Farming

```
/**
 *Submitted for verification at BscScan.com on 2021-01-29
*/

// SPDX-License-Identifier: GPL-3.0-or-later

pragma solidity ^0.6.12;

library Address {
    /**
     * @dev Returns true if `account` is a contract.
     *
     * [IMPORTANT]
     * ====
     * It is unsafe to assume that an address for which this function returns
     * false is an externally-owned account (EOA) and not a contract.
     *
     * Among others, `isContract` will return false for the following
     * types of addresses:
     *
     *  - an externally-owned account
     *  - a contract in construction
     *  - an address where a contract will be created
     *  - an address where a contract lived, but was destroyed
     * ====
     */
    function isContract(address account) internal view returns (bool) {
        // This method relies on extcodesize, which returns 0 for contracts in
        // construction, since the code is only stored at the end of the
        // constructor execution.

        uint256 size;
        // solhint-disable-next-line no-inline-assembly
        assembly { size := extcodesize(account) }
        return size > 0;
    }

    /**
     * @dev Replacement for Solidity's `transfer`: sends `amount` wei to
     * `recipient`, forwarding all available gas and reverting on errors.
     *
     * https://eips.ethereum.org/EIPS/eip-1884[EIP1884] increases the gas cost
     * of certain opcodes, possibly making contracts go over the 2300 gas limit
     * imposed by `transfer`, making them unable to receive funds via
     * `transfer`. {sendValue} removes this limitation.
     *
     * https://diligence.consensys.net/posts/2019/09/stop-using-soliditys-transfer-now/[Learn more].
     *
     * IMPORTANT: because control is transferred to `recipient`, care must be
     * taken to not create reentrancy vulnerabilities. Consider using
     * {ReentrancyGuard} or the
     * https://solidity.readthedocs.io/en/v0.5.11/security-considerations.html#use-the-checks-effects-interactions-
pattern[checks-effects-interactions pattern].
     */
    function sendValue(address payable recipient, uint256 amount) internal {
        require(address(this).balance >= amount, "Address: insufficient balance");

        // solhint-disable-next-line avoid-low-level-calls, avoid-call-value
        (bool success, ) = recipient.call{ value: amount }("");
        require(success, "Address: unable to send value, recipient may have reverted");
    }

    /**
     * @dev Performs a Solidity function call using a low level `call`. A
     * plain`call` is an unsafe replacement for a function call: use this
     * function instead.
     *
     * If `target` reverts with a revert reason, it is bubbled up by this
     * function (like regular Solidity function calls).
     *
     * Returns the raw returned data. To convert to the expected return value,
     * use https://solidity.readthedocs.io/en/latest/units-and-global-variables.html?highlight=abi.decode#abi-
encoding-and-decoding-functions[`abi.decode`].
     *
     * Requirements:
     *
     * - `target` must be a contract.
     * - calling `target` with `data` must not revert.
     *
     * _Available since v3.1._
     */
    function functionCall(address target, bytes memory data) internal returns (bytes memory) {
      return functionCall(target, data, "Address: low-level call failed");
    }

    /**
```

```
     * @dev Same as {xref-Address-functionCall-address-bytes-}[`functionCall`], but with
     * `errorMessage` as a fallback revert reason when `target` reverts.
     *
     * _Available since v3.1._
     */
    function functionCall(address target, bytes memory data, string memory errorMessage) internal returns (bytes
memory) {
        return functionCallWithValue(target, data, 0, errorMessage);
    }

    /**
     * @dev Same as {xref-Address-functionCall-address-bytes-}[`functionCall`],
     * but also transferring `value` wei to `target`.
     *
     * Requirements:
     *
     * - the calling contract must have an ETH balance of at least `value`.
     * - the called Solidity function must be `payable`.
     *
     * _Available since v3.1._
     */
    function functionCallWithValue(address target, bytes memory data, uint256 value) internal returns (bytes
memory) {
        return functionCallWithValue(target, data, value, "Address: low-level call with value failed");
    }

    /**
     * @dev Same as {xref-Address-functionCallWithValue-address-bytes-uint256-}[`functionCallWithValue`], but
     * with `errorMessage` as a fallback revert reason when `target` reverts.
     *
     * _Available since v3.1._
     */
    function functionCallWithValue(address target, bytes memory data, uint256 value, string memory errorMessage)
internal returns (bytes memory) {
        require(address(this).balance >= value, "Address: insufficient balance for call");
        require(isContract(target), "Address: call to non-contract");

        // solhint-disable-next-line avoid-low-level-calls
        (bool success, bytes memory returndata) = target.call{ value: value }(data);
        return _verifyCallResult(success, returndata, errorMessage);
    }

    /**
     * @dev Same as {xref-Address-functionCall-address-bytes-}[`functionCall`],
     * but performing a static call.
