CPSC 466/566: Blockchains and Cryptocurrencies[a] **Spring 2023**

Programming Project #2

Due: 11:59pm on **Thu., May 4, 2023**
Submit via Gradescope **Course code: RZ2Z2W**

[a] *From materials of Stanford University, CS 251, Dan Boneh*

For this project, you will be working on a web client and two Solidity contracts to implement a decentralized cryptocurrency exchange. By the end of the project, your exchange will have much of the functionality possessed by full fledged decentralized exchanges such as Uniswap. Additionally, you will create your own ERC20 token, which you will then trade over your exchange.

Throughout this project, make sure to write code that is resilient against adversarial attacks. Remember that attackers can call your smart contracts with arbitrary inputs, not just through the provided user interface. As such, make liberal use of `require` statements to test any assumptions baked into your code.

For most students, this project will likely represent their most comprehensive project implemented so far in Solidity. As such, we encourage you to start early and ask questions. We will guide you through implementing a decentralized exchange in several phases, and we believe your end product will give you something to be very proud of! Let's get started.

As part of this, we will also build a user interface that computes useful information for the user and allows non-programmers to use the DApp.

1 Getting Started

1.1 Setup

1. Install the prerequisite software: you'll need to download and install Node.js. Hardhat only supports Node.js V12.xx, 14.xx, and 16.xx. You can find the previous releases here, and select an appropriate version to install.

2. Download and extract the starter code from the course website.

3. `cd` into the starter code directory.

4. Run `npm install --save-dev hardhat` to install the Ethereum development environment Hardhat, which you will use to simulate an Ethereum node on your local machine. If you encounter an error in regards to an incompatible Node version, please install a supported Node.js version and use `nvm use <version>` to switch to it.

5. Run `npm install --save-dev @nomiclabs/hardhat-ethers ethers` to install a Hardhat plugin to deploy a Hardhat node which your scripts will use.

6. Run `npm install --save-dev @openzeppelin/contracts` to install OpenZeppelin libraries.

1.2 Compile, Deploy and Test

1. Open the starter code directory in your favorite IDE or text editor (something like Sublime Text, Atom, or Visual Studio Code works nicely). You'll be modifying `contracts/token.sol` and `contracts/exchange.sol` to define your Solidity contracts and `web_app/exchange.js` to build the Javascript backend. Looking at the other files may help understand how the Hardhat node and web client works. There are places marked with functions to modify and you can add helper functions to the three files listed above. Please do not modify any other code or install additional node packages.

2. Peruse the starter code, the ethers.js documentation, the Solidity documentation, and the OpenZeppelin ERC-20 contract implementation. Think carefully about the overall design of your system before you write code. What data should be stored on chain? What computation will be done by the contract vs. on the client?

3. After you finish implementation, run `npx hardhat node` to start the local node. If the node is started correctly, you should see in terminal: *Started HTTP and WebSocket JSON-RPC server at https://localhost:8545*

4. **Deploying the Contracts:** Open another terminal tab or window, and `cd` into the starter code directory.
   Run `npx hardhat run --network localhost scripts/deploy_token.js` to compile and deploy your token contract. Upon success, you should see this message on the terminal: *Finished writing token contract address: ....* Copy that value and paste it into the `address tokenAddr` field in `contracts/exchange.sol`.

5. Run `npx hardhat run --network localhost scripts/deploy_exchange.js` to deploy your exchange contract. Upon success, you should see *Finished writing exchange contract address: ....* Save that address for the next step.
   Step 5 is the same as Step 4 but instead for the contract `exchange.sol`. The newly created contract address will be copied in Step 6.

6. **Update the contract address and ABI** in `web_app/exchange.js`.
   Update the `const token_address` and `const exchange_address` variables with the two contract addresses. Unlike in Solidity, the addresses in Javascript do not need to be checksummed.

   The ABIs can be copied from `artifacts/contracts/token.sol/token.json` and `artifacts/contracts/exchange.sol/exchange.json`. To correctly update the ABI, please copy the whole list after the 'abi' field, starting from the square bracket. The address is saved in the previous step. Make sure your contract address is a string.

7. Open the `web_app/index.html` file in your browser. Until you have finished implementing Section 3 (`contracts/token.sol`), there will be an error in the browser console. Otherwise, you can play around with the page, and run sanityCheck! See Section 10 for more details.

