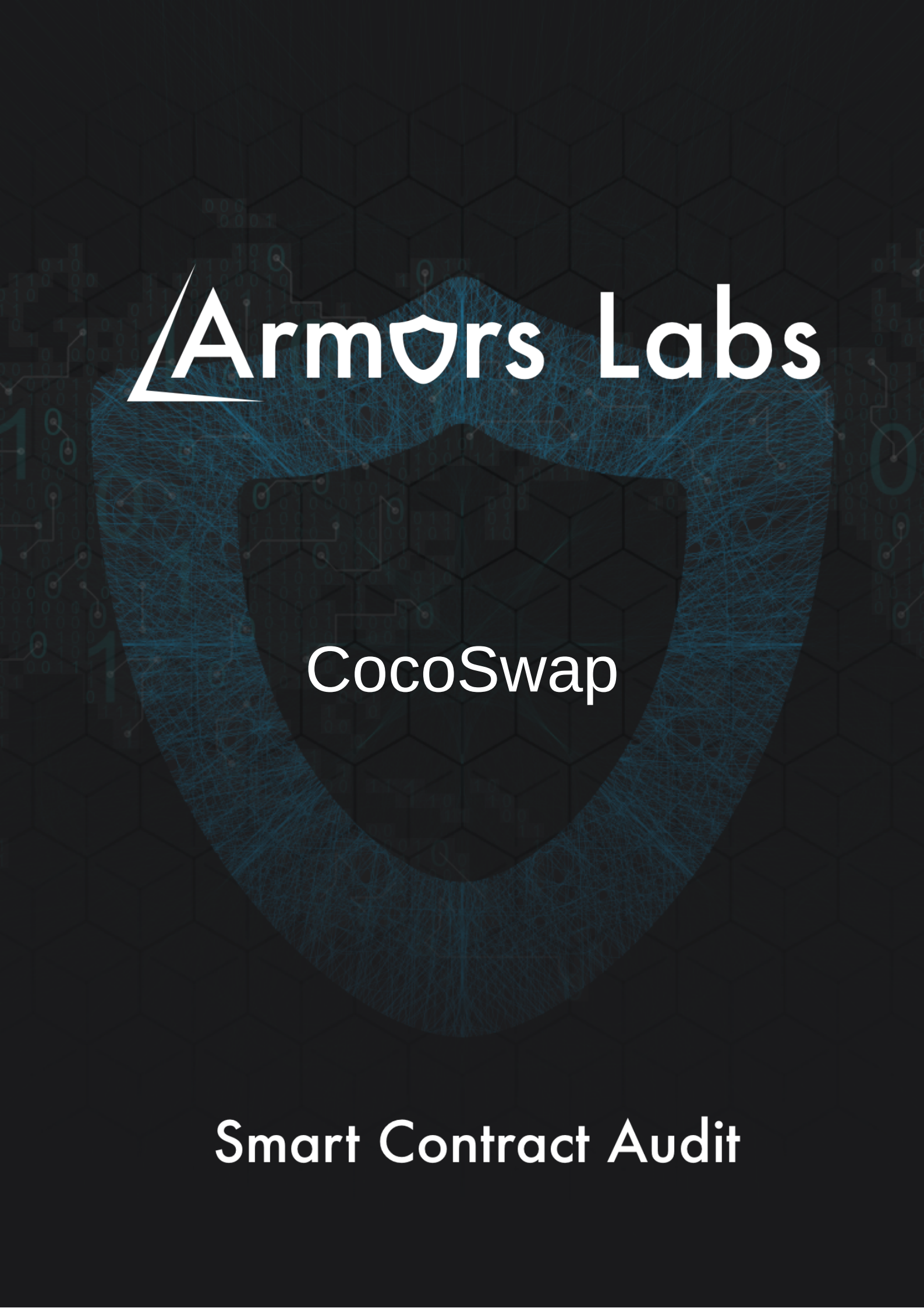

Armors Labs
CocoSwap
Smart Contract Audit

CocoSwap Audit Summary

Project name : CocoSwap Contract

Project address: None

Code URL : https://www.oklink.com/okexchain/address/0x748dEF2e7fbB37111761Aa78260B0ce71e41d4CA

Code URL : https://www.oklink.com/okexchain/address/0x83cc87c7dc5f0486bd79525b96e8177a2a14471c

Code URL : https://www.oklink.com/okexchain/address/0x32d8c121aDE1f6F916e975cFeD6DcF9B40c0D293

Code URL : https://www.oklink.com/okexchain/address/0x96B7639F5a67c5Ab3C6f223b7888d68Aad5EE006

Code URL : https://www.oklink.com/okexchain/address/0xB464d06A4111c79092171d52eC918eaf6CE71ac3

Code URL : https://www.oklink.com/okexchain/address/0x648C6B73825bAe6e1ac6D800a0cD751A6De8002D

Code URL : https://www.oklink.com/okexchain/address/0x1D813269B0A68CC12977329C850b1Ae510793B1F

Code URL : https://www.oklink.com/okexchain/address/0xEf2957f94e2e2BFD7AcA85062b25a7829166e28D

Code URL : https://www.oklink.com/okexchain/address/0x835D6fD31cE9a421ab9D2d910A822066459AbcAD

Commit : None

Project target : CocoSwap Contract Audit

Blockchain : OKExChain

Test result : PASSED

Audit Info

Audit NO : 0X202108050006

Audit Team : Armors Labs

Audit Proofreading: https://armors.io/#project-cases

CocoSwap Audit

The CocoSwap team asked us to review and audit their CocoSwap contract. We looked at the code and now publish our results.

Here is our assessment and recommendations, in order of importance.

Document information

Name	Auditor	Version	Date
CocoSwap Audit	Rock, Sophia, Rushairer, Rico, David, Alice	1.0.0	2021-08-05

Audit results

Note that as of the date of publishing, the above review reflects the current understanding of known security patterns as they relate to the CocoSwap contract. The above should not be construed as investment advice.

Based on the widely recognized security status of the current underlying blockchain and smart contract, this audit report is valid for 3 months from the date of output.

Disclaimer

Armors Labs Reports is not and should not be regarded as an "approval" or "disapproval" of any particular project or team. These reports are not and should not be regarded as indicators of the economy or value of any "product" or "asset" created by any team. Armors do not cover testing or auditing the integration with external contract or services (such as Unicrypt, Uniswap, PancakeSwap etc'…)

Armors Labs Reports represent an extensive auditing process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology. Armors does not guarantee the safety or functionality of the technology agreed to be analyzed.

Armors Labs postulates that the information provided is not missing, tampered, deleted or hidden. If the information provided is missing, tampered, deleted, hidden or reflected in a way that is not consistent with the actual situation, Armors Labs shall not be responsible for the losses and adverse effects caused. Armors Labs Audits should not be used in any way to make decisions around investment or involvement with any particular project. These reports in no way provide investment advice, nor should be leveraged as investment advice of any sort.

Audited target file

file	md5
./BoardRoom.sol	5d72f2b4576375722b38fb9151ba062a
./Oracle.sol	178fcbc31adf55993c262226f7ad6e41
./DexRouter.sol	f6ea3c82863ad4cee20e4c3639bca81d
./xTokenPool.sol	36b13cf262f3725d77e7967faf6511ae
./DexToken.sol	77e6edc8620e71f99979690cbd43c174
./HecoPool.sol	3955aeda47bed74670e60664303c2e9a
./DexFactory.sol	aab615af36b4a337ba75916230d1d11d
./SwapMinging.sol	68eb0178512ef4d2d808938feef2558b
./Repurchase.sol	14c673d7650bc60cd2c0a50b05621ecb

Vulnerability analysis

Vulnerability distribution

vulnerability level	number
Critical severity	0
High severity	0
Medium severity	0

vulnerability level	number
Low severity	0

Summary of audit results

Vulnerability	status
Re-Entrancy	safe
Arithmetic Over/Under Flows	safe
Unexpected Blockchain Currency	safe
Delegatecall	safe
Default Visibilities	safe
Entropy Illusion	safe
External Contract Referencing	safe
Short Address/Parameter Attack	safe
Unchecked CALL Return Values	safe
Race Conditions / Front Running	safe
Denial Of Service (DOS)	safe
Block Timestamp Manipulation	safe
Constructors with Care	safe
Unintialised Storage Pointers	safe
Floating Points and Numerical Precision	safe
tx.origin Authentication	safe
Permission restrictions	safe

Contract file

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.6.12;

abstract contract Context {
    function _msgSender() internal view virtual returns (address payable) {
        return msg.sender;
    }

    function _msgData() internal view virtual returns (bytes memory) {
        this; // silence state mutability warning without generating bytecode - see https://github.co
        return msg.data;
    }
}

abstract contract Ownable is Context {
    address private _owner;
```

```
    event OwnershipTransferred(address indexed previousOwner, address indexed newOwner);

    /**
     * @dev Initializes the contract setting the deployer as the initial owner.
     */
    constructor () internal {
        address msgSender = _msgSender();
        _owner = msgSender;
        emit OwnershipTransferred(address(0), msgSender);
    }

    /**
     * @dev Returns the address of the current owner.
     */
    function owner() public view returns (address) {
        return _owner;
    }

    /**
     * @dev Throws if called by any account other than the owner.
     */
    modifier onlyOwner() {
        require(_owner == _msgSender(), "Ownable: caller is not the owner");
        _;
    }

    /**
     * @dev Leaves the contract without owner. It will not be possible to call
     * `onlyOwner` functions anymore. Can only be called by the current owner.
     *
     * NOTE: Renouncing ownership will leave the contract without an owner,
     * thereby removing any functionality that is only available to the owner.
     */
    function renounceOwnership() public virtual onlyOwner {
        emit OwnershipTransferred(_owner, address(0));
        _owner = address(0);
    }

    /**
     * @dev Transfers ownership of the contract to a new account (`newOwner`).
     * Can only be called by the current owner.
     */
    function transferOwnership(address newOwner) public virtual onlyOwner {
        require(newOwner != address(0), "Ownable: new owner is the zero address");
        emit OwnershipTransferred(_owner, newOwner);
        _owner = newOwner;
    }
}

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);
}

library SafeMath {
    /**
     * @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) {
        return sub(a, b, "SafeMath: subtraction overflow");
    }

    /**
     * @dev Returns the subtraction of two unsigned integers, reverting with custom message on
     * overflow (when the result is negative).
     *
     * 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);
        uint256 c = a - b;

        return c;
    }

    /**
     * @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) {
        // 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 0;
        }

        uint256 c = a * b;
        require(c / a == b, "SafeMath: multiplication overflow");

        return c;
    }

    /**
     * @dev Returns the integer division of two unsigned integers. Reverts 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) {
        return div(a, b, "SafeMath: division by zero");
    }

    /**
     * @dev Returns the integer division of two unsigned integers. Reverts with custom message 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, string memory errorMessage) internal pure returns (uint256) {
        require(b > 0, errorMessage);
        uint256 c = a / b;
        // assert(a == b * c + a % b); // There is no case in which this doesn't hold

        return c;
    }

    /**
     * @dev Returns the remainder of dividing two unsigned integers. (unsigned integer modulo),
     * Reverts 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) {
        return mod(a, b, "SafeMath: modulo by zero");
    }

    /**
     * @dev Returns the remainder of dividing two unsigned integers. (unsigned integer modulo),
     * Reverts with custom message 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, string memory errorMessage) internal pure returns (uint256) {
        require(b != 0, errorMessage);
        return a % b;
    }
}
```

```
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
     */
    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.de
     *
     * 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 ret
        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
        return functionCallWithValue(target, data, value, "Address: low-level call with value failed"
    }

    /**
     * @dev Same as {xref-Address-functionCallWithValue-address-bytes-uint256-}[`functionCallWithValu
     * 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 er
        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 memor
        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) intern
```

```
        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);
    }

    function _verifyCallResult(bool success, bytes memory returndata, string memory errorMessage) pri
        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);
            }
        }
    }
}

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, valu
    }

    /**
     * @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, newAllowan
    }