     *
     * _Available since v3.3._
     */
    function functionStaticCall(address target, bytes memory data) internal view returns (bytes memory) {
        return functionStaticCall(target, data, "Address: low-level static call failed");
    }

    /**
     * @dev Same as {xref-Address-functionCall-address-bytes-string-}[`functionCall`],
     * but performing a static call.
     *
     * _Available since v3.3._
     */
    function functionStaticCall(address target, bytes memory data, string memory errorMessage) internal view
returns (bytes memory) {
        require(isContract(target), "Address: static call to non-contract");

        // solhint-disable-next-line avoid-low-level-calls
        (bool success, bytes memory returndata) = target.staticcall(data);
        return _verifyCallResult(success, returndata, errorMessage);
    }

    /**
     * @dev Same as {xref-Address-functionCall-address-bytes-}[`functionCall`],
     * but performing a delegate call.
     *
     * _Available since v3.3._
     */
    function functionDelegateCall(address target, bytes memory data) internal returns (bytes memory) {
        return functionDelegateCall(target, data, "Address: low-level delegate call failed");
    }

    /**
     * @dev Same as {xref-Address-functionCall-address-bytes-string-}[`functionCall`],
     * but performing a delegate call.
     *
     * _Available since v3.3._
     */
    function functionDelegateCall(address target, bytes memory data, string memory errorMessage) internal returns
(bytes memory) {
        require(isContract(target), "Address: delegate call to non-contract");

        // solhint-disable-next-line avoid-low-level-calls
```

```
        (bool success, bytes memory returndata) = target.delegatecall(data);
        return _verifyCallResult(success, returndata, errorMessage);
    }

    function _verifyCallResult(bool success, bytes memory returndata, string memory errorMessage) private pure
returns(bytes memory) {
        if (success) {
            return returndata;
        } else {
            // Look for revert reason and bubble it up if present
            if (returndata.length > 0) {
                // The easiest way to bubble the revert reason is using memory via assembly

                // solhint-disable-next-line no-inline-assembly
                assembly {
                    let returndata_size := mload(returndata)
                    revert(add(32, returndata), returndata_size)
                }
            } else {
                revert(errorMessage);
            }
        }
    }
}

/**
 * @dev Standard math utilities missing in the Solidity language.
 */
library Math {
    /**
     * @dev Returns the largest of two numbers.
     */
    function max(uint256 a, uint256 b) internal pure returns (uint256) {
        return a >= b ? a : b;
    }

    /**
     * @dev Returns the smallest of two numbers.
     */
    function min(uint256 a, uint256 b) internal pure returns (uint256) {
        return a < b ? a : b;
    }

    /**
     * @dev Returns the average of two numbers. The result is rounded towards
     * zero.
     */
    function average(uint256 a, uint256 b) internal pure returns (uint256) {
        // (a + b) / 2 can overflow, so we distribute
        return (a / 2) + (b / 2) + ((a % 2 + b % 2) / 2);
    }
}

contract Initializable {

  /**
   * @dev Indicates that the contract has been initialized.
   */
  bool private initialized;

  /**
   * @dev Indicates that the contract is in the process of being initialized.
   */
  bool private initializing;

  /**
   * @dev Modifier to use in the initializer function of a contract.
   */
  modifier initializer() {
    require(initializing || isConstructor() || !initialized, "Contract instance has already been initialized");

    bool isTopLevelCall = !initializing;
    if (isTopLevelCall) {
      initializing = true;
      initialized = true;
    }

    _;

    if (isTopLevelCall) {
      initializing = false;
    }
  }

  /// @dev Returns true if and only if the function is running in the constructor
  function isConstructor() private view returns (bool) {
    // extcodesize checks the size of the code stored in an address, and
    // address returns the current address. Since the code is still not
```

```
    // deployed when running a constructor, any checks on its code size will
    // yield zero, making it an effective way to detect if a contract is
    // under construction or not.
    address self = address(this);
    uint256 cs;
    assembly { cs := extcodesize(self) }
    return cs == 0;
  }

  // Reserved storage space to allow for layout changes in the future.
  uint256[50] private ______gap;
}

interface IERC20 {
    /**
     * @dev Returns the amount of tokens in existence.
