Homework 7

Due **Monday Nov 7 at 11:59pm EST**: on coursesite

Built-in extension to Monday Nov 14 at 11:59 pm EST

CSE 242: Fall 2022

1 General Description

This assignment will have you use Solidity to create a smart contract that allows users to swap between two cryptocurrencies. In decentralized finance (defi), this type of a contract is called an Automatic Market Maker (AMM). AMMs are a cornerstone of defi. They allow participants to exchange assets with each other trustlessly and automatically. Anyone with cryptocurrency A who wants to exchange it for cryptocurrency B can use this contract to deposit cryptocurrency A and receive an amount of cryptocurrency B.

Our contract will handle exchanges between Ether and an ERC-20 token. ERC-20 is a token standard that allows anyone to create their own token using Solidity. ERC-20 tokens have their own functions for sending and receiving. Graders have created three ERC-20 tokens for your contract to interact with.

You will write one Solidity smart contract named Exchange.sol and deploy it three times to create an Ether/ERC-20 exchange for each ERC-20. This is done by accepting an address of an ERC-20 token in your contract's constructor.

All work and deployments should be done on the lu-eth network. The wallets you created in the previous assignment should hold a sufficient amount of Ether to cover any transaction fees. You will be sent a sufficient amount of all three ERC-20 tokens to your wallet for testing.

It is recommended to use the Remix IDE for development and calling functions: `https://remix.ethereum.org/`

2 Automatic Market Maker

This smart contract holds two types of tokens and determines the price ratio of these tokens by a mathematical market making formula (this is where the automatic part of Automatic Market Maker comes into play). Instead of buyers and sellers trading between one another, they interact with the contract. Users who deposit the two types of tokens into the contract are referred to as liquidity providers. Liquidity providers are issued an amount of liquidity positions which represents the ratio of their provided tokens to the rest of the tokens in the contract. This ratio is taken at the time of calling provideLiquidity(), so it will change when others provide liquidity. In your contract, liquidity positions will be represented as a mapping of liquidity providers (addresses) to the amount of liquidity positions they have (uint) ex: mapping(address=¿uint) liquidityPositions . When liquidity providers wish to withdraw their liquidity, the amount of tokens they receive is based on the amount of liquidity positions they are giving up. Traditionally, liquidity providers are incentivized by being paid a small fee for each exchange (token swap) made by other users, but for simplicity, this reward mechanism is not part of this assignment.

The most common market making formula is the constant product formula x*y=k where x and y are the balances of the two tokens in the contract and k is a global constant. K is updated whenever liquidity is provided or withdrawn. We will use this to calculate the amount of tokens to give a user when they swap one token for another.

In the real world, assets have a “real” market price, which is the price at which people will be willing to buy and sell the asset at, this constant product formula acts to find that market price.

Here is a nice explanation of the constant product formula: `https://www.youtube.com/watch?v=1PbZMudPP5E`

3 Details

The solution file is < 100 lines of code (excluding comments). This may be your first time writing Solidity code, and you are encouraged to post questions to Piazza to eliminate any sources of confusion[1]. Beyond the details of using Solidity for the first time, your team will face some issues with interacting with the ERC-20 token standard.

Spend some time reading up on how to interact with ERC-20 tokens: `https://docs.openzeppelin.com/contracts/4.x/erc20`

And have the Solidity docs bookmarked: `https://docs.soliditylang.org/en/latest/`

Once your contract is working, review your code. Be sure that your contract does not implement the DAO bug (re-entrancy), for example.

3.1 Formulas

Use these as a starting point for the calculations in your functions.

- `amountERC20TokenSentOrReceived / contractERC20TokenBalance =`
  `amountEthSentOrReceived / contractEthBalance`
- `amountTokenSentOrReceived / contractTokenBalance =`
  `liquidityPositionsIssuedOrBurned / totalLiquidityPositions`
  - ‘Token’ refers to either ERC-20 or Ether
  - Mappings cannot be iterated over in solidity, so you will want to have another global variable *totalLiquidityPositions* and update it whenever liquidity positions are issued (in the function *provideLiquidity*) or destroyed (in the function *withdrawLiquidity*).
- `K = contractEthBalance * contractERC20TokenBalance`

3.2 Tokens

Three ERC-20 tokens were created, deployed, and sent to your wallets for you to use. To see them in your metamask wallet, connect to lu-eth and in the assets tab “import tokens” then paste in an address. Each contract deploy will handle exchanging between a different ERC-20 token and Ether.

- AsaToken (ASA) address: 0x1A5Cf8a4611CA718B6F0218141aC0Bfa114AAf7D
- HawKoin (HAW) address: 0x42cD7B2c632E3F589933275095566DE6d8c1bfa5
- KorthCoin (KOR) address: 0x0B09AC43C6b788146fe0223159EcEa12b2EC6361

Here is the source code for each ERC-20 token:

AsaToken.sol

[1] if you are posting code, be sure to make the post *private to instructors*.