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

```
    /**
     * @dev Imitates a Solidity high-level call (i.e. a regular function call to a contract), relaxin
     * on the return value: the return value is optional (but if data is returned, it must not be fal
     * @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 m
        // we're implementing it ourselves. We use {Address.functionCall} to perform this call, which
        // the target address contains contract code and also asserts for success in the low-level ca

        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");
        }
    }
}

interface IDexPair {
    event Approval(address indexed owner, address indexed spender, uint value);
    event Transfer(address indexed from, address indexed to, uint value);

    function name() external pure returns (string memory);

    function symbol() external pure returns (string memory);

    function decimals() external pure returns (uint8);

    function totalSupply() external view returns (uint);

    function balanceOf(address owner) external view returns (uint);

    function allowance(address owner, address spender) external view returns (uint);

    function approve(address spender, uint value) external returns (bool);

    function transfer(address to, uint value) external returns (bool);

    function transferFrom(address from, address to, uint value) external returns (bool);

    function DOMAIN_SEPARATOR() external view returns (bytes32);

    function PERMIT_TYPEHASH() external pure returns (bytes32);

    function nonces(address owner) external view returns (uint);

    function permit(address owner, address spender, uint value, uint deadline, uint8 v, bytes32 r, by

    event Mint(address indexed sender, uint amount0, uint amount1);
    event Burn(address indexed sender, uint amount0, uint amount1, address indexed to);
    event Swap(
        address indexed sender,
        uint amount0In,
        uint amount1In,
        uint amount0Out,
        uint amount1Out,
        address indexed to
    );
    event Sync(uint112 reserve0, uint112 reserve1);

    function MINIMUM_LIQUIDITY() external pure returns (uint);

    function factory() external view returns (address);
```

```
    function token0() external view returns (address);

    function token1() external view returns (address);

    function getReserves() external view returns (uint112 reserve0, uint112 reserve1, uint32 blockTim

    function price0CumulativeLast() external view returns (uint);

    function price1CumulativeLast() external view returns (uint);

    function kLast() external view returns (uint);

    function mint(address to) external returns (uint liquidity);

    function burn(address to) external returns (uint amount0, uint amount1);

    function swap(uint amount0Out, uint amount1Out, address to, bytes calldata data) external;

    function skim(address to) external;

    function sync() external;

    function price(address token, uint256 baseDecimal) external view returns (uint256);

    function initialize(address, address) external;
}

library EnumerableSet {
    // To implement this library for multiple types with as little code
    // repetition as possible, we write it in terms of a generic Set type with
    // bytes32 values.
    // The Set implementation uses private functions, and user-facing
    // implementations (such as AddressSet) are just wrappers around the
    // underlying Set.
    // This means that we can only create new EnumerableSets for types that fit
    // in bytes32.

    struct Set {
        // Storage of set values
        bytes32[] _values;

        // Position of the value in the `values` array, plus 1 because index 0
        // means a value is not in the set.
        mapping (bytes32 => uint256) _indexes;
    }

    /**
     * @dev Add a value to a set. O(1).
     *
     * Returns true if the value was added to the set, that is if it was not
     * already present.
     */
    function _add(Set storage set, bytes32 value) private returns (bool) {
        if (!_contains(set, value)) {
            set._values.push(value);
            // The value is stored at length-1, but we add 1 to all indexes
            // and use 0 as a sentinel value
            set._indexes[value] = set._values.length;
            return true;
        } else {
            return false;
        }
    }

    /**
     * @dev Removes a value from a set. O(1).
```

```
     *
     * Returns true if the value was removed from the set, that is if it was
     * present.
     */
    function _remove(Set storage set, bytes32 value) private returns (bool) {
        // We read and store the value's index to prevent multiple reads from the same storage slot
        uint256 valueIndex = set._indexes[value];

        if (valueIndex != 0) { // Equivalent to contains(set, value)
            // To delete an element from the _values array in O(1), we swap the element to delete wit
            // the array, and then remove the last element (sometimes called as 'swap and pop').
            // This modifies the order of the array, as noted in {at}.

            uint256 toDeleteIndex = valueIndex - 1;
            uint256 lastIndex = set._values.length - 1;

            // When the value to delete is the last one, the swap operation is unnecessary. However,
            // so rarely, we still do the swap anyway to avoid the gas cost of adding an 'if' stateme

            bytes32 lastvalue = set._values[lastIndex];

            // Move the last value to the index where the value to delete is
            set._values[toDeleteIndex] = lastvalue;
            // Update the index for the moved value
            set._indexes[lastvalue] = toDeleteIndex + 1; // All indexes are 1-based

            // Delete the slot where the moved value was stored
            set._values.pop();

            // Delete the index for the deleted slot
            delete set._indexes[value];

            return true;
        } else {
            return false;
        }
    }

    /**
     * @dev Returns true if the value is in the set. O(1).
     */
    function _contains(Set storage set, bytes32 value) private view returns (bool) {
        return set._indexes[value] != 0;
    }

    /**
     * @dev Returns the number of values on the set. O(1).
     */
    function _length(Set storage set) private view returns (uint256) {
        return set._values.length;
    }

   /**
    * @dev Returns the value stored at position `index` in the set. O(1).
    *
    * Note that there are no guarantees on the ordering of values inside the
    * array, and it may change when more values are added or removed.
    *
    * Requirements:
    *
    * - `index` must be strictly less than {length}.
    */
    function _at(Set storage set, uint256 index) private view returns (bytes32) {
        require(set._values.length > index, "EnumerableSet: index out of bounds");
        return set._values[index];
    }
```

```
// Bytes32Set

struct Bytes32Set {
    Set _inner;
}

/**
 * @dev Add a value to a set. O(1).
 *
 * Returns true if the value was added to the set, that is if it was not
 * already present.
 */
function add(Bytes32Set storage set, bytes32 value) internal returns (bool) {
    return _add(set._inner, value);
}

/**
 * @dev Removes a value from a set. O(1).
 *
 * Returns true if the value was removed from the set, that is if it was
 * present.
 */
function remove(Bytes32Set storage set, bytes32 value) internal returns (bool) {
    return _remove(set._inner, value);
}

/**
 * @dev Returns true if the value is in the set. O(1).
 */
function contains(Bytes32Set storage set, bytes32 value) internal view returns (bool) {
    return _contains(set._inner, value);
}

/**
 * @dev Returns the number of values in the set. O(1).
 */
function length(Bytes32Set storage set) internal view returns (uint256) {
    return _length(set._inner);
}

/**
 * @dev Returns the value stored at position `index` in the set. O(1).
 *
 * Note that there are no guarantees on the ordering of values inside the
 * array, and it may change when more values are added or removed.
 *
 * Requirements:
 *
 * - `index` must be strictly less than {length}.
 */
function at(Bytes32Set storage set, uint256 index) internal view returns (bytes32) {
    return _at(set._inner, index);
}

// AddressSet

struct AddressSet {
    Set _inner;
}

/**
 * @dev Add a value to a set. O(1).
 *
 * Returns true if the value was added to the set, that is if it was not
 * already present.
```

```
     */
    function add(AddressSet storage set, address value) internal returns (bool) {
        return _add(set._inner, bytes32(uint256(value)));
    }

    /**
     * @dev Removes a value from a set. O(1).
     *
     * Returns true if the value was removed from the set, that is if it was
     * present.
     */
    function remove(AddressSet storage set, address value) internal returns (bool) {
        return _remove(set._inner, bytes32(uint256(value)));
    }

    /**
     * @dev Returns true if the value is in the set. O(1).
     */
    function contains(AddressSet storage set, address value) internal view returns (bool) {
        return _contains(set._inner, bytes32(uint256(value)));
    }

    /**
     * @dev Returns the number of values in the set. O(1).
     */
    function length(AddressSet storage set) internal view returns (uint256) {
        return _length(set._inner);
    }

   /**
    * @dev Returns the value stored at position `index` in the set. O(1).
    *
    * Note that there are no guarantees on the ordering of values inside the
    * array, and it may change when more values are added or removed.
    *
    * Requirements:
    *
    * - `index` must be strictly less than {length}.
    */
    function at(AddressSet storage set, uint256 index) internal view returns (address) {
        return address(uint256(_at(set._inner, index)));
    }


    // UintSet

    struct UintSet {
        Set _inner;
    }

    /**
     * @dev Add a value to a set. O(1).
     *
     * Returns true if the value was added to the set, that is if it was not
     * already present.
     */
    function add(UintSet storage set, uint256 value) internal returns (bool) {
        return _add(set._inner, bytes32(value));
    }

    /**
     * @dev Removes a value from a set. O(1).
     *
     * Returns true if the value was removed from the set, that is if it was
     * present.
     */
```

```
    function remove(UintSet storage set, uint256 value) internal returns (bool) {
        return _remove(set._inner, bytes32(value));
    }

    /**
     * @dev Returns true if the value is in the set. O(1).
     */
    function contains(UintSet storage set, uint256 value) internal view returns (bool) {
        return _contains(set._inner, bytes32(value));
    }

    /**
     * @dev Returns the number of values on the set. O(1).
     */
    function length(UintSet storage set) internal view returns (uint256) {
        return _length(set._inner);
    }

   /**
    * @dev Returns the value stored at position `index` in the set. O(1).
    *
    * Note that there are no guarantees on the ordering of values inside the
    * array, and it may change when more values are added or removed.
    *
    * Requirements:
    *
    * - `index` must be strictly less than {length}.
    */
    function at(UintSet storage set, uint256 index) internal view returns (uint256) {
        return uint256(_at(set._inner, index));
    }
}


contract Repurchase is Ownable {
    using SafeMath for uint256;
    using SafeERC20 for IERC20;

    using EnumerableSet for EnumerableSet.AddressSet;
    EnumerableSet.AddressSet private _caller;

    address public constant USDT = 0x382bB369d343125BfB2117af9c149795C6C65C50;
    address public constant COCO = 0x748dEF2e7fbB37111761Aa78260B0ce71e41d4CA;
    address public constant COCO_USDT = 0xcf5C645da27e0164eDA3E2a1002D078ebe80b5c5;
    address public constant destroyAddress = 0x000000000000000000000000000000000000C0C0;
    address public emergencyAddress;
    uint256 public amountIn;

    constructor (uint256 _amount, address _emergencyAddress) public {
        require(_amount > 0, "Amount must be greater than zero");
        require(_emergencyAddress != address(0), "Is zero address");
        amountIn = _amount;
        emergencyAddress = _emergencyAddress;
    }

    function setAmountIn(uint256 _newIn) public onlyOwner {
        amountIn = _newIn;
    }

    function setEmergencyAddress(address _newAddress) public onlyOwner {
        require(_newAddress != address(0), "Is zero address");
        emergencyAddress = _newAddress;
    }

    function addCaller(address _newCaller) public onlyOwner returns (bool) {
        require(_newCaller != address(0), "NewCaller is the zero address");
```

```
        return EnumerableSet.add(_caller, _newCaller);
    }

    function delCaller(address _delCaller) public onlyOwner returns (bool) {
        require(_delCaller != address(0), "DelCaller is the zero address");
        return EnumerableSet.remove(_caller, _delCaller);
    }

    function getCallerLength() public view returns (uint256) {
        return EnumerableSet.length(_caller);
    }

    function isCaller(address _call) public view returns (bool) {
        return EnumerableSet.contains(_caller, _call);
    }

    function getCaller(uint256 _index) public view returns (address){
        require(_index <= getCallerLength() - 1, "index out of bounds");
        return EnumerableSet.at(_caller, _index);
    }

    function swap() external onlyCaller returns (uint256 amountOut){
        require(IERC20(USDT).balanceOf(address(this)) >= amountIn, "Insufficient contract balance");
        (uint256 reserve0, uint256 reserve1,) = IDexPair(COCO_USDT).getReserves();
        uint256 amountInWithFee = amountIn.mul(997);
        amountOut = amountIn.mul(997).mul(reserve0) / reserve1.mul(1000).add(amountInWithFee);
        IERC20(USDT).safeTransfer(COCO_USDT, amountIn);
        IDexPair(COCO_USDT).swap(amountOut, 0, destroyAddress, new bytes(0));
    }

    modifier onlyCaller() {
        require(isCaller(msg.sender), "Not the caller");
        _;
    }

    function emergencyWithdraw(address _token) public onlyOwner {
        require(IERC20(_token).balanceOf(address(this)) > 0, "Insufficient contract balance");
        IERC20(_token).transfer(emergencyAddress, IERC20(_token).balanceOf(address(this)));
    }
}// SPDX-License-Identifier: MIT
pragma solidity >=0.5.0 <0.7.0;

contract Ownable {
    address private _owner;

    constructor () internal {
        _owner = msg.sender;
        emit OwnershipTransferred(address(0), _owner);
    }

    function owner() public view returns (address) {
        return _owner;
    }

    function isOwner(address account) public view returns (bool) {
        return account == _owner;
    }

    function renounceOwnership() public onlyOwner {
        emit OwnershipTransferred(_owner, address(0));
        _owner = address(0);
    }

    function _transferOwnership(address newOwner) internal {
        require(newOwner != address(0), "Ownable: new owner is the zero address");
        emit OwnershipTransferred(_owner, newOwner);
```

```
        _owner = newOwner;
    }

    function transferOwnership(address newOwner) public onlyOwner {
        _transferOwnership(newOwner);
    }


    modifier onlyOwner() {
        require(isOwner(msg.sender), "Ownable: caller is not the owner");
        _;
    }

    event OwnershipTransferred(address indexed previousOwner, address indexed newOwner);
}

library SafeMath {
    uint256 constant WAD = 10 ** 18;
    uint256 constant RAY = 10 ** 27;

    function wad() public pure returns (uint256) {
        return WAD;
    }

    function ray() public pure returns (uint256) {
        return RAY;
    }

    function add(uint256 a, uint256 b) internal pure returns (uint256) {
        uint256 c = a + b;
        require(c >= a, "SafeMath: addition overflow");

        return c;
    }

    function sub(uint256 a, uint256 b) internal pure returns (uint256) {
        return sub(a, b, "SafeMath: subtraction overflow");
    }

    function sub(uint256 a, uint256 b, string memory errorMessage) internal pure returns (uint256) {
        require(b <= a, errorMessage);
        uint256 c = a - b;

        return c;
    }

    function mul(uint256 a, uint256 b) internal pure returns (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 0;
        }

        uint256 c = a * b;
        require(c / a == b, "SafeMath: multiplication overflow");

        return c;
    }

    function div(uint256 a, uint256 b) internal pure returns (uint256) {
        return div(a, b, "SafeMath: division by zero");
    }

    function div(uint256 a, uint256 b, string memory errorMessage) internal pure returns (uint256) {
        // Solidity only automatically asserts when dividing by 0
```

```
        require(b > 0, errorMessage);
        uint256 c = a / b;
        // assert(a == b * c + a % b); // There is no case in which this doesn't hold

        return c;
    }

    function mod(uint256 a, uint256 b) internal pure returns (uint256) {
        return mod(a, b, "SafeMath: modulo by zero");
    }

    function mod(uint256 a, uint256 b, string memory errorMessage) internal pure returns (uint256) {
        require(b != 0, errorMessage);
        return a % b;
    }

    function min(uint256 a, uint256 b) internal pure returns (uint256) {
        return a <= b ? a : b;
    }

    function max(uint256 a, uint256 b) internal pure returns (uint256) {
        return a >= b ? a : b;
    }

    function sqrt(uint256 a) internal pure returns (uint256 b) {
        if (a > 3) {
            b = a;
            uint256 x = a / 2 + 1;
            while (x < b) {
                b = x;
                x = (a / x + x) / 2;
            }
        } else if (a != 0) {
            b = 1;
        }
    }

    function wmul(uint256 a, uint256 b) internal pure returns (uint256) {
        return mul(a, b) / WAD;
    }

    function wmulRound(uint256 a, uint256 b) internal pure returns (uint256) {
        return add(mul(a, b), WAD / 2) / WAD;
    }

    function rmul(uint256 a, uint256 b) internal pure returns (uint256) {
        return mul(a, b) / RAY;
    }

    function rmulRound(uint256 a, uint256 b) internal pure returns (uint256) {
        return add(mul(a, b), RAY / 2) / RAY;
    }

    function wdiv(uint256 a, uint256 b) internal pure returns (uint256) {
        return div(mul(a, WAD), b);
    }

    function wdivRound(uint256 a, uint256 b) internal pure returns (uint256) {
        return add(mul(a, WAD), b / 2) / b;
    }

    function rdiv(uint256 a, uint256 b) internal pure returns (uint256) {
        return div(mul(a, RAY), b);
    }

    function rdivRound(uint256 a, uint256 b) internal pure returns (uint256) {
```

```
        return add(mul(a, RAY), b / 2) / b;
    }

    function wpow(uint256 x, uint256 n) internal pure returns (uint256) {
        uint256 result = WAD;
        while (n > 0) {
            if (n % 2 != 0) {
                result = wmul(result, x);
            }
            x = wmul(x, x);
            n /= 2;
        }
        return result;
    }

    function rpow(uint256 x, uint256 n) internal pure returns (uint256) {
        uint256 result = RAY;
        while (n > 0) {
            if (n % 2 != 0) {
                result = rmul(result, x);
            }
            x = rmul(x, x);
            n /= 2;
        }
        return result;
    }
}

library EnumerableSet {
    // To implement this library for multiple types with as little code
    // repetition as possible, we write it in terms of a generic Set type with
    // bytes32 values.
    // The Set implementation uses private functions, and user-facing
    // implementations (such as AddressSet) are just wrappers around the
    // underlying Set.
    // This means that we can only create new EnumerableSets for types that fit
    // in bytes32.

    struct Set {
        // Storage of set values
        bytes32[] _values;

        // Position of the value in the `values` array, plus 1 because index 0
        // means a value is not in the set.
        mapping(bytes32 => uint256) _indexes;
    }

    /**
     * @dev Add a value to a set. O(1).
     *
     * Returns true if the value was added to the set, that is if it was not
     * already present.
     */
    function _add(Set storage set, bytes32 value) private returns (bool) {
        if (!_contains(set, value)) {
            set._values.push(value);
            // The value is stored at length-1, but we add 1 to all indexes
            // and use 0 as a sentinel value
            set._indexes[value] = set._values.length;
            return true;
        } else {
            return false;
        }
    }

    /**
```

```
     * @dev Removes a value from a set. O(1).
     *
     * Returns true if the value was removed from the set, that is if it was
     * present.
     */
    function _remove(Set storage set, bytes32 value) private returns (bool) {
        // We read and store the value's index to prevent multiple reads from the same storage slot
        uint256 valueIndex = set._indexes[value];

        if (valueIndex != 0) {// Equivalent to contains(set, value)
            // To delete an element from the _values array in O(1), we swap the element to delete wit
            // the array, and then remove the last element (sometimes called as 'swap and pop').
            // This modifies the order of the array, as noted in {at}.

            uint256 toDeleteIndex = valueIndex - 1;
            uint256 lastIndex = set._values.length - 1;

            // When the value to delete is the last one, the swap operation is unnecessary. However,
            // so rarely, we still do the swap anyway to avoid the gas cost of adding an 'if' stateme

            bytes32 lastvalue = set._values[lastIndex];

            // Move the last value to the index where the value to delete is
            set._values[toDeleteIndex] = lastvalue;
            // Update the index for the moved value
            set._indexes[lastvalue] = toDeleteIndex + 1;
            // All indexes are 1-based

            // Delete the slot where the moved value was stored
            set._values.pop();

            // Delete the index for the deleted slot
            delete set._indexes[value];

            return true;
        } else {
            return false;
        }
    }

    /**
     * @dev Returns true if the value is in the set. O(1).
     */
    function _contains(Set storage set, bytes32 value) private view returns (bool) {
        return set._indexes[value] != 0;
    }

    /**
     * @dev Returns the number of values on the set. O(1).
     */
    function _length(Set storage set) private view returns (uint256) {
        return set._values.length;
    }

    /**
     * @dev Returns the value stored at position `index` in the set. O(1).
     *
     * Note that there are no guarantees on the ordering of values inside the
     * array, and it may change when more values are added or removed.
     *
     * Requirements:
     *
     * - `index` must be strictly less than {length}.
     */
    function _at(Set storage set, uint256 index) private view returns (bytes32) {
        require(set._values.length > index, "EnumerableSet: index out of bounds");
```

```
        return set._values[index];
    }

    // Bytes32Set

    struct Bytes32Set {
        Set _inner;
    }

    /**
     * @dev Add a value to a set. O(1).
     *
     * Returns true if the value was added to the set, that is if it was not
     * already present.
     */
    function add(Bytes32Set storage set, bytes32 value) internal returns (bool) {
        return _add(set._inner, value);
    }

    /**
     * @dev Removes a value from a set. O(1).
     *
     * Returns true if the value was removed from the set, that is if it was
     * present.
     */
    function remove(Bytes32Set storage set, bytes32 value) internal returns (bool) {
        return _remove(set._inner, value);
    }

    /**
     * @dev Returns true if the value is in the set. O(1).
     */
    function contains(Bytes32Set storage set, bytes32 value) internal view returns (bool) {
        return _contains(set._inner, value);
    }

    /**
     * @dev Returns the number of values in the set. O(1).
     */
    function length(Bytes32Set storage set) internal view returns (uint256) {
        return _length(set._inner);
    }

    /**
     * @dev Returns the value stored at position `index` in the set. O(1).
     *
     * Note that there are no guarantees on the ordering of values inside the
     * array, and it may change when more values are added or removed.
     *
     * Requirements:
     *
     * - `index` must be strictly less than {length}.
     */
    function at(Bytes32Set storage set, uint256 index) internal view returns (bytes32) {
        return _at(set._inner, index);
    }

    // AddressSet

    struct AddressSet {
        Set _inner;
    }

    /**
     * @dev Add a value to a set. O(1).
     *
```

```
     * Returns true if the value was added to the set, that is if it was not
     * already present.
     */
    function add(AddressSet storage set, address value) internal returns (bool) {
        return _add(set._inner, bytes32(uint256(value)));
    }

    /**
     * @dev Removes a value from a set. O(1).
     *
     * Returns true if the value was removed from the set, that is if it was
     * present.
     */
    function remove(AddressSet storage set, address value) internal returns (bool) {
        return _remove(set._inner, bytes32(uint256(value)));
    }

    /**
     * @dev Returns true if the value is in the set. O(1).
     */
    function contains(AddressSet storage set, address value) internal view returns (bool) {
        return _contains(set._inner, bytes32(uint256(value)));
    }

    /**
     * @dev Returns the number of values in the set. O(1).
     */
    function length(AddressSet storage set) internal view returns (uint256) {
        return _length(set._inner);
    }

    /**
     * @dev Returns the value stored at position `index` in the set. O(1).
     *
     * Note that there are no guarantees on the ordering of values inside the
     * array, and it may change when more values are added or removed.
     *
     * Requirements:
     *
     * - `index` must be strictly less than {length}.
     */
    function at(AddressSet storage set, uint256 index) internal view returns (address) {
        return address(uint256(_at(set._inner, index)));
    }


    // UintSet

    struct UintSet {
        Set _inner;
    }

    /**
     * @dev Add a value to a set. O(1).
     *
     * Returns true if the value was added to the set, that is if it was not
     * already present.
     */
    function add(UintSet storage set, uint256 value) internal returns (bool) {
        return _add(set._inner, bytes32(value));
    }

    /**
     * @dev Removes a value from a set. O(1).
     *
     * Returns true if the value was removed from the set, that is if it was
```

```
     * present.
     */
    function remove(UintSet storage set, uint256 value) internal returns (bool) {
        return _remove(set._inner, bytes32(value));
    }

    /**
     * @dev Returns true if the value is in the set. O(1).
     */
    function contains(UintSet storage set, uint256 value) internal view returns (bool) {
        return _contains(set._inner, bytes32(value));
    }

    /**
     * @dev Returns the number of values on the set. O(1).
     */
    function length(UintSet storage set) internal view returns (uint256) {
        return _length(set._inner);
    }

    /**
     * @dev Returns the value stored at position `index` in the set. O(1).
     *
     * Note that there are no guarantees on the ordering of values inside the
     * array, and it may change when more values are added or removed.
     *
     * Requirements:
     *
     * - `index` must be strictly less than {length}.
     */
    function at(UintSet storage set, uint256 index) internal view returns (uint256) {
        return uint256(_at(set._inner, index));
    }
}

interface IERC20 {
    event Approval(address indexed owner, address indexed spender, uint value);
    event Transfer(address indexed from, address indexed to, uint value);

    function name() external view returns (string memory);

    function symbol() external view returns (string memory);

    function decimals() external view returns (uint8);

    function totalSupply() external view returns (uint);

    function balanceOf(address owner) external view returns (uint);

    function allowance(address owner, address spender) external view returns (uint);

    function approve(address spender, uint value) external returns (bool);

    function transfer(address to, uint value) external returns (bool);

    function transferFrom(address from, address to, uint value) external returns (bool);
}

interface IDex is IERC20 {
    function mint(address to, uint256 amount) external returns (bool);
}

interface IDexFactory {
    event PairCreated(address indexed token0, address indexed token1, address pair, uint);

    function FEE_RATE_DENOMINATOR() external view returns (uint256);
```

```
    function feeRateNumerator() external view returns (uint256);
    function feeTo() external view returns (address);
    function feeToSetter() external view returns (address);
    function feeToRate() external view returns (uint256);
    function initCodeHash() external view returns (bytes32);
    function pairFeeToRate(address) external view returns (uint256);
    function pairFees(address) external view returns (uint256);
    function getPair(address tokenA, address tokenB) external view returns (address pair);
    function allPairs(uint) external view returns (address pair);
    function allPairsLength() external view returns (uint);
    function createPair(address tokenA, address tokenB) external returns (address pair);
    function setFeeTo(address) external;
    function setFeeToSetter(address) external;
    function addPair(address) external returns (bool);
    function delPair(address) external returns (bool);
    function getSupportListLength() external view returns (uint256);
    function isSupportPair(address pair) external view returns (bool);
    function getSupportPair(uint256 index) external view returns (address);
    function setFeeRateNumerator(uint256) external;
    function setPairFees(address pair, uint256 fee) external;
    function setDefaultFeeToRate(uint256) external;
    function setPairFeeToRate(address pair, uint256 rate) external;
    function getPairFees(address) external view returns (uint256);
    function getPairRate(address) external view returns (uint256);
    function sortTokens(address tokenA, address tokenB) external pure returns (address token0, addres
    function pairFor(address tokenA, address tokenB) external view returns (address pair);
    function getReserves(address tokenA, address tokenB) external view returns (uint256 reserveA, uin
    function quote(uint256 amountA, uint256 reserveA, uint256 reserveB) external pure returns (uint25
    function getAmountOut(uint256 amountIn, uint256 reserveIn, uint256 reserveOut, address token0, ad
    function getAmountIn(uint256 amountOut, uint256 reserveIn, uint256 reserveOut, address token0, ad
    function getAmountsOut(uint256 amountIn, address[] calldata path) external view returns (uint256[
    function getAmountsIn(uint256 amountOut, address[] calldata path) external view returns (uint256[
}
```

```
interface IDexPair {
    event Approval(address indexed owner, address indexed spender, uint value);
    event Transfer(address indexed from, address indexed to, uint value);

    function name() external pure returns (string memory);

    function symbol() external pure returns (string memory);

    function decimals() external pure returns (uint8);

    function totalSupply() external view returns (uint);

    function balanceOf(address owner) external view returns (uint);

    function allowance(address owner, address spender) external view returns (uint);

    function approve(address spender, uint value) external returns (bool);

    function transfer(address to, uint value) external returns (bool);

    function transferFrom(address from, address to, uint value) external returns (bool);

    function DOMAIN_SEPARATOR() external view returns (bytes32);

    function PERMIT_TYPEHASH() external pure returns (bytes32);

    function nonces(address owner) external view returns (uint);

    function permit(address owner, address spender, uint value, uint deadline, uint8 v, bytes32 r, by

    event Mint(address indexed sender, uint amount0, uint amount1);
    event Burn(address indexed sender, uint amount0, uint amount1, address indexed to);
    event Swap(
        address indexed sender,
        uint amount0In,
        uint amount1In,
        uint amount0Out,
        uint amount1Out,
        address indexed to
    );
    event Sync(uint112 reserve0, uint112 reserve1);

    function MINIMUM_LIQUIDITY() external pure returns (uint);

    function factory() external view returns (address);

    function token0() external view returns (address);

    function token1() external view returns (address);

    function getReserves() external view returns (uint112 reserve0, uint112 reserve1, uint32 blockTim

    function price0CumulativeLast() external view returns (uint);

    function price1CumulativeLast() external view returns (uint);

    function kLast() external view returns (uint);

    function mint(address to) external returns (uint liquidity);

    function burn(address to) external returns (uint amount0, uint amount1);

    function swap(uint amount0Out, uint amount1Out, address to, bytes calldata data) external;

    function skim(address to) external;
```

```
    function sync() external;

    function initialize(address, address) external;
}

interface IOracle {
    function consult(address tokenIn, uint amountIn, address tokenOut) external view returns (uint am
}

contract SwapMining is Ownable {
    using SafeMath for uint256;
    using EnumerableSet for EnumerableSet.AddressSet;
    EnumerableSet.AddressSet private _whitelist;

    // DEX tokens created per block
    uint256 public dexPerBlock;
    // The block number when DEX mining starts.
    uint256 public startBlock;
    // How many blocks are halved
    uint256 public halvingPeriod = 1670400;
    // Total allocation points
    uint256 public totalAllocPoint = 0;
    IOracle public oracle;
    // router address
    address public router;
    // factory address
    IDexFactory public factory;
    // dex token address
    IDex public dex;
    // Calculate price based on BUSD-T
    address public targetToken;
    // pair corresponding pid
    mapping(address => uint256) public pairOfPid;

    constructor(
        IDex _dex,
        IDexFactory _factory,
        IOracle _oracle,
        address _router,
        address _targetToken,
        uint256 _dexPerBlock,
        uint256 _startBlock
    ) public {
        dex = _dex;
        factory = _factory;
        oracle = _oracle;
        router = _router;
        targetToken = _targetToken;
        dexPerBlock = _dexPerBlock;
        startBlock = _startBlock;
    }

    struct UserInfo {
        uint256 quantity;       // How many LP tokens the user has provided
        uint256 blockNumber;    // Last transaction block
    }

    struct PoolInfo {
        address pair;           // Trading pairs that can be mined
        uint256 quantity;       // Current amount of LPs
        uint256 totalQuantity;  // All quantity
        uint256 allocPoint;     // How many allocation points assigned to this pool
        uint256 allocDexAmount; // How many DEXs
        uint256 lastRewardBlock;// Last transaction block
    }
```

```
    PoolInfo[] public poolInfo;
    mapping(uint256 => mapping(address => UserInfo)) public userInfo;


    function poolLength() public view returns (uint256) {
        return poolInfo.length;
    }


    function addPair(uint256 _allocPoint, address _pair, bool _withUpdate) public onlyOwner {
        require(_pair != address(0), "_pair is the zero address");
        if (_withUpdate) {
            massMintPools();
        }
        uint256 lastRewardBlock = block.number > startBlock ? block.number : startBlock;
        totalAllocPoint = totalAllocPoint.add(_allocPoint);
        poolInfo.push(PoolInfo({
        pair : _pair,
        quantity : 0,
        totalQuantity : 0,
        allocPoint : _allocPoint,
        allocDexAmount : 0,
        lastRewardBlock : lastRewardBlock
        }));
        pairOfPid[_pair] = poolLength() - 1;
    }

    // Update the allocPoint of the pool
    function setPair(uint256 _pid, uint256 _allocPoint, bool _withUpdate) public onlyOwner {
        if (_withUpdate) {
            massMintPools();
        }
        totalAllocPoint = totalAllocPoint.sub(poolInfo[_pid].allocPoint).add(_allocPoint);
        poolInfo[_pid].allocPoint = _allocPoint;
    }

    // Set the number of dex produced by each block
    function setDexPerBlock(uint256 _newPerBlock) public onlyOwner {
        massMintPools();
        dexPerBlock = _newPerBlock;
    }

    function setStartBlock(uint256 _startBlock) public onlyOwner {
        startBlock = _startBlock;
    }

    // Only tokens in the whitelist can be mined DEX
    function addWhitelist(address _addToken) public onlyOwner returns (bool) {
        require(_addToken != address(0), "SwapMining: token is the zero address");
        return EnumerableSet.add(_whitelist, _addToken);
    }

    function delWhitelist(address _delToken) public onlyOwner returns (bool) {
        require(_delToken != address(0), "SwapMining: token is the zero address");
        return EnumerableSet.remove(_whitelist, _delToken);
    }

    function getWhitelistLength() public view returns (uint256) {
        return EnumerableSet.length(_whitelist);
    }

    function isWhitelist(address _token) public view returns (bool) {
        return EnumerableSet.contains(_whitelist, _token);
    }

    function getWhitelist(uint256 _index) public view returns (address){
```

```
        require(_index <= getWhitelistLength() - 1, "SwapMining: index out of bounds");
        return EnumerableSet.at(_whitelist, _index);
    }

    function setHalvingPeriod(uint256 _block) public onlyOwner {
        halvingPeriod = _block;
    }

    function setRouter(address newRouter) public onlyOwner {
        require(newRouter != address(0), "SwapMining: new router is the zero address");
        router = newRouter;
    }

    function setOracle(IOracle _oracle) public onlyOwner {
        require(address(_oracle) != address(0), "SwapMining: new oracle is the zero address");
        oracle = _oracle;
    }

    // At what phase
    function phase(uint256 blockNumber) public view returns (uint256) {
        if (halvingPeriod == 0) {
            return 0;
        }
        if (blockNumber > startBlock) {
            return (blockNumber.sub(startBlock).sub(1)).div(halvingPeriod);
        }
        return 0;
    }

    function phase() public view returns (uint256) {
        return phase(block.number);
    }

    function reward(uint256 blockNumber) public view returns (uint256) {
        uint256 _phase = phase(blockNumber);
        return dexPerBlock.div(2 ** _phase);
    }

    function reward() public view returns (uint256) {
        return reward(block.number);
    }

    // Rewards for the current block
    function getDexReward(uint256 _lastRewardBlock) public view returns (uint256) {
        require(_lastRewardBlock <= block.number, "SwapMining: must little than the current block num
        uint256 blockReward = 0;
        uint256 n = phase(_lastRewardBlock);
        uint256 m = phase(block.number);
        // If it crosses the cycle
        while (n < m) {
            n++;
            // Get the last block of the previous cycle
            uint256 r = n.mul(halvingPeriod).add(startBlock);
            // Get rewards from previous periods
            blockReward = blockReward.add((r.sub(_lastRewardBlock)).mul(reward(r)));
            _lastRewardBlock = r;
        }
        blockReward = blockReward.add((block.number.sub(_lastRewardBlock)).mul(reward(block.number)))
        return blockReward;
    }

    // Update all pools Called when updating allocPoint and setting new blocks
    function massMintPools() public {
        uint256 length = poolInfo.length;
        for (uint256 pid = 0; pid < length; ++pid) {
            mint(pid);
```

```
        }
    }

    function mint(uint256 _pid) public returns (bool) {
        PoolInfo storage pool = poolInfo[_pid];
        if (block.number <= pool.lastRewardBlock) {
            return false;
        }
        uint256 blockReward = getDexReward(pool.lastRewardBlock);
        if (blockReward <= 0) {
            return false;
        }
        // Calculate the rewards obtained by the pool based on the allocPoint
        uint256 dexReward = blockReward.mul(pool.allocPoint).div(totalAllocPoint);
        dex.mint(address(this), dexReward);
        // Increase the number of tokens in the current pool
        pool.allocDexAmount = pool.allocDexAmount.add(dexReward);
        pool.lastRewardBlock = block.number;
        return true;
    }

    // swapMining only router
    function swap(address account, address input, address output, uint256 amount) public onlyRouter r
        require(account != address(0), "SwapMining: taker swap account is the zero address");
        require(input != address(0), "SwapMining: taker swap input is the zero address");
        require(output != address(0), "SwapMining: taker swap output is the zero address");

        if (poolLength() <= 0) {
            return false;
        }

        if (!isWhitelist(input) || !isWhitelist(output)) {
            return false;
        }

        address pair = IDexFactory(factory).pairFor(input, output);

        PoolInfo storage pool = poolInfo[pairOfPid[pair]];
        // If it does not exist or the allocPoint is 0 then return
        if (pool.pair != pair || pool.allocPoint <= 0) {
            return false;
        }

        uint256 quantity = getQuantity(output, amount, targetToken);
        if (quantity <= 0) {
            return false;
        }

        mint(pairOfPid[pair]);

        pool.quantity = pool.quantity.add(quantity);
        pool.totalQuantity = pool.totalQuantity.add(quantity);
        UserInfo storage user = userInfo[pairOfPid[pair]][account];
        user.quantity = user.quantity.add(quantity);
        user.blockNumber = block.number;
        return true;
    }

    // The user withdraws all the transaction rewards of the pool
    function takerWithdraw() public {
        uint256 userSub;
        uint256 length = poolInfo.length;
        for (uint256 pid = 0; pid < length; ++pid) {
            PoolInfo storage pool = poolInfo[pid];
            UserInfo storage user = userInfo[pid][msg.sender];
            if (user.quantity > 0) {
```

```
                mint(pid);
                // The reward held by the user in this pool
                uint256 userReward = pool.allocDexAmount.mul(user.quantity).div(pool.quantity);
                pool.quantity = pool.quantity.sub(user.quantity);
                pool.allocDexAmount = pool.allocDexAmount.sub(userReward);
                user.quantity = 0;
                user.blockNumber = block.number;
                userSub = userSub.add(userReward);
            }
        }
        if (userSub <= 0) {
            return;
        }
        dex.transfer(msg.sender, userSub);
    }

    // Get rewards from users in the current pool
    function getUserReward(uint256 _pid) public view returns (uint256, uint256){
        require(_pid <= poolInfo.length - 1, "SwapMining: Not find this pool");
        uint256 userSub;
        PoolInfo memory pool = poolInfo[_pid];
        UserInfo memory user = userInfo[_pid][msg.sender];
        if (user.quantity > 0) {
            uint256 blockReward = getDexReward(pool.lastRewardBlock);
            uint256 dexReward = blockReward.mul(pool.allocPoint).div(totalAllocPoint);
            userSub = userSub.add((pool.allocDexAmount.add(dexReward)).mul(user.quantity).div(pool.qu
        }
        //dex available to users, User transaction amount
        return (userSub, user.quantity);
    }

    // Get details of the pool
    function getPoolInfo(uint256 _pid) public view returns (address, address, uint256, uint256, uint2