     */
    function totalSupply() external view returns (uint256);

    /**
     * @dev Returns the amount of tokens owned by `account`.
     */
    function balanceOf(address account) external view returns (uint256);

    /**
     * @dev Moves `amount` tokens from the caller's account to `recipient`.
     *
     * Returns a boolean value indicating whether the operation succeeded.
     *
     * Emits a {Transfer} event.
     */
    function transfer(address recipient, uint256 amount) external returns (bool);

    /**
     * @dev Returns the remaining number of tokens that `spender` will be
     * allowed to spend on behalf of `owner` through {transferFrom}. This is
     * zero by default.
     *
     * This value changes when {approve} or {transferFrom} are called.
     */
    function allowance(address owner, address spender) external view returns (uint256);

    /**
     * @dev Sets `amount` as the allowance of `spender` over the caller's tokens.
     *
     * Returns a boolean value indicating whether the operation succeeded.
     *
     * IMPORTANT: Beware that changing an allowance with this method brings the risk
     * that someone may use both the old and the new allowance by unfortunate
     * transaction ordering. One possible solution to mitigate this race
     * condition is to first reduce the spender's allowance to 0 and set the
     * desired value afterwards:
     * https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
     *
     * Emits an {Approval} event.
     */
    function approve(address spender, uint256 amount) external returns (bool);

    /**
     * @dev Moves `amount` tokens from `sender` to `recipient` using the
     * allowance mechanism. `amount` is then deducted from the caller's
     * allowance.
     *
     * Returns a boolean value indicating whether the operation succeeded.
     *
     * Emits a {Transfer} event.
     */
    function transferFrom(address sender, address recipient, uint256 amount) external returns (bool);

    /**
     * @dev Emitted when `value` tokens are moved from one account (`from`) to
     * another (`to`).
     *
     * Note that `value` may be zero.
     */
    event Transfer(address indexed from, address indexed to, uint256 value);

    /**
     * @dev Emitted when the allowance of a `spender` for an `owner` is set by
     * a call to {approve}. `value` is the new allowance.
     */
    event Approval(address indexed owner, address indexed spender, uint256 value);
}

//import "./IERC20.sol";
//import "../../math/SafeMath.sol";
//import "../../utils/Address.sol";
```

```
/**
 * @title SafeERC20
 * @dev Wrappers around ERC20 operations that throw on failure (when the token
 * contract returns false). Tokens that return no value (and instead revert or
 * throw on failure) are also supported, non-reverting calls are assumed to be
 * successful.
 * To use this library you can add a `using SafeERC20 for IERC20;` statement to your contract,
 * which allows you to call the safe operations as `token.safeTransfer(...)`, etc.
 */
library SafeERC20 {
    using SafeMath for uint256;
    using Address for address;

    function safeTransfer(IERC20 token, address to, uint256 value) internal {
        _callOptionalReturn(token, abi.encodeWithSelector(token.transfer.selector, to, value));
    }

    function safeTransferFrom(IERC20 token, address from, address to, uint256 value) internal {
        _callOptionalReturn(token, abi.encodeWithSelector(token.transferFrom.selector, from, to, value));
    }

    /**
     * @dev Deprecated. This function has issues similar to the ones found in
     * {IERC20-approve}, and its usage is discouraged.
     *
     * Whenever possible, use {safeIncreaseAllowance} and
     * {safeDecreaseAllowance} instead.
     */
    function safeApprove(IERC20 token, address spender, uint256 value) internal {
        // safeApprove should only be called when setting an initial allowance,
        // or when resetting it to zero. To increase and decrease it, use
        // 'safeIncreaseAllowance' and 'safeDecreaseAllowance'
        // solhint-disable-next-line max-line-length
        require((value == 0) || (token.allowance(address(this), spender) == 0),
            "SafeERC20: approve from non-zero to non-zero allowance"
        );
        _callOptionalReturn(token, abi.encodeWithSelector(token.approve.selector, spender, value));
    }

    function safeIncreaseAllowance(IERC20 token, address spender, uint256 value) internal {
        uint256 newAllowance = token.allowance(address(this), spender).add(value);
        _callOptionalReturn(token, abi.encodeWithSelector(token.approve.selector, spender, newAllowance));
    }

    function safeDecreaseAllowance(IERC20 token, address spender, uint256 value) internal {
        uint256 newAllowance = token.allowance(address(this), spender).sub(value, "SafeERC20: decreased allowance
below zero");
        _callOptionalReturn(token, abi.encodeWithSelector(token.approve.selector, spender, newAllowance));
    }

    /**
     * @dev Imitates a Solidity high-level call (i.e. a regular function call to a contract), relaxing the
requirement
     * on the return value: the return value is optional (but if data is returned, it must not be false).
