Jeffrey Scholz Nov 21, 2023 8 min read

The staking algorithm of Sushiswap MasterChef and Synthetix

Updated: Dec 12, 2023

The MasterChef and Synthetix staking algorithms distribute a fixed reward pool among stakers according to their time-weighted contributions to a pool. To save on gas, the algorithms use a cumulative counter of token-level reward and defer the reward distribution.

Imagine we have a fixed pool of 100,000 REWARD tokens we wish to fairly distribute to stakers from block 1 to 100.

Our goal is to distribute 1,000 REWARD every block divided up among the stakers according to their stake.

For example, if on a particular block, the staking balances in the contract were as follows:

	Amount Staked	% of Pool
Alice	100	25
Bob	100	25
Chad	200	50

Then the 1,000 REWARD distributed that block would be as follows

	Amount Staked	% of Pool	Reward Distributed
Alice	100	25	**250**
Bob	100	25	**250**
Chad	200	50	**500**

Terminology for token and reward

The staked token and staking reward may or may not be the same currency. For sake of clarity we will refer to them as token and reward -- sometimes as TOKEN and REWARD.

Sending a transaction every block to distribute a reward is impractical

The naïve solution would be to have an offchain bot send transactions every block to read the stated balances of TOKEN for each of the stakers in the contract and mint them each REWARD according to their percentage of the pool.

However, there is no reliable way to get transactions included in each block. If the bot misses one block, then the users will get less reward than they are expecting.

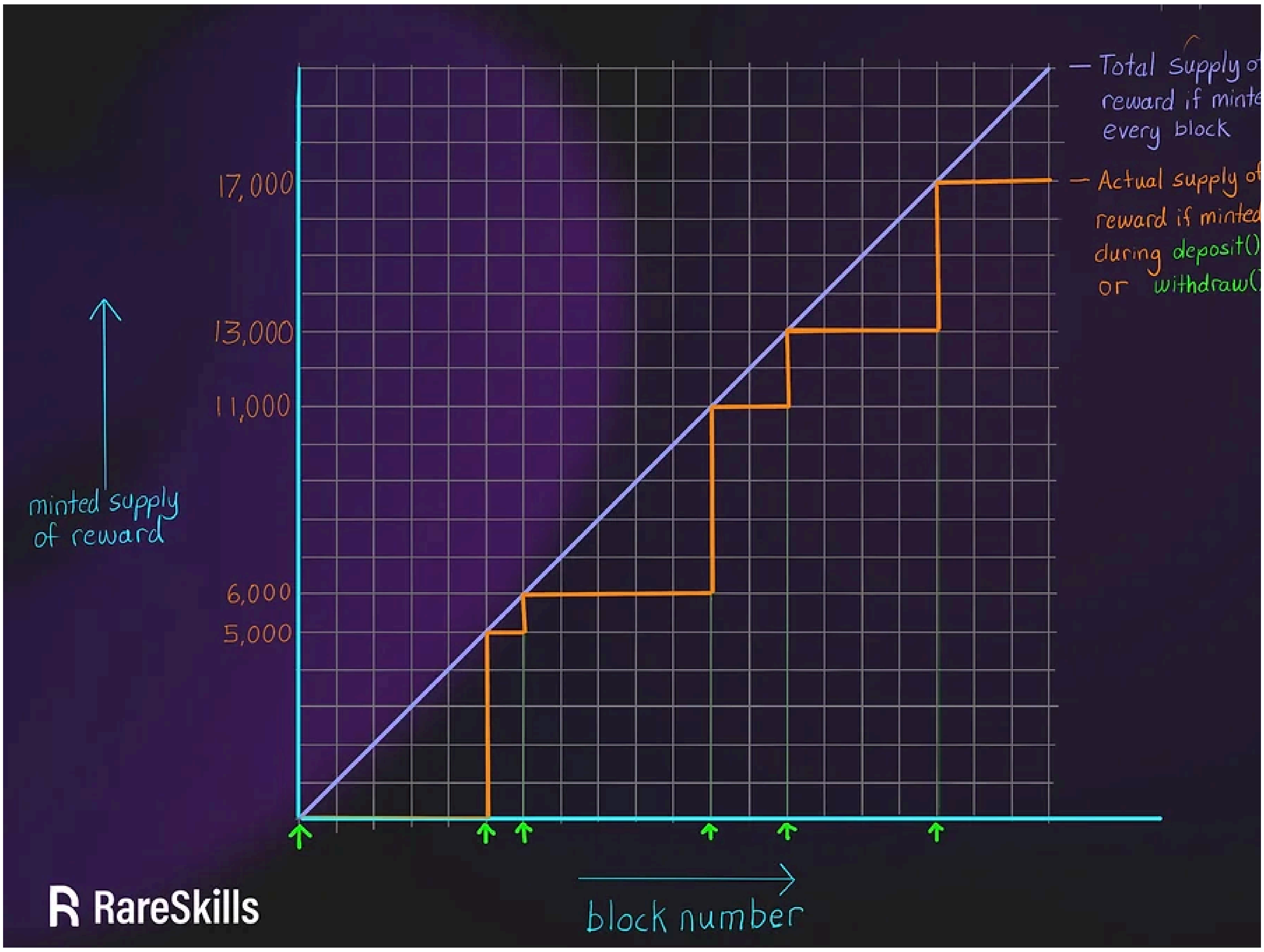
This strategy is also going to incur a lot of transaction fees.