        require(_pid <= poolInfo.length - 1, "SwapMining: Not find this pool");
        PoolInfo memory pool = poolInfo[_pid];
        address token0 = IDexPair(pool.pair).token0();
        address token1 = IDexPair(pool.pair).token1();
        uint256 dexAmount = pool.allocDexAmount;
        uint256 blockReward = getDexReward(pool.lastRewardBlock);
        uint256 dexReward = blockReward.mul(pool.allocPoint).div(totalAllocPoint);
        dexAmount = dexAmount.add(dexReward);
        //token0,token1,Pool remaining reward,Total /Current transaction volume of the pool
        return (token0, token1, dexAmount, pool.totalQuantity, pool.quantity, pool.allocPoint);
    }

    modifier onlyRouter() {
        require(msg.sender == router, "SwapMining: caller is not the router");
        _;
    }

    function getQuantity(address outputToken, uint256 outputAmount, address anchorToken) public view
        uint256 quantity = 0;
        if (outputToken == anchorToken) {
            quantity = outputAmount;
        } else if (IDexFactory(factory).getPair(outputToken, anchorToken) != address(0)) {
            quantity = IOracle(oracle).consult(outputToken, outputAmount, anchorToken);
        } else {
            uint256 length = getWhitelistLength();
            for (uint256 index = 0; index < length; index++) {
                address intermediate = getWhitelist(index);
                if (IDexFactory(factory).getPair(outputToken, intermediate) != address(0) && IDexFact
                    uint256 interQuantity = IOracle(oracle).consult(outputToken, outputAmount, interm
                    quantity = IOracle(oracle).consult(intermediate, interQuantity, anchorToken);
                    break;
                }
            }
```

```
        }
        return quantity;
    }

}pragma solidity >=0.5.0 <0.8.0;

interface IDexFactory {
    event PairCreated(address indexed token0, address indexed token1, address pair, uint);

    function FEE_RATE_DENOMINATOR() external view returns (uint256);

    function feeRateNumerator() external view returns (uint256);

    function feeTo() external view returns (address);

    function feeToSetter() external view returns (address);

    function feeToRate() external view returns (uint256);

    function initCodeHash() external view returns (bytes32);

    function pairFeeToRate(address) external view returns (uint256);

    function pairFees(address) external view returns (uint256);

    function getPair(address tokenA, address tokenB) external view returns (address pair);

    function allPairs(uint) external view returns (address pair);

    function allPairsLength() external view returns (uint);

    function createPair(address tokenA, address tokenB) external returns (address pair);

    function setFeeTo(address) external;

    function setFeeToSetter(address) external;

    function addPair(address) external returns (bool);

    function delPair(address) external returns (bool);

    function getSupportListLength() external view returns (uint256);

    function isSupportPair(address pair) external view returns (bool);

    function getSupportPair(uint256 index) external view returns (address);

    function setFeeRateNumerator(uint256) external;

    function setPairFees(address pair, uint256 fee) external;

    function setDefaultFeeToRate(uint256) external;

    function setPairFeeToRate(address pair, uint256 rate) external;

    function getPairFees(address) external view returns (uint256);

    function getPairRate(address) external view returns (uint256);

    function sortTokens(address tokenA, address tokenB) external pure returns (address token0, addres

    function pairFor(address tokenA, address tokenB) external view returns (address pair);

    function getReserves(address tokenA, address tokenB) external view returns (uint256 reserveA, uin

    function quote(uint256 amountA, uint256 reserveA, uint256 reserveB) external pure returns (uint25
```

```
    function getAmountOut(uint256 amountIn, uint256 reserveIn, uint256 reserveOut, address token0, ad

    function getAmountIn(uint256 amountOut, uint256 reserveIn, uint256 reserveOut, address token0, ad

    function getAmountsOut(uint256 amountIn, address[] calldata path) external view returns (uint256[

    function getAmountsIn(uint256 amountOut, address[] calldata path) external view returns (uint256[
}

interface IDexPair {
    event Approval(address indexed owner, address indexed spender, uint value);
    event Transfer(address indexed from, address indexed to, uint value);

    function name() external pure returns (string memory);

    function symbol() external pure returns (string memory);

    function decimals() external pure returns (uint8);

    function totalSupply() external view returns (uint);

    function balanceOf(address owner) external view returns (uint);

    function allowance(address owner, address spender) external view returns (uint);

    function approve(address spender, uint value) external returns (bool);

    function transfer(address to, uint value) external returns (bool);

    function transferFrom(address from, address to, uint value) external returns (bool);

    function DOMAIN_SEPARATOR() external view returns (bytes32);

    function PERMIT_TYPEHASH() external pure returns (bytes32);

    function nonces(address owner) external view returns (uint);

    function permit(address owner, address spender, uint value, uint deadline, uint8 v, bytes32 r, by

    event Mint(address indexed sender, uint amount0, uint amount1);
    event Burn(address indexed sender, uint amount0, uint amount1, address indexed to);
    event Swap(
        address indexed sender,
        uint amount0In,
        uint amount1In,
        uint amount0Out,
        uint amount1Out,
        address indexed to
    );
    event Sync(uint112 reserve0, uint112 reserve1);

    function MINIMUM_LIQUIDITY() external pure returns (uint);

    function factory() external view returns (address);

    function token0() external view returns (address);

    function token1() external view returns (address);

    function getReserves() external view returns (uint112 reserve0, uint112 reserve1, uint32 blockTim

    function price0CumulativeLast() external view returns (uint);

    function price1CumulativeLast() external view returns (uint);
```

```
    function kLast() external view returns (uint);

    function mint(address to) external returns (uint liquidity);

    function burn(address to) external returns (uint amount0, uint amount1);

    function swap(uint amount0Out, uint amount1Out, address to, bytes calldata data) external;

    function skim(address to) external;

    function sync() external;

    function initialize(address, address) external;
}

interface IDexERC20 {
    event Approval(address indexed owner, address indexed spender, uint value);
    event Transfer(address indexed from, address indexed to, uint value);

    function name() external pure returns (string memory);

    function symbol() external pure returns (string memory);

    function decimals() external pure returns (uint8);

    function totalSupply() external view returns (uint);

    function balanceOf(address owner) external view returns (uint);

    function allowance(address owner, address spender) external view returns (uint);

    function approve(address spender, uint value) external returns (bool);

    function transfer(address to, uint value) external returns (bool);

    function transferFrom(address from, address to, uint value) external returns (bool);

    function DOMAIN_SEPARATOR() external view returns (bytes32);

    function PERMIT_TYPEHASH() external pure returns (bytes32);

    function nonces(address owner) external view returns (uint);

    function permit(address owner, address spender, uint value, uint deadline, uint8 v, bytes32 r, by
}

interface IERC20 {
    event Approval(address indexed owner, address indexed spender, uint value);
    event Transfer(address indexed from, address indexed to, uint value);

    function name() external view returns (string memory);

    function symbol() external view returns (string memory);

    function decimals() external view returns (uint8);

    function totalSupply() external view returns (uint);

    function balanceOf(address owner) external view returns (uint);

    function allowance(address owner, address spender) external view returns (uint);

    function approve(address spender, uint value) external returns (bool);

    function transfer(address to, uint value) external returns (bool);
```

```
    function transferFrom(address from, address to, uint value) external returns (bool);
}

interface IswapV2Callee {
    function swapV2Call(address sender, uint amount0, uint amount1, bytes calldata data) external;
}

library SafeMath {
    uint256 constant WAD = 10 ** 18;
    uint256 constant RAY = 10 ** 27;

    function wad() public pure returns (uint256) {
        return WAD;
    }

    function ray() public pure returns (uint256) {
        return RAY;
    }

    function add(uint256 a, uint256 b) internal pure returns (uint256) {
        uint256 c = a + b;
        require(c >= a, "SafeMath: addition overflow");

        return c;
    }

    function sub(uint256 a, uint256 b) internal pure returns (uint256) {
        return sub(a, b, "SafeMath: subtraction overflow");
    }

    function sub(uint256 a, uint256 b, string memory errorMessage) internal pure returns (uint256) {
        require(b <= a, errorMessage);
        uint256 c = a - b;

        return c;
    }

    function mul(uint256 a, uint256 b) internal pure returns (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 0;
        }

        uint256 c = a * b;
        require(c / a == b, "SafeMath: multiplication overflow");

        return c;
    }

    function div(uint256 a, uint256 b) internal pure returns (uint256) {
        return div(a, b, "SafeMath: division by zero");
    }

    function div(uint256 a, uint256 b, string memory errorMessage) internal pure returns (uint256) {
        // Solidity only automatically asserts when dividing by 0
        require(b > 0, errorMessage);
        uint256 c = a / b;
        // assert(a == b * c + a % b); // There is no case in which this doesn't hold

        return c;
    }

    function mod(uint256 a, uint256 b) internal pure returns (uint256) {
        return mod(a, b, "SafeMath: modulo by zero");
```

```
    }

    function mod(uint256 a, uint256 b, string memory errorMessage) internal pure returns (uint256) {
        require(b != 0, errorMessage);
        return a % b;
    }

    function min(uint256 a, uint256 b) internal pure returns (uint256) {
        return a <= b ? a : b;
    }

    function max(uint256 a, uint256 b) internal pure returns (uint256) {
        return a >= b ? a : b;
    }

    function sqrt(uint256 a) internal pure returns (uint256 b) {
        if (a > 3) {
            b = a;
            uint256 x = a / 2 + 1;
            while (x < b) {
                b = x;
                x = (a / x + x) / 2;
            }
        } else if (a != 0) {
            b = 1;
        }
    }

    function wmul(uint256 a, uint256 b) internal pure returns (uint256) {
        return mul(a, b) / WAD;
    }

    function wmulRound(uint256 a, uint256 b) internal pure returns (uint256) {
        return add(mul(a, b), WAD / 2) / WAD;
    }

    function rmul(uint256 a, uint256 b) internal pure returns (uint256) {
        return mul(a, b) / RAY;
    }

    function rmulRound(uint256 a, uint256 b) internal pure returns (uint256) {
        return add(mul(a, b), RAY / 2) / RAY;
    }

    function wdiv(uint256 a, uint256 b) internal pure returns (uint256) {
        return div(mul(a, WAD), b);
    }

    function wdivRound(uint256 a, uint256 b) internal pure returns (uint256) {
        return add(mul(a, WAD), b / 2) / b;
    }

    function rdiv(uint256 a, uint256 b) internal pure returns (uint256) {
        return div(mul(a, RAY), b);
    }

    function rdivRound(uint256 a, uint256 b) internal pure returns (uint256) {
        return add(mul(a, RAY), b / 2) / b;
    }

    function wpow(uint256 x, uint256 n) internal pure returns (uint256) {
        uint256 result = WAD;
        while (n > 0) {
            if (n % 2 != 0) {
                result = wmul(result, x);
            }
```

```
            x = wmul(x, x);
            n /= 2;
        }
        return result;
    }

    function rpow(uint256 x, uint256 n) internal pure returns (uint256) {
        uint256 result = RAY;
        while (n > 0) {
            if (n % 2 != 0) {
                result = rmul(result, x);
            }
            x = rmul(x, x);
            n /= 2;
        }
        return result;
    }
}

library UQ112x112 {
    uint224 constant Q112 = 2 ** 112;

    // encode a uint112 as a UQ112x112
    function encode(uint112 y) internal pure returns (uint224 z) {
        z = uint224(y) * Q112;
        // never overflows
    }

    // divide a UQ112x112 by a uint112, returning a UQ112x112
    function uqdiv(uint224 x, uint112 y) internal pure returns (uint224 z) {
        z = x / uint224(y);
    }
}

library EnumerableSet {
    // To implement this library for multiple types with as little code
    // repetition as possible, we write it in terms of a generic Set type with
    // bytes32 values.
    // The Set implementation uses private functions, and user-facing
    // implementations (such as AddressSet) are just wrappers around the
    // underlying Set.
    // This means that we can only create new EnumerableSets for types that fit
    // in bytes32.

    struct Set {
        // Storage of set values
        bytes32[] _values;

        // Position of the value in the `values` array, plus 1 because index 0
        // means a value is not in the set.
        mapping(bytes32 => uint256) _indexes;
    }

    /**
     * @dev Add a value to a set. O(1).
     *
     * Returns true if the value was added to the set, that is if it was not
     * already present.
     */
    function _add(Set storage set, bytes32 value) private returns (bool) {
        if (!_contains(set, value)) {
            set._values.push(value);
            // The value is stored at length-1, but we add 1 to all indexes
            // and use 0 as a sentinel value
            set._indexes[value] = set._values.length;
            return true;
```

```
        } else {
            return false;
        }
    }

    /**
     * @dev Removes a value from a set. O(1).
     *
     * Returns true if the value was removed from the set, that is if it was
     * present.
     */
    function _remove(Set storage set, bytes32 value) private returns (bool) {
        // We read and store the value's index to prevent multiple reads from the same storage slot
        uint256 valueIndex = set._indexes[value];

        if (valueIndex != 0) {// Equivalent to contains(set, value)
            // To delete an element from the _values array in O(1), we swap the element to delete wit
            // the array, and then remove the last element (sometimes called as 'swap and pop').
            // This modifies the order of the array, as noted in {at}.

            uint256 toDeleteIndex = valueIndex - 1;
            uint256 lastIndex = set._values.length - 1;

            // When the value to delete is the last one, the swap operation is unnecessary. However,
            // so rarely, we still do the swap anyway to avoid the gas cost of adding an 'if' stateme

            bytes32 lastvalue = set._values[lastIndex];

            // Move the last value to the index where the value to delete is
            set._values[toDeleteIndex] = lastvalue;
            // Update the index for the moved value
            set._indexes[lastvalue] = toDeleteIndex + 1;
            // All indexes are 1-based

            // Delete the slot where the moved value was stored
            set._values.pop();

            // Delete the index for the deleted slot
            delete set._indexes[value];

            return true;
        } else {
            return false;
        }
    }

    /**
     * @dev Returns true if the value is in the set. O(1).
     */
    function _contains(Set storage set, bytes32 value) private view returns (bool) {
        return set._indexes[value] != 0;
    }

    /**
     * @dev Returns the number of values on the set. O(1).
     */
    function _length(Set storage set) private view returns (uint256) {
        return set._values.length;
    }

    /**
     * @dev Returns the value stored at position `index` in the set. O(1).
     *
     * Note that there are no guarantees on the ordering of values inside the
     * array, and it may change when more values are added or removed.
     *
```

```
     * Requirements:
     *
     * - `index` must be strictly less than {length}.
     */
    function _at(Set storage set, uint256 index) private view returns (bytes32) {
        require(set._values.length > index, "EnumerableSet: index out of bounds");
        return set._values[index];
    }

    // Bytes32Set

    struct Bytes32Set {
        Set _inner;
    }

    /**
     * @dev Add a value to a set. O(1).
     *
     * Returns true if the value was added to the set, that is if it was not
     * already present.
     */
    function add(Bytes32Set storage set, bytes32 value) internal returns (bool) {
        return _add(set._inner, value);
    }

    /**
     * @dev Removes a value from a set. O(1).
     *
     * Returns true if the value was removed from the set, that is if it was
     * present.
     */
    function remove(Bytes32Set storage set, bytes32 value) internal returns (bool) {
        return _remove(set._inner, value);
    }

    /**
     * @dev Returns true if the value is in the set. O(1).
     */
    function contains(Bytes32Set storage set, bytes32 value) internal view returns (bool) {
        return _contains(set._inner, value);
    }

    /**
     * @dev Returns the number of values in the set. O(1).
     */
    function length(Bytes32Set storage set) internal view returns (uint256) {
        return _length(set._inner);
    }

    /**
     * @dev Returns the value stored at position `index` in the set. O(1).
     *
     * Note that there are no guarantees on the ordering of values inside the
     * array, and it may change when more values are added or removed.
     *
     * Requirements:
     *
     * - `index` must be strictly less than {length}.
     */
    function at(Bytes32Set storage set, uint256 index) internal view returns (bytes32) {
        return _at(set._inner, index);
    }

    // AddressSet

    struct AddressSet {
```

```
        Set _inner;
    }

    /**
     * @dev Add a value to a set. O(1).
     *
     * Returns true if the value was added to the set, that is if it was not
     * already present.
     */
    function add(AddressSet storage set, address value) internal returns (bool) {
        return _add(set._inner, bytes32(uint256(value)));
    }

    /**
     * @dev Removes a value from a set. O(1).
     *
     * Returns true if the value was removed from the set, that is if it was
     * present.
     */
    function remove(AddressSet storage set, address value) internal returns (bool) {
        return _remove(set._inner, bytes32(uint256(value)));
    }

    /**
     * @dev Returns true if the value is in the set. O(1).
     */
    function contains(AddressSet storage set, address value) internal view returns (bool) {
        return _contains(set._inner, bytes32(uint256(value)));
    }

    /**
     * @dev Returns the number of values in the set. O(1).
     */
    function length(AddressSet storage set) internal view returns (uint256) {
        return _length(set._inner);
    }

    /**
     * @dev Returns the value stored at position `index` in the set. O(1).
     *
     * Note that there are no guarantees on the ordering of values inside the
     * array, and it may change when more values are added or removed.
     *
     * Requirements:
     *
     * - `index` must be strictly less than {length}.
     */
    function at(AddressSet storage set, uint256 index) internal view returns (address) {
        return address(uint256(_at(set._inner, index)));
    }


    // UintSet

    struct UintSet {
        Set _inner;
    }

    /**
     * @dev Add a value to a set. O(1).
     *
     * Returns true if the value was added to the set, that is if it was not
     * already present.
     */
    function add(UintSet storage set, uint256 value) internal returns (bool) {
        return _add(set._inner, bytes32(value));
```

```
    }

    /**
     * @dev Removes a value from a set. O(1).
     *
     * Returns true if the value was removed from the set, that is if it was
     * present.
     */
    function remove(UintSet storage set, uint256 value) internal returns (bool) {
        return _remove(set._inner, bytes32(value));
    }

    /**
     * @dev Returns true if the value is in the set. O(1).
     */
    function contains(UintSet storage set, uint256 value) internal view returns (bool) {
        return _contains(set._inner, bytes32(value));
    }

    /**
     * @dev Returns the number of values on the set. O(1).
     */
    function length(UintSet storage set) internal view returns (uint256) {
        return _length(set._inner);
    }

    /**
     * @dev Returns the value stored at position `index` in the set. O(1).
     *
     * Note that there are no guarantees on the ordering of values inside the
     * array, and it may change when more values are added or removed.
     *
     * Requirements:
     *
     * - `index` must be strictly less than {length}.
     */
    function at(UintSet storage set, uint256 index) internal view returns (uint256) {
        return uint256(_at(set._inner, index));
    }
}

contract DexERC20 is IDexERC20 {
    using SafeMath for uint;

    string public constant name = 'COCO LP Token';
    string public constant symbol = 'COCO LP';
    uint8 public constant decimals = 18;
    uint  public totalSupply;
    mapping(address => uint) public balanceOf;
    mapping(address => mapping(address => uint)) public allowance;

    bytes32 public DOMAIN_SEPARATOR;
    // keccak256("Permit(address owner,address spender,uint256 value,uint256 nonce,uint256 deadline)"
    bytes32 public constant PERMIT_TYPEHASH = 0x6e71edae12b1b97f4d1f60370fef10105fa2faae0126114a169c6
    mapping(address => uint) public nonces;

    event Approval(address indexed owner, address indexed spender, uint value);
    event Transfer(address indexed from, address indexed to, uint value);

    constructor() public {
        uint chainId;
        assembly {
            chainId := chainid
        }
        DOMAIN_SEPARATOR = keccak256(
            abi.encode(
```

```
                keccak256('EIP712Domain(string name,string version,uint256 chainId,address verifyingC
                keccak256(bytes(name)),
                keccak256(bytes('1')),
                chainId,
                address(this)
            )
        );
    }

    function _mint(address to, uint value) internal {
        totalSupply = totalSupply.add(value);
        balanceOf[to] = balanceOf[to].add(value);
        emit Transfer(address(0), to, value);
    }

    function _burn(address from, uint value) internal {
        balanceOf[from] = balanceOf[from].sub(value);
        totalSupply = totalSupply.sub(value);
        emit Transfer(from, address(0), value);
    }

    function _approve(address owner, address spender, uint value) private {
        allowance[owner][spender] = value;
        emit Approval(owner, spender, value);
    }

    function _transfer(address from, address to, uint value) private {
        balanceOf[from] = balanceOf[from].sub(value);
        balanceOf[to] = balanceOf[to].add(value);
        emit Transfer(from, to, value);
    }

    function approve(address spender, uint value) external returns (bool) {
        _approve(msg.sender, spender, value);
        return true;
    }

    function transfer(address to, uint value) external returns (bool) {
        _transfer(msg.sender, to, value);
        return true;
    }

    function transferFrom(address from, address to, uint value) external returns (bool) {
        if (allowance[from][msg.sender] != uint(- 1)) {
            allowance[from][msg.sender] = allowance[from][msg.sender].sub(value);
        }
        _transfer(from, to, value);
        return true;
    }

    function permit(address owner, address spender, uint value, uint deadline, uint8 v, bytes32 r, by
        require(deadline >= block.timestamp, 'DexSwap: EXPIRED');
        bytes32 digest = keccak256(
            abi.encodePacked(
                '\x19\x01',
                DOMAIN_SEPARATOR,
                keccak256(abi.encode(PERMIT_TYPEHASH, owner, spender, value, nonces[owner]++, deadlin
            )
        );
        address recoveredAddress = ecrecover(digest, v, r, s);
        require(recoveredAddress != address(0) && recoveredAddress == owner, 'DexSwap: INVALID_SIGNAT
        _approve(owner, spender, value);
    }
}

contract DexPair is IDexPair, DexERC20 {
```

```
using SafeMath  for uint;
using UQ112x112 for uint224;

uint public constant MINIMUM_LIQUIDITY = 10 ** 3;
bytes4 private constant SELECTOR = bytes4(keccak256(bytes('transfer(address,uint256)')));

address public factory;
address public token0;
address public token1;

uint112 private reserve0;           // uses single storage slot, accessible via getReserves
uint112 private reserve1;           // uses single storage slot, accessible via getReserves
uint32  private blockTimestampLast; // uses single storage slot, accessible via getReserves

uint public price0CumulativeLast;
uint public price1CumulativeLast;
uint public kLast; // reserve0 * reserve1, as of immediately after the most recent liquidity even

uint private unlocked = 1;
modifier lock() {
    require(unlocked == 1, 'DexSwap: LOCKED');
    unlocked = 0;
    _;
    unlocked = 1;
}

function getReserves() public view returns (uint112 _reserve0, uint112 _reserve1, uint32 _blockTi
    _reserve0 = reserve0;
    _reserve1 = reserve1;
    _blockTimestampLast = blockTimestampLast;
}

function _safeTransfer(address token, address to, uint value) private {
    (bool success, bytes memory data) = token.call(abi.encodeWithSelector(SELECTOR, to, value));
    require(success && (data.length == 0 || abi.decode(data, (bool))), 'DexSwap: TRANSFER_FAILED'
}

event Mint(address indexed sender, uint amount0, uint amount1);
event Burn(address indexed sender, uint amount0, uint amount1, address indexed to);
event Swap(
    address indexed sender,
    uint amount0In,
    uint amount1In,
    uint amount0Out,
    uint amount1Out,
    address indexed to
);
event Sync(uint112 reserve0, uint112 reserve1);

constructor() public {
    factory = msg.sender;
}

// called once by the factory at time of deployment
function initialize(address _token0, address _token1) external {
    require(msg.sender == factory, 'DexSwap: FORBIDDEN');
    // sufficient check
    token0 = _token0;
    token1 = _token1;
}

// update reserves and, on the first call per block, price accumulators
function _update(uint balance0, uint balance1, uint112 _reserve0, uint112 _reserve1) private {
    require(balance0 <= uint112(- 1) && balance1 <= uint112(- 1), 'DexSwap: OVERFLOW');
    uint32 blockTimestamp = uint32(block.timestamp % 2 ** 32);
    uint32 timeElapsed = blockTimestamp - blockTimestampLast;
```

```
        // overflow is desired
        if (timeElapsed > 0 && _reserve0 != 0 && _reserve1 != 0) {
            // * never overflows, and + overflow is desired
            price0CumulativeLast += uint(UQ112x112.encode(_reserve1).uqdiv(_reserve0)) * timeElapsed;
            price1CumulativeLast += uint(UQ112x112.encode(_reserve0).uqdiv(_reserve1)) * timeElapsed;
        }
        reserve0 = uint112(balance0);
        reserve1 = uint112(balance1);
        blockTimestampLast = blockTimestamp;
        emit Sync(reserve0, reserve1);
    }

    // if fee is on, mint liquidity equivalent to 1/6th of the growth in sqrt(k)
    function _mintFee(uint112 _reserve0, uint112 _reserve1) private returns (bool feeOn) {
        address feeTo = IDexFactory(factory).feeTo();
        feeOn = feeTo != address(0) && IDexFactory(factory).getPairRate(address(this)) != 99;
        uint _kLast = kLast;
        // gas savings
        if (feeOn) {
            if (_kLast != 0) {
                uint rootK = SafeMath.sqrt(uint(_reserve0).mul(_reserve1));
                uint rootKLast = SafeMath.sqrt(_kLast);
                if (rootK > rootKLast) {
                    uint numerator = totalSupply.mul(rootK.sub(rootKLast)).mul(10);
                    uint denominator = rootK.mul(IDexFactory(factory).getPairRate(address(this))).add
                    uint liquidity = numerator / denominator;
                    if (liquidity > 0) _mint(feeTo, liquidity);
                }
            }
        } else if (_kLast != 0) {
            kLast = 0;
        }
    }

    // this low-level function should be called from a contract which performs important safety check
    function mint(address to) external lock returns (uint liquidity) {
        (uint112 _reserve0, uint112 _reserve1,) = getReserves();
        // gas savings
        uint balance0 = IERC20(token0).balanceOf(address(this));
        uint balance1 = IERC20(token1).balanceOf(address(this));
        uint amount0 = balance0.sub(_reserve0);
        uint amount1 = balance1.sub(_reserve1);

        bool feeOn = _mintFee(_reserve0, _reserve1);
        uint _totalSupply = totalSupply;
        // gas savings, must be defined here since totalSupply can update in _mintFee
        if (_totalSupply == 0) {
            liquidity = SafeMath.sqrt(amount0.mul(amount1)).sub(MINIMUM_LIQUIDITY);
            _mint(address(0), MINIMUM_LIQUIDITY);
            // permanently lock the first MINIMUM_LIQUIDITY tokens
        } else {
            liquidity = SafeMath.min(amount0.mul(_totalSupply) / _reserve0, amount1.mul(_totalSupply)
        }
        require(liquidity > 0, 'DexSwap: INSUFFICIENT_LIQUIDITY_MINTED');
        _mint(to, liquidity);

        _update(balance0, balance1, _reserve0, _reserve1);
        if (feeOn) kLast = uint(reserve0).mul(reserve1);
        // reserve0 and reserve1 are up-to-date
        emit Mint(msg.sender, amount0, amount1);
    }

    // this low-level function should be called from a contract which performs important safety check
    function burn(address to) external lock returns (uint amount0, uint amount1) {
        (uint112 _reserve0, uint112 _reserve1,) = getReserves();
        // gas savings
```

```
        address _token0 = token0;
        // gas savings
        address _token1 = token1;
        // gas savings
        uint balance0 = IERC20(_token0).balanceOf(address(this));
        uint balance1 = IERC20(_token1).balanceOf(address(this));
        uint liquidity = balanceOf[address(this)];

        bool feeOn = _mintFee(_reserve0, _reserve1);
        uint _totalSupply = totalSupply;
        // gas savings, must be defined here since totalSupply can update in _mintFee
        amount0 = liquidity.mul(balance0) / _totalSupply;
        // using balances ensures pro-rata distribution
        amount1 = liquidity.mul(balance1) / _totalSupply;
        // using balances ensures pro-rata distribution
        require(amount0 > 0 && amount1 > 0, 'DexSwap: INSUFFICIENT_LIQUIDITY_BURNED');
        _burn(address(this), liquidity);
        _safeTransfer(_token0, to, amount0);
        _safeTransfer(_token1, to, amount1);
        balance0 = IERC20(_token0).balanceOf(address(this));
        balance1 = IERC20(_token1).balanceOf(address(this));

        _update(balance0, balance1, _reserve0, _reserve1);
        if (feeOn) kLast = uint(reserve0).mul(reserve1);
        // reserve0 and reserve1 are up-to-date
        emit Burn(msg.sender, amount0, amount1, to);
    }

    // this low-level function should be called from a contract which performs important safety check
    function swap(uint amount0Out, uint amount1Out, address to, bytes calldata data) external lock {
        require(amount0Out > 0 || amount1Out > 0, 'DexSwap: INSUFFICIENT_OUTPUT_AMOUNT');
        (uint112 _reserve0, uint112 _reserve1,) = getReserves();
        // gas savings
        require(amount0Out < _reserve0 && amount1Out < _reserve1, 'DexSwap: INSUFFICIENT_LIQUIDITY');

        uint balance0;
        uint balance1;
        {// scope for _token{0,1}, avoids stack too deep errors
            address _token0 = token0;
            address _token1 = token1;
            require(to != _token0 && to != _token1, 'DexSwap: INVALID_TO');
            if (amount0Out > 0) _safeTransfer(_token0, to, amount0Out);
            // optimistically transfer tokens
            if (amount1Out > 0) _safeTransfer(_token1, to, amount1Out);
            // optimistically transfer tokens
            if (data.length > 0) IswapV2Callee(to).swapV2Call(msg.sender, amount0Out, amount1Out, dat
            balance0 = IERC20(_token0).balanceOf(address(this));
            balance1 = IERC20(_token1).balanceOf(address(this));
        }
        uint amount0In = balance0 > _reserve0 - amount0Out ? balance0 - (_reserve0 - amount0Out) : 0;
        uint amount1In = balance1 > _reserve1 - amount1Out ? balance1 - (_reserve1 - amount1Out) : 0;
        require(amount0In > 0 || amount1In > 0, 'DexSwap: INSUFFICIENT_INPUT_AMOUNT');
        {// scope for reserve{0,1}Adjusted, avoids stack too deep errors
            uint balance0Adjusted = balance0.mul(1e4).sub(amount0In.mul(IDexFactory(factory).getPairF
            uint balance1Adjusted = balance1.mul(1e4).sub(amount1In.mul(IDexFactory(factory).getPairF
            require(balance0Adjusted.mul(balance1Adjusted) >= uint(_reserve0).mul(_reserve1).mul(1e8)
        }

        _update(balance0, balance1, _reserve0, _reserve1);
        emit Swap(msg.sender, amount0In, amount1In, amount0Out, amount1Out, to);
    }

    // force balances to match reserves
    function skim(address to) external lock {
        address _token0 = token0;
        // gas savings
```

```
        address _token1 = token1;
        // gas savings
        _safeTransfer(_token0, to, IERC20(_token0).balanceOf(address(this)).sub(reserve0));
        _safeTransfer(_token1, to, IERC20(_token1).balanceOf(address(this)).sub(reserve1));
    }

    // force reserves to match balances
    function sync() external lock {
        _update(IERC20(token0).balanceOf(address(this)), IERC20(token1).balanceOf(address(this)), res
    }

}

contract DexFactory is IDexFactory {
    using SafeMath for uint256;
    using EnumerableSet for EnumerableSet.AddressSet;
    EnumerableSet.AddressSet private _supportList;

    uint256 public constant FEE_RATE_DENOMINATOR = 1e4;
    uint256 public feeRateNumerator = 30;
    address public feeTo;
    address public feeToSetter;
    uint256 public feeToRate = 5;
    bytes32 public initCodeHash;

    mapping(address => uint256) public pairFeeToRate;
    mapping(address => uint256) public pairFees;
    mapping(address => mapping(address => address)) public getPair;
    address[] public allPairs;

    event PairCreated(address indexed token0, address indexed token1, address pair, uint);

    constructor(address _feeToSetter) public {
        feeToSetter = _feeToSetter;
        initCodeHash = keccak256(abi.encodePacked(type(DexPair).creationCode));
    }

    function allPairsLength() external view returns (uint) {
        return allPairs.length;
    }

    function createPair(address tokenA, address tokenB) external returns (address pair) {
        require(tokenA != tokenB, 'DexSwapFactory: IDENTICAL_ADDRESSES');
        (address token0, address token1) = tokenA < tokenB ? (tokenA, tokenB) : (tokenB, tokenA);
        require(token0 != address(0), 'DexSwapFactory: ZERO_ADDRESS');
        require(getPair[token0][token1] == address(0), 'DexSwapFactory: PAIR_EXISTS');
        // single check is sufficient
        bytes memory bytecode = type(DexPair).creationCode;
        bytes32 salt = keccak256(abi.encodePacked(token0, token1));
        assembly {
            pair := create2(0, add(bytecode, 32), mload(bytecode), salt)
        }
        IDexPair(pair).initialize(token0, token1);
        getPair[token0][token1] = pair;
        getPair[token1][token0] = pair;
        // populate mapping in the reverse direction
        allPairs.push(pair);
        emit PairCreated(token0, token1, pair, allPairs.length);
    }

    function setFeeTo(address _feeTo) external {
        require(msg.sender == feeToSetter, 'DexSwapFactory: FORBIDDEN');
        feeTo = _feeTo;
    }

    function setFeeToSetter(address _feeToSetter) external {
```

```
        require(msg.sender == feeToSetter, 'DexSwapFactory: FORBIDDEN');
        require(_feeToSetter != address(0), "DexSwapFactory: FeeToSetter is zero address");
        feeToSetter = _feeToSetter;
    }

    function addPair(address pair) external returns (bool){
        require(msg.sender == feeToSetter, 'DexSwapFactory: FORBIDDEN');
        require(pair != address(0), 'DexSwapFactory: pair is the zero address');
        return EnumerableSet.add(_supportList, pair);
    }

    function delPair(address pair) external returns (bool){
        require(msg.sender == feeToSetter, 'DexSwapFactory: FORBIDDEN');
        require(pair != address(0), 'DexSwapFactory: pair is the zero address');
        return EnumerableSet.remove(_supportList, pair);
    }

    function getSupportListLength() public view returns (uint256) {
        return EnumerableSet.length(_supportList);
    }

    function isSupportPair(address pair) public view returns (bool){
        return EnumerableSet.contains(_supportList, pair);
    }

    function getSupportPair(uint256 index) external view returns (address) {
        require(msg.sender == feeToSetter, 'DexSwapFactory: FORBIDDEN');
        require(index <= getSupportListLength() - 1, "index out of bounds");
        return EnumerableSet.at(_supportList, index);
    }

    // Set default fee  ,max is 0.003%
    function setFeeRateNumerator(uint256 _feeRateNumerator) external {
        require(msg.sender == feeToSetter, 'DexSwapFactory: FORBIDDEN');
        require(_feeRateNumerator <= 30, "DexSwapFactory: EXCEEDS_FEE_RATE_DENOMINATOR");
        feeRateNumerator = _feeRateNumerator;
    }

    // Set pair fee , max is 0.003%
    function setPairFees(address pair, uint256 fee) external {
        require(msg.sender == feeToSetter, 'DexSwapFactory: FORBIDDEN');
        require(fee <= 30, 'DexSwapFactory: EXCEEDS_FEE_RATE_DENOMINATOR');
        pairFees[pair] = fee;
    }

    // Set the default fee rate  ,if set to 1/100 no handling fee. ** should multi by 10 **
    function setDefaultFeeToRate(uint256 rate) external {
        require(msg.sender == feeToSetter, 'DexSwapFactory: FORBIDDEN');
        require(rate > 0 && rate <= 100, "DexSwapFactory: FEE_TO_RATE_OVERFLOW");
        feeToRate = rate.sub(1);
    }

    // Set the commission rate of the pair  ,if set to 1/10 no handling fee. ** should multi by 10 **
    function setPairFeeToRate(address pair, uint256 rate) external {
        require(msg.sender == feeToSetter, 'DexSwapFactory: FORBIDDEN');
        require(rate > 0 && rate <= 100, "DexSwapFactory: FEE_TO_RATE_OVERFLOW");
        pairFeeToRate[pair] = rate.sub(1);
    }

    function getPairFees(address pair) public view returns (uint256){
        require(pair != address(0), 'DexSwapFactory: pair is the zero address');
        if (isSupportPair(pair)) {
            return pairFees[pair];
        } else {
            return feeRateNumerator;
        }
```

```
    }

    function getPairRate(address pair) external view returns (uint256) {
        require(pair != address(0), 'DexSwapFactory: pair is the zero address');
        if (isSupportPair(pair)) {
            return pairFeeToRate[pair];
        } else {
            return feeToRate;
        }
    }

    // returns sorted token addresses, used to handle return values from pairs sorted in this order
    function sortTokens(address tokenA, address tokenB) public pure returns (address token0, address
        require(tokenA != tokenB, 'DexSwapFactory: IDENTICAL_ADDRESSES');
        (token0, token1) = tokenA < tokenB ? (tokenA, tokenB) : (tokenB, tokenA);
        require(token0 != address(0), 'DexSwapFactory: ZERO_ADDRESS');
    }

    // calculates the CREATE2 address for a pair without making any external calls
    function pairFor(address tokenA, address tokenB) public view returns (address pair) {
        (address token0, address token1) = sortTokens(tokenA, tokenB);
        pair = address(uint(keccak256(abi.encodePacked(
                hex'ff',
                address(this),
                keccak256(abi.encodePacked(token0, token1)),
                initCodeHash
            ))));
    }

    // fetches and sorts the reserves for a pair
    function getReserves(address tokenA, address tokenB) public view returns (uint reserveA, uint res
        (address token0,) = sortTokens(tokenA, tokenB);
        (uint reserve0, uint reserve1,) = IDexPair(pairFor(tokenA, tokenB)).getReserves();
        (reserveA, reserveB) = tokenA == token0 ? (reserve0, reserve1) : (reserve1, reserve0);
    }

    // given some amount of an asset and pair reserves, returns an equivalent amount of the other ass
    function quote(uint amountA, uint reserveA, uint reserveB) public pure returns (uint amountB) {
        require(amountA > 0, 'DexSwapFactory: INSUFFICIENT_AMOUNT');
        require(reserveA > 0 && reserveB > 0, 'DexSwapFactory: INSUFFICIENT_LIQUIDITY');
        amountB = amountA.mul(reserveB) / reserveA;
    }

    // given an input amount of an asset and pair reserves, returns the maximum output amount of the
    function getAmountOut(uint amountIn, uint reserveIn, uint reserveOut, address token0, address tok
        require(amountIn > 0, 'DexSwapFactory: INSUFFICIENT_INPUT_AMOUNT');
        require(reserveIn > 0 && reserveOut > 0, 'DexSwapFactory: INSUFFICIENT_LIQUIDITY');
        uint256 fee = getPairFees(pairFor(token0, token1));
        uint amountInWithFee = amountIn.mul(FEE_RATE_DENOMINATOR.sub(fee));
        uint numerator = amountInWithFee.mul(reserveOut);
        uint denominator = reserveIn.mul(FEE_RATE_DENOMINATOR).add(amountInWithFee);
        amountOut = numerator / denominator;
    }

    // given an output amount of an asset and pair reserves, returns a required input amount of the o
    function getAmountIn(uint amountOut, uint reserveIn, uint reserveOut, address token0, address tok
        require(amountOut > 0, 'DexSwapFactory: INSUFFICIENT_OUTPUT_AMOUNT');
        require(reserveIn > 0 && reserveOut > 0, 'DexSwapFactory: INSUFFICIENT_LIQUIDITY');
        uint256 fee = getPairFees(pairFor(token0, token1));
        uint numerator = reserveIn.mul(amountOut).mul(FEE_RATE_DENOMINATOR);
        uint denominator = reserveOut.sub(amountOut).mul(FEE_RATE_DENOMINATOR.sub(fee));
        amountIn = (numerator / denominator).add(1);
    }

    // performs chained getAmountOut calculations on any number of pairs
    function getAmountsOut(uint amountIn, address[] memory path) public view returns (uint[] memory a
```

```
        require(path.length >= 2, 'DexSwapFactory: INVALID_PATH');
        amounts = new uint[](path.length);
        amounts[0] = amountIn;
        for (uint i; i < path.length - 1; i++) {
            (uint reserveIn, uint reserveOut) = getReserves(path[i], path[i + 1]);
            amounts[i + 1] = getAmountOut(amounts[i], reserveIn, reserveOut, path[i], path[i + 1]);
        }
    }

    // performs chained getAmountIn calculations on any number of pairs
    function getAmountsIn(uint amountOut, address[] memory path) public view returns (uint[] memory a
        require(path.length >= 2, 'DexSwapFactory: INVALID_PATH');
        amounts = new uint[](path.length);
        amounts[amounts.length - 1] = amountOut;
        for (uint i = path.length - 1; i > 0; i--) {
            (uint reserveIn, uint reserveOut) = getReserves(path[i - 1], path[i]);
            amounts[i - 1] = getAmountIn(amounts[i], reserveIn, reserveOut, path[i - 1], path[i]);
        }
    }
}// SPDX-License-Identifier: MIT
pragma solidity ^0.6.0;

/**
 * @dev Interface of the ERC20 standard as defined in the EIP.
 */
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);
}


pragma solidity ^0.6.0;

/**
 * @dev Wrappers over Solidity's arithmetic operations with added overflow
 * checks.
 *
 * Arithmetic operations in Solidity wrap on overflow. This can easily result