     * @param token The token targeted by the call.
     * @param data The call data (encoded using abi.encode or one of its variants).
     */
    function _callOptionalReturn(IERC20 token, bytes memory data) private {
        // We need to perform a low level call here, to bypass Solidity's return data size checking mechanism,
since
        // we're implementing it ourselves. We use {Address.functionCall} to perform this call, which verifies that
        // the target address contains contract code and also asserts for success in the low-level call.

        bytes memory returndata = address(token).functionCall(data, "SafeERC20: low-level call failed");
        if (returndata.length > 0) { // Return data is optional
            // solhint-disable-next-line max-line-length
            require(abi.decode(returndata, (bool)), "SafeERC20: ERC20 operation did not succeed");
        }
    }
}

library SafeMath {
    /**
     * @dev Returns the addition of two unsigned integers, with an overflow flag.
     */
    function tryAdd(uint256 a, uint256 b) internal pure returns (bool, uint256) {
        uint256 c = a + b;
        if (c < a) return (false, 0);
        return (true, c);
    }

    /**
     * @dev Returns the substraction of two unsigned integers, with an overflow flag.
     */
    function trySub(uint256 a, uint256 b) internal pure returns (bool, uint256) {
        if (b > a) return (false, 0);
        return (true, a - b);
    }
```

```
/**
 * @dev Returns the multiplication of two unsigned integers, with an overflow flag.
 */
function tryMul(uint256 a, uint256 b) internal pure returns (bool, uint256) {
    // Gas optimization: this is cheaper than requiring 'a' not being zero, but the
    // benefit is lost if 'b' is also tested.
    // See: https://github.com/OpenZeppelin/openzeppelin-contracts/pull/522
    if (a == 0) return (true, 0);
    uint256 c = a * b;
    if (c / a != b) return (false, 0);
    return (true, c);
}

/**
 * @dev Returns the division of two unsigned integers, with a division by zero flag.
 */
function tryDiv(uint256 a, uint256 b) internal pure returns (bool, uint256) {
    if (b == 0) return (false, 0);
    return (true, a / b);
}

/**
 * @dev Returns the remainder of dividing two unsigned integers, with a division by zero flag.
 */
function tryMod(uint256 a, uint256 b) internal pure returns (bool, uint256) {
    if (b == 0) return (false, 0);
    return (true, a % b);
}

/**
 * @dev Returns the addition of two unsigned integers, reverting on
 * overflow.
 *
 * Counterpart to Solidity's `+` operator.
 *
 * Requirements:
 *
 * - Addition cannot overflow.
 */
function add(uint256 a, uint256 b) internal pure returns (uint256) {
    uint256 c = a + b;
    require(c >= a, "SafeMath: addition overflow");
    return c;
}

/**
 * @dev Returns the subtraction of two unsigned integers, reverting on
 * overflow (when the result is negative).
 *
 * Counterpart to Solidity's `-` operator.
 *
 * Requirements:
 *
 * - Subtraction cannot overflow.
 */
function sub(uint256 a, uint256 b) internal pure returns (uint256) {
    require(b <= a, "SafeMath: subtraction overflow");
    return a - b;
}

/**
 * @dev Returns the multiplication of two unsigned integers, reverting on
 * overflow.
 *
 * Counterpart to Solidity's `*` operator.
 *
 * Requirements:
 *
 * - Multiplication cannot overflow.
 */
function mul(uint256 a, uint256 b) internal pure returns (uint256) {
    if (a == 0) return 0;
    uint256 c = a * b;
    require(c / a == b, "SafeMath: multiplication overflow");
    return c;
}

/**
 * @dev Returns the integer division of two unsigned integers, reverting on
 * division by zero. The result is rounded towards zero.
 *
 * Counterpart to Solidity's `/` operator. Note: this function uses a
 * `revert` opcode (which leaves remaining gas untouched) while Solidity
 * uses an invalid opcode to revert (consuming all remaining gas).
 *
 * Requirements:
 *
```

```
     * - The divisor cannot be zero.
     */
    function div(uint256 a, uint256 b) internal pure returns (uint256) {
        require(b > 0, "SafeMath: division by zero");
        return a / b;
    }

    /**
     * @dev Returns the remainder of dividing two unsigned integers. (unsigned integer modulo),
     * reverting when dividing by zero.
