Homework 3: Building your own Decentralized Exchange

CS765: Introduction to Blockchains,
Cryptocurrencies and Smart Contracts.

1 Assignment Overview

This assignment is about Decentralized Finance, in particular - Decentralized Exchanges (DEX). If you are new to finance, this is a great starting point and you will find useful resources for some jargons in the financial lingo in the last section. The objective of this assignment is to write, understand, compile and test Solidity Smart contracts on Remix IDE (online) which implement a DEX. The assignment has a total of 100 points.

2 Introduction to DeFi and DEX

DeFi is an all-inclusive term for any application that uses blockchain and cryptocurrency techniques or technology to offer financial services. Some of these applications can provide anything from basic services like savings accounts to more advanced services like providing liquidity (liquidity $\approx$ easily transferable asset/currency) to businesses or investors.

A DEX (Decentralized Exchange) is like a peer-to-peer money swap machine that lets you trade crypto without a bank or middleman. Instead of using a company to match buyers and sellers, it uses smart contracts on a blockchain to do the job automatically.

One of the more notable DeFi service providers is UniSwap, which is a decentralized cryptocurrency exchange that uses a set of smart contracts to create liquidity pools for the execution of trades.

Uniswap is like a crypto vending machine: you put in one token and get another without needing a buyer or seller.

In an *Automated Market Maker* (AMM) model like Uniswap, liquidity providers (LPs) deposit tokens into a *liquidity pool*, maintaining a fixed ratio of assets. The pool follows the **constant product formula**:

$$x \times y = k \tag{1}$$

where:

- x is the reserve of token X in the pool,
- y is the reserve of token Y in the pool,
- k is a constant that remains unchanged during swaps, ensuring price discovery.

However, when LPs **deposit their tokens**, they typically deposit in proportion to the current price, preserving the ratio x/y rather than preserving k. Since new liquidity changes the total reserves, the product $x \times y$ increases, but the relative price remains stable.

During a **swap**, a user provides an input amount of one token, and the pool returns an output amount of the other token, ensuring that the product $x \times y$ remains **constant** (except for small fees). This mechanism prevents risk-free infinite arbitrage and maintains an automated pricing model.

3 Task 1 [5 points]

Deploy two (simple) smart contracts that implement a token. Call these TokenA and TokenB. You should implement these tokens using the ERC20 template (available on Remix IDE).

- Successfully compile and deploy the tokens on the Remix VM.
- Remix provides multiple addresses (up to 15 accounts) for testing.
- Ensure that you can transfer tokenA between users and that balances are recorded correctly. Ensure the same for tokenB.

4 Task 2 [25+15 points]

Implement a simple automated market maker (AMM) with constant product as a decentralized exchange (DEX) that satisfies the requirements below. The requirements are made clearer by the example in the end.

4.1 Functional Requirements [20 points]

4.1.1 Liquidity Pool & LP Tokens

- LPs deposit/withdraw `tokenA` and `tokenB` while preserving the ratio of the reserves: (i.e. $\frac{x}{y} = \frac{tokenA\_deposited}{tokenB\_deposited}$). IMPORTANT: Note that this step does not preserve the product $x \times y$ of the reserves.
- First depositor sets initial ratio; subsequent deposits mint LP tokens proportionally
- LP tokens (ERC-20) represent share of reserves and are 'burned" on withdrawals.
- LP operations preserve proportional ownership. The reserve ratio of tokenA to tokenB should be possible to read by a function. The return value of this function is the *Spot Price*.

4.1.2 Swapping Mechanism ($x \times y = k$ AMM Model)

- traders can swap tokenA for tokenB and vice versa.
- The contract must follow the constant product formula: $x \times y = k$. (recall that x and y are the reserves of tokens A and B).
- This means that a conversion from tokenA to tokenB preserves the product of reserves of the two tokens before and after the swap.
- A small swap fee of 0.3% per swap must be applied.
- The swap price must be dynamically calculated on the basis of the pool reserves.

4.1.3 Tracking Metrics

- The contract should maintain internal reserves for tokenA and tokenB.
- Functions to read the current value of the reserves of tokenA and tokenB should be implemented.
- Functions to retrieve the current price of tokenA in terms of tokenB and vice versa should be available.