 * in bugs, because programmers usually assume that an overflow raises an
 * error, which is the standard behavior in high level programming languages.
 * `SafeMath` restores this intuition by reverting the transaction when an
 * operation overflows.
 *
 * Using this library instead of the unchecked operations eliminates an entire
 * class of bugs, so it's recommended to use it always.
 */
library SafeMath {
    /**
     * @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) {
        return sub(a, b, "SafeMath: subtraction overflow");
    }

    /**
     * @dev Returns the subtraction of two unsigned integers, reverting with custom message on
     * overflow (when the result is negative).
     *
     * 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);
        uint256 c = a - b;

        return c;
    }

    /**
     * @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) {
        // 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 0;
        }

        uint256 c = a * b;
        require(c / a == b, "SafeMath: multiplication overflow");

        return c;
    }

    /**
     * @dev Returns the integer division of two unsigned integers. Reverts 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) {
```

```
        return div(a, b, "SafeMath: division by zero");
    }

    /**
     * @dev Returns the integer division of two unsigned integers. Reverts with custom message 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, string memory errorMessage) internal pure returns (uint256) {
        require(b > 0, errorMessage);
        uint256 c = a / b;
        // assert(a == b * c + a % b); // There is no case in which this doesn't hold

        return c;
    }

    /**
     * @dev Returns the remainder of dividing two unsigned integers. (unsigned integer modulo),
     * Reverts 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) {
        return mod(a, b, "SafeMath: modulo by zero");
    }

    /**
     * @dev Returns the remainder of dividing two unsigned integers. (unsigned integer modulo),
     * Reverts with custom message 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, string memory errorMessage) internal pure returns (uint256) {
        require(b != 0, errorMessage);
        return a % b;
    }
}


pragma solidity ^0.6.2;

/**
 * @dev Collection of functions related to the address type
 */
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
     */
    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.de
     *
     * 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 ret
        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
        return functionCallWithValue(target, data, value, "Address: low-level call with value failed"
    }

    /**
     * @dev Same as {xref-Address-functionCallWithValue-address-bytes-uint256-}[`functionCallWithValu
     * 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 er
        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 memor
        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) intern
        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);
    }

    function _verifyCallResult(bool success, bytes memory returndata, string memory errorMessage) pri
        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);
            }
        }
    }
}



pragma solidity ^0.6.0;

/**
 * @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, valu
    }

    /**
     * @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, newAllowan
    }

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

    /**
     * @dev Imitates a Solidity high-level call (i.e. a regular function call to a contract), relaxin
     * on the return value: the return value is optional (but if data is returned, it must not be fal
     * @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 m
        // we're implementing it ourselves. We use {Address.functionCall} to perform this call, which
        // the target address contains contract code and also asserts for success in the low-level ca

        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");
        }
    }
}


pragma solidity ^0.6.0;

/*
 * @dev Provides information about the current execution context, including the
 * sender of the transaction and its data. While these are generally available
 * via msg.sender and msg.data, they should not be accessed in such a direct
 * manner, since when dealing with GSN meta-transactions the account sending and
 * paying for execution may not be the actual sender (as far as an application
 * is concerned).
 *
 * This contract is only required for intermediate, library-like contracts.
 */
abstract contract Context {
    function _msgSender() internal view virtual returns (address payable) {
        return msg.sender;
    }

    function _msgData() internal view virtual returns (bytes memory) {
        this;
        // silence state mutability warning without generating bytecode - see https://github.com/ethe
        return msg.data;
    }
}


pragma solidity ^0.6.0;

/**
 * @dev Contract module which provides a basic access control mechanism, where
 * there is an account (an owner) that can be granted exclusive access to
 * specific functions.
 *
 * By default, the owner account will be the one that deploys the contract. This
 * can later be changed with {transferOwnership}.
 *
```

```
 * This module is used through inheritance. It will make available the modifier
 * `onlyOwner`, which can be applied to your functions to restrict their use to
 * the owner.
 */
contract Ownable is Context {
    address private _owner;

    event OwnershipTransferred(address indexed previousOwner, address indexed newOwner);

    /**
     * @dev Initializes the contract setting the deployer as the initial owner.
     */
    constructor () internal {
        address msgSender = _msgSender();
        _owner = msgSender;
        emit OwnershipTransferred(address(0), msgSender);
    }

    /**
     * @dev Returns the address of the current owner.
     */
    function owner() public view returns (address) {
        return _owner;
    }

    /**
     * @dev Throws if called by any account other than the owner.
     */
    modifier onlyOwner() {
        require(_owner == _msgSender(), "Ownable: caller is not the owner");
        _;
    }

    /**
     * @dev Leaves the contract without owner. It will not be possible to call
     * `onlyOwner` functions anymore. Can only be called by the current owner.
     *
     * NOTE: Renouncing ownership will leave the contract without an owner,
     * thereby removing any functionality that is only available to the owner.
     */
    function renounceOwnership() public virtual onlyOwner {
        emit OwnershipTransferred(_owner, address(0));
        _owner = address(0);
    }

    /**
     * @dev Transfers ownership of the contract to a new account (`newOwner`).
     * Can only be called by the current owner.
     */
    function transferOwnership(address newOwner) public virtual onlyOwner {
        require(newOwner != address(0), "Ownable: new owner is the zero address");
        emit OwnershipTransferred(_owner, newOwner);
        _owner = newOwner;
    }
}


pragma solidity ^0.6.0;

/**
 * @dev Library for managing
 * https://en.wikipedia.org/wiki/Set_(abstract_data_type)[sets] of primitive
 * types.
 *
 * Sets have the following properties:
 *
```

```
 * - Elements are added, removed, and checked for existence in constant time
 * (O(1)).
 * - Elements are enumerated in O(n). No guarantees are made on the ordering.
 *
 * ``
 * contract Example {
 *     // Add the library methods
 *     using EnumerableSet for EnumerableSet.AddressSet;
 *
 *     // Declare a set state variable
 *     EnumerableSet.AddressSet private mySet;
 * }
 * ``
 *
 * As of v3.0.0, only sets of type `address` (`AddressSet`) and `uint256`
 * (`UintSet`) are supported.
 */
library EnumerableSet {
    // To implement this library for multiple types with as little code
    // repetition as possible, we write it in terms of a generic Set type with
    // bytes32 values.
    // The Set implementation uses private functions, and user-facing
    // implementations (such as AddressSet) are just wrappers around the
    // underlying Set.
    // This means that we can only create new EnumerableSets for types that fit
    // in bytes32.

    struct Set {
        // Storage of set values
        bytes32[] _values;

        // Position of the value in the `values` array, plus 1 because index 0
        // means a value is not in the set.
        mapping(bytes32 => uint256) _indexes;
    }

    /**
     * @dev Add a value to a set. O(1).
     *
     * Returns true if the value was added to the set, that is if it was not
     * already present.
     */
    function _add(Set storage set, bytes32 value) private returns (bool) {
        if (!_contains(set, value)) {
            set._values.push(value);
            // The value is stored at length-1, but we add 1 to all indexes
            // and use 0 as a sentinel value
            set._indexes[value] = set._values.length;
            return true;
        } else {
            return false;
        }
    }

    /**
     * @dev Removes a value from a set. O(1).
     *
     * Returns true if the value was removed from the set, that is if it was
     * present.
     */
    function _remove(Set storage set, bytes32 value) private returns (bool) {
        // We read and store the value's index to prevent multiple reads from the same storage slot
        uint256 valueIndex = set._indexes[value];

        if (valueIndex != 0) {// Equivalent to contains(set, value)
            // To delete an element from the _values array in O(1), we swap the element to delete wit
```

```
        // the array, and then remove the last element (sometimes called as 'swap and pop').
        // This modifies the order of the array, as noted in {at}.

        uint256 toDeleteIndex = valueIndex - 1;
        uint256 lastIndex = set._values.length - 1;

        // When the value to delete is the last one, the swap operation is unnecessary. However,
        // so rarely, we still do the swap anyway to avoid the gas cost of adding an 'if' stateme

        bytes32 lastvalue = set._values[lastIndex];

        // Move the last value to the index where the value to delete is
        set._values[toDeleteIndex] = lastvalue;
        // Update the index for the moved value
        set._indexes[lastvalue] = toDeleteIndex + 1;
        // All indexes are 1-based

        // Delete the slot where the moved value was stored
        set._values.pop();

        // Delete the index for the deleted slot
        delete set._indexes[value];

        return true;
    } else {
        return false;
    }
}

/**
 * @dev Returns true if the value is in the set. O(1).
 */
function _contains(Set storage set, bytes32 value) private view returns (bool) {
    return set._indexes[value] != 0;
}

/**
 * @dev Returns the number of values on the set. O(1).
 */
function _length(Set storage set) private view returns (uint256) {
    return set._values.length;
}

/**
 * @dev Returns the value stored at position `index` in the set. O(1).
 *
 * Note that there are no guarantees on the ordering of values inside the
 * array, and it may change when more values are added or removed.
 *
 * Requirements:
 *
 * - `index` must be strictly less than {length}.
 */
function _at(Set storage set, uint256 index) private view returns (bytes32) {
    require(set._values.length > index, "EnumerableSet: index out of bounds");
    return set._values[index];
}

// AddressSet

struct AddressSet {
    Set _inner;
}

/**
 * @dev Add a value to a set. O(1).
```

```
     *
     * Returns true if the value was added to the set, that is if it was not
     * already present.
     */
    function add(AddressSet storage set, address value) internal returns (bool) {
        return _add(set._inner, bytes32(uint256(value)));
    }

    /**
     * @dev Removes a value from a set. O(1).
     *
     * Returns true if the value was removed from the set, that is if it was
     * present.
     */
    function remove(AddressSet storage set, address value) internal returns (bool) {
        return _remove(set._inner, bytes32(uint256(value)));
    }

    /**
     * @dev Returns true if the value is in the set. O(1).
     */
    function contains(AddressSet storage set, address value) internal view returns (bool) {
        return _contains(set._inner, bytes32(uint256(value)));
    }

    /**
     * @dev Returns the number of values in the set. O(1).
     */
    function length(AddressSet storage set) internal view returns (uint256) {
        return _length(set._inner);
    }

    /**
     * @dev Returns the value stored at position `index` in the set. O(1).
     *
     * Note that there are no guarantees on the ordering of values inside the
     * array, and it may change when more values are added or removed.
     *
     * Requirements:
     *
     * - `index` must be strictly less than {length}.
     */
    function at(AddressSet storage set, uint256 index) internal view returns (address) {
        return address(uint256(_at(set._inner, index)));
    }


    // UintSet

    struct UintSet {
        Set _inner;
    }

    /**
     * @dev Add a value to a set. O(1).
     *
     * Returns true if the value was added to the set, that is if it was not
     * already present.
     */
    function add(UintSet storage set, uint256 value) internal returns (bool) {
        return _add(set._inner, bytes32(value));
    }

    /**
     * @dev Removes a value from a set. O(1).
     *
```

```
     * Returns true if the value was removed from the set, that is if it was
     * present.
     */
    function remove(UintSet storage set, uint256 value) internal returns (bool) {
        return _remove(set._inner, bytes32(value));
    }

    /**
     * @dev Returns true if the value is in the set. O(1).
     */
    function contains(UintSet storage set, uint256 value) internal view returns (bool) {
        return _contains(set._inner, bytes32(value));
    }

    /**
     * @dev Returns the number of values on the set. O(1).
     */
    function length(UintSet storage set) internal view returns (uint256) {
        return _length(set._inner);
    }

    /**
     * @dev Returns the value stored at position `index` in the set. O(1).
     *
     * Note that there are no guarantees on the ordering of values inside the
     * array, and it may change when more values are added or removed.
     *
     * Requirements:
     *
     * - `index` must be strictly less than {length}.
     */
    function at(UintSet storage set, uint256 index) internal view returns (uint256) {
        return uint256(_at(set._inner, index));
    }
}

interface IDex is IERC20 {
    function mint(address to, uint256 amount) external returns (bool);
}

interface IMasterChef {
    function pending(uint256 pid, address user) external view returns (uint256);

    function deposit(uint256 pid, uint256 amount) external;

    function withdraw(uint256 pid, uint256 amount) external;

    function emergencyWithdraw(uint256 pid) external;
}

contract HecoPool is Ownable {
    using SafeMath for uint256;
    using SafeERC20 for IERC20;

    using EnumerableSet for EnumerableSet.AddressSet;
    EnumerableSet.AddressSet private _multLP;
    EnumerableSet.AddressSet private _blackList;

    // Info of each user.
    struct UserInfo {
        uint256 amount;     // How many LP tokens the user has provided.
        uint256 rewardDebt; // Reward debt.
        uint256 multLpRewardDebt; //multLp Reward debt.
    }

    // Info of each pool.
```

```
    struct PoolInfo {
        IERC20 lpToken;           // Address of LP token contract.
        uint256 allocPoint;       // How many allocation points assigned to this pool. DEXs to distri
        uint256 lastRewardBlock;  // Last block number that DEXs distribution occurs.
        uint256 accDexPerShare; // Accumulated DEXs per share, times 1e12.
        uint256 accMultLpPerShare; //Accumulated multLp per share
        uint256 totalAmount;    // Total amount of current pool deposit.
    }

    // The DEX Token!
    IDex public dex;
    // DEX tokens created per block.
    uint256 public dexPerBlock;
    // Info of each pool.
    PoolInfo[] public poolInfo;
    // Info of each user that stakes LP tokens.
    mapping(uint256 => mapping(address => UserInfo)) public userInfo;
    // Corresponding to the pid of the multLP pool
    mapping(uint256 => uint256) public poolCorrespond;
    // pid corresponding address
    mapping(address => uint256) public LpOfPid;
    // Control mining
    bool public paused = false;
    // Total allocation points. Must be the sum of all allocation points in all pools.
    uint256 public totalAllocPoint = 0;
    // The block number when DEX mining starts.
    uint256 public startBlock;
    // multLP MasterChef
    address public multLpChef;
    // multLP Token
    address public multLpToken;
    // How many blocks are halved
    uint256 public halvingPeriod = 1670400;

    event Deposit(address indexed user, uint256 indexed pid, uint256 amount);
    event Withdraw(address indexed user, uint256 indexed pid, uint256 amount);
    event EmergencyWithdraw(address indexed user, uint256 indexed pid, uint256 amount);

    constructor(
        IDex _dex,
        uint256 _dexPerBlock,
        uint256 _startBlock
    ) public {
        dex = _dex;
        dexPerBlock = _dexPerBlock;
        startBlock = _startBlock;
    }

    function setHalvingPeriod(uint256 _block) public onlyOwner {
        halvingPeriod = _block;
    }

    // Set the number of dex produced by each block
    function setDexPerBlock(uint256 newPerBlock) public onlyOwner {
        massUpdatePools();
        dexPerBlock = newPerBlock;
    }

    function setStartBlock(uint256 _startBlock) public onlyOwner {
        startBlock = _startBlock;
    }

    function poolLength() public view returns (uint256) {
        return poolInfo.length;
    }
```

```
    function addBadAddress(address _bad) public onlyOwner returns (bool) {
        require(_bad != address(0), "_bad is the zero address");
        return EnumerableSet.add(_blackList, _bad);
    }

    function delBadAddress(address _bad) public onlyOwner returns (bool) {
        require(_bad != address(0), "_bad is the zero address");
        return EnumerableSet.remove(_blackList, _bad);
    }

    function getBlackListLength() public view returns (uint256) {
        return EnumerableSet.length(_blackList);
    }

    function isBadAddress(address account) public view returns (bool) {
        return EnumerableSet.contains(_blackList, account);
    }

    function getBadAddress(uint256 _index) public view onlyOwner returns (address){
        require(_index <= getBlackListLength() - 1, "index out of bounds");
        return EnumerableSet.at(_blackList, _index);
    }

    function addMultLP(address _addLP) public onlyOwner returns (bool) {
        require(_addLP != address(0), "LP is the zero address");
        IERC20(_addLP).approve(multLpChef, uint256(- 1));
        return EnumerableSet.add(_multLP, _addLP);
    }

    function isMultLP(address _LP) public view returns (bool) {
        return EnumerableSet.contains(_multLP, _LP);
    }

    function getMultLPLength() public view returns (uint256) {
        return EnumerableSet.length(_multLP);
    }

    function getMultLPAddress(uint256 _pid) public view returns (address){
        require(_pid <= getMultLPLength() - 1, "not find this multLP");
        return EnumerableSet.at(_multLP, _pid);
    }

    function setPause() public onlyOwner {
        paused = !paused;
    }

    function setMultLP(address _multLpToken, address _multLpChef) public onlyOwner {
        require(_multLpToken != address(0) && _multLpChef != address(0), "is the zero address");
        multLpToken = _multLpToken;
        multLpChef = _multLpChef;
    }

    function replaceMultLP(address _multLpToken, address _multLpChef) public onlyOwner {
        require(_multLpToken != address(0) && _multLpChef != address(0), "is the zero address");
        require(paused == true, "No mining suspension");
        multLpToken = _multLpToken;
        multLpChef = _multLpChef;
        uint256 length = getMultLPLength();
        while (length > 0) {
            address dAddress = EnumerableSet.at(_multLP, 0);
            uint256 pid = LpOfPid[dAddress];
            IMasterChef(multLpChef).emergencyWithdraw(poolCorrespond[pid]);
            EnumerableSet.remove(_multLP, dAddress);
            length--;
        }
    }
```

```
// Add a new lp to the pool. Can only be called by the owner.
// XXX DO NOT add the same LP token more than once. Rewards will be messed up if you do.
function add(uint256 _allocPoint, IERC20 _lpToken, bool _withUpdate) public onlyOwner {
    require(address(_lpToken) != address(0), "_lpToken is the zero address");
    if (_withUpdate) {
        massUpdatePools();
    }
    uint256 lastRewardBlock = block.number > startBlock ? block.number : startBlock;
    totalAllocPoint = totalAllocPoint.add(_allocPoint);
    poolInfo.push(PoolInfo({
    lpToken : _lpToken,
    allocPoint : _allocPoint,
    lastRewardBlock : lastRewardBlock,
    accDexPerShare : 0,
    accMultLpPerShare : 0,
    totalAmount : 0
    }));
    LpOfPid[address(_lpToken)] = poolLength() - 1;
}

// Update the given pool's DEX allocation point. Can only be called by the owner.
function set(uint256 _pid, uint256 _allocPoint, bool _withUpdate) public onlyOwner {
    if (_withUpdate) {
        massUpdatePools();
    }
    totalAllocPoint = totalAllocPoint.sub(poolInfo[_pid].allocPoint).add(_allocPoint);
    poolInfo[_pid].allocPoint = _allocPoint;
}

// The current pool corresponds to the pid of the multLP pool
function setPoolCorr(uint256 _pid, uint256 _sid) public onlyOwner {
    require(_pid <= poolLength() - 1, "not find this pool");
    poolCorrespond[_pid] = _sid;
}

function phase(uint256 blockNumber) public view returns (uint256) {
    if (halvingPeriod == 0) {
        return 0;
    }
    if (blockNumber > startBlock) {
        return (blockNumber.sub(startBlock).sub(1)).div(halvingPeriod);
    }
    return 0;
}

function reward(uint256 blockNumber) public view returns (uint256) {
    uint256 _phase = phase(blockNumber);
    return dexPerBlock.div(2 ** _phase);
}

function getDexBlockReward(uint256 _lastRewardBlock) public view returns (uint256) {
    uint256 blockReward = 0;
    uint256 n = phase(_lastRewardBlock);
    uint256 m = phase(block.number);
    while (n < m) {
        n++;
        uint256 r = n.mul(halvingPeriod).add(startBlock);
        blockReward = blockReward.add((r.sub(_lastRewardBlock)).mul(reward(r)));
        _lastRewardBlock = r;
    }
    blockReward = blockReward.add((block.number.sub(_lastRewardBlock)).mul(reward(block.number)))
    return blockReward;
}

// Update reward variables for all pools. Be careful of gas spending!
```

```
    function massUpdatePools() public {
        uint256 length = poolInfo.length;
        for (uint256 pid = 0; pid < length; ++pid) {
            updatePool(pid);
        }
    }

    // Update reward variables of the given pool to be up-to-date.
    function updatePool(uint256 _pid) public {
        PoolInfo storage pool = poolInfo[_pid];
        if (block.number <= pool.lastRewardBlock) {
            return;
        }
        uint256 lpSupply;
        if (isMultLP(address(pool.lpToken))) {
            if (pool.totalAmount == 0) {
                pool.lastRewardBlock = block.number;
                return;
            }
            lpSupply = pool.totalAmount;
        } else {
            lpSupply = pool.lpToken.balanceOf(address(this));
            if (lpSupply == 0) {
                pool.lastRewardBlock = block.number;
                return;
            }
        }
        uint256 blockReward = getDexBlockReward(pool.lastRewardBlock);
        if (blockReward <= 0) {
            return;
        }
        uint256 dexReward = blockReward.mul(pool.allocPoint).div(totalAllocPoint);
        bool minRet = dex.mint(address(this), dexReward);
        if (minRet) {
            pool.accDexPerShare = pool.accDexPerShare.add(dexReward.mul(1e12).div(lpSupply));
        }
        pool.lastRewardBlock = block.number;
    }

    // View function to see pending DEXs on frontend.
    function pending(uint256 _pid, address _user) external view returns (uint256, uint256){
        PoolInfo storage pool = poolInfo[_pid];
        if (isMultLP(address(pool.lpToken))) {
            (uint256 dexAmount, uint256 tokenAmount) = pendingDexAndToken(_pid, _user);
            return (dexAmount, tokenAmount);
        } else {
            uint256 dexAmount = pendingDex(_pid, _user);
            return (dexAmount, 0);
        }
    }

    function pendingDexAndToken(uint256 _pid, address _user) private view returns (uint256, uint256){
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][_user];
        uint256 accDexPerShare = pool.accDexPerShare;
        uint256 accMultLpPerShare = pool.accMultLpPerShare;
        if (user.amount > 0) {
            uint256 TokenPending = IMasterChef(multLpChef).pending(poolCorrespond[_pid], address(this
            accMultLpPerShare = accMultLpPerShare.add(TokenPending.mul(1e12).div(pool.totalAmount));
            uint256 userPending = user.amount.mul(accMultLpPerShare).div(1e12).sub(user.multLpRewardD
            if (block.number > pool.lastRewardBlock) {
                uint256 blockReward = getDexBlockReward(pool.lastRewardBlock);
                uint256 dexReward = blockReward.mul(pool.allocPoint).div(totalAllocPoint);
                accDexPerShare = accDexPerShare.add(dexReward.mul(1e12).div(pool.totalAmount));
                return (user.amount.mul(accDexPerShare).div(1e12).sub(user.rewardDebt), userPending);
            }
```

```
            if (block.number == pool.lastRewardBlock) {
                return (user.amount.mul(accDexPerShare).div(1e12).sub(user.rewardDebt), userPending);
            }
        }
        return (0, 0);
    }

    function pendingDex(uint256 _pid, address _user) private view returns (uint256){
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][_user];
        uint256 accDexPerShare = pool.accDexPerShare;
        uint256 lpSupply = pool.lpToken.balanceOf(address(this));
        if (user.amount > 0) {
            if (block.number > pool.lastRewardBlock) {
                uint256 blockReward = getDexBlockReward(pool.lastRewardBlock);
                uint256 dexReward = blockReward.mul(pool.allocPoint).div(totalAllocPoint);
                accDexPerShare = accDexPerShare.add(dexReward.mul(1e12).div(lpSupply));
                return user.amount.mul(accDexPerShare).div(1e12).sub(user.rewardDebt);
            }
            if (block.number == pool.lastRewardBlock) {
                return user.amount.mul(accDexPerShare).div(1e12).sub(user.rewardDebt);
            }
        }
        return 0;
    }

    // Deposit LP tokens to HecoPool for DEX allocation.
    function deposit(uint256 _pid, uint256 _amount) public notPause {
        require(!isBadAddress(msg.sender), 'Illegal, rejected ');
        PoolInfo storage pool = poolInfo[_pid];
        if (isMultLP(address(pool.lpToken))) {
            depositDexAndToken(_pid, _amount, msg.sender);
        } else {
            depositDex(_pid, _amount, msg.sender);
        }
    }

    function depositDexAndToken(uint256 _pid, uint256 _amount, address _user) private {
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][_user];
        updatePool(_pid);
        if (user.amount > 0) {
            uint256 pendingAmount = user.amount.mul(pool.accDexPerShare).div(1e12).sub(user.rewardDeb
            if (pendingAmount > 0) {
                safeDexTransfer(_user, pendingAmount);
            }
            uint256 beforeToken = IERC20(multLpToken).balanceOf(address(this));
            IMasterChef(multLpChef).deposit(poolCorrespond[_pid], 0);
            uint256 afterToken = IERC20(multLpToken).balanceOf(address(this));
            pool.accMultLpPerShare = pool.accMultLpPerShare.add(afterToken.sub(beforeToken).mul(1e12)
            uint256 tokenPending = user.amount.mul(pool.accMultLpPerShare).div(1e12).sub(user.multLpR
            if (tokenPending > 0) {
                IERC20(multLpToken).safeTransfer(_user, tokenPending);
            }
        }
        if (_amount > 0) {
            pool.lpToken.safeTransferFrom(_user, address(this), _amount);
            if (pool.totalAmount == 0) {
                IMasterChef(multLpChef).deposit(poolCorrespond[_pid], _amount);
                user.amount = user.amount.add(_amount);
                pool.totalAmount = pool.totalAmount.add(_amount);
            } else {
                uint256 beforeToken = IERC20(multLpToken).balanceOf(address(this));
                IMasterChef(multLpChef).deposit(poolCorrespond[_pid], _amount);
                uint256 afterToken = IERC20(multLpToken).balanceOf(address(this));
                pool.accMultLpPerShare = pool.accMultLpPerShare.add(afterToken.sub(beforeToken).mul(1
```

```
                user.amount = user.amount.add(_amount);
                pool.totalAmount = pool.totalAmount.add(_amount);
            }
        }
        user.rewardDebt = user.amount.mul(pool.accDexPerShare).div(1e12);
        user.multLpRewardDebt = user.amount.mul(pool.accMultLpPerShare).div(1e12);
        emit Deposit(_user, _pid, _amount);
    }

    function depositDex(uint256 _pid, uint256 _amount, address _user) private {
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][_user];
        updatePool(_pid);
        if (user.amount > 0) {
            uint256 pendingAmount = user.amount.mul(pool.accDexPerShare).div(1e12).sub(user.rewardDeb
            if (pendingAmount > 0) {
                safeDexTransfer(_user, pendingAmount);
            }
        }
        if (_amount > 0) {
            pool.lpToken.safeTransferFrom(_user, address(this), _amount);
            user.amount = user.amount.add(_amount);
            pool.totalAmount = pool.totalAmount.add(_amount);
        }
        user.rewardDebt = user.amount.mul(pool.accDexPerShare).div(1e12);
        emit Deposit(_user, _pid, _amount);
    }

    // Withdraw LP tokens from HecoPool.
    function withdraw(uint256 _pid, uint256 _amount) public notPause {
        PoolInfo storage pool = poolInfo[_pid];
        if (isMultLP(address(pool.lpToken))) {
            withdrawDexAndToken(_pid, _amount, msg.sender);
        } else {
            withdrawDex(_pid, _amount, msg.sender);
        }
    }

    function withdrawDexAndToken(uint256 _pid, uint256 _amount, address _user) private {
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][_user];
        require(user.amount >= _amount, "withdrawDexAndToken: not good");
        updatePool(_pid);
        uint256 pendingAmount = user.amount.mul(pool.accDexPerShare).div(1e12).sub(user.rewardDebt);
        if (pendingAmount > 0) {
            safeDexTransfer(_user, pendingAmount);
        }
        if (_amount > 0) {
            uint256 beforeToken = IERC20(multLpToken).balanceOf(address(this));
            IMasterChef(multLpChef).withdraw(poolCorrespond[_pid], _amount);
            uint256 afterToken = IERC20(multLpToken).balanceOf(address(this));
            pool.accMultLpPerShare = pool.accMultLpPerShare.add(afterToken.sub(beforeToken).mul(1e12)
            uint256 tokenPending = user.amount.mul(pool.accMultLpPerShare).div(1e12).sub(user.multLpR
            if (tokenPending > 0) {
                IERC20(multLpToken).safeTransfer(_user, tokenPending);
            }
            user.amount = user.amount.sub(_amount);
            pool.totalAmount = pool.totalAmount.sub(_amount);
            pool.lpToken.safeTransfer(_user, _amount);
        }
        user.rewardDebt = user.amount.mul(pool.accDexPerShare).div(1e12);
        user.multLpRewardDebt = user.amount.mul(pool.accMultLpPerShare).div(1e12);
        emit Withdraw(_user, _pid, _amount);
    }

    function withdrawDex(uint256 _pid, uint256 _amount, address _user) private {
```

```
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][_user];
        require(user.amount >= _amount, "withdrawDex: not good");
        updatePool(_pid);
        uint256 pendingAmount = user.amount.mul(pool.accDexPerShare).div(1e12).sub(user.rewardDebt);
        if (pendingAmount > 0) {
            safeDexTransfer(_user, pendingAmount);
        }
        if (_amount > 0) {
            user.amount = user.amount.sub(_amount);
            pool.totalAmount = pool.totalAmount.sub(_amount);
            pool.lpToken.safeTransfer(_user, _amount);
        }
        user.rewardDebt = user.amount.mul(pool.accDexPerShare).div(1e12);
        emit Withdraw(_user, _pid, _amount);
    }

    // Withdraw without caring about rewards. EMERGENCY ONLY.
    function emergencyWithdraw(uint256 _pid) public notPause {
        PoolInfo storage pool = poolInfo[_pid];
        if (isMultLP(address(pool.lpToken))) {
            emergencyWithdrawDexAndToken(_pid, msg.sender);
        } else {
            emergencyWithdrawDex(_pid, msg.sender);
        }
    }

    function emergencyWithdrawDexAndToken(uint256 _pid, address _user) private {
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][_user];
        uint256 amount = user.amount;
        uint256 beforeToken = IERC20(multLpToken).balanceOf(address(this));
        IMasterChef(multLpChef).withdraw(poolCorrespond[_pid], amount);
        uint256 afterToken = IERC20(multLpToken).balanceOf(address(this));
        pool.accMultLpPerShare = pool.accMultLpPerShare.add(afterToken.sub(beforeToken).mul(1e12).div
        user.amount = 0;
        user.rewardDebt = 0;
        pool.lpToken.safeTransfer(_user, amount);
        pool.totalAmount = pool.totalAmount.sub(amount);
        emit EmergencyWithdraw(_user, _pid, amount);
    }

    function emergencyWithdrawDex(uint256 _pid, address _user) private {
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][_user];
        uint256 amount = user.amount;
        user.amount = 0;
        user.rewardDebt = 0;
        pool.lpToken.safeTransfer(_user, amount);
        pool.totalAmount = pool.totalAmount.sub(amount);
        emit EmergencyWithdraw(_user, _pid, amount);
    }

    // Safe DEX transfer function, just in case if rounding error causes pool to not have enough DEXs
    function safeDexTransfer(address _to, uint256 _amount) internal {
        uint256 dexBal = dex.balanceOf(address(this));
        if (_amount > dexBal) {
            dex.transfer(_to, dexBal);
        } else {
            dex.transfer(_to, _amount);
        }
    }

    modifier notPause() {
        require(paused == false, "Mining has been suspended");
        _;
```

```
    }

    // addresses not allowed to be represented to harvest
    mapping(address => bool) public notRepresents;

    function represent(bool _allow) public {
        if (!_allow) {
            notRepresents[msg.sender] = true;
        } else if (notRepresents[msg.sender]) {
            delete notRepresents[msg.sender];
        }
    }

    function harvest() public {
        uint256 length = poolInfo.length;
        for (uint256 pid = 0; pid < length; ++pid) {
            deposit(pid, 0);
        }
    }

    function harvestOf(address account) public {
        require(!isBadAddress(account), 'Illegal, rejected ');
        require(!notRepresents[account], 'not allowed');
        uint256 length = poolInfo.length;
        for (uint256 pid = 0; pid < length; ++pid) {
            PoolInfo storage pool = poolInfo[pid];
            if (isMultLP(address(pool.lpToken))) {
                depositDexAndToken(pid, 0, account);
            } else {
                depositDex(pid, 0, account);
            }
        }
    }
}/**
 *Submitted for verification at BscScan.com on 2021-04-08
*/

// SPDX-License-Identifier: MIT
pragma solidity ^0.6.0;


/*
 * @dev Provides information about the current execution context, including the
 * sender of the transaction and its data. While these are generally available
 * via msg.sender and msg.data, they should not be accessed in such a direct
 * manner, since when dealing with GSN meta-transactions the account sending and
 * paying for execution may not be the actual sender (as far as an application
 * is concerned).
 *
 * This contract is only required for intermediate, library-like contracts.
 */
abstract contract Context {
    function _msgSender() internal view virtual returns (address payable) {
        return msg.sender;
    }

    function _msgData() internal view virtual returns (bytes memory) {
        this;
        // silence state mutability warning without generating bytecode - see https://github.com/ethe
        return msg.data;
    }
}


/**
 * @dev Contract module which provides a basic access control mechanism, where
```

```
 * there is an account (an owner) that can be granted exclusive access to
 * specific functions.
 *
 * By default, the owner account will be the one that deploys the contract. This
 * can later be changed with {transferOwnership}.
 *
 * This module is used through inheritance. It will make available the modifier
 * `onlyOwner`, which can be applied to your functions to restrict their use to
 * the owner.
 */
contract Ownable is Context {
    address private _owner;

    event OwnershipTransferred(address indexed previousOwner, address indexed newOwner);

    /**
     * @dev Initializes the contract setting the deployer as the initial owner.
     */
    constructor () internal {
        address msgSender = _msgSender();
        _owner = msgSender;
        emit OwnershipTransferred(address(0), msgSender);
    }

    /**
     * @dev Returns the address of the current owner.
     */
    function owner() public view returns (address) {
        return _owner;
    }

    /**
     * @dev Throws if called by any account other than the owner.
     */
    modifier onlyOwner() {
        require(_owner == _msgSender(), "Ownable: caller is not the owner");
        _;
    }

    /**
     * @dev Leaves the contract without owner. It will not be possible to call
     * `onlyOwner` functions anymore. Can only be called by the current owner.
     *
     * NOTE: Renouncing ownership will leave the contract without an owner,
     * thereby removing any functionality that is only available to the owner.
     */
    function renounceOwnership() public virtual onlyOwner {
        emit OwnershipTransferred(_owner, address(0));
        _owner = address(0);
    }

    /**
     * @dev Transfers ownership of the contract to a new account (`newOwner`).
     * Can only be called by the current owner.
     */
    function transferOwnership(address newOwner) public virtual onlyOwner {
        require(newOwner != address(0), "Ownable: new owner is the zero address");
        emit OwnershipTransferred(_owner, newOwner);
        _owner = newOwner;
    }
}


/**
 * @dev Interface of the ERC20 standard as defined in the EIP.
 */
```

```
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);
}


/**
 * @dev Wrappers over Solidity's arithmetic operations with added overflow
 * checks.
 *
 * Arithmetic operations in Solidity wrap on overflow. This can easily result
 * in bugs, because programmers usually assume that an overflow raises an
 * error, which is the standard behavior in high level programming languages.
 * `SafeMath` restores this intuition by reverting the transaction when an
 * operation overflows.
 *
 * Using this library instead of the unchecked operations eliminates an entire
 * class of bugs, so it's recommended to use it always.
 */
library SafeMath {
    /**
     * @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) {
        return sub(a, b, "SafeMath: subtraction overflow");
    }

    /**
     * @dev Returns the subtraction of two unsigned integers, reverting with custom message on
     * overflow (when the result is negative).
     *
     * 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);
        uint256 c = a - b;

        return c;
```

```
    }

    /**
     * @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) {
        // 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 0;
        }

        uint256 c = a * b;
        require(c / a == b, "SafeMath: multiplication overflow");

        return c;
    }

    /**
     * @dev Returns the integer division of two unsigned integers. Reverts 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) {
        return div(a, b, "SafeMath: division by zero");
    }

    /**
     * @dev Returns the integer division of two unsigned integers. Reverts with custom message 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, string memory errorMessage) internal pure returns (uint256) {
        require(b > 0, errorMessage);
        uint256 c = a / b;
        // assert(a == b * c + a % b); // There is no case in which this doesn't hold

        return c;
    }

    /**
     * @dev Returns the remainder of dividing two unsigned integers. (unsigned integer modulo),
     * Reverts 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) {
        return mod(a, b, "SafeMath: modulo by zero");
    }

    /**
     * @dev Returns the remainder of dividing two unsigned integers. (unsigned integer modulo),
     * Reverts with custom message 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, string memory errorMessage) internal pure returns (uint256) {
        require(b != 0, errorMessage);
        return a % b;
    }
}


/**
 * @dev Collection of functions related to the address type
 */
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) {
        // According to EIP-1052, 0x0 is the value returned for not-yet created accounts
        // and 0xc5d2460186f7233c927e7db2dcc703c0e500b653ca82273b7bfad8045d85a470 is returned
        // for accounts without code, i.e. `keccak256('')`
        bytes32 codehash;
        bytes32 accountHash = 0xc5d2460186f7233c927e7db2dcc703c0e500b653ca82273b7bfad8045d85a470;
        // solhint-disable-next-line no-inline-assembly
        assembly {codehash := extcodehash(account)}
        return (codehash != accountHash && codehash != 0x0);
    }

    /**
     * @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
     */
    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.de
     *
     * 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 ret
        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
        return functionCallWithValue(target, data, value, "Address: low-level call with value failed"
```

```
    }

    /**
     * @dev Same as {xref-Address-functionCallWithValue-address-bytes-uint256-}[`functionCallWithValu
     * 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 er
        require(address(this).balance >= value, "Address: insufficient balance for call");
        return _functionCallWithValue(target, data, value, errorMessage);
    }

    function _functionCallWithValue(address target, bytes memory data, uint256 weiValue, string memor
        require(isContract(target), "Address: call to non-contract");

        // solhint-disable-next-line avoid-low-level-calls
        (bool success, bytes memory returndata) = target.call{value : weiValue}(data);
        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 Implementation of the {IERC20} interface.
 *
 * This implementation is agnostic to the way tokens are created. This means
 * that a supply mechanism has to be added in a derived contract using {_mint}.
 * For a generic mechanism see {ERC20PresetMinterPauser}.
 *
 * TIP: For a detailed writeup see our guide
 * https://forum.zeppelin.solutions/t/how-to-implement-erc20-supply-mechanisms/226[How
 * to implement supply mechanisms].
 *
 * We have followed general OpenZeppelin guidelines: functions revert instead
 * of returning `false` on failure. This behavior is nonetheless conventional
 * and does not conflict with the expectations of ERC20 applications.
 *
 * Additionally, an {Approval} event is emitted on calls to {transferFrom}.
 * This allows applications to reconstruct the allowance for all accounts just
 * by listening to said events. Other implementations of the EIP may not emit
 * these events, as it isn't required by the specification.
 *
 * Finally, the non-standard {decreaseAllowance} and {increaseAllowance}
 * functions have been added to mitigate the well-known issues around setting
 * allowances. See {IERC20-approve}.
 */
contract ERC20 is Context, IERC20 {
    using SafeMath for uint256;
    using Address for address;
```