Note on OSs: All of the above steps should work on Unix-based systems and Windows. The commands we ask you to execute will work in a standard Unix terminal and the Windows Command Prompt.

2 Components

The project has three major components:

- `contracts/token.sol` is a smart contract written in Solidity that will be deployed on the blockchain. You will have to modify it to create and deploy your own token. See Section 3 for implementation details.
- `contracts/exchange.sol` is another smart contract written in Solidity that will be deployed on the blockchain. You will have to modify it to create and deploy a decentralized exchange (DEX), also called an automated market maker (AMM), modeled after Uniswap V1. See Section 4 for implementation details.
- `web_app/exchange.js` is a client running locally in a web browser written in JavaScript. It observes the blockchain using the ethers.js library and can calls functions in the smart contracts `token.sol` and `exchange.sol`. See Section 4 for more details. For more information on how ethers.js works and setting up this assignment, watch the Section 5 recording on Panopto.

Also, file `web_app/index.html` when opened in your browser (works best in Chrome) allows you to access the user interface through which you can test your exchange. On this page, you can pick an address and add liquidity, remove liquidity, and swap your token for ETH and vice-versa. This file and other files should not be modified. Only `token.sol`, `exchange.sol` and `exchange.js` should be modified.

3 Create and Deploy Your Own Token

In the first part of this project, you will create and deploy your own ERC-20 token. ERC-20, as mentioned in class, is a standard for implementing fungible tokens. Luckily for us, much of the code for ERC-20 standard has already been written and is open source. In this project, we will use the standard ERC-20 implementation from the OpenZeppelin project. Make sure you understand the ERC-20 standard, as well as the OpenZeppelin implementations of ERC-20 and Ownable.

Once you've read through the starter code, complete the following steps in `contracts/token.sol` and `web_app/exchange.js`:

1. Come up with a fun (but appropriate!) name for your token. Any name is fine, so feel free to have fun and come up with something creative. Set the private string `_name` of `token.sol` to be the name of your token. In addition, update the `token_name` variable at the top of `exchange.js` to be the same name.
2. Decide on a short symbol for your token (e.g. ETH, instead of Ethereum). Set the private string `_symbol` to be that symbol, and update the `token_symbol` variable at the top of `exchange.js` with the same name.
3. Implement the `mint(uint amount)` and `disable_mint()` functions. `mint` is a public function that creates `amount` tokens. While the OpenZeppelin ERC-20 contract does not provide a `mint()` function, you can still call `_mint()` appropriately to do so. `disable_mint()` makes it such that calling `mint()` will never succeed again. As such, your `mint` implementation must

fail if `disable_mint()` has been called. Also note the `Ownable` modifier on both functions, which makes it such that only the contract administrator can call either one. The `mint()`/ `disable_mint()` paradigm is a simple way of initially generating tokens and guaranteeing that the total supply will remain constant after the supply of the token has been created.

4. Deploy your token contract. Copy the address and ABI of the token contract to the `token_address` and `token_abi` variables in `exchange.js`.

5. Lastly, copy the address of the token contract to the `tokenAddr` variable in `exchange.sol`. Remember to use the checksummed version of the address in the contract. **Every time you redeploy your token contract, you must repeat this step.**

After completing the above steps, you should have your own token all deployed on Hardhat!

You may now try deploying your exchange contract as well. Copy the address and ABI of the token contract to the `exchange_address` and `exchange_abi` variables in `exchange.js`. All the functionality except initial pool setup (which we implemented for you) will be missing. That said, if you have completed Section 2 and the deployment process properly, you should see no errors occur in your browser console and should see a 1-to-1 exchange rate between ETH and your token. There should be 5000 ETH and 5000 of your tokens in under the "Current Liquidity" display.

4 Setting Up Your Basic Exchange

In this part of the assignment, you will implement the basic functionality of your cryptocurrency exchange. Our exchange is modeled after Uniswap V1. Your exchange will only allow for swaps between your Token and test ETH. The changes in this section will primarily affect two files: `exchange.js` and `exchange.sol`. Familiarize yourself with the starter code for those files.