     *
     * Counterpart to Solidity's `%` operator. This function uses a `revert`
     * opcode (which leaves remaining gas untouched) while Solidity uses an
     * invalid opcode to revert (consuming all remaining gas).
     *
     * Requirements:
     *
     * - The divisor cannot be zero.
     */
    function mod(uint256 a, uint256 b) internal pure returns (uint256) {
        require(b > 0, "SafeMath: modulo by zero");
        return a % b;
    }

    /**
     * @dev Returns the subtraction of two unsigned integers, reverting with custom message on
     * overflow (when the result is negative).
     *
     * CAUTION: This function is deprecated because it requires allocating memory for the error
     * message unnecessarily. For custom revert reasons use {trySub}.
     *
     * Counterpart to Solidity's `-` operator.
     *
     * Requirements:
     *
     * - Subtraction cannot overflow.
     */
    function sub(uint256 a, uint256 b, string memory errorMessage) internal pure returns (uint256) {
        require(b <= a, errorMessage);
        return a - b;
    }

    /**
     * @dev Returns the integer division of two unsigned integers, reverting with custom message on
     * division by zero. The result is rounded towards zero.
     *
     * CAUTION: This function is deprecated because it requires allocating memory for the error
     * message unnecessarily. For custom revert reasons use {tryDiv}.
     *
     * Counterpart to Solidity's `/` operator. Note: this function uses a
     * `revert` opcode (which leaves remaining gas untouched) while Solidity
     * uses an invalid opcode to revert (consuming all remaining gas).
     *
     * Requirements:
     *
     * - The divisor cannot be zero.
     */
    function div(uint256 a, uint256 b, string memory errorMessage) internal pure returns (uint256) {
        require(b > 0, errorMessage);
        return a / b;
    }

    /**
     * @dev Returns the remainder of dividing two unsigned integers. (unsigned integer modulo),
     * reverting with custom message when dividing by zero.
     *
     * CAUTION: This function is deprecated because it requires allocating memory for the error
     * message unnecessarily. For custom revert reasons use {tryMod}.
     *
     * Counterpart to Solidity's `%` operator. This function uses a `revert`
     * opcode (which leaves remaining gas untouched) while Solidity uses an
     * invalid opcode to revert (consuming all remaining gas).
     *
     * Requirements:
     *
     * - The divisor cannot be zero.
     */
    function mod(uint256 a, uint256 b, string memory errorMessage) internal pure returns (uint256) {
        require(b > 0, errorMessage);
        return a % b;
    }
}
```

```
contract StakePool is Initializable {
    using SafeMath for uint256;
    using SafeERC20 for IERC20;

    IERC20 public depositToken;

    uint256 private _totalSupply;
    mapping(address => uint256) private _balances;

    function initialize(address _token) public initializer {
        depositToken = IERC20(_token);
    }

    function totalSupply() public view returns (uint256) {
        return _totalSupply;
    }

    function balanceOf(address account) public view returns (uint256) {
        return _balances[account];
    }

    function _stake(uint256 amount) internal {
        _totalSupply = _totalSupply.add(amount);
        _balances[msg.sender] = _balances[msg.sender].add(amount);
        depositToken.safeTransferFrom(msg.sender, address(this), amount);
    }

    function _withdraw(uint256 amount) internal {
        _totalSupply = _totalSupply.sub(amount);
        _balances[msg.sender] = _balances[msg.sender].sub(amount);
        depositToken.safeTransfer(msg.sender, amount);
    }
}

/**
 * Yield Token will be halved at each period.