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

    mapping(address => mapping(address => uint256)) private _allowances;

    uint256 private _totalSupply;

    string private _name;
    string private _symbol;
    uint8 private _decimals;

    /**
     * @dev Sets the values for {name} and {symbol}, initializes {decimals} with
     * a default value of 18.
     *
     * To select a different value for {decimals}, use {_setupDecimals}.
     *
     * All three of these values are immutable: they can only be set once during
     * construction.
     */
    constructor (string memory name, string memory symbol) public {
        _name = name;
        _symbol = symbol;
        _decimals = 18;
    }

    /**
     * @dev Returns the name of the token.
     */
    function name() public view returns (string memory) {
        return _name;
    }

    /**
     * @dev Returns the symbol of the token, usually a shorter version of the
     * name.
     */
    function symbol() public view returns (string memory) {
        return _symbol;
    }

    /**
     * @dev Returns the number of decimals used to get its user representation.
     * For example, if `decimals` equals `2`, a balance of `505` tokens should
     * be displayed to a user as `5,05` (`505 / 10 ** 2`).
     *
     * Tokens usually opt for a value of 18, imitating the relationship between
     * Ether and Wei. This is the value {ERC20} uses, unless {_setupDecimals} is
     * called.
     *
     * NOTE: This information is only used for _display_ purposes: it in
     * no way affects any of the arithmetic of the contract, including
     * {IERC20-balanceOf} and {IERC20-transfer}.
     */
    function decimals() public view returns (uint8) {
        return _decimals;
    }

    /**
     * @dev See {IERC20-totalSupply}.
     */
    function totalSupply() public view override returns (uint256) {
        return _totalSupply;
    }

    /**
     * @dev See {IERC20-balanceOf}.
```

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

    /**
     * @dev See {IERC20-transfer}.
     *
     * Requirements:
     *
     * - `recipient` cannot be the zero address.
     * - the caller must have a balance of at least `amount`.
     */
    function transfer(address recipient, uint256 amount) public virtual override returns (bool) {
        _transfer(_msgSender(), recipient, amount);
        return true;
    }

    /**
     * @dev See {IERC20-allowance}.
     */
    function allowance(address owner, address spender) public view virtual override returns (uint256)
        return _allowances[owner][spender];
    }

    /**
     * @dev See {IERC20-approve}.
     *
     * Requirements:
     *
     * - `spender` cannot be the zero address.
     */
    function approve(address spender, uint256 amount) public virtual override returns (bool) {
        _approve(_msgSender(), spender, amount);
        return true;
    }

    /**
     * @dev See {IERC20-transferFrom}.
     *
     * Emits an {Approval} event indicating the updated allowance. This is not
     * required by the EIP. See the note at the beginning of {ERC20};
     *
     * Requirements:
     * - `sender` and `recipient` cannot be the zero address.
     * - `sender` must have a balance of at least `amount`.
     * - the caller must have allowance for ``sender``'s tokens of at least
     * `amount`.
     */
    function transferFrom(address sender, address recipient, uint256 amount) public virtual override
        _transfer(sender, recipient, amount);
        _approve(sender, _msgSender(), _allowances[sender][_msgSender()].sub(amount, "ERC20: transfer
        return true;
    }

    /**
     * @dev Atomically increases the allowance granted to `spender` by the caller.
     *
     * This is an alternative to {approve} that can be used as a mitigation for
     * problems described in {IERC20-approve}.
     *
     * Emits an {Approval} event indicating the updated allowance.
     *
     * Requirements:
     *
     * - `spender` cannot be the zero address.
```

```
     */
    function increaseAllowance(address spender, uint256 addedValue) public virtual returns (bool) {
        _approve(_msgSender(), spender, _allowances[_msgSender()][spender].add(addedValue));
        return true;
    }

    /**
     * @dev Atomically decreases the allowance granted to `spender` by the caller.
     *
     * This is an alternative to {approve} that can be used as a mitigation for
     * problems described in {IERC20-approve}.
     *
     * Emits an {Approval} event indicating the updated allowance.
     *
     * Requirements:
     *
     * - `spender` cannot be the zero address.
     * - `spender` must have allowance for the caller of at least
     * `subtractedValue`.
     */
    function decreaseAllowance(address spender, uint256 subtractedValue) public virtual returns (bool
        _approve(_msgSender(), spender, _allowances[_msgSender()][spender].sub(subtractedValue, "ERC2
        return true;
    }

    /**
     * @dev Moves tokens `amount` from `sender` to `recipient`.
     *
     * This is internal function is equivalent to {transfer}, and can be used to
     * e.g. implement automatic token fees, slashing mechanisms, etc.
     *
     * Emits a {Transfer} event.
     *
     * Requirements:
     *
     * - `sender` cannot be the zero address.
     * - `recipient` cannot be the zero address.
     * - `sender` must have a balance of at least `amount`.
     */
    function _transfer(address sender, address recipient, uint256 amount) internal virtual {
        require(sender != address(0), "ERC20: transfer from the zero address");
        require(recipient != address(0), "ERC20: transfer to the zero address");

        _beforeTokenTransfer(sender, recipient, amount);

        _balances[sender] = _balances[sender].sub(amount, "ERC20: transfer amount exceeds balance");
        _balances[recipient] = _balances[recipient].add(amount);
        emit Transfer(sender, recipient, amount);
    }

    /** @dev Creates `amount` tokens and assigns them to `account`, increasing
     * the total supply.
     *
     * Emits a {Transfer} event with `from` set to the zero address.
     *
     * Requirements
     *
     * - `to` cannot be the zero address.
     */
    function _mint(address account, uint256 amount) internal virtual {
        require(account != address(0), "ERC20: mint to the zero address");

        _beforeTokenTransfer(address(0), account, amount);

        _totalSupply = _totalSupply.add(amount);
        _balances[account] = _balances[account].add(amount);
```

```
        emit Transfer(address(0), account, amount);
    }

    /**
     * @dev Destroys `amount` tokens from `account`, reducing the
     * total supply.
     *
     * Emits a {Transfer} event with `to` set to the zero address.
     *
     * Requirements
     *
     * - `account` cannot be the zero address.
     * - `account` must have at least `amount` tokens.
     */
    function _burn(address account, uint256 amount) internal virtual {
        require(account != address(0), "ERC20: burn from the zero address");

        _beforeTokenTransfer(account, address(0), amount);

        _balances[account] = _balances[account].sub(amount, "ERC20: burn amount exceeds balance");
        _totalSupply = _totalSupply.sub(amount);
        emit Transfer(account, address(0), amount);
    }

    /**
     * @dev Sets `amount` as the allowance of `spender` over the `owner`s tokens.
     *
     * This is internal function is equivalent to `approve`, and can be used to
     * e.g. set automatic allowances for certain subsystems, etc.
     *
     * Emits an {Approval} event.
     *
     * Requirements:
     *
     * - `owner` cannot be the zero address.
     * - `spender` cannot be the zero address.
     */
    function _approve(address owner, address spender, uint256 amount) internal virtual {
        require(owner != address(0), "ERC20: approve from the zero address");
        require(spender != address(0), "ERC20: approve to the zero address");

        _allowances[owner][spender] = amount;
        emit Approval(owner, spender, amount);
    }

    /**
     * @dev Sets {decimals} to a value other than the default one of 18.
     *
     * WARNING: This function should only be called from the constructor. Most
     * applications that interact with token contracts will not expect
     * {decimals} to ever change, and may work incorrectly if it does.
     */
    function _setupDecimals(uint8 decimals_) internal {
        _decimals = decimals_;
    }

    /**
     * @dev Hook that is called before any transfer of tokens. This includes
     * minting and burning.
     *
     * Calling conditions:
     *
     * - when `from` and `to` are both non-zero, `amount` of ``from``'s tokens
     * will be to transferred to `to`.
     * - when `from` is zero, `amount` tokens will be minted for `to`.
     * - when `to` is zero, `amount` of ``from``'s tokens will be burned.
```

```
     * - `from` and `to` are never both zero.
     *
     * To learn more about hooks, head to xref:ROOT:extending-contracts.adoc#using-hooks[Using Hooks]
     */
    function _beforeTokenTransfer(address from, address to, uint256 amount) internal virtual {}
}


library EnumerableSet {
    // To implement this library for multiple types with as little code
    // repetition as possible, we write it in terms of a generic Set type with
    // bytes32 values.
    // The Set implementation uses private functions, and user-facing
    // implementations (such as AddressSet) are just wrappers around the
    // underlying Set.
    // This means that we can only create new EnumerableSets for types that fit
    // in bytes32.

    struct Set {
        // Storage of set values
        bytes32[] _values;

        // Position of the value in the `values` array, plus 1 because index 0
        // means a value is not in the set.
        mapping(bytes32 => uint256) _indexes;
    }

    /**
     * @dev Add a value to a set. O(1).
     *
     * Returns true if the value was added to the set, that is if it was not
     * already present.
     */
    function _add(Set storage set, bytes32 value) private returns (bool) {
        if (!_contains(set, value)) {
            set._values.push(value);
            // The value is stored at length-1, but we add 1 to all indexes
            // and use 0 as a sentinel value
            set._indexes[value] = set._values.length;
            return true;
        } else {
            return false;
        }
    }

    /**
     * @dev Removes a value from a set. O(1).
     *
     * Returns true if the value was removed from the set, that is if it was
     * present.
     */
    function _remove(Set storage set, bytes32 value) private returns (bool) {
        // We read and store the value's index to prevent multiple reads from the same storage slot
        uint256 valueIndex = set._indexes[value];

        if (valueIndex != 0) {// Equivalent to contains(set, value)
            // To delete an element from the _values array in O(1), we swap the element to delete wit
            // the array, and then remove the last element (sometimes called as 'swap and pop').
            // This modifies the order of the array, as noted in {at}.

            uint256 toDeleteIndex = valueIndex - 1;
            uint256 lastIndex = set._values.length - 1;

            // When the value to delete is the last one, the swap operation is unnecessary. However,
            // so rarely, we still do the swap anyway to avoid the gas cost of adding an 'if' stateme
```

```
            bytes32 lastvalue = set._values[lastIndex];

            // Move the last value to the index where the value to delete is
            set._values[toDeleteIndex] = lastvalue;
            // Update the index for the moved value
            set._indexes[lastvalue] = toDeleteIndex + 1;
            // All indexes are 1-based

            // Delete the slot where the moved value was stored
            set._values.pop();

            // Delete the index for the deleted slot
            delete set._indexes[value];

            return true;
        } else {
            return false;
        }
    }

    /**
     * @dev Returns true if the value is in the set. O(1).
     */
    function _contains(Set storage set, bytes32 value) private view returns (bool) {
        return set._indexes[value] != 0;
    }

    /**
     * @dev Returns the number of values on the set. O(1).
     */
    function _length(Set storage set) private view returns (uint256) {
        return set._values.length;
    }

    /**
     * @dev Returns the value stored at position `index` in the set. O(1).
     *
     * Note that there are no guarantees on the ordering of values inside the
     * array, and it may change when more values are added or removed.
     *
     * Requirements:
     *
     * - `index` must be strictly less than {length}.
     */
    function _at(Set storage set, uint256 index) private view returns (bytes32) {
        require(set._values.length > index, "EnumerableSet: index out of bounds");
        return set._values[index];
    }

    // Bytes32Set

    struct Bytes32Set {
        Set _inner;
    }

    /**
     * @dev Add a value to a set. O(1).
     *
     * Returns true if the value was added to the set, that is if it was not
     * already present.
     */
    function add(Bytes32Set storage set, bytes32 value) internal returns (bool) {
        return _add(set._inner, value);
    }

    /**
```

```
     * @dev Removes a value from a set. O(1).
     *
     * Returns true if the value was removed from the set, that is if it was
     * present.
     */
    function remove(Bytes32Set storage set, bytes32 value) internal returns (bool) {
        return _remove(set._inner, value);
    }

    /**
     * @dev Returns true if the value is in the set. O(1).
     */
    function contains(Bytes32Set storage set, bytes32 value) internal view returns (bool) {
        return _contains(set._inner, value);
    }

    /**
     * @dev Returns the number of values in the set. O(1).
     */
    function length(Bytes32Set storage set) internal view returns (uint256) {
        return _length(set._inner);
    }

    /**
     * @dev Returns the value stored at position `index` in the set. O(1).
     *
     * Note that there are no guarantees on the ordering of values inside the
     * array, and it may change when more values are added or removed.
     *
     * Requirements:
     *
     * - `index` must be strictly less than {length}.
     */
    function at(Bytes32Set storage set, uint256 index) internal view returns (bytes32) {
        return _at(set._inner, index);
    }

    // AddressSet

    struct AddressSet {
        Set _inner;
    }

    /**
     * @dev Add a value to a set. O(1).
     *
     * Returns true if the value was added to the set, that is if it was not
     * already present.
     */
    function add(AddressSet storage set, address value) internal returns (bool) {
        return _add(set._inner, bytes32(uint256(value)));
    }

    /**
     * @dev Removes a value from a set. O(1).
     *
     * Returns true if the value was removed from the set, that is if it was
     * present.
     */
    function remove(AddressSet storage set, address value) internal returns (bool) {
        return _remove(set._inner, bytes32(uint256(value)));
    }

    /**
     * @dev Returns true if the value is in the set. O(1).
     */
```

```
    function contains(AddressSet storage set, address value) internal view returns (bool) {
        return _contains(set._inner, bytes32(uint256(value)));
    }

    /**
     * @dev Returns the number of values in the set. O(1).
     */
    function length(AddressSet storage set) internal view returns (uint256) {
        return _length(set._inner);
    }

    /**
     * @dev Returns the value stored at position `index` in the set. O(1).
     *
     * Note that there are no guarantees on the ordering of values inside the
     * array, and it may change when more values are added or removed.
     *
     * Requirements:
     *
     * - `index` must be strictly less than {length}.
     */
    function at(AddressSet storage set, uint256 index) internal view returns (address) {
        return address(uint256(_at(set._inner, index)));
    }


    // UintSet

    struct UintSet {
        Set _inner;
    }

    /**
     * @dev Add a value to a set. O(1).
     *
     * Returns true if the value was added to the set, that is if it was not
     * already present.
     */
    function add(UintSet storage set, uint256 value) internal returns (bool) {
        return _add(set._inner, bytes32(value));
    }

    /**
     * @dev Removes a value from a set. O(1).
     *
     * Returns true if the value was removed from the set, that is if it was
     * present.
     */
    function remove(UintSet storage set, uint256 value) internal returns (bool) {
        return _remove(set._inner, bytes32(value));
    }

    /**
     * @dev Returns true if the value is in the set. O(1).
     */
    function contains(UintSet storage set, uint256 value) internal view returns (bool) {
        return _contains(set._inner, bytes32(value));
    }

    /**
     * @dev Returns the number of values on the set. O(1).
     */
    function length(UintSet storage set) internal view returns (uint256) {
        return _length(set._inner);
    }
```

```
    /**
     * @dev Returns the value stored at position `index` in the set. O(1).
     *
     * Note that there are no guarantees on the ordering of values inside the
     * array, and it may change when more values are added or removed.
     *
     * Requirements:
     *
     * - `index` must be strictly less than {length}.
     */
    function at(UintSet storage set, uint256 index) internal view returns (uint256) {
        return uint256(_at(set._inner, index));
    }
}

pragma solidity ^0.6.0;
pragma experimental ABIEncoderV2;

abstract contract DelegateERC20 is ERC20 {
    // @notice A record of each accounts delegate
    mapping(address => address) internal _delegates;

    /// @notice A checkpoint for marking number of votes from a given block
    struct Checkpoint {
        uint32 fromBlock;
        uint256 votes;
    }

    /// @notice A record of votes checkpoints for each account, by index
    mapping(address => mapping(uint32 => Checkpoint)) public checkpoints;

    /// @notice The number of checkpoints for each account
    mapping(address => uint32) public numCheckpoints;

    /// @notice The EIP-712 typehash for the contract's domain
    bytes32 public constant DOMAIN_TYPEHASH = keccak256("EIP712Domain(string name,uint256 chainId,add

    /// @notice The EIP-712 typehash for the delegation struct used by the contract
    bytes32 public constant DELEGATION_TYPEHASH = keccak256("Delegation(address delegatee,uint256 non

    /// @notice A record of states for signing / validating signatures
    mapping(address => uint) public nonces;


    // support delegates mint
    function _mint(address account, uint256 amount) internal override virtual {
        super._mint(account, amount);

        // add delegates to the minter
        _moveDelegates(address(0), _delegates[account], amount);
    }


    function _transfer(address sender, address recipient, uint256 amount) internal override virtual {
        super._transfer(sender, recipient, amount);
        _moveDelegates(_delegates[sender], _delegates[recipient], amount);
    }


    /**
    * @notice Delegate votes from `msg.sender` to `delegatee`
    * @param delegatee The address to delegate votes to
    */
    function delegate(address delegatee) external {
        return _delegate(msg.sender, delegatee);
    }
```

```
    /**
     * @notice Delegates votes from signatory to `delegatee`
     * @param delegatee The address to delegate votes to
     * @param nonce The contract state required to match the signature
     * @param expiry The time at which to expire the signature
     * @param v The recovery byte of the signature
     * @param r Half of the ECDSA signature pair
     * @param s Half of the ECDSA signature pair
     */
    function delegateBySig(
        address delegatee,
        uint nonce,
        uint expiry,
        uint8 v,
        bytes32 r,
        bytes32 s
    )
    external
    {
        bytes32 domainSeparator = keccak256(
            abi.encode(
                DOMAIN_TYPEHASH,
                keccak256(bytes(name())),
                getChainId(),
                address(this)
            )
        );

        bytes32 structHash = keccak256(
            abi.encode(
                DELEGATION_TYPEHASH,
                delegatee,
                nonce,
                expiry
            )
        );

        bytes32 digest = keccak256(
            abi.encodePacked(
                "\x19\x01",
                domainSeparator,
                structHash
            )
        );

        address signatory = ecrecover(digest, v, r, s);
        require(signatory != address(0), "DexToken::delegateBySig: invalid signature");
        require(nonce == nonces[signatory]++, "DexToken::delegateBySig: invalid nonce");
        require(now <= expiry, "DexToken::delegateBySig: signature expired");
        return _delegate(signatory, delegatee);
    }

    /**
     * @notice Gets the current votes balance for `account`
     * @param account The address to get votes balance
     * @return The number of current votes for `account`
     */
    function getCurrentVotes(address account)
    external
    view
    returns (uint256)
    {
        uint32 nCheckpoints = numCheckpoints[account];
        return nCheckpoints > 0 ? checkpoints[account][nCheckpoints - 1].votes : 0;
    }
```

```
    /**
     * @notice Determine the prior number of votes for an account as of a block number
     * @dev Block number must be a finalized block or else this function will revert to prevent misin
     * @param account The address of the account to check
     * @param blockNumber The block number to get the vote balance at
     * @return The number of votes the account had as of the given block
     */
    function getPriorVotes(address account, uint blockNumber)
    external
    view
    returns (uint256)
    {
        require(blockNumber < block.number, "DexToken::getPriorVotes: not yet determined");

        uint32 nCheckpoints = numCheckpoints[account];
        if (nCheckpoints == 0) {
            return 0;
        }

        // First check most recent balance
        if (checkpoints[account][nCheckpoints - 1].fromBlock <= blockNumber) {
            return checkpoints[account][nCheckpoints - 1].votes;
        }

        // Next check implicit zero balance
        if (checkpoints[account][0].fromBlock > blockNumber) {
            return 0;
        }

        uint32 lower = 0;
        uint32 upper = nCheckpoints - 1;
        while (upper > lower) {
            uint32 center = upper - (upper - lower) / 2;
            // ceil, avoiding overflow
            Checkpoint memory cp = checkpoints[account][center];
            if (cp.fromBlock == blockNumber) {
                return cp.votes;
            } else if (cp.fromBlock < blockNumber) {
                lower = center;
            } else {
                upper = center - 1;
            }
        }
        return checkpoints[account][lower].votes;
    }

    function _delegate(address delegator, address delegatee)
    internal
    {
        address currentDelegate = _delegates[delegator];
        uint256 delegatorBalance = balanceOf(delegator);
        // balance of underlying balances (not scaled);
        _delegates[delegator] = delegatee;

        _moveDelegates(currentDelegate, delegatee, delegatorBalance);

        emit DelegateChanged(delegator, currentDelegate, delegatee);
    }

    function _moveDelegates(address srcRep, address dstRep, uint256 amount) internal {
        if (srcRep != dstRep && amount > 0) {
            if (srcRep != address(0)) {
                // decrease old representative
                uint32 srcRepNum = numCheckpoints[srcRep];
                uint256 srcRepOld = srcRepNum > 0 ? checkpoints[srcRep][srcRepNum - 1].votes : 0;
```

```
                uint256 srcRepNew = srcRepOld.sub(amount);
                _writeCheckpoint(srcRep, srcRepNum, srcRepOld, srcRepNew);
            }

            if (dstRep != address(0)) {
                // increase new representative
                uint32 dstRepNum = numCheckpoints[dstRep];
                uint256 dstRepOld = dstRepNum > 0 ? checkpoints[dstRep][dstRepNum - 1].votes : 0;
                uint256 dstRepNew = dstRepOld.add(amount);
                _writeCheckpoint(dstRep, dstRepNum, dstRepOld, dstRepNew);
            }
        }
    }

    function _writeCheckpoint(
        address delegatee,
        uint32 nCheckpoints,
        uint256 oldVotes,
        uint256 newVotes
    )
    internal
    {
        uint32 blockNumber = safe32(block.number, "DexToken::_writeCheckpoint: block number exceeds 3

        if (nCheckpoints > 0 && checkpoints[delegatee][nCheckpoints - 1].fromBlock == blockNumber) {
            checkpoints[delegatee][nCheckpoints - 1].votes = newVotes;
        } else {
            checkpoints[delegatee][nCheckpoints] = Checkpoint(blockNumber, newVotes);
            numCheckpoints[delegatee] = nCheckpoints + 1;
        }

        emit DelegateVotesChanged(delegatee, oldVotes, newVotes);
    }

    function safe32(uint n, string memory errorMessage) internal pure returns (uint32) {
        require(n < 2 ** 32, errorMessage);
        return uint32(n);
    }

    function getChainId() internal pure returns (uint) {
        uint256 chainId;
        assembly {chainId := chainid()}

        return chainId;
    }

    /// @notice An event thats emitted when an account changes its delegate
    event DelegateChanged(address indexed delegator, address indexed fromDelegate, address indexed to

    /// @notice An event thats emitted when a delegate account's vote balance changes
    event DelegateVotesChanged(address indexed delegate, uint previousBalance, uint newBalance);

}

contract COCOToken is DelegateERC20, Ownable {
    uint256 private constant preMineSupply = 3000000 * 1e18;
    uint256 private constant maxSupply = 500000000 * 1e18;     // the total supply

    using EnumerableSet for EnumerableSet.AddressSet;
    EnumerableSet.AddressSet private _minters;

    constructor() public ERC20("COCO Token", "COCO"){
        _mint(msg.sender, preMineSupply);
    }

    // mint with max supply
```

```
    function mint(address _to, uint256 _amount) public onlyMinter returns (bool) {
        if (_amount.add(totalSupply()) > maxSupply) {
            return false;
        }
        _mint(_to, _amount);
        return true;
    }

    function addMinter(address _addMinter) public onlyOwner returns (bool) {
        require(_addMinter != address(0), "DexToken: _addMinter is the zero address");
        return EnumerableSet.add(_minters, _addMinter);
    }

    function delMinter(address _delMinter) public onlyOwner returns (bool) {
        require(_delMinter != address(0), "DexToken: _delMinter is the zero address");
        return EnumerableSet.remove(_minters, _delMinter);
    }

    function getMinterLength() public view returns (uint256) {
        return EnumerableSet.length(_minters);
    }

    function isMinter(address account) public view returns (bool) {
        return EnumerableSet.contains(_minters, account);
    }

    function getMinter(uint256 _index) public view onlyOwner returns (address){
        require(_index <= getMinterLength() - 1, "DexToken: index out of bounds");
        return EnumerableSet.at(_minters, _index);
    }

    // modifier for mint function
    modifier onlyMinter() {
        require(isMinter(msg.sender), "caller is not the minter");
        _;
    }
}// SPDX-License-Identifier: GPL-3.0-or-later

pragma solidity >=0.4.0;

/*
 * @dev Provides information about the current execution context, including the
 * sender of the transaction and its data. While these are generally available
 * via msg.sender and msg.data, they should not be accessed in such a direct
 * manner, since when dealing with GSN meta-transactions the account sending and
 * paying for execution may not be the actual sender (as far as an application
 * is concerned).
 *
 * This contract is only required for intermediate, library-like contracts.
 */
contract Context {
    // Empty internal constructor, to prevent people from mistakenly deploying
    // an instance of this contract, which should be used via inheritance.
    constructor() internal {}

    function _msgSender() internal view returns (address payable) {
        return msg.sender;
    }

    function _msgData() internal view returns (bytes memory) {
        this; // silence state mutability warning without generating bytecode - see https://github.co
        return msg.data;
    }
}

pragma solidity >=0.4.0;
```

```
/**
 * @dev Contract module which provides a basic access control mechanism, where
 * there is an account (an owner) that can be granted exclusive access to
 * specific functions.
 *
 * By default, the owner account will be the one that deploys the contract. This
 * can later be changed with {transferOwnership}.
 *
 * This module is used through inheritance. It will make available the modifier
 * `onlyOwner`, which can be applied to your functions to restrict their use to
 * the owner.
 */
contract Ownable is Context {
    address private _owner;

    event OwnershipTransferred(address indexed previousOwner, address indexed newOwner);

    /**
     * @dev Initializes the contract setting the deployer as the initial owner.
     */
    constructor() internal {
        address msgSender = _msgSender();
        _owner = msgSender;
        emit OwnershipTransferred(address(0), msgSender);
    }

    /**
     * @dev Returns the address of the current owner.
     */
    function owner() public view returns (address) {
        return _owner;
    }

    /**
     * @dev Throws if called by any account other than the owner.
     */
    modifier onlyOwner() {
        require(_owner == _msgSender(), 'Ownable: caller is not the owner');
        _;
    }

    /**
     * @dev Leaves the contract without owner. It will not be possible to call
     * `onlyOwner` functions anymore. Can only be called by the current owner.
     *
     * NOTE: Renouncing ownership will leave the contract without an owner,
     * thereby removing any functionality that is only available to the owner.
     */
    function renounceOwnership() public onlyOwner {
        emit OwnershipTransferred(_owner, address(0));
        _owner = address(0);
    }

    /**
     * @dev Transfers ownership of the contract to a new account (`newOwner`).
     * Can only be called by the current owner.
     */
    function transferOwnership(address newOwner) public onlyOwner {
        _transferOwnership(newOwner);
    }

    /**
     * @dev Transfers ownership of the contract to a new account (`newOwner`).
     */
    function _transferOwnership(address newOwner) internal {
```

```
        require(newOwner != address(0), 'Ownable: new owner is the zero address');
        emit OwnershipTransferred(_owner, newOwner);
        _owner = newOwner;
    }
}

pragma solidity >=0.4.0;

/**
 * @dev Wrappers over Solidity's arithmetic operations with added overflow
 * checks.
 *
 * Arithmetic operations in Solidity wrap on overflow. This can easily result
 * in bugs, because programmers usually assume that an overflow raises an
 * error, which is the standard behavior in high level programming languages.
 * `SafeMath` restores this intuition by reverting the transaction when an
 * operation overflows.
 *
 * Using this library instead of the unchecked operations eliminates an entire
 * class of bugs, so it's recommended to use it always.
 */
library SafeMath {
    /**
     * @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) {
        return sub(a, b, 'SafeMath: subtraction overflow');
    }

    /**
     * @dev Returns the subtraction of two unsigned integers, reverting with custom message on
     * overflow (when the result is negative).
     *
     * 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);
        uint256 c = a - b;

        return c;
    }

    /**
     * @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) {
        // 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 0;
        }

        uint256 c = a * b;
        require(c / a == b, 'SafeMath: multiplication overflow');

        return c;
    }

    /**
     * @dev Returns the integer division of two unsigned integers. Reverts 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) {
        return div(a, b, 'SafeMath: division by zero');
    }

    /**
     * @dev Returns the integer division of two unsigned integers. Reverts with custom message 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,
        string memory errorMessage
    ) internal pure returns (uint256) {
        require(b > 0, errorMessage);
```

```
        uint256 c = a / b;
        // assert(a == b * c + a % b); // There is no case in which this doesn't hold

        return c;
    }

    /**
     * @dev Returns the remainder of dividing two unsigned integers. (unsigned integer modulo),
     * Reverts 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) {
        return mod(a, b, 'SafeMath: modulo by zero');
    }

    /**
     * @dev Returns the remainder of dividing two unsigned integers. (unsigned integer modulo),
     * Reverts with custom message 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,
        string memory errorMessage
    ) internal pure returns (uint256) {
        require(b != 0, errorMessage);
        return a % b;
    }

    function min(uint256 x, uint256 y) internal pure returns (uint256 z) {
        z = x < y ? x : y;
    }

    // babylonian method (https://en.wikipedia.org/wiki/Methods_of_computing_square_roots#Babylonian_
    function sqrt(uint256 y) internal pure returns (uint256 z) {
        if (y > 3) {
            z = y;
            uint256 x = y / 2 + 1;
            while (x < z) {
                z = x;
                x = (y / x + x) / 2;
            }
        } else if (y != 0) {
            z = 1;
        }
    }
}

pragma solidity >=0.4.0;

interface IHRC20 {
    /**
```