A decentralized exchange consists of two types of participants: liquidity providers (LP) and traders. For a given exchange pool between two tokens, liquidity providers provide some equal value amount of both types of liquidity (in your case, ETH and your own token). When traders swap between the two currencies, they will add some amount of one currency to the liquidity pool, and will be sent an equal value of the other currency from the pool. The exchange rate between the two currencies is determined by the constant product formula:

> Let x be the amount of currency A that is in the liquidity pool, and let y be the amount of currency B. Let k be some constant. After every swap, it must be true that
>
> $$x * y = k.$$

During each swap, the exchange must send out the correct amount of the swapped-to currency such that the pool always remains on the constant product curve. The price of currency B in terms of currency A can the be calculated as x/y, whereas the price of currency A in terms of currency B can be calculated as y/x. Every swap will thus modify the exchange rate. This makes sense, as each swap is an indication of demand for a given currency. The effects of each swap on the price of the currencies will be relevant in section 6.

When a liquidity provider adds liquidity, they must provide equal values of currency A and B, as determined by the current exchange rate. Note then that adding liquidity will not change the

exchange rate between the currencies, but will increase the value of the constant k. Similarly, when a liquidity provider goes to withdraw their liquidity, they must withdraw equal values of currency A and currency B. Therefore, k will decrease but the exchange rate will remain the same.

This has another notable consequence: since a liquidity provider can only withdraw equal values of each currency, they are not actually entitled to withdraw their exact initial investment (in terms of quantity of each token). Rather, providing liquidity is analagous to owning a percentage share of the liquidity pool, which the provider is then entitled to withdraw at a later time. A liquidity provider who provided 10% currency A and currency B is entitled to withdraw 10% of each of the reserves for those currencies (assuming their percentage was not diluted by other providers), even if 10% does not line up precisely with the quantities of each token they provided. In this way, liquidity providers can suffer impermanent loss, if the value of what they are entitled to withdraw is less than the value that they initially invested. If you are unfamiliar with impermanent loss, please review the lecture and section on decentralized exchanges.

With the above in mind, you will now implement the basic functionality of your exchange. We take care of initializing the pool for you by implementing and calling the `createPool` function. We take ETH and tokens from the first address to initialize the pool, and *you do not need to track this initial amount when tracking liquidity providers.* **In order for other addresses to obtain tokens and/or provide liquidity, they must first swap for tokens on the exchange.** In `exchange.sol`, implement the following functions:

- `function addLiquidity() external payable`:
  Add liquidity to the pool if the provider possesses sufficient ETH and tokens (otherwise the transaction should fail). The caller will send ETH to the contract, which can be accessed using `msg.value`. This function should also transfer the equivalent amount of tokens based on the current exchange rate from the sender's address to the contract (using the token's `transfer` or `transferFrom` method), and update the exchange state accordingly. The transaction must fail if the provider's funds are insufficient. See Section 9 for advice on how to best keep track of liquidity.

- `function removeLiquidity(uint amountETH) public payable`:
  Remove a specified amount of liquidity from the pool (if the provider is entitled to remove given amount of liquidity) and update the exchange state accordingly. `amountETH` is numeric amount of ETH the liquidity provider wants to take out, so they should receive a total value equivalent to `2 * amountETH` after they receive tokens and ETH. Be sure to update the amount of liquidity provided by each liquidity provider accordingly. The function should fail if users try to remove more liquidity than they are entitled to, or if they try to deplete the ETH or token reserves to 0.

- `function removeAllLiquidity() external payable`:
  Remove the maximum amount of liquidity that the sender is allowed to remove and update the exchange state accordingly. In addition, be sure to update the amount of liquidity provided by each liquidity provider. Similarly, this function should fail if the liquidity provider will drain the ETH or token reserves to 0.

- `function swapTokensForETH(uint amountTokens) external payable`:
  Swap the given amount of tokens for the equivalent value of ETH and update the exchange state accordingly. If the provider does not have sufficient tokens for the swap, the transaction

should fail. Additionally, if completing the swap would completely remove all ETH from the pool, the transaction should fail to avoid having zero ETH and (therefore) an undefined exchange rate. Be sure to leave at least 1 ETH and 1 token in the pool at all times.

- `function swapETHForTokens() external payable`:
  Swap the given amount of ETH for the equivalent value in your token and update the exchange state accordingly. Similar to `addLiquidity()`, the sender would send ETH into the contract, which can be accessed through `msg.value`. If completing the swap would completely remove all tokens from the pool, the transaction should fail to avoid having zero tokens and (therefore) an undefined exchange rate. Be sure to leave at least 1 ETH and 1 token in the pool at all times.