Sending transactions every block is unnecessary, we can issue "catch up" rewards for blocks where rewards weren't distributed

Suppose we kept a variable **lastUpdateBlockNumber** that tracked the last time we issued a reward. We would calculate the number of blocks since the last reward as **block.number - lastUpdateBlockNumber**.

We could then skip some blocks where we distribute the reward and "catch up" when we actually distribute the reward.

The plot below illustrates this.



However, we still don't have a good way to distribute the rewards we just minted to all the stakers according to their stake percentage.

Also, we don't know if the balances of token staked were constant over the previous interval since the last reward distribution. For example, what if Chad knew we were going to measure the balances at block 100 and made a big deposit at block 99 to get a bigger share of the reward?

That problem turns out to be easy to solve.

Key invariant: no transactions, no balance changes

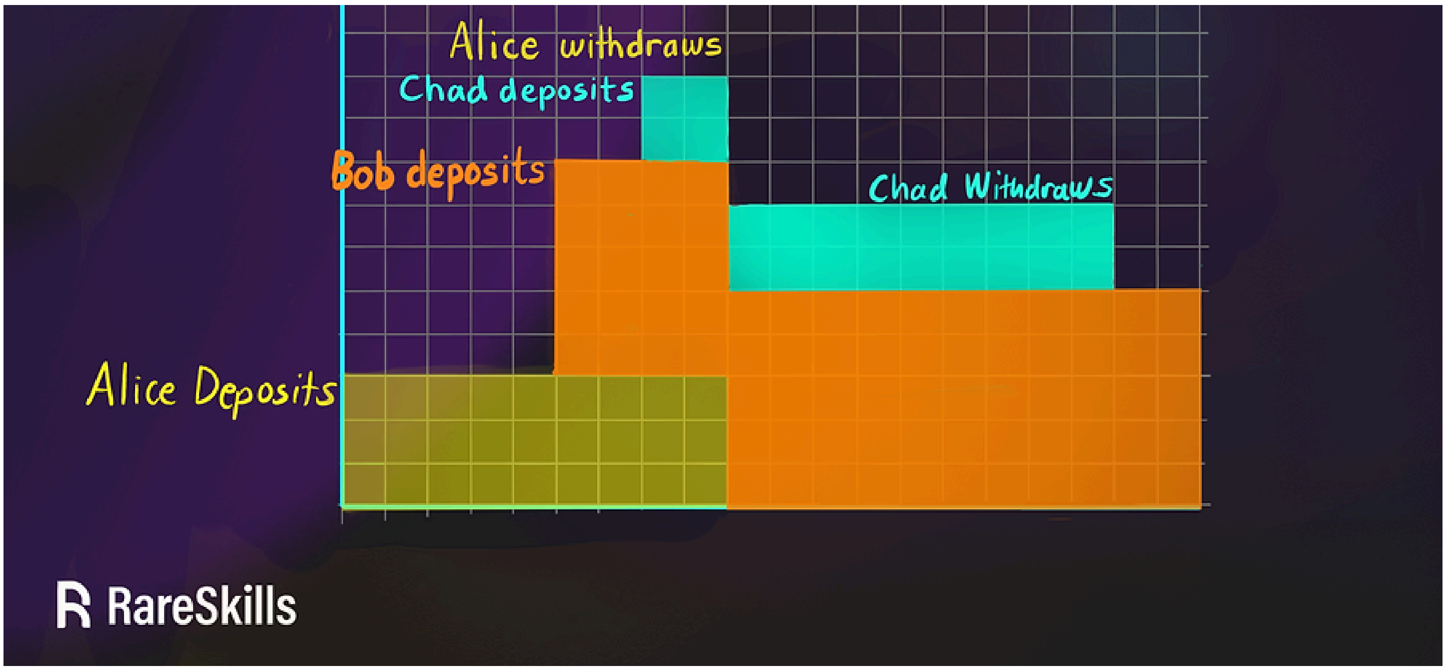
Instead of having a bot fire of transactions every 20 blocks or so, we can just wait for a user to interact with the contract via state changing functions like **deposit()** or **withdraw()**.

