by setting up the initial parameters like initialSupply, token name and token symbol.

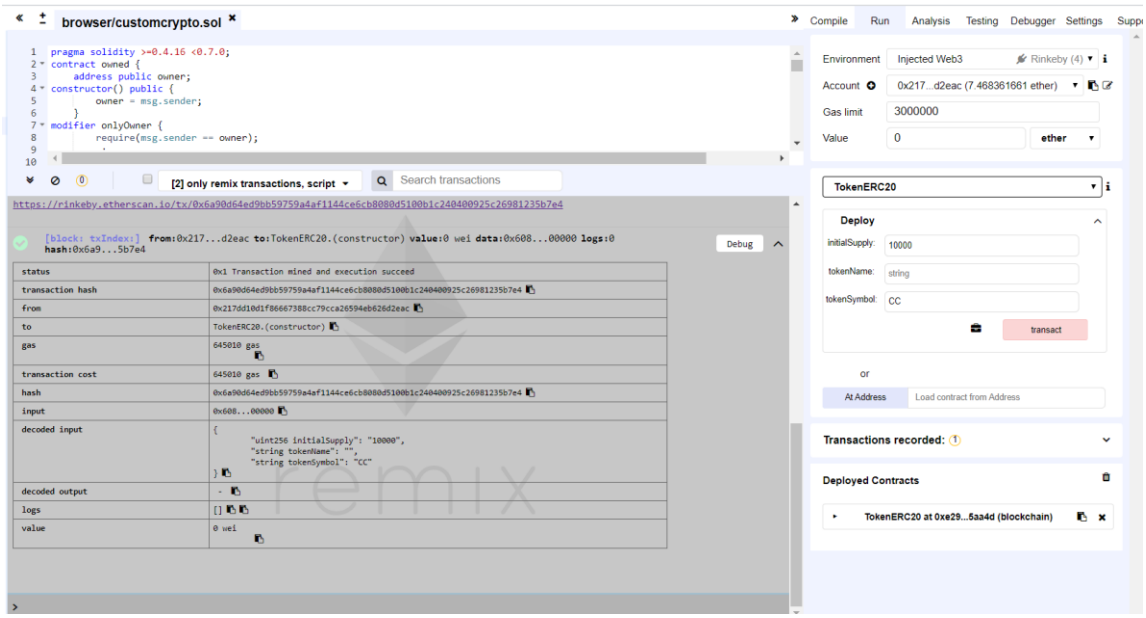


Figure 5: Deploying contract with the set values

→ Note that when starting MetaMask, you will be notified regarding linking the remix smart contract with MetaMask address. Accept it and now your account address will be the MetaMask account address. Also we need to select Rinkeby test network.

→ We now take a look at Rinkeby network and find out our executed transaction.

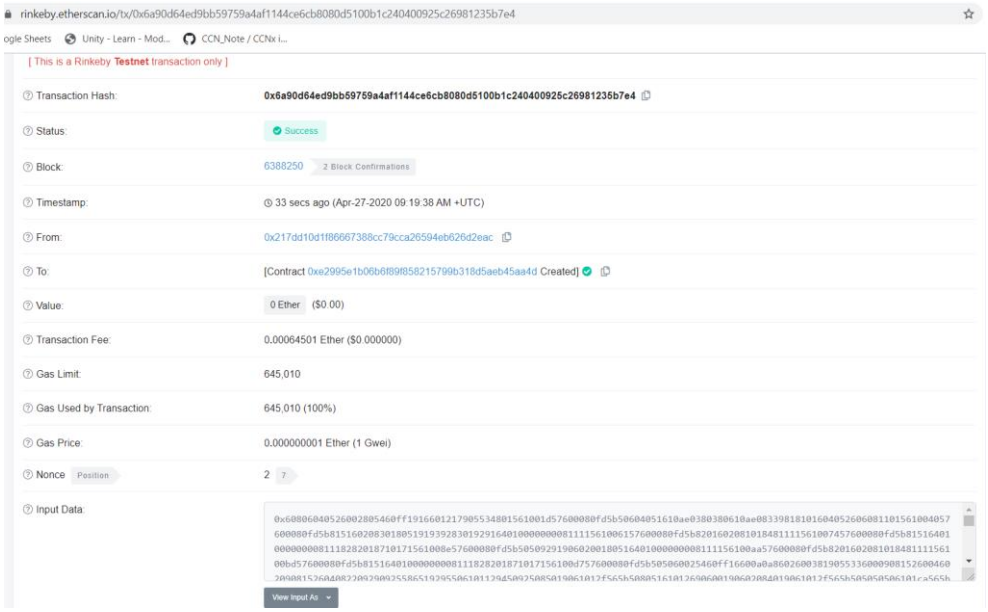


Figure 6: Displaying transaction details executed by account contract address

→ We select the contract address created and add the custom token in MetaMask. The token is displayed as shown below.