In each of the above functions, be sure that you are adjusting `token_reserves`, `eth_reserves`, and/or `k` in the correct way such that the exchange is always on the constant product curve described above. Additionally, be sure that functions fail when the caller does not possess sufficient funds. Finally, remember to set `address tokenAddr` to be your deployed token contract's address. You can now run `npx hardhat run --network localhost scripts/deploy_exchange.js` to debug and deploy your code.

Handling round-off errors. Swapping tokens following the constant product rule involves performing operations such as additions and divisions, and this inevitably leads to round-off errors. It is crucial to ensure that these errors cannot compound. This could result in losses for liquidity providers if slightly more tokens than expected are withdrawn from the pool at each swap, for example.

As a consequence, the product $x*y$ might not be strictly equal to k due to some round-off errors - which is not a problem - but since k remains strictly unmodified these errors don't compound but rather compensate themselves when many swaps are performed. Round-off errors can also occur when adding or withdrawing liquidity because your token and ETH cannot be splitted further than 1 unit (in this project at least, see section 8 for more details). *We don't expect you to handle these types of round-off errors in this project.* Thus, it is ok if $x * y$ deviates from k when performing arithmetic, as well as other similar roundoffs when calculating liquidity. Nevertheless, it is still important to update `k` at appropriate places.

5 Implementing the Backend

After you finish implementing the contract functions, implement the following functions in `exchange.js`. You can ignore the `maxSlippagePct` variable for now – this will be used in Section 6. For the most part, these would just call the token and exchange functions you wrote above:

- async function addLiquidity(amountEth, maxSlippagePct)
- async function removeLiquidity(amountEth, maxSlippagePct)
- async function removeAllLiquidity(maxSlippagePct)
- async function swapTokensForETH(amountToken, maxSlippagePct)
- async function swapETHForTokens(amountEth, maxSlippagePct)

You can call contract code with `await contract.functionName(args)` or `await contract.connect(anotherSigner).functionName(args)`. For more syntax help, we strongly encourage you to check out the ethers.js documentation, as well as Section 9 for some tips.

Once you've fully implemented your smart contract and the corresponding JavaScript code, update the `token_abi` and `exchange_abi` variables at the top of the file, and copy the contract addresses to the `token_address` and `exchange_address` variables in `exchange.js`. Be sure to include the outermost brackets when copying the ABI. If you reload `index.html`, you should now be able to provide liquidity, remove liquidity, and perform swaps.

6 Handling Slippage

There is a significant issue with our exchange as we implemented it in Section 3, as it does not account for "slippage". Recall that with every swap on a decentralized exchange, the price of each asset will shift slightly. Since many users may be trying to swap currency at once on a decentralized exchange, there may be a shift in the exchange rate between the submission of a swap transaction and the actual processing of that transaction. This shift in the exchange rate between the exchange's quote price and actual price is called "slippage." Slippage is of particular concern while trading volatile assets. For example, if a user submits a swap transaction to swap some amount of currency A for currency B, and then the price of currency B dramatically increases from the quote price, the user might not actually wish to complete the swap transaction.

Additionally, not sufficiently handling opens users up to a type of attack known as a sandwich attack. A sandwich attack works as follows:

1. Alice submits a swap transaction to convert some large amount of currency A into currency B.
2. An adversary sees Alice's transaction and front-runs it with a very large purchase of currency B, thus raising the price of asset B.
3. Alice buys currency B at the new higher price, even further raising the price of currency B.
4. The attacker then immediately sells all their newly acquired currency B at the higher price, making a quick profit.

Vulnerability to sandwich attacks is bad for users of a decentralized exchange, as users consistently pay higher exchange rates than the true asset value. As such, it is important that we upgrade our exchange to properly handle slippage and defend against sandwich attacks.

The most common defense against sandwich attacks is to allow users to set some maximum slippage while submitting the transaction. This parameter, typically a percentage, will cause the transaction to fail if the price of the assets has changed by more than the maximum allowed slippage. This limits the damage that can be done by sandwich attacks and protects users who are swapping volatile assets.

To implement a maximum slippage requirement, perform the following steps:

1. In `exchange.sol`, update your `swapTokensForETH` and `swapETHForTokens` functions to take in a `uint max_exchange_rate` parameter. You may also pass in other parameters if needed for your design. While swapping, the swap should fail if the current price of the new asset (i.e. the asset the user is swapping to) has increased to more than the maximum exchange

rate. Note that the price of the asset decreasing is good for the user, so we don't have to fail in that case.

