1. Programing, math, problem solving

Submission:

- Once you have completed your implementation, upload it to the GitHub and then share the repository link

Overview:

- Implement a math model of a Liquidity Pool in the Rust programming language, which provides immediate liquidity unstaking. The model originated from the Marinade Protocol and is described here: https://docs.marinade.finance/marinade-protocol/system-overview/unstake-liquidity-pool
- Your task involves implementing functions based on structures similar to the ones outlined below:

```
struct TokenAmount(u64);
struct StakedTokenAmount(u64);
struct LpTokenAmount(u64);
struct Price(u64);
struct Percentage(u64);

struct LpPool {
    price: Price,
    token_amount: TokenAmount,
    st_token_amount: StakedTokenAmount,
    lp_token_amount: LpTokenAmount,
    liquidity_target: TokenAmount,
    min_fee: Percentage,
    max_fee: Percentage,
}

impl LpPool {
  pub fn init(price: Price, min_fee: Percentage, max_fee: Percentage, liquidity_target: TokenAmount)
    -> Result <Self, Errors> {
  //PROVIDE IMPLEMENTATION
  }

  pub fn add_liquidity(self: &mut Self, token_amount: TokenAmount)
    -> Result <LpTokenAmount, Errors> {
  //PROVIDE IMPLEMENTATION
  }

  pub fn remove_liquidity(self: &mut Self, lp_token_amount: LpTokenAmount)
    -> Result <(TokenAmount, StakedTokenAmount), Errors> {
  //PROVIDE IMPLEMENTATION
  }

  pub fn swap(self: &mut Self, staked_token_amount: StakedTokenAmount)
    -> Result <TokenAmount, Errors> {
  //PROVIDE IMPLEMENTATION
  }
}
```

Details:

- Assume there are 3 types of tokens: Token, StakedToken, LPToken and corresponding token amounts: TokenAmount, StakedTokenAmount, LPTokenAmount. These tokens can be defined as follows:
    - Token: base unit.
    - StakedToken: unit backed by specific amounts of the Token. The exchange ratio is determined by the Price.
    - LpToken: unit represents a share of the liquidity pool. You can mint LpTokens by invoking the “add_liquidity” function, which requires a certain amount of Token. Conversely, you can redeem LpTokens using the “remove_liquidity” function, resulting in proportional amounts of Token and StakedToken being returned.
- The Liquidity Pool functions as a mechanism for exchanging StakedTokens for Tokens. The LpPool serves as a system where actors can exchange with each other. However, the actual swapping doesn't happen directly among these involved actors. Instead, the LpPool acts as a trusted third party responsible for settling the transactions.
- The liquidity pool involves two actors: the Swapper and the Liquidity Provider:
    - The swapper is an actor that uses LpPool to exchange StakedToken for Tokens.
    - A Liquidity Provider is an actor that supplies Tokens to the Liquidity Pool, enabling these exchanges. As compensation for the provided Tokens, the Liquidity Provider receives a share of ownership in the liquidity pool, represented as LpTokens. Importantly, the Liquidity Pool charges a fee after each transaction, with the fee percentage varying based on factors such as available Tokens in the LpPool and target liquidity.
- The LpPool methods are analyzed in terms of their interface: passed arguments, instance state changes, and return type:
    - LpPool::init
        - Params: Configuration parameters such as price, fee_min, fee_max, liquidity target
        - State change: Initializes all LpPool fields
        - Return: instance of LpPool
    - LpPool::add_liquidity
        - Params: Amount of Token that the liquidity provider wants to add to the LpPool
        - State change: Increases the Token reserve and the amount of LpToken
        - Return: New amount of minted LpToken
    - LpPool::remove_liquidity
        - Params: Amount of LpToken that the liquidity provider wants to redeem from the LpPool
        - State change: Decreases Token reserve, decreases StakedToken reserve, and decreases the amount of LpToken
        - Return: Specific amounts of Token and StakedToken. The amount of returned tokens is proportional to the LpToken passed, considering all LpTokens minted by the LpPool
    - LpPool::swap

    - Params: Amount of StakedToken that the Swapper wants to exchange
    - State change: Decreases Token reserve and increases StakedToken reserve in the LpPool
    - Return: Amount of Token received as a result of the exchange. The received token amount depends on the StakedToken passed during invocation and the fee charged by the LpPool.

Requirements:

- Use fixed-point decimals based on the u64 type for all of these parameters, instead of floating points.
- Assume that the price is constant for simplicity.
- Implement a math model in pure Rust; integration with blockchain or UI is not necessary.
- Include unit tests for at least the most important functions.
- Choose any implementation paradigm (such as OOP, functional programming, etc.) based on your preferences.

Story example:

Here is a brief overview of calls made on the LpPool instance, demonstrating operations using example data:

1. LpPool::init(price=1.5, min_fee=0.1%, max_fee9%, liquidity_target=90.0 Token) -> return lp_pool
2. lp_pool.add_liquidity(100.0 Token) -> return 100.0 LpToken
3. lp_pool.swap(6 StakedToken) -> return 8.991 Token
4. lp_pool.add_liquidity(10.0 Token) -> 9.9991 LpToken
5. lp_pool.swap(30.0 StakedToken) -> return 43.44237 Token
6. lp_pool.remove_liquidity(109.9991) -> return (57.56663 Token, 36 StakedToken)

Tips:

- Please take note of how to deal with fixed-point decimals in a smart contract: https://medium.com/asecuritysite-when-bob-met-alice/dealing-with-decimals-in-smart-contracts-fd27eea9209a
- If you're having trouble understanding staked tokens, you can refer to some additional information (though it's not essential for completing the task). Check out the Marinade documentation: https://docs.marinade.finance
- Marinade is an open-source project, and the implementation, including the Liquidity Pool, is available here: https://github.com/marinade-finance/liquid-staking-program
- If you encounter any issues, you can always ask Marinade developers on Discord: https://discord.com/invite/6EtUf4Euu6
- Alternatively, you can contact us via email at contact@invariant.app or on Telegram: @WojciechCichocki

If you have trouble understanding the concept of the task, familiarizing yourself with these terms will help:

- DeFi (Decentralized Finance)
- DEX (Decentralized Exchange)
- Tokens (such as ERC20 or SPL standards)
- LP (Liquidity pool), LP tokens
- Proof of Stake / Staking
- Liquid Staking / LSD (Liquid Staking Derivatives)

2. Smart contract development

Submission:

- Once you have completed your implementation, upload it to the GitHub and then share the repository link

Overview:

- Create a contract using one of the following ecosystems: Solana, Aleph Zero, or Casper. Your task is to implement the Escrow pattern (third-party; middleman).
- Choose one ecosystem from the following options:

1. Solana (Anchor framework, Rust programing language)
2. Aleph zero (Ink! framework, Rust programing language)
3. Casper (Odra framework, Rust programing language)

Details:

- Escrow contracts should maintain separate accounts for each user, allowing efficient handling of multiple users simultaneously
- Escrow contracts allow for the multi-call of entrypoints: deposit and withdraw
- Only the owner of an account should be able to withdraw funds from their respective escrow account.

Requirements:

- Utilize a generic token type (SPL Token, Erc20, PSP22) based on your chosen ecosystem.
- Address potential security concerns, particularly vulnerabilities like reentrancy.
- Include unit tests.
- Bonus points if you add end-to-end tests in TypeScript.
- Bonus points if you implement deposit and withdrawal in a single transaction (swap). The swap may have different levels of complexity, e.g., always 1 to 1, AMM, order book, depending on your skills.

Useful links:

- Solana:
  - Solana docs: https://docs.solana.com/
  - Anchor docs: https://www.anchor-lang.com/
- Aleph zero:
  - Aleph Zero docs: https://docs.alephzero.org/aleph-zero/
  - Ink docs: https://use.ink/
- Casper:
  - Casper docs: https://docs.casper.network/
  - Odra docs: https://odra.dev/docs/

If you're interested in checking out our projects as examples within those ecosystems, below are the links:

- Solana: https://github.com/invariant-labs/protocol
- Aleph zero: https://github.com/invariant-labs/protocol-a0
- Casper: https://github.com/invariant-labs/protocol-cspr

Tips:

- Check out simple escrow example (solidity based): https://www.geeksforgeeks.org/what-is-escrow-smart-contract/
- Try to ask questions on the discord channels if you get stuck.
- Alternatively, you can contact us via email at contact@invariant.app or on Telegram: @WojciechCichocki

3. Frontend development

Task submission:

- To complete the task, you'll be working on adding a new feature to an existing repository located at: https://github.com/invariant-labs/webapp
- To submit the task, create a fork of the above-mentioned repository and create a pull request containing your solution to the original repository

Tech stack:

- React
- Storybook
- Material UI
- Nivo

Task details:

- Your objective is to enhance the user experience by modifying the list view of user positions to include an indicator showing whether a specific market (liquidity pool) has been indexed by the Jupiter aggregator. When a position is open on a pool indexed by Jupiter, it should be indicated by a glowing Jupiter icon. Conversely, if a position is open on a pool not indexed by Jupiter, the icon should remain unlit.
- You can find the design mockup for this task here: https://www.figma.com/file/Jq3r23IXCsxbyAOOsEZuJi/FRONTEND-TASK?type=design&node-id=0-1&mode=design&t=sVpgpVtJobcpdC28-0
  Please note that this mockup includes a feature that has not yet been implemented, namely, the icon indicating whether the position is open on a pair indexed by Jupiter.
- Determining whether a pool is indexed by Jupiter is straightforward as Jupiter exposes an API providing information about all indexed pools. You can access a list of indexed pools via this link: https://cache.jup.ag/markets?v=3
- For instance, within the JSON data, an entry like *{"pubkey":"5dX3tkVDmbHBWMCQMerAHTmd9wsRvmtKLoQt6qv9fHy7",...}* indicates that the pool with the address *"5dX3tkVDmbHBWMCQMerAHTmd9wsRvmtKLoQt6qv9fHy7"* has been indexed. Consequently, all positions on the USDC/USDT 0.01% pool should feature the illuminated Jupiter icon.

Tips:

- The pool address serves as a unique identifier for a pool, designated for a token pair A/B and fee tiers specifying the percentage level of fees for exchanges.
- Market ID, pool address, and pair address essentially refer to the same thing, and these terms are interchangeable.
- For instance, the address for the USDC/USDT pool at a 0.01% fee tier is "*5dX3tkVDmbHBWMCQMerAHTmd9wsRvmtKLoQt6qv9fHy7*". You can find this address for each pair under the label Market ID. You can also find it at this link: https://invariant.app/newPosition/USDC/USDT/0_01

- The presence of the pool address at https://cache.jup.ag/markets?v=3 is sufficient information to determine whether the pool has been indexed.
- To run the app in the repository, simply execute: "*npm i*" followed by "*npm run vite*".
- Alternatively, you can contact us via email at contact@invariant.app or on Telegram: @WojciechCichocki