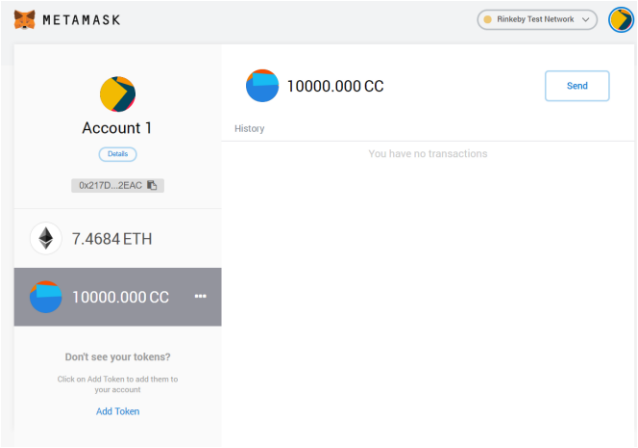


Figure 7: MetaMask wallet with customized cryptocurrency CC

B) Byzantine Problem:

This problem was introduced by Leslie Lamport in his paper "**The Byzantine Generals Problem**", where he describes the problem as follows:

"Reliable computer systems must handle malfunctioning components that give conflicting information to different parts of the system. This situation can be expressed abstractly in terms of a group of generals of the Byzantine army camped with their troops around an enemy city. Communicating only by messenger, the generals must agree upon a common battle plan. However, one or more of them may be traitors who will try to confuse the others. The problem is to find an algorithm to ensure that the loyal generals will reach agreement."

Lamport tries to proves that:

"For any m, Algorithm OM(m) satisfies conditions that All loyal generals decide upon the same plan of action and A small number of traitors cannot cause the loyal generals to adopt a bad plan if there are more than 3m generals and at most m traitors."