2. Update `addLiquidity`, `removeLiquidity`, and `removeAllLiquidity` functions to take in `uint max_exchange_rate` and `uint min_exchange_rate` parameters. You may also pass in other parameters if needed for your design. While providing liquidity, the transaction should fail if the current price of the new asset (i.e. the asset the user is swapping to) has increased to more than the maximum exchange rate or fallen below the minimum exchange rate. Sudden price shifts in either direction can subject providers to impermanent loss before they deposit their liquidity, so thus we want the liquidity providers to specify a maximum and minimum exchange rate.

3. Now update your `exchange.js` file to communicate with the contract about the max/min exchange rates. The `maxSlippagePct` parameter is provided, which represents the maximum allowable percent price change before the transaction should fail. In testing and in the browser interface, this parameter is passed as an int, not as a float - i.e. 4% is passed as 4, not 0.04. This parameter can be used in each of the JavaScript functions to calculate the correct values for `max_exchange_rate` and/or `min_exchange_rate`, which can then be passed to the contract. The `getPoolState` function that we provide to you may be useful here.

As always, after updating your contract make sure to recompile, redeploy, and copy the new ABI and contract address to the variable at the top of your `exchange.js` file. At this point, you can also uncomment the sanityCheck() function to check your implementation. See Section 10 for more details about the sanityCheck.

7 Rewarding Liquidity Providers

After completing the above sections, you now have a working exchange that allows users to limit the amount of slippage they wish to tolerate! There is one more big issue, however. We have discussed several times how liquidity providers are taking on risk in the form of impermanent loss. That is, the value of their liquidity stake may decrease if the price of either asset changes. In practice, since many cryptocurrencies are quite volatile, this is a level of risk that no liquidity provider would be willing to take on for free.

As such, we need to incentivize liquidity providers to give liquidity to the pool. In real world exchanges, liquidity providers are incentivized to provide liquidity because they receive a small fee from every swap transaction. These fees are automatically reinvested into the liquidity pool on behalf of each liquidity provider. When a provider goes to withdraw their liquidity, the amount that they are entitled to withdraw includes all fees they have been awarded since providing their liquidity.

You will now implement the same fee reward scheme for liquidity providers. **You are free to design your own as long as it meets the requirements stated at the end of this section. However, we strongly suggest the following one, explained in plain text.**

- Each liquidity provider has ownership of a fraction f of the mining pool. This fraction is stored in the smart contract for each liquidity provider (ie. an array stores for each address the fraction of the mining pool owned by this address). For this assignment, we'll be setting the swap fee to 3% to aid with the autograder, as represented by the `swap_fee_numerator`

and `swap_fee_denominator` fields. The liquidity providers' ownership proportions should be unchanged when swaps are performed, since liquidity rewards are distributed based on each lp's ownership percentages. Note that because of fees, the total liquidity $l = \sqrt{xy}$ grows, and thus k would shift a bit. This results in profits for liquidity providers if the exchange rate doesn't change, and serves to combat impermanent loss.

- As mentioned above, due to liquidity rewards, k will now slightly change on every swap. For the purposes of this project, however, **you should only change k when adding or removing liquidity**. Thus, you do not need to worry about these small shifts in k due to the change in $x * y$ during swapping.

- When a liquidity provider withdraws its liquidity, they get a fraction f of both tokens in the pool corresponding to their ownership fraction, as well as their rewards. Other liquidity providers have their ownership fraction increased accordingly. All the fractions should add up to 1.

- When a liquidity provider adds liquidity in the pool, they get an ownership fraction f on the pool equal to the proportion of their tokens in the new state of the pool. Other liquidity providers have their ownership fraction shrunk accordingly. Again, all the fractions should add up to 1.

If you have implemented liquidity tracking with fractions, then this section should not take too much additional work. Alternatively, we will also accept any design that fulfills the requirements listed below.

Liquidity Rewards Requirements:

1. Your pool must charge the person performing the swap some nonzero percent fee on every swap transaction.[1] You can define this value using `swap_fee_numerator` and `swap_fee_denominator`, as explained above.

2. When a swap occurs, the value of tokens or ETH sent to the trader should be equal to $(1-p)$ times the value of the assets they are swapping, where p is the percent fee taken for liquidity providers. For example, if the fee is 1% and a user is swapping 100 ETH for your tokens, they should only be sent the equivalent of 99 ETH.