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

    /**
     * @dev Returns the token decimals.
     */
    function decimals() external view returns (uint8);

    /**
     * @dev Returns the token symbol.
     */
    function symbol() external view returns (string memory);

    /**
     * @dev Returns the token name.
     */
    function name() external view returns (string memory);

    /**
     * @dev Returns the bep token owner.
     */
    function getOwner() external view returns (address);

    /**
     * @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);
}

pragma solidity ^0.6.2;

/**
 * @dev Collection of functions related to the address type
 */
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) {
        // According to EIP-1052, 0x0 is the value returned for not-yet created accounts
        // and 0xc5d2460186f7233c927e7db2dcc703c0e500b653ca82273b7bfad8045d85a470 is returned
        // for accounts without code, i.e. `keccak256('')`
        bytes32 codehash;
        bytes32 accountHash = 0xc5d2460186f7233c927e7db2dcc703c0e500b653ca82273b7bfad8045d85a470;
        // solhint-disable-next-line no-inline-assembly
        assembly {
            codehash := extcodehash(account)
        }
        return (codehash != accountHash && codehash != 0x0);
    }

    /**
     * @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
     */
    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.de
     *
     * 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-}[`functionCallWithValu
     * 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');
        return _functionCallWithValue(target, data, value, errorMessage);
    }

    function _functionCallWithValue(
        address target,
        bytes memory data,
        uint256 weiValue,
        string memory errorMessage
    ) private returns (bytes memory) {
        require(isContract(target), 'Address: call to non-contract');

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

pragma solidity ^0.6.0;

/**
 * @title SafeHRC20
 * @dev Wrappers around HRC20 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 SafeHRC20 for IHRC20;` statement to your contract,
 * which allows you to call the safe operations as `token.safeTransfer(...)`, etc.
 */
```

```
library SafeHRC20 {
    using SafeMath for uint256;
    using Address for address;

    function safeTransfer(
        IHRC20 token,
        address to,
        uint256 value
    ) internal {
        _callOptionalReturn(token, abi.encodeWithSelector(token.transfer.selector, to, value));
    }

    function safeTransferFrom(
        IHRC20 token,
        address from,
        address to,
        uint256 value
    ) internal {
        _callOptionalReturn(token, abi.encodeWithSelector(token.transferFrom.selector, from, to, valu
    }

    /**
     * @dev Deprecated. This function has issues similar to the ones found in
     * {IHRC20-approve}, and its usage is discouraged.
     *
     * Whenever possible, use {safeIncreaseAllowance} and
     * {safeDecreaseAllowance} instead.
     */
    function safeApprove(
        IHRC20 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),
            'SafeHRC20: approve from non-zero to non-zero allowance'
        );
        _callOptionalReturn(token, abi.encodeWithSelector(token.approve.selector, spender, value));
    }

    function safeIncreaseAllowance(
        IHRC20 token,
        address spender,
        uint256 value
    ) internal {
        uint256 newAllowance = token.allowance(address(this), spender).add(value);
        _callOptionalReturn(token, abi.encodeWithSelector(token.approve.selector, spender, newAllowan
    }

    function safeDecreaseAllowance(
        IHRC20 token,
        address spender,
        uint256 value
    ) internal {
        uint256 newAllowance = token.allowance(address(this), spender).sub(
            value,
            'SafeHRC20: decreased allowance below zero'
        );
        _callOptionalReturn(token, abi.encodeWithSelector(token.approve.selector, spender, newAllowan
    }

    /**
```

```
     * @dev Imitates a Solidity high-level call (i.e. a regular function call to a contract), relaxin
     * on the return value: the return value is optional (but if data is returned, it must not be fal
     * @param token The token targeted by the call.
     * @param data The call data (encoded using abi.encode or one of its variants).
     */
    function _callOptionalReturn(IHRC20 token, bytes memory data) private {
        // We need to perform a low level call here, to bypass Solidity's return data size checking m
        // we're implementing it ourselves. We use {Address.functionCall} to perform this call, which
        // the target address contains contract code and also asserts for success in the low-level ca

        bytes memory returndata = address(token).functionCall(data, 'SafeHRC20: low-level call failed
        if (returndata.length > 0) {
            // Return data is optional
            // solhint-disable-next-line max-line-length
            require(abi.decode(returndata, (bool)), 'SafeHRC20: HRC20 operation did not succeed');
        }
    }
}

pragma solidity 0.6.12;

contract xTokenPool is Ownable {
    using SafeMath for uint256;
    using SafeHRC20 for IHRC20;

    // Info of each user.
    struct UserInfo {
        uint256 amount;     // How many LP tokens the user has provided.
        uint256 rewardDebt; // Reward debt. See explanation below.
    }

    // Info of each pool.
    struct PoolInfo {
        IHRC20 lpToken;           // Address of LP token contract.
        uint256 allocPoint;       // How many allocation points assigned to this pool. reward tokens
        uint256 lastRewardBlock;  // Last block number that reward tokens distribution occurs.
        uint256 accRewardPerShare; // Accumulated reward tokens per share, times 1e12. See below.
    }

    // The x TOKEN!
    IHRC20 public xToken;
    IHRC20 public rewardToken;

    // uint256 public maxStaking;

    // reward tokens created per block.
    uint256 public rewardPerBlock;

    // Info of each pool.
    PoolInfo[] public poolInfo;
    // Info of each user that stakes LP tokens.
    mapping (address => UserInfo) public userInfo;
    // Total allocation poitns. Must be the sum of all allocation points in all pools.
    uint256 private totalAllocPoint = 0;
    // The block number when reward token mining starts.
    uint256 public startBlock;
    // The block number when reward token mining ends.
    uint256 public bonusEndBlock;

    event Deposit(address indexed user, uint256 amount);
    event Withdraw(address indexed user, uint256 amount);
    event EmergencyWithdraw(address indexed user, uint256 amount);

    constructor(
        IHRC20 _xToken,
        IHRC20 _rewardToken,
```

```
        uint256 _rewardPerBlock,
        uint256 _startBlock,
        uint256 _bonusEndBlock
    ) public {
        xToken = _xToken;
        rewardToken = _rewardToken;
        rewardPerBlock = _rewardPerBlock;
        startBlock = _startBlock;
        bonusEndBlock = _bonusEndBlock;

        // staking pool
        poolInfo.push(PoolInfo({
            lpToken: _xToken,
            allocPoint: 1000,
            lastRewardBlock: startBlock,
            accRewardPerShare: 0
        }));

        totalAllocPoint = 1000;
        // maxStaking = 5000000000000000000;

    }

    function stopReward() public onlyOwner {
        bonusEndBlock = block.number;
    }


    // Return reward multiplier over the given _from to _to block.
    function getMultiplier(uint256 _from, uint256 _to) public view returns (uint256) {
        if (_to <= bonusEndBlock) {
            return _to.sub(_from);
        } else if (_from >= bonusEndBlock) {
            return 0;
        } else {
            return bonusEndBlock.sub(_from);
        }
    }

    // View function to see pending Reward on frontend.
    function pendingReward(address _user) external view returns (uint256) {
        PoolInfo storage pool = poolInfo[0];
        UserInfo storage user = userInfo[_user];
        uint256 accRewardPerShare = pool.accRewardPerShare;
        uint256 lpSupply = pool.lpToken.balanceOf(address(this));
        if (block.number > pool.lastRewardBlock && lpSupply != 0) {
            uint256 multiplier = getMultiplier(pool.lastRewardBlock, block.number);
            uint256 tokenReward = multiplier.mul(rewardPerBlock).mul(pool.allocPoint).div(totalAllocP
            accRewardPerShare = accRewardPerShare.add(tokenReward.mul(1e12).div(lpSupply));
        }
        return user.amount.mul(accRewardPerShare).div(1e12).sub(user.rewardDebt);
    }

    // Update reward variables of the given pool to be up-to-date.
    function updatePool(uint256 _pid) public {
        PoolInfo storage pool = poolInfo[_pid];
        if (block.number <= pool.lastRewardBlock) {
            return;
        }
        uint256 lpSupply = pool.lpToken.balanceOf(address(this));
        if (lpSupply == 0) {
            pool.lastRewardBlock = block.number;
            return;
        }
        uint256 multiplier = getMultiplier(pool.lastRewardBlock, block.number);
        uint256 tokenReward = multiplier.mul(rewardPerBlock).mul(pool.allocPoint).div(totalAllocPoint
```

```
        pool.accRewardPerShare = pool.accRewardPerShare.add(tokenReward.mul(1e12).div(lpSupply));
        pool.lastRewardBlock = block.number;
    }

    // Update reward variables for all pools. Be careful of gas spending!
    function massUpdatePools() public {
        uint256 length = poolInfo.length;
        for (uint256 pid = 0; pid < length; ++pid) {
            updatePool(pid);
        }
    }


    // Stake x tokens
    function deposit(uint256 _amount) public {
        PoolInfo storage pool = poolInfo[0];
        UserInfo storage user = userInfo[msg.sender];

        // require (_amount.add(user.amount) <= maxStaking, 'exceed max stake');

        updatePool(0);
        if (user.amount > 0) {
            uint256 pending = user.amount.mul(pool.accRewardPerShare).div(1e12).sub(user.rewardDebt);
            if(pending > 0) {
                rewardToken.safeTransfer(address(msg.sender), pending);
            }
        }
        if(_amount > 0) {
            pool.lpToken.safeTransferFrom(address(msg.sender), address(this), _amount);
            user.amount = user.amount.add(_amount);
        }
        user.rewardDebt = user.amount.mul(pool.accRewardPerShare).div(1e12);

        emit Deposit(msg.sender, _amount);
    }

    // Withdraw x tokens from STAKING.
    function withdraw(uint256 _amount) public {
        PoolInfo storage pool = poolInfo[0];
        UserInfo storage user = userInfo[msg.sender];
        require(user.amount >= _amount, "withdraw: not good");
        updatePool(0);
        uint256 pending = user.amount.mul(pool.accRewardPerShare).div(1e12).sub(user.rewardDebt);
        if(pending > 0) {
            rewardToken.safeTransfer(address(msg.sender), pending);
        }
        if(_amount > 0) {
            user.amount = user.amount.sub(_amount);
            pool.lpToken.safeTransfer(address(msg.sender), _amount);
        }
        user.rewardDebt = user.amount.mul(pool.accRewardPerShare).div(1e12);

        emit Withdraw(msg.sender, _amount);
    }

    // Withdraw without caring about rewards. EMERGENCY ONLY.
    function emergencyWithdraw() public {
        PoolInfo storage pool = poolInfo[0];
        UserInfo storage user = userInfo[msg.sender];
        uint256 amount = user.amount;
        pool.lpToken.safeTransfer(address(msg.sender), amount);
        user.amount = 0;
        user.rewardDebt = 0;
        emit EmergencyWithdraw(msg.sender, amount);
    }
```

```
    // Withdraw reward. EMERGENCY ONLY.
    function emergencyRewardWithdraw(uint256 _amount) public onlyOwner {
        require(_amount <= rewardToken.balanceOf(address(this)), 'not enough token');
        rewardToken.safeTransfer(address(msg.sender), _amount);
    }

}pragma solidity =0.6.6;

interface IDexFactory {
    event PairCreated(address indexed token0, address indexed token1, address pair, uint);

    function FEE_RATE_DENOMINATOR() external view returns (uint256);

    function feeRateNumerator() external view returns (uint256);

    function feeTo() external view returns (address);

    function feeToSetter() external view returns (address);

    function feeToRate() external view returns (uint256);

    function initCodeHash() external view returns (bytes32);

    function pairFeeToRate(address) external view returns (uint256);

    function pairFees(address) external view returns (uint256);

    function getPair(address tokenA, address tokenB) external view returns (address pair);

    function allPairs(uint) external view returns (address pair);

    function allPairsLength() external view returns (uint);

    function createPair(address tokenA, address tokenB) external returns (address pair);

    function setFeeTo(address) external;

    function setFeeToSetter(address) external;

    function addPair(address) external returns (bool);

    function delPair(address) external returns (bool);

    function getSupportListLength() external view returns (uint256);

    function isSupportPair(address pair) external view returns (bool);

    function getSupportPair(uint256 index) external view returns (address);

    function setFeeRateNumerator(uint256) external;

    function setPairFees(address pair, uint256 fee) external;

    function setDefaultFeeToRate(uint256) external;

    function setPairFeeToRate(address pair, uint256 rate) external;

    function getPairFees(address) external view returns (uint256);

    function getPairRate(address) external view returns (uint256);

    function sortTokens(address tokenA, address tokenB) external pure returns (address token0, addres

    function pairFor(address tokenA, address tokenB) external view returns (address pair);

    function getReserves(address tokenA, address tokenB) external view returns (uint256 reserveA, uin
```

```
    function quote(uint256 amountA, uint256 reserveA, uint256 reserveB) external pure returns (uint25

    function getAmountOut(uint256 amountIn, uint256 reserveIn, uint256 reserveOut, address token0, ad

    function getAmountIn(uint256 amountOut, uint256 reserveIn, uint256 reserveOut, address token0, ad

    function getAmountsOut(uint256 amountIn, address[] calldata path) external view returns (uint256[

    function getAmountsIn(uint256 amountOut, address[] calldata path) external view returns (uint256[
}

interface IDexPair {
    event Approval(address indexed owner, address indexed spender, uint value);
    event Transfer(address indexed from, address indexed to, uint value);

    function name() external pure returns (string memory);

    function symbol() external pure returns (string memory);

    function decimals() external pure returns (uint8);

    function totalSupply() external view returns (uint);

    function balanceOf(address owner) external view returns (uint);

    function allowance(address owner, address spender) external view returns (uint);

    function approve(address spender, uint value) external returns (bool);

    function transfer(address to, uint value) external returns (bool);

    function transferFrom(address from, address to, uint value) external returns (bool);

    function DOMAIN_SEPARATOR() external view returns (bytes32);

    function PERMIT_TYPEHASH() external pure returns (bytes32);

    function nonces(address owner) external view returns (uint);

    function permit(address owner, address spender, uint value, uint deadline, uint8 v, bytes32 r, by

    event Mint(address indexed sender, uint amount0, uint amount1);
    event Burn(address indexed sender, uint amount0, uint amount1, address indexed to);
    event Swap(
        address indexed sender,
        uint amount0In,
        uint amount1In,
        uint amount0Out,
        uint amount1Out,
        address indexed to
    );
    event Sync(uint112 reserve0, uint112 reserve1);

    function MINIMUM_LIQUIDITY() external pure returns (uint);

    function factory() external view returns (address);

    function token0() external view returns (address);

    function token1() external view returns (address);

    function getReserves() external view returns (uint112 reserve0, uint112 reserve1, uint32 blockTim

    function price0CumulativeLast() external view returns (uint);
```

```
    function price1CumulativeLast() external view returns (uint);

    function kLast() external view returns (uint);

    function mint(address to) external returns (uint liquidity);

    function burn(address to) external returns (uint amount0, uint amount1);

    function swap(uint amount0Out, uint amount1Out, address to, bytes calldata data) external;

    function skim(address to) external;

    function sync() external;

    function initialize(address, address) external;
}

interface IDexRouter {
    function factory() external pure returns (address);

    function WHT() external pure returns (address);

    function swapMining() external pure returns (address);

    function addLiquidity(
        address tokenA,
        address tokenB,
        uint amountADesired,
        uint amountBDesired,
        uint amountAMin,
        uint amountBMin,
        address to,
        uint deadline
    ) external returns (uint amountA, uint amountB, uint liquidity);

    function addLiquidityETH(
        address token,
        uint amountTokenDesired,
        uint amountTokenMin,
        uint amountETHMin,
        address to,
        uint deadline
    ) external payable returns (uint amountToken, uint amountETH, uint liquidity);

    function removeLiquidity(
        address tokenA,
        address tokenB,
        uint liquidity,
        uint amountAMin,
        uint amountBMin,
        address to,
        uint deadline
    ) external returns (uint amountA, uint amountB);

    function removeLiquidityETH(
        address token,
        uint liquidity,
        uint amountTokenMin,
        uint amountETHMin,
        address to,
        uint deadline
    ) external returns (uint amountToken, uint amountETH);

    function removeLiquidityWithPermit(
        address tokenA,
        address tokenB,
```

```
        uint liquidity,
        uint amountAMin,
        uint amountBMin,
        address to,
        uint deadline,
        bool approveMax, uint8 v, bytes32 r, bytes32 s
    ) external returns (uint amountA, uint amountB);

    function removeLiquidityETHWithPermit(
        address token,
        uint liquidity,
        uint amountTokenMin,
        uint amountETHMin,
        address to,
        uint deadline,
        bool approveMax, uint8 v, bytes32 r, bytes32 s
    ) external returns (uint amountToken, uint amountETH);

    function swapExactTokensForTokens(
        uint amountIn,
        uint amountOutMin,
        address[] calldata path,
        address to,
        uint deadline
    ) external returns (uint[] memory amounts);

    function swapTokensForExactTokens(
        uint amountOut,
        uint amountInMax,
        address[] calldata path,
        address to,
        uint deadline
    ) external returns (uint[] memory amounts);

    function swapExactETHForTokens(uint amountOutMin, address[] calldata path, address to, uint deadl
    external
    payable
    returns (uint[] memory amounts);

    function swapTokensForExactETH(uint amountOut, uint amountInMax, address[] calldata path, address
    external
    returns (uint[] memory amounts);

    function swapExactTokensForETH(uint amountIn, uint amountOutMin, address[] calldata path, address
    external
    returns (uint[] memory amounts);

    function swapETHForExactTokens(uint amountOut, address[] calldata path, address to, uint deadline
    external
    payable
    returns (uint[] memory amounts);

    function quote(uint256 amountA, uint256 reserveA, uint256 reserveB) external view returns (uint25

    function getAmountOut(uint256 amountIn, uint256 reserveIn, uint256 reserveOut, address token0, ad

    function getAmountIn(uint256 amountOut, uint256 reserveIn, uint256 reserveOut, address token0, ad

    function getAmountsOut(uint256 amountIn, address[] calldata path) external view returns (uint256[

    function getAmountsIn(uint256 amountOut, address[] calldata path) external view returns (uint256[

    function removeLiquidityETHSupportingFeeOnTransferTokens(
        address token,
        uint liquidity,
        uint amountTokenMin,
```

```
        uint amountETHMin,
        address to,
        uint deadline
    ) external returns (uint amountETH);

    function removeLiquidityETHWithPermitSupportingFeeOnTransferTokens(
        address token,
        uint liquidity,
        uint amountTokenMin,
        uint amountETHMin,
        address to,
        uint deadline,
        bool approveMax, uint8 v, bytes32 r, bytes32 s
    ) external returns (uint amountETH);

    function swapExactTokensForTokensSupportingFeeOnTransferTokens(
        uint amountIn,
        uint amountOutMin,
        address[] calldata path,
        address to,
        uint deadline
    ) external;

    function swapExactETHForTokensSupportingFeeOnTransferTokens(
        uint amountOutMin,
        address[] calldata path,
        address to,
        uint deadline
    ) external payable;

    function swapExactTokensForETHSupportingFeeOnTransferTokens(
        uint amountIn,
        uint amountOutMin,
        address[] calldata path,
        address to,
        uint deadline
    ) external;
}

interface ISwapMining {
    function swap(address account, address input, address output, uint256 amount) external returns (b
}

interface IERC20 {
    event Approval(address indexed owner, address indexed spender, uint value);
    event Transfer(address indexed from, address indexed to, uint value);

    function name() external view returns (string memory);

    function symbol() external view returns (string memory);

    function decimals() external view returns (uint8);

    function totalSupply() external view returns (uint);

    function balanceOf(address owner) external view returns (uint);

    function allowance(address owner, address spender) external view returns (uint);

    function approve(address spender, uint value) external returns (bool);

    function transfer(address to, uint value) external returns (bool);

    function transferFrom(address from, address to, uint value) external returns (bool);
}
```

```
interface IWHT {
    function deposit() external payable;

    function transfer(address to, uint value) external returns (bool);

    function withdraw(uint) external;
}

library SafeMath {
    uint256 constant WAD = 10 ** 18;
    uint256 constant RAY = 10 ** 27;

    function wad() public pure returns (uint256) {
        return WAD;
    }

    function ray() public pure returns (uint256) {
        return RAY;
    }

    function add(uint256 a, uint256 b) internal pure returns (uint256) {
        uint256 c = a + b;
        require(c >= a, "SafeMath: addition overflow");

        return c;
    }

    function sub(uint256 a, uint256 b) internal pure returns (uint256) {
        return sub(a, b, "SafeMath: subtraction overflow");
    }

    function sub(uint256 a, uint256 b, string memory errorMessage) internal pure returns (uint256) {
        require(b <= a, errorMessage);
        uint256 c = a - b;

        return c;
    }

    function mul(uint256 a, uint256 b) internal pure returns (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 0;
        }

        uint256 c = a * b;
        require(c / a == b, "SafeMath: multiplication overflow");

        return c;
    }

    function div(uint256 a, uint256 b) internal pure returns (uint256) {
        return div(a, b, "SafeMath: division by zero");
    }

    function div(uint256 a, uint256 b, string memory errorMessage) internal pure returns (uint256) {
        // Solidity only automatically asserts when dividing by 0
        require(b > 0, errorMessage);
        uint256 c = a / b;
        // assert(a == b * c + a % b); // There is no case in which this doesn't hold

        return c;
    }

    function mod(uint256 a, uint256 b) internal pure returns (uint256) {
```

```
        return mod(a, b, "SafeMath: modulo by zero");
    }

    function mod(uint256 a, uint256 b, string memory errorMessage) internal pure returns (uint256) {
        require(b != 0, errorMessage);
        return a % b;
    }

    function min(uint256 a, uint256 b) internal pure returns (uint256) {
        return a <= b ? a : b;
    }

    function max(uint256 a, uint256 b) internal pure returns (uint256) {
        return a >= b ? a : b;
    }

    function sqrt(uint256 a) internal pure returns (uint256 b) {
        if (a > 3) {
            b = a;
            uint256 x = a / 2 + 1;
            while (x < b) {
                b = x;
                x = (a / x + x) / 2;
            }
        } else if (a != 0) {
            b = 1;
        }
    }

    function wmul(uint256 a, uint256 b) internal pure returns (uint256) {
        return mul(a, b) / WAD;
    }

    function wmulRound(uint256 a, uint256 b) internal pure returns (uint256) {
        return add(mul(a, b), WAD / 2) / WAD;
    }

    function rmul(uint256 a, uint256 b) internal pure returns (uint256) {
        return mul(a, b) / RAY;
    }

    function rmulRound(uint256 a, uint256 b) internal pure returns (uint256) {
        return add(mul(a, b), RAY / 2) / RAY;
    }

    function wdiv(uint256 a, uint256 b) internal pure returns (uint256) {
        return div(mul(a, WAD), b);
    }

    function wdivRound(uint256 a, uint256 b) internal pure returns (uint256) {
        return add(mul(a, WAD), b / 2) / b;
    }

    function rdiv(uint256 a, uint256 b) internal pure returns (uint256) {
        return div(mul(a, RAY), b);
    }

    function rdivRound(uint256 a, uint256 b) internal pure returns (uint256) {
        return add(mul(a, RAY), b / 2) / b;
    }

    function wpow(uint256 x, uint256 n) internal pure returns (uint256) {
        uint256 result = WAD;
        while (n > 0) {
            if (n % 2 != 0) {
                result = wmul(result, x);
```

```
            }
            x = wmul(x, x);
            n /= 2;
        }
        return result;
    }

    function rpow(uint256 x, uint256 n) internal pure returns (uint256) {
        uint256 result = RAY;
        while (n > 0) {
            if (n % 2 != 0) {
                result = rmul(result, x);
            }
            x = rmul(x, x);
            n /= 2;
        }
        return result;
    }
}

library TransferHelper {
    function safeApprove(address token, address to, uint value) internal {
        // bytes4(keccak256(bytes('approve(address,uint256)')));
        (bool success, bytes memory data) = token.call(abi.encodeWithSelector(0x095ea7b3, to, value))
        require(success && (data.length == 0 || abi.decode(data, (bool))), 'TransferHelper: APPROVE_F
    }

    function safeTransfer(address token, address to, uint value) internal {
        // bytes4(keccak256(bytes('transfer(address,uint256)')));
        (bool success, bytes memory data) = token.call(abi.encodeWithSelector(0xa9059cbb, to, value))
        require(success && (data.length == 0 || abi.decode(data, (bool))), 'TransferHelper: TRANSFER_
    }

    function safeTransferFrom(address token, address from, address to, uint value) internal {
        // bytes4(keccak256(bytes('transferFrom(address,address,uint256)')));
        (bool success, bytes memory data) = token.call(abi.encodeWithSelector(0x23b872dd, from, to, v
        require(success && (data.length == 0 || abi.decode(data, (bool))), 'TransferHelper: TRANSFER_
    }

    function safeTransferETH(address to, uint value) internal {
        (bool success,) = to.call{value : value}(new bytes(0));
        require(success, 'TransferHelper: ETH_TRANSFER_FAILED');
    }
}

contract Ownable {
    address private _owner;

    constructor () internal {
        _owner = msg.sender;
        emit OwnershipTransferred(address(0), _owner);
    }

    function owner() public view returns (address) {
        return _owner;
    }

    function isOwner(address account) public view returns (bool) {
        return account == _owner;
    }

    function renounceOwnership() public onlyOwner {
        emit OwnershipTransferred(_owner, address(0));
        _owner = address(0);
    }
```