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.7;

import "@openzeppelin/contracts/token/ERC20/ERC20.sol";

/// @notice Token with faucet for grading purposes only
contract AsaToken is ERC20 {

    constructor() ERC20("AsaToken", "ASA") {
        _mint(msg.sender, 99999*1e18);
    }

    /// @notice In case anyone runs out of tokens.
    /// @dev Normally never do this, but it is fine for a class project.
    function mintMe(uint256 amount) external {
        _mint(msg.sender, amount*1e18);
    }
}
```

HawKoin.sol

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.7;

import "@openzeppelin/contracts/token/ERC20/ERC20.sol";

/// @notice Token with faucet for grading purposes only
contract HawKoin is ERC20 {

    constructor() ERC20("HawKoin", "HAW") {
        _mint(msg.sender, 99999*1e18);
    }

    /// @notice In case anyone runs out of tokens.
    /// @dev Normally never do this, but it is fine for a class project.
    function mintMe(uint256 amount) external {
        _mint(msg.sender, amount*1e18);
    }
}
```

KorthCoin.sol

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.7;

import "@openzeppelin/contracts/token/ERC20/ERC20.sol";

/// @notice Token with faucet for grading purposes only
contract KorthCoin is ERC20 {

    constructor() ERC20("KorthCoin", "KOR") {
        _mint(msg.sender, 99999*1e18);
    }
    /// @notice In case anyone runs out of tokens.
    /// @dev Normally never do this, but it is fine for a class project.
    function mintMe(uint256 amount) external {
        _mint(msg.sender, amount*1e18);
    }
}
```

Example: how to interact with and call functions on AsaToken in Remix

- Copy above AsaToken source code into Remix in a new AsaToken.sol file in the contract folder.
- Go to the "Solidity compiler" tab and click "Compile AsaToken.sol".
- Go to the "Deploy & run transactions" tab.
- In the "Environment" dropdown menu select "Injected Provider - Metamask" (Make sure you are connected to lu-eth in Metamask!).
- In the "Contract" dropdown menu select AsaToken which you just compiled in step 2.
- In the "At address" field where it says "Load contract from address" paste in the deployed AsaToken address from above, then click the button "At address".
- You can now call functions on the AsaToken smart contract that the graders deployed for this assignment. Repeat these steps for each token above.

3.3 Functions

For ease of reading here, we refer to "Ether" but your code will be operating using a smaller unit because Solidity does not support floats and 1 ETH is too large a unit of funds. In Solidity, Ether is handled as Wei where 1 Ether = 1000000000000000000 Wei (1 ETH = 1e18 Wei).

Your contract will have to include the following functions that any user should be able to call. These will be called for grading. Feel free to make other helper functions.

- `provideLiquidity(uint _amountERC20Token)`
  - Caller deposits Ether and ERC20 token in ratio equal to the current ratio of tokens in the contract and receives liquidity positions (that is:
    $totalLiquidityPositions * amountERC20Token/contractERC20TokenBalance ==$
    $totalLiquidityPositions * amountEth/contractEthBalance$)
  - Transfer Ether and ERC-20 tokens from caller into contract
  - If caller is the first to provide liquidity, give them 100 liquidity positions
  - Otherwise, give them $liquidityPositions =$
    $totalLiquidityPositions * amountERC20Token \;/\; contractERC20TokenBalance$
  - Update K: K = $newContractEthBalance * newContractERC20TokenBalance$
  - Return a uint of the amount of liquidity positions issued
- `estimateEthToProvide(uint _amountERC20Token)`
  - Users who want to provide liquidity won't know the current ratio of the tokens in the contract so they'll have to call this function to find out how much Ether to deposit if they want to deposit a particular amount of ERC-20 tokens.

  - Return a uint of the amount of Ether to provide to match the ratio in the contract if caller wants to provide a given amount of ERC20 tokens
    Use the above to get *amountEth =*
    *contractEthBalance * _amountERC20Token / contractERC20TokenBalance*)

- `estimateERC20TokenToProvide(uint _amountEth)`
  - Users who want to provide liquidity won't know the current ratio of the tokens in the contract so they'll have to call this function to find out how much ERC-20 token to deposit if they want to deposit an amount of Ether
  - Return a uint of the amount of ERC20 token to provide to match the ratio in the contract if the caller wants to provide a given amount of Ether
    Use the above to get *amountERC20 =*
    *contractERC20TokenBalance * _amountEth/contractEthBalance*)

- `getMyLiquidityPositions()`
  - Return a uint of the amount of the caller's liquidity positions (the uint associated to the address calling in your liquidityPositions mapping) for when a user wishes to view their liquidity positions

- `withdrawLiquidity(uint _liquidityPositionsToBurn)`
  - Caller gives up some of their liquidity positions and receives some Ether and ERC20 tokens in return.
    Use the above to get
    *amountEthToSend = _liquidityPositionsToBurn*contractEthBalance / totalLiquidityPositions*