3. When a fee is taken during a swap, it should be distributed to the liquidity providers such that each provider should later be able to withdraw their original stake plus fees. Fees should be distributed proportionally based on providers' fractional share of the liquidity pool at the time that the swap took place. It would be incorrect, for example, if a liquidity provider who provided half of all liquidity in the pool at time t was allowed to withdraw half of all fees taken prior to time t. Liquidity providers should not have to take any additional steps to claim their fees beyond calling removeLiquidity. Additionally, liquidity rewards should *not* be sent out of the exchange to the providers each time a swap takes place, since doing so would be prohibitively expensive in practice.

[1] For reference, the default fee on Uniswap is 0.3%, whereas centralized exchanges typically charge around 1-4% to swap currencies.

4. While deciding between different design options, we encourage you to opt for the solution that minimizes gas costs. We will not grade strictly on gas usage; however, you will be required to justify your design decisions in the design doc in Section 6.

After designing and implementing the above section, you should have a fully working exchange! Congratulations! Test your functions using the provided UI in `index.html`, or write testing code in JavaScript. Implementing this project represents a very impressive achievement, so give yourself a pat on the back. In fact, with some security modifications, you can deploy both your token and your exchange onto the Ethereum mainnet, and thus have an exchange you can call your own!

8 Note on Solidity and Javascript Decimals

Unlike most programming languages, Solidity does not support floating point arithmetic. Thus, all ERC-20 tokens keep track of a *decimals* variable, which indicates how many decimals to the left the numbers should be interpreted as. For example, ether uses 18 decimal points, so 1 ETH would be represented as 10^{18} in the contract. Similarly, 1 wei = 10^{-18} ETH, so 1 wei is represented as just 1. Unfortunately, Javascript also has a limit to how large integers can be: Numbers.MAX_INT $= 9 * 10^{15}$. To balance out between the two, in our exchange, we initialize the pool to have 10^10 tokens, which means that the pool starts off with 10^{-8} ETH and 10^{-8} of your tokens.

The consequence of Solidity not supporting float operations is that all decimal numbers would be truncated. For example, 5 / 2 = 2. This is why the decimals variable is needed, as representing 5 ETH as $5 * 10^{18}$ would lead to $5 * 10^{18}$ / $2 = 25 * 10^{16}$, and since we know there are 18 decimal places, we can see that this corresponds to 2.5 ETH, as desired. However, in some cases, underflow can still occur, especially if the numerator is smaller than the denominator in the calculations and thus leading to 0. In this case, you would need to be careful with the order of operations, such as multiplying or adding first before dividing. In general, it is a good idea to delay division until as late as possible to prevent encountering this rounding error.

In this project, we will not be testing you on overflow, and we have tried our best to abstract away all decimals. Thus, when testing our your exchange, we will choose values that won't have your exchange exceed Javascript's INT_MAX. However, we will be checking your implementation to make sure you have accounted for underflow in your exchange.

9 Implementation Advice

While the overall design of your contract is open-ended, here is some advice that might help streamline your implementation process:

1. **Keep track of the liquidity providers' proportions, rather than absolute values**. For example, if there is 1000 ETH and 1000 tokens in the pool, and Alice provides 500 ETH and 500 tokens, then Alice is entitled to 1/3 of the pool. Rather than storing 500 ETH for her liquidity, we recommend you to store that she owns 1/3. During swaps, this value should not change (even when implementing lp rewards). However, when a liquidity provider adds or removes liquidity, this value (and all the other liquidity providers' fractions) should be updated accordingly. In addition, since Solidity does not support floating-point decimals, we

recommend you to fix a denominator (i.e. 100, 1000, etc.) so you can store the numerator of the percentage as `uints`.

2. **Handing edge rounding errors**. Again, since Solidity does not support floating points, we recommend you to *perform multiplication before division* whenever possible. This avoids running into division rounding to 0, and then multiplying to get 0 again. Similarly, with uint types, be sure to *perform addition before subtraction* when possible to prevent underflow.

3. **Approving token transfers.** In order for a third-party address (for example, the contract) to send or receive your tokens on your behalf, you must first grant them permission to by using the token contract's approve() function. This function would need to be initiated by the user, you will not be able to run approve() from the contract itself. Thus, be sure to call this function in the Javascript before calling the exchange function when appropriate. The details of this function can be found in Openzeppelin's ERC20 implementation.