Between calls to these functions, we can be sure nobody's balance changed.

For example, if between block 10 and block 15 Alice had 50% of the stake and Bob had 50% of the stake, then we can issue 5,000 REWARD (5 blocks times 1,000) and give each of them 50%.

There's no way for Chad or Bob to "jump in" and increase their balance because when they call **deposit()**, they will trigger a reward distribution. And the reward distribution function is programmed to not include their recent deposit.

Consider the plot below showing the changes in balances over time. The only way for those changes to occur is for a transaction with smart contract to have happened.



However, this solution doesn't scale.

Looping over all the stakers is gas-intensive

Distributing every staker their REWARD whenever someone calls **deposit()** or **withdraw()** will be very gas expensive if there are dozens of stakers. Transferring and ERC 20 token isn't cheap, and doing it dozens of times in a loop is prohibitive.

To do this staking efficiently, people can only get rewards transferred to them if they initiate a state changing transaction. For those who don't claim their rewards, their rewards are deferred. The rewards sit in the contract waiting to be claimed by them.

This will prevent us from having to do a bunch of ERC 20 transfers.
To have an efficient solution:

- we can only update account variables associated with the account initiating the transaction
- we can only update a single global variable that tracks everyone else's increased reward allocation, we cannot explicitly update each account

Rather than tracking reward accruals in accounts, we track the gains of a single staked token

Suppose we can accurately track how much a single staked token has accumulated in reward "since the beginning of time" (when the contract started distributing rewards).

If so, then tracking how much reward an account has accrued is simply multiplying their token balance by how much reward a single token has accrued since the beginning of time.

Suppose we know that a token staked since the beginning of time to the current time has collected 12 reward. If Alice has a stake of 100, then she is due 1,200 rewards.

This is a bit like saying "a dollar saved in our bank has earned $0.40 in interest since we opened the bank. If you opened an account when we opened the bank, and haven't deposited or withdrawn since then, you've earned 40% in interest."

This leads us to two questions:

1. How do we track reward accruals for a single token since the beginning of time.
2. What if Alice has not been staking since the beginning of time, but only deposited recently