```
    function _transferOwnership(address newOwner) internal {
        require(newOwner != address(0), "Ownable: new owner is the zero address");
        emit OwnershipTransferred(_owner, newOwner);
        _owner = newOwner;
    }

    function transferOwnership(address newOwner) public onlyOwner {
        _transferOwnership(newOwner);
    }


    modifier onlyOwner() {
        require(isOwner(msg.sender), "Ownable: caller is not the owner");
        _;
    }

    event OwnershipTransferred(address indexed previousOwner, address indexed newOwner);
}

contract DexRouter is IDexRouter, Ownable {
    using SafeMath for uint256;

    address public immutable override factory;
    address public immutable override WHT;
    address public override swapMining;

    modifier ensure(uint deadline) {
        require(deadline >= block.timestamp, 'DexRouter: EXPIRED');
        _;
    }

    constructor(address _factory, address _WHT) public {
        factory = _factory;
        WHT = _WHT;
    }

    receive() external payable {
        assert(msg.sender == WHT);
        // only accept BNB via fallback from the WBNB contract
    }

    function pairFor(address tokenA, address tokenB) public view returns (address pair){
        pair = IDexFactory(factory).pairFor(tokenA, tokenB);
    }

    function setSwapMining(address _swapMininng) public onlyOwner {
        swapMining = _swapMininng;
    }

    // **** ADD LIQUIDITY ****
    function _addLiquidity(
        address tokenA,
        address tokenB,
        uint amountADesired,
        uint amountBDesired,
        uint amountAMin,
        uint amountBMin
    ) internal virtual returns (uint amountA, uint amountB) {
        // create the pair if it doesn't exist yet
        if (IDexFactory(factory).getPair(tokenA, tokenB) == address(0)) {
            IDexFactory(factory).createPair(tokenA, tokenB);
        }
        (uint reserveA, uint reserveB) = IDexFactory(factory).getReserves(tokenA, tokenB);
        if (reserveA == 0 && reserveB == 0) {
            (amountA, amountB) = (amountADesired, amountBDesired);
        } else {
```

```
            uint amountBOptimal = IDexFactory(factory).quote(amountADesired, reserveA, reserveB);
            if (amountBOptimal <= amountBDesired) {
                require(amountBOptimal >= amountBMin, 'DexRouter: INSUFFICIENT_B_AMOUNT');
                (amountA, amountB) = (amountADesired, amountBOptimal);
            } else {
                uint amountAOptimal = IDexFactory(factory).quote(amountBDesired, reserveB, reserveA);
                assert(amountAOptimal <= amountADesired);
                require(amountAOptimal >= amountAMin, 'DexRouter: INSUFFICIENT_A_AMOUNT');
                (amountA, amountB) = (amountAOptimal, amountBDesired);
            }
        }
    }

    function addLiquidity(
        address tokenA,
        address tokenB,
        uint amountADesired,
        uint amountBDesired,
        uint amountAMin,
        uint amountBMin,
        address to,
        uint deadline
    ) external virtual override ensure(deadline) returns (uint amountA, uint amountB, uint liquidity)
        (amountA, amountB) = _addLiquidity(tokenA, tokenB, amountADesired, amountBDesired, amountAMin
        address pair = pairFor(tokenA, tokenB);
        TransferHelper.safeTransferFrom(tokenA, msg.sender, pair, amountA);
        TransferHelper.safeTransferFrom(tokenB, msg.sender, pair, amountB);
        liquidity = IDexPair(pair).mint(to);
    }

    function addLiquidityETH(
        address token,
        uint amountTokenDesired,
        uint amountTokenMin,
        uint amountETHMin,
        address to,
        uint deadline
    ) external virtual override payable ensure(deadline) returns (uint amountToken, uint amountETH, u
        (amountToken, amountETH) = _addLiquidity(
            token,
            WHT,
            amountTokenDesired,
            msg.value,
            amountTokenMin,
            amountETHMin
        );
        address pair = pairFor(token, WHT);
        TransferHelper.safeTransferFrom(token, msg.sender, pair, amountToken);
        IWHT(WHT).deposit{value : amountETH}();
        assert(IWHT(WHT).transfer(pair, amountETH));
        liquidity = IDexPair(pair).mint(to);
        // refund dust eth, if any
        if (msg.value > amountETH) TransferHelper.safeTransferETH(msg.sender, msg.value - amountETH);
    }

    // **** REMOVE LIQUIDITY ****
    function removeLiquidity(
        address tokenA,
        address tokenB,
        uint liquidity,
        uint amountAMin,
        uint amountBMin,
        address to,
        uint deadline
    ) public virtual override ensure(deadline) returns (uint amountA, uint amountB) {
        address pair = pairFor(tokenA, tokenB);
```

```
        IDexPair(pair).transferFrom(msg.sender, pair, liquidity);
        // send liquidity to pair
        (uint amount0, uint amount1) = IDexPair(pair).burn(to);
        (address token0,) = IDexFactory(factory).sortTokens(tokenA, tokenB);
        (amountA, amountB) = tokenA == token0 ? (amount0, amount1) : (amount1, amount0);
        require(amountA >= amountAMin, 'DexRouter: INSUFFICIENT_A_AMOUNT');
        require(amountB >= amountBMin, 'DexRouter: INSUFFICIENT_B_AMOUNT');
    }

    function removeLiquidityETH(
        address token,
        uint liquidity,
        uint amountTokenMin,
        uint amountETHMin,
        address to,
        uint deadline
    ) public virtual override ensure(deadline) returns (uint amountToken, uint amountETH) {
        (amountToken, amountETH) = removeLiquidity(
            token,
            WHT,
            liquidity,
            amountTokenMin,
            amountETHMin,
            address(this),
            deadline
        );
        TransferHelper.safeTransfer(token, to, amountToken);
        IWHT(WHT).withdraw(amountETH);
        TransferHelper.safeTransferETH(to, amountETH);
    }

    function removeLiquidityWithPermit(
        address tokenA,
        address tokenB,
        uint liquidity,
        uint amountAMin,
        uint amountBMin,
        address to,
        uint deadline,
        bool approveMax, uint8 v, bytes32 r, bytes32 s
    ) external virtual override returns (uint amountA, uint amountB) {
        address pair = pairFor(tokenA, tokenB);
        uint value = approveMax ? uint(- 1) : liquidity;
        IDexPair(pair).permit(msg.sender, address(this), value, deadline, v, r, s);
        (amountA, amountB) = removeLiquidity(tokenA, tokenB, liquidity, amountAMin, amountBMin, to, d
    }

    function removeLiquidityETHWithPermit(
        address token,
        uint liquidity,
        uint amountTokenMin,
        uint amountETHMin,
        address to,
        uint deadline,
        bool approveMax, uint8 v, bytes32 r, bytes32 s
    ) external virtual override returns (uint amountToken, uint amountETH) {
        address pair = pairFor(token, WHT);
        uint value = approveMax ? uint(- 1) : liquidity;
        IDexPair(pair).permit(msg.sender, address(this), value, deadline, v, r, s);
        (amountToken, amountETH) = removeLiquidityETH(token, liquidity, amountTokenMin, amountETHMin,
    }

    // **** REMOVE LIQUIDITY (supporting fee-on-transfer tokens) ****
    function removeLiquidityETHSupportingFeeOnTransferTokens(
        address token,
        uint liquidity,
```

```
        uint amountTokenMin,
        uint amountETHMin,
        address to,
        uint deadline
    ) public virtual override ensure(deadline) returns (uint amountETH) {
        (, amountETH) = removeLiquidity(
            token,
            WHT,
            liquidity,
            amountTokenMin,
            amountETHMin,
            address(this),
            deadline
        );
        TransferHelper.safeTransfer(token, to, IERC20(token).balanceOf(address(this)));
        IWHT(WHT).withdraw(amountETH);
        TransferHelper.safeTransferETH(to, amountETH);
    }

    function removeLiquidityETHWithPermitSupportingFeeOnTransferTokens(
        address token,
        uint liquidity,
        uint amountTokenMin,
        uint amountETHMin,
        address to,
        uint deadline,
        bool approveMax, uint8 v, bytes32 r, bytes32 s
    ) external virtual override returns (uint amountETH) {
        address pair = pairFor(token, WHT);
        uint value = approveMax ? uint(- 1) : liquidity;
        IDexPair(pair).permit(msg.sender, address(this), value, deadline, v, r, s);
        amountETH = removeLiquidityETHSupportingFeeOnTransferTokens(
            token, liquidity, amountTokenMin, amountETHMin, to, deadline
        );
    }

    // **** SWAP ****
    // requires the initial amount to have already been sent to the first pair
    function _swap(uint[] memory amounts, address[] memory path, address _to) internal virtual {
        for (uint i; i < path.length - 1; i++) {
            (address input, address output) = (path[i], path[i + 1]);
            (address token0,) = IDexFactory(factory).sortTokens(input, output);
            uint amountOut = amounts[i + 1];
            if (swapMining != address(0)) {
                ISwapMining(swapMining).swap(msg.sender, input, output, amountOut);
            }
            (uint amount0Out, uint amount1Out) = input == token0 ? (uint(0), amountOut) : (amountOut,
            address to = i < path.length - 2 ? pairFor(output, path[i + 2]) : _to;
            IDexPair(pairFor(input, output)).swap(
                amount0Out, amount1Out, to, new bytes(0)
            );
        }
    }

    function swapExactTokensForTokens(
        uint amountIn,
        uint amountOutMin,
        address[] calldata path,
        address to,
        uint deadline
    ) external virtual override ensure(deadline) returns (uint[] memory amounts) {
        amounts = IDexFactory(factory).getAmountsOut(amountIn, path);
        require(amounts[amounts.length - 1] >= amountOutMin, 'DexRouter: INSUFFICIENT_OUTPUT_AMOUNT')
        TransferHelper.safeTransferFrom(
            path[0], msg.sender, pairFor(path[0], path[1]), amounts[0]
        );
```

```
        _swap(amounts, path, to);
    }

    function swapTokensForExactTokens(
        uint amountOut,
        uint amountInMax,
        address[] calldata path,
        address to,
        uint deadline
    ) external virtual override ensure(deadline) returns (uint[] memory amounts) {
        amounts = IDexFactory(factory).getAmountsIn(amountOut, path);
        require(amounts[0] <= amountInMax, 'DexRouter: EXCESSIVE_INPUT_AMOUNT');
        TransferHelper.safeTransferFrom(
            path[0], msg.sender, pairFor(path[0], path[1]), amounts[0]
        );
        _swap(amounts, path, to);
    }

    function swapExactETHForTokens(uint amountOutMin, address[] calldata path, address to, uint deadl
    external
    virtual
    override
    payable
    ensure(deadline)
    returns (uint[] memory amounts)
    {
        require(path[0] == WHT, 'DexRouter: INVALID_PATH');
        amounts = IDexFactory(factory).getAmountsOut(msg.value, path);
        require(amounts[amounts.length - 1] >= amountOutMin, 'DexRouter: INSUFFICIENT_OUTPUT_AMOUNT')
        IWHT(WHT).deposit{value : amounts[0]}();
        assert(IWHT(WHT).transfer(pairFor(path[0], path[1]), amounts[0]));
        _swap(amounts, path, to);
    }

    function swapTokensForExactETH(uint amountOut, uint amountInMax, address[] calldata path, address
    external
    virtual
    override
    ensure(deadline)
    returns (uint[] memory amounts)
    {
        require(path[path.length - 1] == WHT, 'DexRouter: INVALID_PATH');
        amounts = IDexFactory(factory).getAmountsIn(amountOut, path);
        require(amounts[0] <= amountInMax, 'DexRouter: EXCESSIVE_INPUT_AMOUNT');
        TransferHelper.safeTransferFrom(
            path[0], msg.sender, pairFor(path[0], path[1]), amounts[0]
        );
        _swap(amounts, path, address(this));
        IWHT(WHT).withdraw(amounts[amounts.length - 1]);
        TransferHelper.safeTransferETH(to, amounts[amounts.length - 1]);
    }

    function swapExactTokensForETH(uint amountIn, uint amountOutMin, address[] calldata path, address
    external
    virtual
    override
    ensure(deadline)
    returns (uint[] memory amounts)
    {
        require(path[path.length - 1] == WHT, 'DexRouter: INVALID_PATH');
        amounts = IDexFactory(factory).getAmountsOut(amountIn, path);
        require(amounts[amounts.length - 1] >= amountOutMin, 'DexRouter: INSUFFICIENT_OUTPUT_AMOUNT')
        TransferHelper.safeTransferFrom(
            path[0], msg.sender, pairFor(path[0], path[1]), amounts[0]
        );
        _swap(amounts, path, address(this));
```

```
        IWHT(WHT).withdraw(amounts[amounts.length - 1]);
        TransferHelper.safeTransferETH(to, amounts[amounts.length - 1]);
    }

    function swapETHForExactTokens(uint amountOut, address[] calldata path, address to, uint deadline
    external
    virtual
    override
    payable
    ensure(deadline)
    returns (uint[] memory amounts)
    {
        require(path[0] == WHT, 'DexRouter: INVALID_PATH');
        amounts = IDexFactory(factory).getAmountsIn(amountOut, path);
        require(amounts[0] <= msg.value, 'DexRouter: EXCESSIVE_INPUT_AMOUNT');
        IWHT(WHT).deposit{value : amounts[0]}();
        assert(IWHT(WHT).transfer(pairFor(path[0], path[1]), amounts[0]));
        _swap(amounts, path, to);
        // refund dust eth, if any
        if (msg.value > amounts[0]) TransferHelper.safeTransferETH(msg.sender, msg.value - amounts[0]
    }

    // **** SWAP (supporting fee-on-transfer tokens) ****
    // requires the initial amount to have already been sent to the first pair
    function _swapSupportingFeeOnTransferTokens(address[] memory path, address _to) internal virtual
        for (uint i; i < path.length - 1; i++) {
            (address input, address output) = (path[i], path[i + 1]);
            (address token0,) = IDexFactory(factory).sortTokens(input, output);
            IDexPair pair = IDexPair(pairFor(input, output));
            uint amountInput;
            uint amountOutput;
            {// scope to avoid stack too deep errors
                (uint reserve0, uint reserve1,) = pair.getReserves();
                (uint reserveInput, uint reserveOutput) = input == token0 ? (reserve0, reserve1) : (r
                amountInput = IERC20(input).balanceOf(address(pair)).sub(reserveInput);
                amountOutput = IDexFactory(factory).getAmountOut(amountInput, reserveInput, reserveOu
            }
            if (swapMining != address(0)) {
                ISwapMining(swapMining).swap(msg.sender, input, output, amountOutput);
            }
            (uint amount0Out, uint amount1Out) = input == token0 ? (uint(0), amountOutput) : (amountO
            address to = i < path.length - 2 ? pairFor(output, path[i + 2]) : _to;
            pair.swap(amount0Out, amount1Out, to, new bytes(0));
        }
    }

    function swapExactTokensForTokensSupportingFeeOnTransferTokens(
        uint amountIn,
        uint amountOutMin,
        address[] calldata path,
        address to,
        uint deadline
    ) external virtual override ensure(deadline) {
        TransferHelper.safeTransferFrom(
            path[0], msg.sender, pairFor(path[0], path[1]), amountIn
        );
        uint balanceBefore = IERC20(path[path.length - 1]).balanceOf(to);
        _swapSupportingFeeOnTransferTokens(path, to);
        require(
            IERC20(path[path.length - 1]).balanceOf(to).sub(balanceBefore) >= amountOutMin,
            'DexRouter: INSUFFICIENT_OUTPUT_AMOUNT'
        );
    }

    function swapExactETHForTokensSupportingFeeOnTransferTokens(
        uint amountOutMin,
```

```
        address[] calldata path,
        address to,
        uint deadline
    )
    external
    virtual
    override
    payable
    ensure(deadline)
    {
        require(path[0] == WHT, 'DexRouter: INVALID_PATH');
        uint amountIn = msg.value;
        IWHT(WHT).deposit{value : amountIn}();
        assert(IWHT(WHT).transfer(pairFor(path[0], path[1]), amountIn));
        uint balanceBefore = IERC20(path[path.length - 1]).balanceOf(to);
        _swapSupportingFeeOnTransferTokens(path, to);
        require(
            IERC20(path[path.length - 1]).balanceOf(to).sub(balanceBefore) >= amountOutMin,
            'DexRouter: INSUFFICIENT_OUTPUT_AMOUNT'
        );
    }

    function swapExactTokensForETHSupportingFeeOnTransferTokens(
        uint amountIn,
        uint amountOutMin,
        address[] calldata path,
        address to,
        uint deadline
    )
    external
    virtual
    override
    ensure(deadline)
    {
        require(path[path.length - 1] == WHT, 'DexRouter: INVALID_PATH');
        TransferHelper.safeTransferFrom(
            path[0], msg.sender, pairFor(path[0], path[1]), amountIn
        );
        _swapSupportingFeeOnTransferTokens(path, address(this));
        uint amountOut = IERC20(WHT).balanceOf(address(this));
        require(amountOut >= amountOutMin, 'DexRouter: INSUFFICIENT_OUTPUT_AMOUNT');
        IWHT(WHT).withdraw(amountOut);
        TransferHelper.safeTransferETH(to, amountOut);
    }

    // **** LIBRARY FUNCTIONS ****
    function quote(uint256 amountA, uint256 reserveA, uint256 reserveB) public view override returns
        return IDexFactory(factory).quote(amountA, reserveA, reserveB);
    }

    function getAmountOut(uint256 amountIn, uint256 reserveIn, uint256 reserveOut, address token0, ad
        return IDexFactory(factory).getAmountOut(amountIn, reserveIn, reserveOut, token0, token1);
    }

    function getAmountIn(uint256 amountOut, uint256 reserveIn, uint256 reserveOut, address token0, ad
        return IDexFactory(factory).getAmountIn(amountOut, reserveIn, reserveOut, token0, token1);
    }

    function getAmountsOut(uint256 amountIn, address[] memory path) public view override returns (uin
        return IDexFactory(factory).getAmountsOut(amountIn, path);
    }

    function getAmountsIn(uint256 amountOut, address[] memory path) public view override returns (uin
        return IDexFactory(factory).getAmountsIn(amountOut, path);
    }
```

```
}pragma solidity =0.6.6;

library SafeMath {
    function add(uint x, uint y) internal pure returns (uint z) {
        require((z = x + y) >= x, 'ds-math-add-overflow');
    }

    function sub(uint x, uint y) internal pure returns (uint z) {
        require((z = x - y) <= x, 'ds-math-sub-underflow');
    }

    function mul(uint x, uint y) internal pure returns (uint z) {
        require(y == 0 || (z = x * y) / y == x, 'ds-math-mul-overflow');
    }
}

library FixedPoint {
    // range: [0, 2**112 - 1]
    // resolution: 1 / 2**112
    struct uq112x112 {
        uint224 _x;
    }

    // range: [0, 2**144 - 1]
    // resolution: 1 / 2**112
    struct uq144x112 {
        uint _x;
    }

    uint8 private constant RESOLUTION = 112;

    // encode a uint112 as a UQ112x112
    function encode(uint112 x) internal pure returns (uq112x112 memory) {
        return uq112x112(uint224(x) << RESOLUTION);
    }

    // encodes a uint144 as a UQ144x112
    function encode144(uint144 x) internal pure returns (uq144x112 memory) {
        return uq144x112(uint256(x) << RESOLUTION);
    }

    // divide a UQ112x112 by a uint112, returning a UQ112x112
    function div(uq112x112 memory self, uint112 x) internal pure returns (uq112x112 memory) {
        require(x != 0, 'FixedPoint: DIV_BY_ZERO');
        return uq112x112(self._x / uint224(x));
    }

    // multiply a UQ112x112 by a uint, returning a UQ144x112
    // reverts on overflow
    function mul(uq112x112 memory self, uint y) internal pure returns (uq144x112 memory) {
        uint z;
        require(y == 0 || (z = uint(self._x) * y) / y == uint(self._x), "FixedPoint: MULTIPLICATION_O
        return uq144x112(z);
    }

    // returns a UQ112x112 which represents the ratio of the numerator to the denominator
    // equivalent to encode(numerator).div(denominator)
    function fraction(uint112 numerator, uint112 denominator) internal pure returns (uq112x112 memory
        require(denominator > 0, "FixedPoint: DIV_BY_ZERO");
        return uq112x112((uint224(numerator) << RESOLUTION) / denominator);
    }

    // decode a UQ112x112 into a uint112 by truncating after the radix point
    function decode(uq112x112 memory self) internal pure returns (uint112) {
        return uint112(self._x >> RESOLUTION);
    }
```

```
    // decode a UQ144x112 into a uint144 by truncating after the radix point
    function decode144(uq144x112 memory self) internal pure returns (uint144) {
        return uint144(self._x >> RESOLUTION);
    }
}

interface IDexPair {
    event Approval(address indexed owner, address indexed spender, uint value);
    event Transfer(address indexed from, address indexed to, uint value);

    function name() external pure returns (string memory);

    function symbol() external pure returns (string memory);

    function decimals() external pure returns (uint8);

    function totalSupply() external view returns (uint);

    function balanceOf(address owner) external view returns (uint);

    function allowance(address owner, address spender) external view returns (uint);

    function approve(address spender, uint value) external returns (bool);

    function transfer(address to, uint value) external returns (bool);

    function transferFrom(address from, address to, uint value) external returns (bool);

    function DOMAIN_SEPARATOR() external view returns (bytes32);

    function PERMIT_TYPEHASH() external pure returns (bytes32);

    function nonces(address owner) external view returns (uint);

    function permit(address owner, address spender, uint value, uint deadline, uint8 v, bytes32 r, by

    event Mint(address indexed sender, uint amount0, uint amount1);
    event Burn(address indexed sender, uint amount0, uint amount1, address indexed to);
    event Swap(
        address indexed sender,
        uint amount0In,
        uint amount1In,
        uint amount0Out,
        uint amount1Out,
        address indexed to
    );
    event Sync(uint112 reserve0, uint112 reserve1);

    function MINIMUM_LIQUIDITY() external pure returns (uint);

    function factory() external view returns (address);

    function token0() external view returns (address);

    function token1() external view returns (address);

    function getReserves() external view returns (uint112 reserve0, uint112 reserve1, uint32 blockTim

    function price0CumulativeLast() external view returns (uint);

    function price1CumulativeLast() external view returns (uint);

    function kLast() external view returns (uint);

    function mint(address to) external returns (uint liquidity);
```

```
    function burn(address to) external returns (uint amount0, uint amount1);

    function swap(uint amount0Out, uint amount1Out, address to, bytes calldata data) external;

    function skim(address to) external;

    function sync() external;

    function price(address token, uint256 baseDecimal) external view returns (uint256);

    function initialize(address, address) external;
}

library DexOracleLibrary {
    using FixedPoint for *;

    // helper function that returns the current block timestamp within the range of uint32, i.e. [0,
    function currentBlockTimestamp() internal view returns (uint32) {
        return uint32(block.timestamp % 2 ** 32);
    }

    // produces the cumulative price using counterfactuals to save gas and avoid a call to sync.
    function currentCumulativePrices(
        address pair
    ) internal view returns (uint price0Cumulative, uint price1Cumulative, uint32 blockTimestamp) {
        blockTimestamp = currentBlockTimestamp();
        price0Cumulative = IDexPair(pair).price0CumulativeLast();
        price1Cumulative = IDexPair(pair).price1CumulativeLast();

        // if time has elapsed since the last update on the pair, mock the accumulated price values
        (uint112 reserve0, uint112 reserve1, uint32 blockTimestampLast) = IDexPair(pair).getReserves(
        if (blockTimestampLast != blockTimestamp) {
            // subtraction overflow is desired
            uint32 timeElapsed = blockTimestamp - blockTimestampLast;
            // addition overflow is desired
            // counterfactual
            price0Cumulative += uint(FixedPoint.fraction(reserve1, reserve0)._x) * timeElapsed;
            // counterfactual
            price1Cumulative += uint(FixedPoint.fraction(reserve0, reserve1)._x) * timeElapsed;
        }
    }
}

interface IDexFactory {
    event PairCreated(address indexed token0, address indexed token1, address pair, uint);

    function feeTo() external view returns (address);

    function feeToSetter() external view returns (address);

    function feeToRate() external view returns (uint256);

    function initCodeHash() external view returns (bytes32);

    function getPair(address tokenA, address tokenB) external view returns (address pair);

    function allPairs(uint) external view returns (address pair);

    function allPairsLength() external view returns (uint);

    function createPair(address tokenA, address tokenB) external returns (address pair);

    function setFeeTo(address) external;

    function setFeeToSetter(address) external;
```

```
    function setFeeToRate(uint256) external;

    function setInitCodeHash(bytes32) external;

    function sortTokens(address tokenA, address tokenB) external pure returns (address token0, addres

    function pairFor(address tokenA, address tokenB) external view returns (address pair);

    function getReserves(address tokenA, address tokenB) external view returns (uint256 reserveA, uin

    function quote(uint256 amountA, uint256 reserveA, uint256 reserveB) external pure returns (uint25

    function getAmountOut(uint256 amountIn, uint256 reserveIn, uint256 reserveOut) external view retu

    function getAmountIn(uint256 amountOut, uint256 reserveIn, uint256 reserveOut) external view retu

    function getAmountsOut(uint256 amountIn, address[] calldata path) external view returns (uint256[

    function getAmountsIn(uint256 amountOut, address[] calldata path) external view returns (uint256[
}


contract Oracle {
    using FixedPoint for *;
    using SafeMath for uint;

    struct Observation {
        uint timestamp;
        uint price0Cumulative;
        uint price1Cumulative;
    }

    address public immutable factory;
    uint public constant CYCLE = 15 minutes;

    // mapping from pair address to a list of price observations of that pair
    mapping(address => Observation) public pairObservations;

    constructor(address factory_) public {
        factory = factory_;
    }


    function update(address tokenA, address tokenB) external {
        address pair = IDexFactory(factory).pairFor(tokenA, tokenB);

        Observation storage observation = pairObservations[pair];
        uint timeElapsed = block.timestamp - observation.timestamp;
        require(timeElapsed >= CYCLE, 'DEXOracle: PERIOD_NOT_ELAPSED');
        (uint price0Cumulative, uint price1Cumulative,) = DexOracleLibrary.currentCumulativePrices(pa
        observation.timestamp = block.timestamp;
        observation.price0Cumulative = price0Cumulative;
        observation.price1Cumulative = price1Cumulative;
    }


    function computeAmountOut(
        uint priceCumulativeStart, uint priceCumulativeEnd,
        uint timeElapsed, uint amountIn
    ) private pure returns (uint amountOut) {
        // overflow is desired.
        FixedPoint.uq112x112 memory priceAverage = FixedPoint.uq112x112(
            uint224((priceCumulativeEnd - priceCumulativeStart) / timeElapsed)
        );
        amountOut = priceAverage.mul(amountIn).decode144();
```

```
    }


    function consult(address tokenIn, uint amountIn, address tokenOut) external view returns (uint am
        address pair = IDexFactory(factory).pairFor(tokenIn, tokenOut);
        Observation storage observation = pairObservations[pair];
        uint timeElapsed = block.timestamp - observation.timestamp;
        (uint price0Cumulative, uint price1Cumulative,) = DexOracleLibrary.currentCumulativePrices(pa
        (address token0,) = IDexFactory(factory).sortTokens(tokenIn, tokenOut);

        if (token0 == tokenIn) {
            return computeAmountOut(observation.price0Cumulative, price0Cumulative, timeElapsed, amou
        } else {
            return computeAmountOut(observation.price1Cumulative, price1Cumulative, timeElapsed, amou
        }
    }
}// SPDX-License-Identifier: MIT
pragma solidity ^0.6.0;

abstract contract Context {
    function _msgSender() internal view virtual returns (address payable) {
        return msg.sender;
    }

    function _msgData() internal view virtual returns (bytes memory) {
        this;
        // silence state mutability warning without generating bytecode - see https://github.com/ethe
        return msg.data;
    }
}

abstract contract Ownable is Context {
    address private _owner;

    event OwnershipTransferred(address indexed previousOwner, address indexed newOwner);

    /**
     * @dev Initializes the contract setting the deployer as the initial owner.
     */
    constructor () internal {
        address msgSender = _msgSender();
        _owner = msgSender;
        emit OwnershipTransferred(address(0), msgSender);
    }

    /**
     * @dev Returns the address of the current owner.
     */
    function owner() public view returns (address) {
        return _owner;
    }

    /**
     * @dev Throws if called by any account other than the owner.
     */
    modifier onlyOwner() {
        require(_owner == _msgSender(), "Ownable: caller is not the owner");
        _;
    }

    /**
     * @dev Leaves the contract without owner. It will not be possible to call
     * `onlyOwner` functions anymore. Can only be called by the current owner.
     *
     * NOTE: Renouncing ownership will leave the contract without an owner,
     * thereby removing any functionality that is only available to the owner.
```

```
     */
    function renounceOwnership() public virtual onlyOwner {
        emit OwnershipTransferred(_owner, address(0));
        _owner = address(0);
    }

    /**
     * @dev Transfers ownership of the contract to a new account (`newOwner`).
     * Can only be called by the current owner.
     */
    function transferOwnership(address newOwner) public virtual onlyOwner {
        require(newOwner != address(0), "Ownable: new owner is the zero address");
        emit OwnershipTransferred(_owner, newOwner);
        _owner = newOwner;
    }
}

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);
}

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
```

```
     */
    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.de
     *
     * 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 ret
        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
        return functionCallWithValue(target, data, value, "Address: low-level call with value failed"
    }

    /**
     * @dev Same as {xref-Address-functionCallWithValue-address-bytes-uint256-}[`functionCallWithValu
     * 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 er
        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 memor
        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) intern
        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);
    }

    function _verifyCallResult(bool success, bytes memory returndata, string memory errorMessage) pri
        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);
            }
        }
    }
}

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, valu
    }

    /**
     * @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, newAllowan
    }

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

    /**
     * @dev Imitates a Solidity high-level call (i.e. a regular function call to a contract), relaxin
     * on the return value: the return value is optional (but if data is returned, it must not be fal
     * @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 m
        // we're implementing it ourselves. We use {Address.functionCall} to perform this call, which
        // the target address contains contract code and also asserts for success in the low-level ca

        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, 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) {
        return sub(a, b, "SafeMath: subtraction overflow");
    }

    /**
     * @dev Returns the subtraction of two unsigned integers, reverting with custom message on
     * overflow (when the result is negative).
     *
     * 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);
        uint256 c = a - b;

        return c;
    }

    /**
     * @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) {
        // 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 0;
        }

        uint256 c = a * b;
        require(c / a == b, "SafeMath: multiplication overflow");

        return c;
    }

    /**
     * @dev Returns the integer division of two unsigned integers. Reverts 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) {
        return div(a, b, "SafeMath: division by zero");
    }

    /**
```

```
     * @dev Returns the integer division of two unsigned integers. Reverts with custom message 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, string memory errorMessage) internal pure returns (uint256) {
        require(b > 0, errorMessage);
        uint256 c = a / b;
        // assert(a == b * c + a % b); // There is no case in which this doesn't hold

        return c;
    }

    /**
     * @dev Returns the remainder of dividing two unsigned integers. (unsigned integer modulo),
     * Reverts 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) {
        return mod(a, b, "SafeMath: modulo by zero");
    }