The source code for implementing the problem is as follows:

```
+-----------------------------------------------------------------------------------------------+
from argparse import ArgumentParser
from collections import Counter



class General:
    def __init__(self, id, is_traitor=False):
        self.id = id
        self.other_generals = []
```

```
        self.orders = []
        self.is_traitor = is_traitor

    def __call__(self, m, order):
        """When a general is called, it acts as the commander,
        and begins the OM algorithm by passing its command to
        all the other generals.

        Args:
            m (int): The level of recursion.
            order (str): The order, such that order ∈ {"ATTACK","RETREAT"}.

        """
        self.om_algorithm(commander=self,
                          m=m,
                          order=order,
                          )

    def _next_order(self, is_traitor, order, i):
        """A helper function to determine what each commander
        should pass on as the next order. Traitors will pass-
        on the opposite command if the index of the general
        in their `other_generals` list is odd.

        Args:
            is_traitor (bool): True for traitors.
            order (str): The received order, such that
                order ∈ {"ATTACK","RETREAT"}.
            i(int): The index of the general in question.
```

```
        Returns:
            str: The resulting order ("ATTACK" or "RETREAT").

        """
        if is_traitor:
            if i % 2 == 0:
                return "ATTACK" if order == "RETREAT" else "RETREAT"
        return order

    def om_algorithm(self, commander, m, order):
        """The OM algorithm from Lamport's paper.

        Args:
            commander (General): A reference to the general
                who issued the previous command.
            m (int): The level of recursion .
            order (str): The received order, such that
                order ∈ {"ATTACK","RETREAT"}.

        """
        if m < 0:
            self.orders.append(order)
        elif m == 0:
            for i, l in enumerate(self.other_generals):
                l.om_algorithm(
                    commander=self,
                    m=(m - 1),
                    order=self._next_order(self.is_traitor, order, i)
```

```
                )
        else:
            for i, l in enumerate(self.other_generals):
                if l is not self and l is not commander:
                    l.om_algorithm(
                        commander=self,
                        m=(m - 1),
                        order=self._next_order(self.is_traitor, order, i)
                    )

    @property
    def decision(self):
        """Returns a tally of the General's received commands.

        """
        c = Counter(self.orders)
        return c.most_common()



def init_generals(generals_spec):
    """Creates a list of generals, given a string
    input from arg-parse.

    Args:
        generals_spec (list): A list of generals
            of the form 'l,t,l,t...", where "l"
            is loyal and "t" is a traitor.

    Returns:
```

```
        list: A list of initialized generals.
    """
    generals = []
    for i, spec in enumerate(generals_spec):
        general = General(i)
        if spec == "l":
            pass
        elif spec == "t":
            general.is_traitor = True
        else:
            print("Error, bad input in generals list:
{}".format(generals_spec))
            exit(1)
        generals.append(general)
    # Add list of other generals to each general.
    for general in generals:
        general.other_generals = generals
    return generals



def print_decisions(generals):
    for i, l in enumerate(generals):
        print("General {}: {}".format(i, l.decision))



def main():
    parser = ArgumentParser()
    parser.add_argument("-m", type=int, dest="recursion",
                        help=" The level of recursion in the algorithm, where
M > 0")
```

```
    parser.add_argument("-G", type=str, dest="generals",
                        help=" A string of generals (ie 'l,t,l,l,l'...),
where l is loyal and t is a traitor.  "
                             "The first general is the Commander.")
    parser.add_argument("-O", type=str, dest="order",
                        help=" The order the commander gives to the other
generals (O ∈ {ATTACK,RETREAT})")
    args = parser.parse_args()


    generals_spec = [x.strip() for x in args.generals.split(',')]
    generals = init_generals(generals_spec=generals_spec)
    generals[0](m=args.recursion, order=args.order)
    print_decisions(generals)



if __name__ == "__main__":
    main()
```

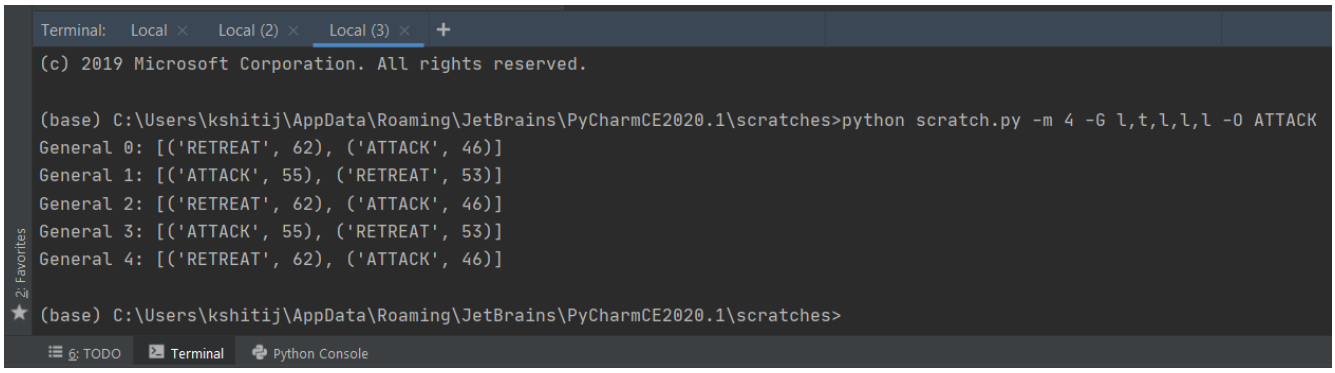
+--+

The ouput is as shown below:

-m Stands for Recursion (Level of recursion in the algorithm)

-G is a string of Generals where ‘l’ is loyal and ‘t’ is traitor. First general is commander.

-O stands for order given by commander. It can be either “ATTACK” or “RETREAT”



References

[1] Create your own Cryptocurrency in Ethereum Blockchain , https://medium.com/coinmonks/create-your-own-cryptocurrency-in-ethereum-blockchain-40865db8a29f

[2] The Byzantine Generals Problem, https://github.com/JVerwolf/byzantine_generals