    and
    amountERC20ToSend =
    *_liquidityPositionsToBurn * contractERC20TokenBalance / totalLiquidityPositions*
  - Decrement the caller's liquidity positions and the total liquidity positions
  - Caller shouldn't be able to give up more liquidity positions than they own
  - Caller shouldn't be able to give up all the liquidity positions in the pool
  - Update K: K = newContractEthBalance * newContractERC20TokenBalance
  - Transfer Ether and ERC-20 from contract to caller
  - Return 2 uints, the amount of ERC20 tokens sent and the amount of Ether sent

- `swapForEth(uint _amountERC20Token)`
  - Caller deposits some ERC20 token in return for some Ether
  - hint: *ethToSend = contractEthBalance - contractEthBalanceAfterSwap*
    where *contractEthBalanceAfterSwap = K / contractERC20TokenBalanceAfterSwap*
  - Transfer ERC-20 tokens from caller to contract
  - Transfer Ether from contract to caller
  - Return a uint of the amount of Ether sent

- `estimateSwapForEth(uint _amountERC20Token)`

  - estimates the amount of Ether to give caller based on amount ERC20 token caller wishes to swap for when a user wants to know how much Ether to expect when calling swapForEth
  - hint: ethToSend = contractEthBalance-contractEthBalanceAfterSwap where contractEthBalanceAfterSwap = K/contractERC20TokenBalanceAfterSwap
  - Return a uint of the amount of Ether caller would receive

- `swapForERC20Token()`
  - Caller deposits some Ether in return for some ERC20 tokens
  - hint: *ERC20TokenToSend = contractERC20TokenBalance - contractERC20TokenBalanceAfterSwap* where *contractERC20TokenBalanceAfterSwap = K / contractEthBalanceAfterSwap*
  - Transfer Ether from caller to contract
  - Transfer ERC-20 tokens from contract to caller
  - Return a uint of the amount of ERC20 tokens sent

- `estimateSwapForERC20Token(uint _amountEth)`
  - estimates the amount of ERC20 token to give caller based on amount Ether caller wishes to swap for when a user wants to know how many ERC-20 tokens to expect when calling swapForERC20Token
  - hint: *ERC20TokenToSend = contractERC20TokenBalance - contractERC20TokenBalanceAfterSwap* where *contractERC20TokenBalanceAfterSwap = K / contractEthBalanceAfterSwap*
  - Return a uint of the amount of ERC20 tokens caller would receive

3.4 Events

Events are ways to log specific information onto a blockchain. Your contract should emit the following events:

- `event LiquidityProvided(uint amountERC20TokenDeposited, uint amountEthDeposited, uint liquidityPositionsIssued)`
- `event LiquidityWithdrew(uint amountERC20TokenWithdrew, uint amountEthWithdrew, uint liquidityPositionsBurned)`
- `event SwapForEth(uint amountERC20TokenDeposited, uint amountEthWithdrew)`
- `event SwapForERC20Token(uint amountERC20TokenWithdrew, uint amountEthDeposited)`

For more information on events (and emitting them), see Solidity documentation at `https://docs.soliditylang.org/en/latest/contracts.html#events`

4 Tips

- We are creating contracts that transfer and receive both Ether AND an ERC-20 token. Contrary to intuition, Ether is NOT an ERC-20 token. The syntax for transferring ERC-20 tokens and Ether is different. Reference OpenZeppelin docs `https://docs.openzeppelin.com/contracts/4.x/api/token/erc20` for ERC-20 functions and reference Solidity docs `https://docs.soliditylang.org/en/latest/` for Ether functions.

- Solidity doesn't support floating point numbers (be careful dividing variables!), so numbers are scaled up to represent units of precision. In Solidity, Ether is handled as Wei where 1 Ether = 1000000000000000000 Wei (1 Eth = 1e18 Wei). For example, no floats means you can't represent 0.5 Ether, so it is instead represented as 500000000000000000 Wei (0.5 Eth = 5e17Wei).
- You might run into 'ERC20: insufficient allowance' when testing your Exchange.sol functions which handle receiving ERC-20 tokens. You must first call the 'approve(spender, amount)' on the deployed ERC-20 token which you are trying to send to Exchange.sol (see section 3.2 for how to call functions on the deployed ERC-20 tokens). 'spender' is the address of your deployed Exchange.sol. 'amount' is the maximum amount of the ERC-20 that you are allowing 'spender' to take from your wallet. This grants the deployed Exchange.sol permission to use some of your ERC-20 tokens. This 'approve(spender, amount)' pattern is for security purposes. Every user of a deployed Exchange.sol will have to call 'approve(spender, amount)' on the corresponding ERC-20's contract before calling functions on Exchange.sol that accept ERC-20 tokens.

5 Submitting

After deploying an exchange for each ERC-20 token, you should provide some liquidity in each so graders can interact with them. Then, create a text document named contractInfo.txt with the following information:

```
Deployed Ether/ASA exchange: {address of deployed exchange}
Deployed Ether/HAW exchange: {address of deployed exchange}
Deployed Ether/KOR exchange: {address of deployed exchange}
```

Zip up this .txt file along with your Exchange.sol Solidity code. Name the zip your team's acronym ex: "abc.zip" and submit it to coursesite.