How we track reward accruals for a single token since the beginning of time

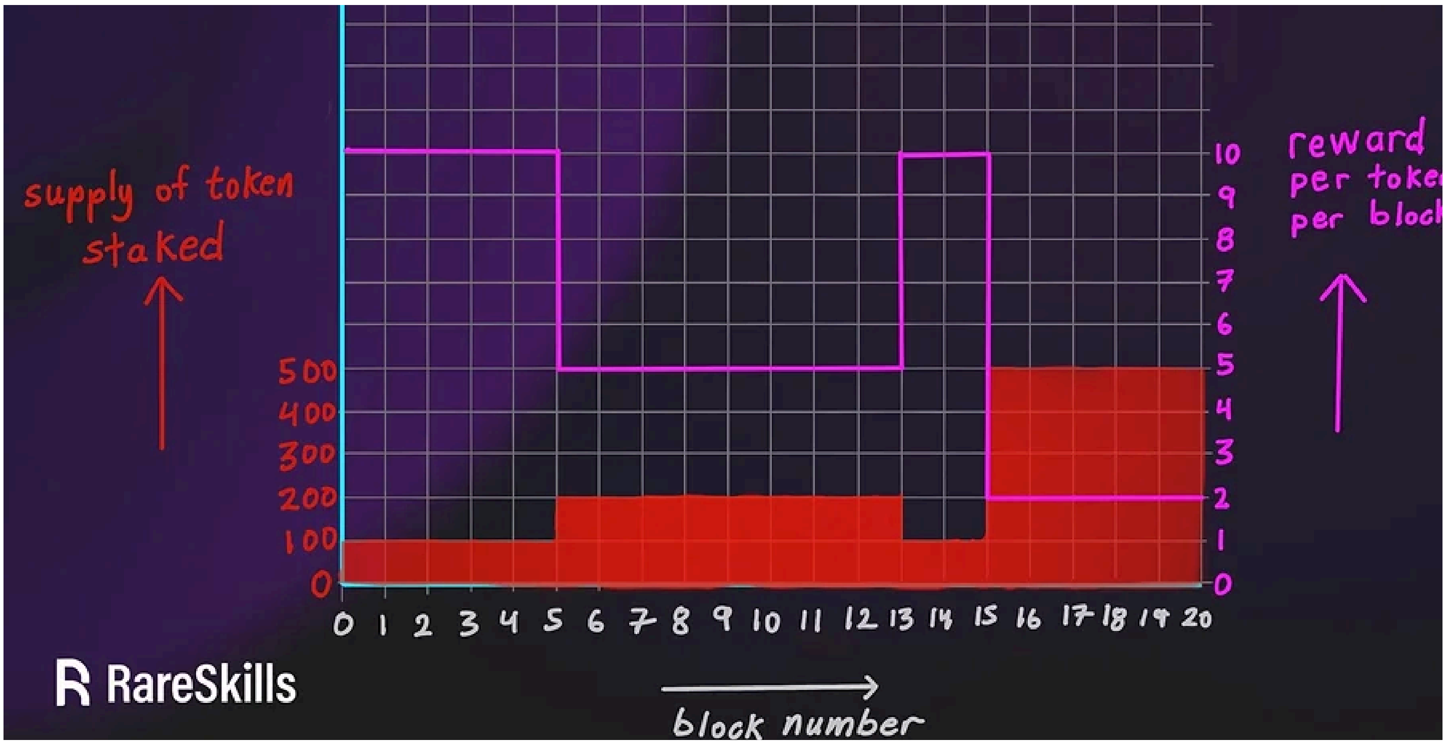
Since a fixed number of reward are issued each block (1,000 in our running example), the more stakers there are, the smaller share of the fixed 1,000 reward they earn. It doesn't matter how many people stake, only the total supply of tokens staked in the contract.

Consider the following hypothetical example.

	rewards issued per block	supply of tokens staked	Reward per token per block
blocks 1-5	1,000	100	10
blocks 6-13	1,000	200	5
blocks 14-15	1,000	100	10
blocks 16-20	1,000	500	2

The more tokens staked, the less reward per token per block. Larger stakes dilute the reward, and a single token earns less as a consequence.