4.1.4 Security and Validations

- Implement sanity checks wherever necessary and ensure safe arithmetic while also minimizing code vulnerabilities.
- Enlist all the measures you took in your report.

4.2 Testing Requirements [15 points]

Run a Javascript simulation (on Remix IDE) in which 5 users are LPs and 8 traders operate as traders generating random transactions. The Javascript code `"ballot_example.js"` can be found on Moodle, which simulates the solidity contract `ballot.sol`; you can use this as a reference.

Generate N (choose some fixed $N \in [50, 100]; N \in \mathbb{Z}$) transactions of swaps, deposits and withdrawals. The performer of each transaction is chosen uniformly randomly from all users. The amount of token deposited/withdrawn in any LP transaction can be determined uniformly randomly based on the maximum tokens available with the LP. The tokens deposited in a swap should be chosen from a uniform random distribution between 0 and min(tokens held by user, 10% of reserves of that token) Study the following quantities and plot vs time:

Table 1: Key Metrics for DEX Simulation

Category	**Metric**	**Description**
Liquidity	Total Value Locked (TVL)	Sum of all token reserves in USD
	Reserve Ratios	TokenA/TokenB ratio in pool
	LP Token Distribution	Holdings across liquidity providers
Trading Activity	Swap Volume	Total tokens swapped by type
	Fee Accumulation	Total fees collected by the DEX
Price Dynamics	Spot Price	Current exchange rate (TokenA/TokenB)
	Slippage	Difference between expected/actual swap amounts

Note: *Slippage* S, refers to the relative difference in (1) the ratio of tokenY received to the tokenX actually deposited (called Actual Price), and (2), the ratio of the reserves of token Y to the reserves of token X (called Expected Price) just before the trade is executed.

$$S := \frac{\frac{\text{token Y received in Swap}}{\text{token X deposited in Swap}} - \frac{\text{token Y reserves before swap}}{\text{token X reserves before swap}}}{\frac{\text{token Y reserves before swap}}{\text{token X reserves before swap}}} \times 100 \ \% \tag{2}$$

4.3 Example:

Let us walk through a simple example.

Suppose that the initial state of the AMM is as given in Table 2. Note that 2 LP Tokens have already been issued.

Now suppose that some LP wants to deposit 100 tokens of A and 150 tokens of B (this preserves the ratio of A and B). Since there is 1 LPT per 100 tokenA's, this LP receives 1 token.

Next, a trader wants to swap 50 token A. Let's suppose the AMM collects 2% fees. So, the AMM takes 1 tokenA for itself, and adds the remaining 49 tokens to the reserves. The tokens she receives in exchange x_B should be such that $(300 + 49)(450 - x_B) = 300 \times 450$. This gives a value $x_B = 63.18$ tokenB, which is transferred to the trader.

Suppose one of the LPs want to cash out against 0.5 LPT, so they deposit 0.5 LPT and take $(0.5 \times 349)/3 = 58.166$ of the tokenA and $58.166 \times 386.82/349 = 64.469B$.

Action	**Reserve A**	**Reserve B**	**LPTs**
Initial State	200	300	2
Add Liquidity	300 (+100)	450 (+150)	3 (+1)
Swap 50A	349 (+49)	386.82 (-63.18)	3
Remove 0.5 LPT	290.83 (-58.17)	322.35 (-64.47)	2.5 (-0.5)

Table 2: AMM state transitions with delta changes shown in parentheses

4.4 Technical Requirements [5 points]

- Implement the following contracts:
  1. `LPToken.sol`: Implement the LPTs to be minted and burned in the DEX
  2. `Token.sol`: TokenA and TokenB should be instances of the deployment of this code.
  3. `DEX.sol`: The core DEX implementation should be in this file.
- Use OpenZeppelin's ERC-20 standard for `LPToken` and use OpenZeppelin's `IERC20` interface for token interactions. (These are already present in your Remix VM at `/browser/.deps/npm/@openzeppelin`)
- The DEX contract should be deployed with two existing ERC-20 token addresses.

5 Task 3 [30 points]

Now that you have a functioning DEX, the next task is to implement a smart contract that performs arbitrage.