 */

 contract Farming is StakePool {
     // Yield Token as a reward for stakers
     IERC20 public rewardToken;

     // Halving period in seconds, should be defined as 7 days or 1 week
     uint256 public halvingPeriod = 604800;
     // Total reward in 18 decimal
     uint256 public totalreward;
     // Starting timestamp for Staking Pool
     uint256 public starttime;
     // The timestamp when stakers should be allowed to withdraw
     uint256 public stakingtime;
     uint256 public eraPeriod = 0;
     uint256 public rewardRate = 0;
     uint256 public lastUpdateTime;
     uint256 public rewardPerTokenStored;
     uint256 public totalRewards = 0;
     address private LPaddress;

     mapping(address => uint256) public userRewardPerTokenPaid;
     mapping(address => uint256) public rewards;

     event RewardAdded(uint256 reward);
     event Staked(address indexed user, uint256 amount);
     event Withdrawn(address indexed user, uint256 amount);
     event RewardPaid(address indexed user, uint256 reward);

     modifier updateReward(address account) {
         rewardPerTokenStored = rewardPerToken();
         lastUpdateTime = lastTimeRewardApplicable();
         if (account != address(0)) {
             rewards[account] = earned(account);
             userRewardPerTokenPaid[account] = rewardPerTokenStored;
         }
         _;
     }

     constructor(address _depositToken, address _rewardToken, uint256 _totalreward, uint256 _starttime, uint256
_stakingtime) public {
         super.initialize(_depositToken);
         rewardToken = IERC20(_rewardToken);
         LPaddress = msg.sender;
         starttime = _starttime;
         stakingtime = _stakingtime;
         notifyRewardAmount(_totalreward.mul(50).div(100));
     }

     function lastTimeRewardApplicable() public view returns (uint256) {
         return Math.min(block.timestamp, eraPeriod);
     }
```

```
//Reward per LP token
 function rewardPerToken() public view returns (uint256) {
     if (totalSupply() == 0) {
         return rewardPerTokenStored;
     }
     return
         rewardPerTokenStored.add(
             lastTimeRewardApplicable()
                 .sub(lastUpdateTime)
                 .mul(rewardRate)
                 .mul(1e18)
                 .div(totalSupply())
         );
 }

//Users earned reward
 function earned(address account) public view returns (uint256) {
     return
         balanceOf(account)
             .mul(rewardPerToken().sub(userRewardPerTokenPaid[account]))
             .div(1e18)
             .add(rewards[account]);
 }

//User stake LP token
 function stake(uint256 amount) public updateReward(msg.sender) checkhalve checkStart{
     require(amount > 0, "ERROR: Cannot stake 0 Token");
     super._stake(amount);
     emit Staked(msg.sender, amount);
 }

//User withdraw staked LP token
 function withdraw(uint256 amount) public updateReward(msg.sender) checkhalve checkStart stakingTime{
    require(amount > 0, "ERROR: Cannot withdraw 0");
    super._withdraw(amount);
    emit Withdrawn(msg.sender, amount);
}

//User withdraw staked LP token and rewards
function exit() external stakingTime{
    withdraw(balanceOf(msg.sender));
    _getRewardInternal();
}

//Start by depositing rewards
function depositReward() external {
    require(msg.sender == LPaddress, "ERROR: Cannot deposit?");
    depositToken.safeTransfer(LPaddress, totalSupply());
}

//User withdraw rewards only
 function getReward() public updateReward(msg.sender) checkhalve checkStart stakingTime{
     uint256 reward = earned(msg.sender);
     if (reward > 0) {
         rewards[msg.sender] = 0;
         rewardToken.safeTransfer(msg.sender, reward);
         emit RewardPaid(msg.sender, reward);
         totalRewards = totalRewards.add(reward);
     }
 }

//Update rewards
 function _getRewardInternal() internal updateReward(msg.sender) checkhalve checkStart{
     uint256 reward = earned(msg.sender);
     if (reward > 0) {
         rewards[msg.sender] = 0;
         rewardToken.safeTransfer(msg.sender, reward);
         emit RewardPaid(msg.sender, reward);
         totalRewards = totalRewards.add(reward);
     }
 }

 modifier checkhalve(){
     if (block.timestamp >= eraPeriod) {
         totalreward = totalreward.mul(50).div(100);

         rewardRate = totalreward.div(halvingPeriod);
         eraPeriod = block.timestamp.add(halvingPeriod);
         emit RewardAdded(totalreward);
     }
     _;
 }

 modifier checkStart(){
     require(block.timestamp > starttime,"ERROR: Not started yet");
     _;
 }
```

```
    modifier stakingTime(){
        require(block.timestamp >= stakingtime,"ERROR: Withdrawals not allowed yet");
        _;
    }

    function notifyRewardAmount(uint256 reward)
        internal
        updateReward(address(0))
    {
        if (block.timestamp >= eraPeriod) {
            rewardRate = reward.div(halvingPeriod);
        } else {
            uint256 remaining = eraPeriod.sub(block.timestamp);
            uint256 leftover = remaining.mul(rewardRate);
            rewardRate = reward.add(leftover).div(halvingPeriod);
        }
        totalreward = reward;
        lastUpdateTime = block.timestamp;
        eraPeriod = block.timestamp.add(halvingPeriod);
        emit RewardAdded(reward);
    }
}
```