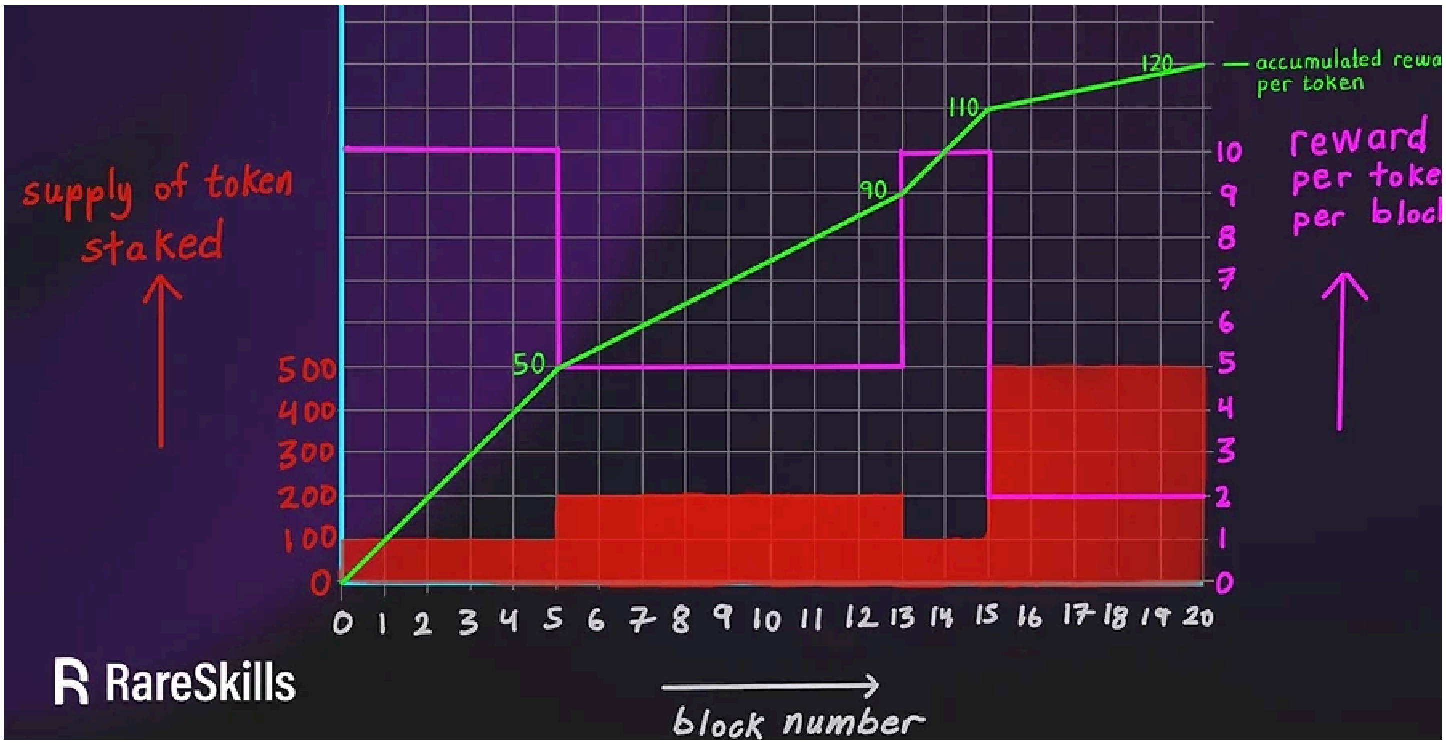
The table is plotted visually below. The red plot is the supply of tokens staked. The purple line is the amount of reward that accrues to a single token in that block. Blocks move to the right on the x axis. The inverse relationship between the two variables should be clear.



Here is the key

Every time we have a state changing transaction, we look back at how many blocks passed, multiply that by the reward per block, then divide that by the total supply staked. This is how much reward a token accrued over that interval. We then add this value to a global accumulator that started at zero at the beginning of time. If we keep repeating this process every time a transaction comes in, we know how much a single token has accumulated in rewards since the beginning of time.

Here is the same plot with the accumulator added.



And here is a table showing the same values.

[illegible]

That is to say, a single token staked from over the course of our plot has accumulated 120 reward.

Test case: rewarding Alice who has been staking since the beginning

Let's consider a very simple example. We are again issuing 1,000 rewards per block. Over the course of 20 blocks, 20,000 reward will be issued.

Here is what Alice and Bob do:

- **Alice has staked 100 tokens from block 1 to block 20.**
- **At block 10, Bob stakes 100 tokens.**
- **At block 20, Alice should be due all 75% of all the rewards issued up to that point, or 15,000 reward.**

Visually, the share of the stake pool per block would look like the following.

From block 1 to block 10, the reward per token per block was 10 (1,000 ÷ 100). Over that interval of 10 blocks, each token accumulated 100 reward (10 reward per token per block x 10 blocks).