5.1 Arbitrage Explanation

Arbitrage is the process of exploiting price differences for the same asset across different markets. In decentralized finance (DeFi), this typically involves: identifying price discrepancies between DEXes, buying the asset at the lower price and selling at the higher price.

The arbitrage opportunity exists when:

$$\frac{Reserve_A^1}{Reserve_B^1} \neq \frac{Reserve_A^2}{Reserve_B^2} \tag{3}$$

where superscripts denote different DEXes.

5.2 Example of arbitrage

The opportunity exists because DEX θ' offers a better exchange rate (2.1 B/A) than DEX θ (2.0 B/A).

The trader profits by buying Token B cheaply on DEX θ' and selling it at a higher price on DEX θ

DEX θ	**DEX** θ'
Token A initial: 1000	Token A initial: 1000
Token B initial: 2000	Token B initial: 2100

Both DEX charge 1% fees.

Swap 10 Token A in DEX θ'...

$$10 \text{ A} \xrightarrow{\text{fees } 1\%} 9.9 \text{ A} \xrightarrow{\theta} \big((1000+9.9)(2100-y) = 21\times 10^5 \implies y = 20.58 \text{ B}\big)$$

$$20.58 \text{ B} \xrightarrow{\text{fees } 1\%} 20.37 \text{ B} \xrightarrow{\theta'} \big((2000+20.37)(1000-x) = 2\times 10^6\big) \implies x = 10.082 \text{ A}$$

$$\textbf{Profit} = 10.082 - 10 = 0.082 \text{ Token A.}$$

5.3 Functional Requirements

- Implement price comparison logic using `spotPrice()` from two DEX contracts
- Calculate arbitrage opportunities in both directions (A→B→A and B→A→B)
- Execute swaps only when profit exceeds a minimum threshold. Your main function, upon being called, should detect the opportunity and execute if profitable. Assume the knowledge of fees charged by each DEX to be fixed at 0.3%
- Return the original capital and the profit to the arbitrageur.

5.4 Testing Requirements

Demonstrate 2 scenarios using your smart contract `arbitrage.sol`, which takes as arguments, addresses of two already deployed DEX contracts.:

- Profitable arbitrage execution
- Failed arbitrage (insufficient profit)

You should write Javascript to demonstrate this on Remix in `simulate_arbitrage.js`. The Javascript code `"ballot_example.js"` can be found on Moodle, which simulates the solidity contract `ballot.sol`; you can use this as a reference.

6 Theory Questions [20 points]

Include the answers to these questions in your report and show how it reflects in your code implementation (if at all it does)

1. Which address(es) should be allowed to mint/burn the LP tokens?
2. In what way do DEXs level the playing ground between a powerful and resourceful trader (HFT/institutional investor) and a lower resource trader (retail investors, like you and me!)?
3. Suppose there are many transaction requests to the DEX sitting in a miner's mempool. How can the miner take undue advantage of this information? Is it possible to make the DEX robust against it?
4. We have left out a very important dimension on the feasibility of this smart contract- the gas fees! Every function call needs gas. How does gas fees influence economic viability of the entire DEX and arbitrage?
5. Could gas fees lead to undue advantages to some transactors over others? How?
6. What are the various ways to minimize slippage (as defined in 4.2) in a swap?
7. Having defined what slippage, plot how slippage varies with the "trade lot fraction" for a constant product AMM?. Trade lot fraction is the ratio of the amount of token X deposited in a swap, to the amount of X in the reserves just before the swap.

7 Submission [5 points]

- Provide Solidity source code (`.sol` files) for `DEX`, `arbitrage`, `Token` and `LPToken`.
- Include the Javascript code `simulate_DEX.js` and `simulate_arbitrage.js` that you used to simulate your code on Remix.
- Describe your implementation and answer the theory questions in your report.
- Add a short README explaining all the contents of each file uploaded, in short. You may also upload any other files you used for simulation/plotting, but duly mention them in the README file. Use of GenAI tools is not allowed.

8 References

- What is a DEX?
- Blog on Constant Product Market Maker
- Remix Documentation
- Arbitrage on DEXs

Good luck, and happy coding!