4. **Sending ETH to and from the Contract.** In order to successfully transfer ETH from a user account to a contract, the function that handles the transfer must be marked as `payable`. It is important to note that simply specifying an argument in the contract function to specify an amount of ETH will not transfer the ETH; rather, ETH is transferred via the `msg.value` parameter. **In order to avoid decimals, please use `ethers.utils.parseUnits()` with the units set to `"wei"` when passing in ETH into the contract**. Similarly, if you want to send ETH from the contract to the user address, then you can use the `payable()` function appropriately. More details can be found here.

5. **Iterating through mapping keys.** Solidity does not support iterating through keys of a mapping. Thus, we define `address[] private lp_providers` for you to store the addresses of liquidity providers. In addition, remember that Solidity arrays do not automatically "shift" all elements when a value in the middle of the array is removed. Thus, we provide a helper function `removeLP()` that removes a liquidity provider from the array while filling the "gap". This will update `lp_providers.length` accordingly. Be careful when calling this function while you are iterating through the array, however, as it is risky to change the length of the array while iterating through it if you want to reach every element.

6. **Accessing the contract address.** You can get the contract's address in Solidity by calling `address(this)`.

7. **Access the contract's token and ETH amounts.** While you can calculate your contract's token and ETH amounts by hand while performing intermediate calculations, you can access the true balances with the following functions:

   - ETH balance: `address(this).balance`
   - Token balance: `token.balanceOf(address(this))`

   We highly recommend using these true values to calculate k rather than using `token_reserves` and `eth_reserves`, and to double check your work.

8. **Javascript Async Functions**. In Javascript, there are many times when functions are run asynchronously. That is, if your code calls an `async function`, then the code will continue to run past that line without waiting for that async function to finish executing. By default,

async functions return a Promise object, which specifies some value will be returned at the end of the function execution. In order to have your code wait for the async function to run and get the result, you will need to use the `await` keyword. You can learn more about Javascript async functions here.

It is important to note that all Solidity function calls will be async from the backend. Thus, in `exchange.js`, in order to call a function from the contract and get the output, be sure to use the `await` keyword, such as `var num = await token_contract.function(args)`.

9. **Security of the Contract**. Remember, since all deployed contracts are public on the chain, it is important to keep your contract safe. While we will not be actively testing your contract for security, we still expect to see some defenses via using `require()` or `assert()` statements.

10 Sanity Check

In order to test your implementation, we have implemented two sanity check programs that run depending on whether you have implemented liquidity rewards. We check this value by reading the `swap_fee_numerator` value in `exchange.sol`: if that value is 0, then we assume you have not implemented swap fees and liquidity rewards. You will not receive full credit until liquidity rewards are fully implemented, but we also want to give credit for completing the basic implementation of this project.

To enable sanity check, uncomment the `setTimeout()` function with `sanityCheck()`, and refresh the page. We tried to design the sanityCheck to run properly even after the first load, so you do not need to redeploy the contracts and reset the pool state every time you want to run sanityCheck. However, due to rounding errors, there might be a point in which the sanityCheck passes when the exchange rate is 1:1 but not with your current exchange status. Your sanityCheck score for this project will be based off of the initial pool state – that is, the exchange rate between tokens is 1:1, and there are 5000 ETH and 5000 tokens in the pool.

11 Design Document

Please fill in `DesignDoc.txt` with your answers to the following questions:

1. Explain why adding and removing liquidity to your exchange does not change the exchange rate.
2. Explain your scheme for rewarding liquidity providers and justify the design decisions you made. How does it satisfy the liquidity rewards requirements outlined in Section 7?
3. Describe at least one method you used to minimize your exchange contract's gas usage. Why was this method effective?
4. Optional Feedback
   (a) How much time did you spend on the assignment?
   (b) What is one thing that would have been useful to know before starting the assignment?

(c) If you could change one with about this assignment, what would you change?

(d) Please feel free to include any other feedback you may have.

12 Submission

When you are ready to submit, please upload a .zip file of your entire project to Gradescope. To do so, navigate to the root folder for the project and zip the entire folder. Be sure to include the following files in your project zip: `exchange.js`, `exchange.sol`, `token.sol`, and `DesignDoc.txt`. If you added or changed any other files, please be sure to include those files as well.

You are allowed to work with 1 project partner (team size ≤ 2). Don't forget to add your project partner to your Gradescope submissions. If you are using late days for this project, make sure that both partners have enough late days remaining according to class policy on the website.