But when Bob deposited at block 10, the reward per token per block was diluted down to 5 (1,000 ÷ 200). Over following 10 block interval (blocks 11 to 20), each token accumulated 50 reward.

Therefore, the total value a token had accumulated from block 1 to block 10 was 100, and the value accumulated from 11 to 20 was 50. Therefore, the total value a token accumulated is 100 + 50 = 150.

Since Alice has 100 tokens deposited, and each token has accumulated 150 rewards, she will be issued 15,000 reward, which is indeed 75% of the total rewards issued.

What if someone hasn't been staking from the beginning?

An obvious corner case in the above example (and one we predicted in an earlier section) is that if Bob were to claim reward, he also would get 15,000 reward because his stake at block 20 is 100, like Alice's.

To solve this, we only want the accumulator to start counting for Bob at the moment Bob deposited.

The intuitive solution is to store the block number where Bob deposited and correct it later.

However, it's even simpler to calculate the amount of rewards he ***would*** have been issued at block 10 deposited and then claimed a reward right away. For example, at block 10, the accumulated reward per token was 100. Since Bob deposited 100 tokens, he theoretically could have claimed 10,000 reward right away.

To prevent this from happening, we have a variable for Bob we call the "reward debt." The moment he deposits, we set the reward debt to be the deposited balance times the reward per token accumulator. That will prevent him from claiming a reward right away since at that moment, the rewards due to him would be zero (current rewards minus reward debt).

We have a separate variable for Bob call "reward debt" or "rewards already issued" and assign it to that hypothetical reward amount. At block 10, the accumulated reward per token was 100, and Bob's

deposit was 100, so his reward debt is 10,000.
If Bob claims rewards at block 20, we subtract the 15,000 reward by the reward debt of 10,000. Bob will only be able to claim 5,000 reward at block 20.

Pseudocode for MasterChef

Below we present a stripped down version of the MasterChef algorithm. We have taken some liberty to change the variable names from the original contract for the sake of clarity. We also omit events and implementation details around scaling token decimals.

Differences between Synthetix and MasterChef

Synthetix and MasterChef both use the same mechanism to accumulate the reward per token based on amount staked. The major difference is that rather than tracking reward debt, Synthetix stores a snapshot of the reward accumulator when the user last interacted with the contract. The difference between the current reward accumulator and the snapshot is used to calculate the rewards to the user's account.

That difference is added to a per-user rewards mapping and accumulates there until the user calls getRewards(). This extra bookkeeping makes the Synthetix algorithm less efficient.

The rest of the differences are fairly minor

- MasterChef has deposit() and withdraw().
    - Synthetix has stake(), withdraw(), and getReward().
- MasterChef uses blocks as a unit of time.
    - Synthetix uses the timestamp.
- MasterChef mints rewards to itself as described in the above sections.
    - Synthetix assumes the admin has already transferred the rewards to the contract and does not mint rewards.
- MasterChef distributes rewards from a configurable startBlock to lastRewardBlock.
    - Synthetix is hardcoded to distribute rewards over the course of a week after the admin starts the clock. Synthetix will not necessarily distribute the entire balance of rewards in the contract, but an amount specified by the admin.
- MasterChef transfers the reward to the user whenever they call deposit() or withdraw() with non-zero amounts.
    - Synthetix accumulates the reward due to the user in a mapping called rewards but does not transfer it to the user until they explicitly call getRewards().
- MasterChef supports multiple pools within the same contract, and divides up the rewards by the weight of the pool.
    - Synthetix only has one pool

The interested reader can consult the code for SushiSwap MasterChef Staking.

Pseudocode for Synthetix

The graphic below shows the bookkeeping subroutine of Synthetix that is called during deposit(), withdraw(), or getRewards(). Specifically, it is done before the balance updates in deposit or withdraw or the reward distribution.

In the graphic below lastUpdateTime is the last time any user called one of the three functions. In the example below, the user claiming rewards is not the same one who previously interacted with the contract. The prime ' marker means the value of the variable after the subroutine completes

The interested reader can consult the Synthetix staking code themselves.

Learn more with RareSkills

Please see our blockchain bootcamp to learn more advanced technical web3 subjects