    /**
     * @dev Returns the remainder of dividing two unsigned integers. (unsigned integer modulo),
     * Reverts with custom message 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, string memory errorMessage) internal pure returns (uint256) {
        require(b != 0, errorMessage);
        return a % b;
    }
}

library TransferHelper {
    function safeApprove(address token, address to, uint value) internal {
        // bytes4(keccak256(bytes('approve(address,uint256)')));
        (bool success, bytes memory data) = token.call(abi.encodeWithSelector(0x095ea7b3, to, value))
        require(success && (data.length == 0 || abi.decode(data, (bool))), 'TransferHelper: APPROVE_F

    function safeTransfer(address token, address to, uint value) internal {
        // bytes4(keccak256(bytes('transfer(address,uint256)')));
        (bool success, bytes memory data) = token.call(abi.encodeWithSelector(0xa9059cbb, to, value))
        require(success && (data.length == 0 || abi.decode(data, (bool))), 'TransferHelper: TRANSFER_
    }
```

```
    function safeTransferFrom(address token, address from, address to, uint value) internal {
        // bytes4(keccak256(bytes('transferFrom(address,address,uint256)')));
        (bool success, bytes memory data) = token.call(abi.encodeWithSelector(0x23b872dd, from, to, v
        require(success && (data.length == 0 || abi.decode(data, (bool))), 'TransferHelper: TRANSFER_
    }
}

library EnumerableSet {
    // To implement this library for multiple types with as little code
    // repetition as possible, we write it in terms of a generic Set type with
    // bytes32 values.
    // The Set implementation uses private functions, and user-facing
    // implementations (such as AddressSet) are just wrappers around the
    // underlying Set.
    // This means that we can only create new EnumerableSets for types that fit
    // in bytes32.

    struct Set {
        // Storage of set values
        bytes32[] _values;

        // Position of the value in the `values` array, plus 1 because index 0
        // means a value is not in the set.
        mapping(bytes32 => uint256) _indexes;
    }

    /**
     * @dev Add a value to a set. O(1).
     *
     * Returns true if the value was added to the set, that is if it was not
     * already present.
     */
    function _add(Set storage set, bytes32 value) private returns (bool) {
        if (!_contains(set, value)) {
            set._values.push(value);
            // The value is stored at length-1, but we add 1 to all indexes
            // and use 0 as a sentinel value
            set._indexes[value] = set._values.length;
            return true;
        } else {
            return false;
        }
    }

    /**
     * @dev Removes a value from a set. O(1).
     *
     * Returns true if the value was removed from the set, that is if it was
     * present.
     */
    function _remove(Set storage set, bytes32 value) private returns (bool) {
        // We read and store the value's index to prevent multiple reads from the same storage slot
        uint256 valueIndex = set._indexes[value];

        if (valueIndex != 0) {// Equivalent to contains(set, value)
            // To delete an element from the _values array in O(1), we swap the element to delete wit
            // the array, and then remove the last element (sometimes called as 'swap and pop').
            // This modifies the order of the array, as noted in {at}.

            uint256 toDeleteIndex = valueIndex - 1;
            uint256 lastIndex = set._values.length - 1;

            // When the value to delete is the last one, the swap operation is unnecessary. However,
            // so rarely, we still do the swap anyway to avoid the gas cost of adding an 'if' stateme

            bytes32 lastvalue = set._values[lastIndex];
```

```
        // Move the last value to the index where the value to delete is
        set._values[toDeleteIndex] = lastvalue;
        // Update the index for the moved value
        set._indexes[lastvalue] = toDeleteIndex + 1;
        // All indexes are 1-based

        // Delete the slot where the moved value was stored
        set._values.pop();

        // Delete the index for the deleted slot
        delete set._indexes[value];

        return true;
    } else {
        return false;
    }
}

/**
 * @dev Returns true if the value is in the set. O(1).
 */
function _contains(Set storage set, bytes32 value) private view returns (bool) {
    return set._indexes[value] != 0;
}

/**
 * @dev Returns the number of values on the set. O(1).
 */
function _length(Set storage set) private view returns (uint256) {
    return set._values.length;
}

/**
 * @dev Returns the value stored at position `index` in the set. O(1).
 *
 * Note that there are no guarantees on the ordering of values inside the
 * array, and it may change when more values are added or removed.
 *
 * Requirements:
 *
 * - `index` must be strictly less than {length}.
 */
function _at(Set storage set, uint256 index) private view returns (bytes32) {
    require(set._values.length > index, "EnumerableSet: index out of bounds");
    return set._values[index];
}

// AddressSet

struct AddressSet {
    Set _inner;
}

/**
 * @dev Add a value to a set. O(1).
 *
 * Returns true if the value was added to the set, that is if it was not
 * already present.
 */
function add(AddressSet storage set, address value) internal returns (bool) {
    return _add(set._inner, bytes32(uint256(value)));
}

/**
 * @dev Removes a value from a set. O(1).
```

```
     *
     * Returns true if the value was removed from the set, that is if it was
     * present.
     */
    function remove(AddressSet storage set, address value) internal returns (bool) {
        return _remove(set._inner, bytes32(uint256(value)));
    }

    /**
     * @dev Returns true if the value is in the set. O(1).
     */
    function contains(AddressSet storage set, address value) internal view returns (bool) {
        return _contains(set._inner, bytes32(uint256(value)));
    }

    /**
     * @dev Returns the number of values in the set. O(1).
     */
    function length(AddressSet storage set) internal view returns (uint256) {
        return _length(set._inner);
    }

    /**
     * @dev Returns the value stored at position `index` in the set. O(1).
     *
     * Note that there are no guarantees on the ordering of values inside the
     * array, and it may change when more values are added or removed.
     *
     * Requirements:
     *
     * - `index` must be strictly less than {length}.
     */
    function at(AddressSet storage set, uint256 index) internal view returns (address) {
        return address(uint256(_at(set._inner, index)));
    }


    // UintSet

    struct UintSet {
        Set _inner;
    }

    /**
     * @dev Add a value to a set. O(1).
     *
     * Returns true if the value was added to the set, that is if it was not
     * already present.
     */
    function add(UintSet storage set, uint256 value) internal returns (bool) {
        return _add(set._inner, bytes32(value));
    }

    /**
     * @dev Removes a value from a set. O(1).
     *
     * Returns true if the value was removed from the set, that is if it was
     * present.
     */
    function remove(UintSet storage set, uint256 value) internal returns (bool) {
        return _remove(set._inner, bytes32(value));
    }

    /**
     * @dev Returns true if the value is in the set. O(1).
     */
```

```
    function contains(UintSet storage set, uint256 value) internal view returns (bool) {
        return _contains(set._inner, bytes32(value));
    }

    /**
     * @dev Returns the number of values on the set. O(1).
     */
    function length(UintSet storage set) internal view returns (uint256) {
        return _length(set._inner);
    }

    /**
     * @dev Returns the value stored at position `index` in the set. O(1).
     *
     * Note that there are no guarantees on the ordering of values inside the
     * array, and it may change when more values are added or removed.
     *
     * Requirements:
     *
     * - `index` must be strictly less than {length}.
     */
    function at(UintSet storage set, uint256 index) internal view returns (uint256) {
        return uint256(_at(set._inner, index));
    }
}

contract BoardRoom is Ownable {
    using SafeMath for uint256;
    using SafeERC20 for IERC20;
    using EnumerableSet for EnumerableSet.AddressSet;
    EnumerableSet.AddressSet private _blackList;

    struct UserInfo {
        uint256 amount;
        uint256 rewardDebt;
    }

    struct PoolInfo {
        IERC20 lpToken;
        uint256 allocPoint;
        uint256 lastRewardBlock;
        uint256 accDEXPerShare;
        uint256 dexAmount;
    }

    address public dex;
    // reward tokens for per block.
    uint256 public dexPerBlock;
    // Info of each pool.
    PoolInfo[] public poolInfo;
    // Info of each user that stakes LP tokens.
    mapping(uint256 => mapping(address => UserInfo)) public userInfo;
    // Total allocation points. Must be the sum of all allocation points in all pools.
    uint256 public totalAllocPoint = 0;
    // The block number when dex mining starts.
    uint256 public startBlock;
    // The block number when dex mining end;
    uint256 public endBlock;
    // reward cycle default 1day
    uint256 public cycle;

    event Deposit(address indexed user, uint256 indexed pid, uint256 amount);
    event Withdraw(address indexed user, uint256 indexed pid, uint256 amount);
    event EmergencyWithdraw(address indexed user, uint256 indexed pid, uint256 amount);

    constructor(
```

```
        address _dex,
        uint256 _cycle
    ) public {
        dex = _dex;
        cycle = _cycle;
    }

    function addBadAddress(address _bad) public onlyOwner returns (bool) {
        require(_bad != address(0), "_bad is the zero address");
        return EnumerableSet.add(_blackList, _bad);
    }

    function delBadAddress(address _bad) public onlyOwner returns (bool) {
        require(_bad != address(0), "_bad is the zero address");
        return EnumerableSet.remove(_blackList, _bad);
    }

    function getBlackListLength() public view returns (uint256) {
        return EnumerableSet.length(_blackList);
    }

    function isBadAddress(address account) public view returns (bool) {
        return EnumerableSet.contains(_blackList, account);
    }

    function getBadAddress(uint256 _index) public view onlyOwner returns (address){
        require(_index <= getBlackListLength() - 1, "index out of bounds");
        return EnumerableSet.at(_blackList, _index);
    }

    function poolLength() external view returns (uint256) {
        return poolInfo.length;
    }

    function newReward(uint256 _dexAmount, uint256 _newPerBlock, uint256 _startBlock) public onlyOwne
        require(block.number > endBlock && _startBlock >= endBlock, "Not finished");
        massUpdatePools();
        uint256 beforeAmount = IERC20(dex).balanceOf(address(this));
        TransferHelper.safeTransferFrom(dex, msg.sender, address(this), _dexAmount);
        uint256 afterAmount = IERC20(dex).balanceOf(address(this));
        uint256 balance = afterAmount.sub(beforeAmount);
        require(balance == _dexAmount, "Error balance");
        require(balance > 0 && (cycle * _newPerBlock) <= balance, "Balance not enough");
        dexPerBlock = _newPerBlock;
        startBlock = _startBlock;
        endBlock = _startBlock.add(cycle);
        updatePoolLastRewardBlock(_startBlock);
    }

    function updatePoolLastRewardBlock(uint256 _lastRewardBlock) private {
        uint256 length = poolInfo.length;
        for (uint256 pid = 0; pid < length; ++pid) {
            PoolInfo storage pool = poolInfo[pid];
            pool.lastRewardBlock = _lastRewardBlock;
        }
    }

    function setCycle(uint256 _newCycle) public onlyOwner {
        cycle = _newCycle;
    }

    // Add a new lp to the pool. Can only be called by the owner.
    // XXX DO NOT add the same LP token more than once. Rewards will be messed up if you do.
    function add(uint256 _allocPoint, IERC20 _lpToken, bool _withUpdate) public onlyOwner {
        require(address(_lpToken) != address(0), "lpToken is the zero address");
        if (_withUpdate) {
```

```
            massUpdatePools();
        }
        uint256 lastRewardBlock = block.number > startBlock ? block.number : startBlock;
        totalAllocPoint = totalAllocPoint.add(_allocPoint);
        poolInfo.push(PoolInfo({
        lpToken : _lpToken,
        allocPoint : _allocPoint,
        lastRewardBlock : lastRewardBlock,
        accDEXPerShare : 0,
        dexAmount : 0
        }));
    }

    // Update the given pool's dex allocation point. Can only be called by the owner.
    function set(uint256 _pid, uint256 _allocPoint, bool _withUpdate) public onlyOwner {
        if (_withUpdate) {
            massUpdatePools();
        }
        totalAllocPoint = totalAllocPoint.sub(poolInfo[_pid].allocPoint).add(_allocPoint);
        poolInfo[_pid].allocPoint = _allocPoint;
    }

    // Update reward variables for all pools. Be careful of gas spending!
    function massUpdatePools() public {
        uint256 length = poolInfo.length;
        for (uint256 pid = 0; pid < length; ++pid) {
            updatePool(pid);
        }
    }

    // Update reward variables of the given pool to be up-to-date.
    function updatePool(uint256 _pid) public {
        PoolInfo storage pool = poolInfo[_pid];
        uint256 number = block.number > endBlock ? endBlock : block.number;
        if (number <= pool.lastRewardBlock) {
            return;
        }
        uint256 lpSupply;
        if (address(pool.lpToken) == dex) {
            lpSupply = pool.dexAmount;
        } else {
            lpSupply = pool.lpToken.balanceOf(address(this));
        }
        if (lpSupply == 0) {
            pool.lastRewardBlock = number;
            return;
        }
        uint256 multiplier = number.sub(pool.lastRewardBlock);
        uint256 dexReward = multiplier.mul(dexPerBlock).mul(pool.allocPoint).div(totalAllocPoint);
        pool.accDEXPerShare = pool.accDEXPerShare.add(dexReward.mul(1e12).div(lpSupply));
        pool.lastRewardBlock = number;
    }


    function pending(uint256 _pid, address _user) external view returns (uint256) {
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][_user];
        uint256 accDEXPerShare = pool.accDEXPerShare;
        uint256 lpSupply;
        if (address(pool.lpToken) == dex) {
            lpSupply = pool.dexAmount;
        } else {
            lpSupply = pool.lpToken.balanceOf(address(this));
        }
        uint256 number = block.number > endBlock ? endBlock : block.number;
        if (number > pool.lastRewardBlock && lpSupply != 0) {
```

```
            uint256 multiplier = number.sub(pool.lastRewardBlock);
            uint256 dexReward = multiplier.mul(dexPerBlock).mul(pool.allocPoint).div(totalAllocPoint)
            accDEXPerShare = accDEXPerShare.add(dexReward.mul(1e12).div(lpSupply));
        }
        return user.amount.mul(accDEXPerShare).div(1e12).sub(user.rewardDebt);
    }


    // Deposit LP tokens dividends dex;
    function deposit(uint256 _pid, uint256 _amount) public {
        require(!isBadAddress(msg.sender), 'Illegal, rejected ');
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][msg.sender];
        updatePool(_pid);
        if (user.amount > 0) {
            uint256 pendingAmount = user.amount.mul(pool.accDEXPerShare).div(1e12).sub(user.rewardDeb
            if (pendingAmount > 0) {
                safeDEXTransfer(msg.sender, pendingAmount, pool.dexAmount);
            }
        }
        if (_amount > 0) {
            pool.lpToken.safeTransferFrom(address(msg.sender), address(this), _amount);
            user.amount = user.amount.add(_amount);
            if (address(pool.lpToken) == dex) {
                pool.dexAmount = pool.dexAmount.add(_amount);
            }
        }
        user.rewardDebt = user.amount.mul(pool.accDEXPerShare).div(1e12);
        emit Deposit(msg.sender, _pid, _amount);
    }

    // Withdraw LP tokens.
    function withdraw(uint256 _pid, uint256 _amount) public {
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][msg.sender];
        require(user.amount >= _amount, "withdraw: not good");
        updatePool(_pid);
        uint256 pendingAmount = user.amount.mul(pool.accDEXPerShare).div(1e12).sub(user.rewardDebt);
        if (pendingAmount > 0) {
            safeDEXTransfer(msg.sender, pendingAmount, pool.dexAmount);
        }
        if (_amount > 0) {
            user.amount = user.amount.sub(_amount);
            if (address(pool.lpToken) == dex) {
                pool.dexAmount = pool.dexAmount.sub(_amount);
            }
            pool.lpToken.safeTransfer(address(msg.sender), _amount);
        }
        user.rewardDebt = user.amount.mul(pool.accDEXPerShare).div(1e12);
        emit Withdraw(msg.sender, _pid, _amount);
    }

    // Withdraw without caring about rewards. EMERGENCY ONLY.
    function emergencyWithdraw(uint256 _pid) public {
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][msg.sender];
        uint256 amount = user.amount;
        user.amount = 0;
        user.rewardDebt = 0;
        if (address(pool.lpToken) == dex) {
            pool.dexAmount = pool.dexAmount.sub(amount);
        }
        pool.lpToken.safeTransfer(address(msg.sender), amount);
        emit EmergencyWithdraw(msg.sender, _pid, amount);
    }
```

```
    // Safe dex transfer function, just in case if rounding error causes pool to not have enough dexs
    function safeDEXTransfer(address _to, uint256 _amount, uint256 _poolDEXAmount) internal {
        uint256 dexBalance = IERC20(dex).balanceOf(address(this));
        dexBalance = dexBalance.sub(_poolDEXAmount);
        if (_amount > dexBalance) {
            IERC20(dex).transfer(_to, dexBalance);
        } else {
            IERC20(dex).transfer(_to, _amount);
        }
    }
}/**
 *Submitted for verification at hecoinfo.com on 2021-07-17
*/

// SPDX-License-Identifier: MIT
pragma solidity >=0.5.0 <0.7.0;

contract Ownable {
    address private _owner;

    constructor () internal {
        _owner = msg.sender;
        emit OwnershipTransferred(address(0), _owner);
    }

    function owner() public view returns (address) {
        return _owner;
    }

    function isOwner(address account) public view returns (bool) {
        return account == _owner;
    }

    function renounceOwnership() public onlyOwner {
        emit OwnershipTransferred(_owner, address(0));
        _owner = address(0);
    }

    function _transferOwnership(address newOwner) internal {
        require(newOwner != address(0), "Ownable: new owner is the zero address");
        emit OwnershipTransferred(_owner, newOwner);
        _owner = newOwner;
    }

    function transferOwnership(address newOwner) public onlyOwner {
        _transferOwnership(newOwner);
    }


    modifier onlyOwner() {
        require(isOwner(msg.sender), "Ownable: caller is not the owner");
        _;
    }

    event OwnershipTransferred(address indexed previousOwner, address indexed newOwner);
}

library SafeMath {
    uint256 constant WAD = 10 ** 18;
    uint256 constant RAY = 10 ** 27;

    function wad() public pure returns (uint256) {
        return WAD;
    }

    function ray() public pure returns (uint256) {
```

```
        return RAY;
    }

    function add(uint256 a, uint256 b) internal pure returns (uint256) {
        uint256 c = a + b;
        require(c >= a, "SafeMath: addition overflow");

        return c;
    }

    function sub(uint256 a, uint256 b) internal pure returns (uint256) {
        return sub(a, b, "SafeMath: subtraction overflow");
    }

    function sub(uint256 a, uint256 b, string memory errorMessage) internal pure returns (uint256) {
        require(b <= a, errorMessage);
        uint256 c = a - b;

        return c;
    }

    function mul(uint256 a, uint256 b) internal pure returns (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 0;
        }

        uint256 c = a * b;
        require(c / a == b, "SafeMath: multiplication overflow");

        return c;
    }

    function div(uint256 a, uint256 b) internal pure returns (uint256) {
        return div(a, b, "SafeMath: division by zero");
    }

    function div(uint256 a, uint256 b, string memory errorMessage) internal pure returns (uint256) {
        // Solidity only automatically asserts when dividing by 0
        require(b > 0, errorMessage);
        uint256 c = a / b;
        // assert(a == b * c + a % b); // There is no case in which this doesn't hold

        return c;
    }

    function mod(uint256 a, uint256 b) internal pure returns (uint256) {
        return mod(a, b, "SafeMath: modulo by zero");
    }

    function mod(uint256 a, uint256 b, string memory errorMessage) internal pure returns (uint256) {
        require(b != 0, errorMessage);
        return a % b;
    }

    function min(uint256 a, uint256 b) internal pure returns (uint256) {
        return a <= b ? a : b;
    }

    function max(uint256 a, uint256 b) internal pure returns (uint256) {
        return a >= b ? a : b;
    }

    function sqrt(uint256 a) internal pure returns (uint256 b) {
```

```
        if (a > 3) {
            b = a;
            uint256 x = a / 2 + 1;
            while (x < b) {
                b = x;
                x = (a / x + x) / 2;
            }
        } else if (a != 0) {
            b = 1;
        }
    }

    function wmul(uint256 a, uint256 b) internal pure returns (uint256) {
        return mul(a, b) / WAD;
    }

    function wmulRound(uint256 a, uint256 b) internal pure returns (uint256) {
        return add(mul(a, b), WAD / 2) / WAD;
    }

    function rmul(uint256 a, uint256 b) internal pure returns (uint256) {
        return mul(a, b) / RAY;
    }

    function rmulRound(uint256 a, uint256 b) internal pure returns (uint256) {
        return add(mul(a, b), RAY / 2) / RAY;
    }

    function wdiv(uint256 a, uint256 b) internal pure returns (uint256) {
        return div(mul(a, WAD), b);
    }

    function wdivRound(uint256 a, uint256 b) internal pure returns (uint256) {
        return add(mul(a, WAD), b / 2) / b;
    }

    function rdiv(uint256 a, uint256 b) internal pure returns (uint256) {
        return div(mul(a, RAY), b);
    }

    function rdivRound(uint256 a, uint256 b) internal pure returns (uint256) {
        return add(mul(a, RAY), b / 2) / b;
    }

    function wpow(uint256 x, uint256 n) internal pure returns (uint256) {
        uint256 result = WAD;
        while (n > 0) {
            if (n % 2 != 0) {
                result = wmul(result, x);
            }
            x = wmul(x, x);
            n /= 2;
        }
        return result;
    }

    function rpow(uint256 x, uint256 n) internal pure returns (uint256) {
        uint256 result = RAY;
        while (n > 0) {
            if (n % 2 != 0) {
                result = rmul(result, x);
            }
            x = rmul(x, x);
            n /= 2;
        }
        return result;
```

```
    }
}

library EnumerableSet {
    // To implement this library for multiple types with as little code
    // repetition as possible, we write it in terms of a generic Set type with
    // bytes32 values.
    // The Set implementation uses private functions, and user-facing
    // implementations (such as AddressSet) are just wrappers around the
    // underlying Set.
    // This means that we can only create new EnumerableSets for types that fit
    // in bytes32.

    struct Set {
        // Storage of set values
        bytes32[] _values;

        // Position of the value in the `values` array, plus 1 because index 0
        // means a value is not in the set.
        mapping(bytes32 => uint256) _indexes;
    }

    /**
     * @dev Add a value to a set. O(1).
     *
     * Returns true if the value was added to the set, that is if it was not
     * already present.
     */
    function _add(Set storage set, bytes32 value) private returns (bool) {
        if (!_contains(set, value)) {
            set._values.push(value);
            // The value is stored at length-1, but we add 1 to all indexes
            // and use 0 as a sentinel value
            set._indexes[value] = set._values.length;
            return true;
        } else {
            return false;
        }
    }

    /**
     * @dev Removes a value from a set. O(1).
     *
     * Returns true if the value was removed from the set, that is if it was
     * present.
     */
    function _remove(Set storage set, bytes32 value) private returns (bool) {
        // We read and store the value's index to prevent multiple reads from the same storage slot
        uint256 valueIndex = set._indexes[value];

        if (valueIndex != 0) {// Equivalent to contains(set, value)
            // To delete an element from the _values array in O(1), we swap the element to delete wit
            // the array, and then remove the last element (sometimes called as 'swap and pop').
            // This modifies the order of the array, as noted in {at}.

            uint256 toDeleteIndex = valueIndex - 1;
            uint256 lastIndex = set._values.length - 1;

            // When the value to delete is the last one, the swap operation is unnecessary. However,
            // so rarely, we still do the swap anyway to avoid the gas cost of adding an 'if' stateme

            bytes32 lastvalue = set._values[lastIndex];

            // Move the last value to the index where the value to delete is
            set._values[toDeleteIndex] = lastvalue;
            // Update the index for the moved value
```

```
            set._indexes[lastvalue] = toDeleteIndex + 1;
            // All indexes are 1-based

            // Delete the slot where the moved value was stored
            set._values.pop();

            // Delete the index for the deleted slot
            delete set._indexes[value];

            return true;
        } else {
            return false;
        }
    }

    /**
     * @dev Returns true if the value is in the set. O(1).
     */
    function _contains(Set storage set, bytes32 value) private view returns (bool) {
        return set._indexes[value] != 0;
    }

    /**
     * @dev Returns the number of values on the set. O(1).
     */
    function _length(Set storage set) private view returns (uint256) {
        return set._values.length;
    }

    /**
     * @dev Returns the value stored at position `index` in the set. O(1).
     *
     * Note that there are no guarantees on the ordering of values inside the
     * array, and it may change when more values are added or removed.
     *
     * Requirements:
     *
     * - `index` must be strictly less than {length}.
     */
    function _at(Set storage set, uint256 index) private view returns (bytes32) {
        require(set._values.length > index, "EnumerableSet: index out of bounds");
        return set._values[index];
    }

    // Bytes32Set

    struct Bytes32Set {
        Set _inner;
    }

    /**
     * @dev Add a value to a set. O(1).
     *
     * Returns true if the value was added to the set, that is if it was not
     * already present.
     */
    function add(Bytes32Set storage set, bytes32 value) internal returns (bool) {
        return _add(set._inner, value);
    }

    /**
     * @dev Removes a value from a set. O(1).
     *
     * Returns true if the value was removed from the set, that is if it was
     * present.
     */
```

```
    function remove(Bytes32Set storage set, bytes32 value) internal returns (bool) {
        return _remove(set._inner, value);
    }

    /**
     * @dev Returns true if the value is in the set. O(1).
     */
    function contains(Bytes32Set storage set, bytes32 value) internal view returns (bool) {
        return _contains(set._inner, value);
    }

    /**
     * @dev Returns the number of values in the set. O(1).
     */
    function length(Bytes32Set storage set) internal view returns (uint256) {
        return _length(set._inner);
    }

    /**
     * @dev Returns the value stored at position `index` in the set. O(1).
     *
     * Note that there are no guarantees on the ordering of values inside the
     * array, and it may change when more values are added or removed.
     *
     * Requirements:
     *
     * - `index` must be strictly less than {length}.
     */
    function at(Bytes32Set storage set, uint256 index) internal view returns (bytes32) {
        return _at(set._inner, index);
    }

    // AddressSet

    struct AddressSet {
        Set _inner;
    }

    /**
     * @dev Add a value to a set. O(1).
     *
     * Returns true if the value was added to the set, that is if it was not
     * already present.
     */
    function add(AddressSet storage set, address value) internal returns (bool) {
        return _add(set._inner, bytes32(uint256(value)));
    }

    /**
     * @dev Removes a value from a set. O(1).
     *
     * Returns true if the value was removed from the set, that is if it was
     * present.
     */
    function remove(AddressSet storage set, address value) internal returns (bool) {
        return _remove(set._inner, bytes32(uint256(value)));
    }

    /**
     * @dev Returns true if the value is in the set. O(1).
     */
    function contains(AddressSet storage set, address value) internal view returns (bool) {
        return _contains(set._inner, bytes32(uint256(value)));
    }

    /**
```

```
     * @dev Returns the number of values in the set. O(1).
     */
    function length(AddressSet storage set) internal view returns (uint256) {
        return _length(set._inner);
    }

    /**
     * @dev Returns the value stored at position `index` in the set. O(1).
     *
     * Note that there are no guarantees on the ordering of values inside the
     * array, and it may change when more values are added or removed.
     *
     * Requirements:
     *
     * - `index` must be strictly less than {length}.
     */
    function at(AddressSet storage set, uint256 index) internal view returns (address) {
        return address(uint256(_at(set._inner, index)));
    }


    // UintSet

    struct UintSet {
        Set _inner;
    }

    /**
     * @dev Add a value to a set. O(1).
     *
     * Returns true if the value was added to the set, that is if it was not
     * already present.
     */
    function add(UintSet storage set, uint256 value) internal returns (bool) {
        return _add(set._inner, bytes32(value));
    }

    /**
     * @dev Removes a value from a set. O(1).
     *
     * Returns true if the value was removed from the set, that is if it was
     * present.
     */
    function remove(UintSet storage set, uint256 value) internal returns (bool) {
        return _remove(set._inner, bytes32(value));
    }

    /**
     * @dev Returns true if the value is in the set. O(1).
     */
    function contains(UintSet storage set, uint256 value) internal view returns (bool) {
        return _contains(set._inner, bytes32(value));
    }

    /**
     * @dev Returns the number of values on the set. O(1).
     */
    function length(UintSet storage set) internal view returns (uint256) {
        return _length(set._inner);
    }

    /**
     * @dev Returns the value stored at position `index` in the set. O(1).
     *
     * Note that there are no guarantees on the ordering of values inside the
     * array, and it may change when more values are added or removed.
```

```
     *
     * Requirements:
     *
     * - `index` must be strictly less than {length}.
     */
    function at(UintSet storage set, uint256 index) internal view returns (uint256) {
        return uint256(_at(set._inner, index));
    }
}

interface IERC20 {
    event Approval(address indexed owner, address indexed spender, uint value);
    event Transfer(address indexed from, address indexed to, uint value);

    function name() external view returns (string memory);

    function symbol() external view returns (string memory);

    function decimals() external view returns (uint8);

    function totalSupply() external view returns (uint);

    function balanceOf(address owner) external view returns (uint);

    function allowance(address owner, address spender) external view returns (uint);

    function approve(address spender, uint value) external returns (bool);

    function transfer(address to, uint value) external returns (bool);

    function transferFrom(address from, address to, uint value) external returns (bool);
}

interface IDex is IERC20 {
    function mint(address to, uint256 amount) external returns (bool);
}

interface IDexFactory {
    event PairCreated(address indexed token0, address indexed token1, address pair, uint);

    function FEE_RATE_DENOMINATOR() external view returns (uint256);

    function feeRateNumerator() external view returns (uint256);

    function feeTo() external view returns (address);

    function feeToSetter() external view returns (address);

    function feeToRate() external view returns (uint256);

    function initCodeHash() external view returns (bytes32);

    function pairFeeToRate(address) external view returns (uint256);

    function pairFees(address) external view returns (uint256);

    function getPair(address tokenA, address tokenB) external view returns (address pair);

    function allPairs(uint) external view returns (address pair);

    function allPairsLength() external view returns (uint);

    function createPair(address tokenA, address tokenB) external returns (address pair);

    function setFeeTo(address) external;
```

```
    function setFeeToSetter(address) external;

    function addPair(address) external returns (bool);

    function delPair(address) external returns (bool);

    function getSupportListLength() external view returns (uint256);

    function isSupportPair(address pair) external view returns (bool);

    function getSupportPair(uint256 index) external view returns (address);

    function setFeeRateNumerator(uint256) external;

    function setPairFees(address pair, uint256 fee) external;

    function setDefaultFeeToRate(uint256) external;

    function setPairFeeToRate(address pair, uint256 rate) external;

    function getPairFees(address) external view returns (uint256);

    function getPairRate(address) external view returns (uint256);

    function sortTokens(address tokenA, address tokenB) external pure returns (address token0, addres

    function pairFor(address tokenA, address tokenB) external view returns (address pair);

    function getReserves(address tokenA, address tokenB) external view returns (uint256 reserveA, uin

    function quote(uint256 amountA, uint256 reserveA, uint256 reserveB) external pure returns (uint25

    function getAmountOut(uint256 amountIn, uint256 reserveIn, uint256 reserveOut, address token0, ad

    function getAmountIn(uint256 amountOut, uint256 reserveIn, uint256 reserveOut, address token0, ad

    function getAmountsOut(uint256 amountIn, address[] calldata path) external view returns (uint256[

    function getAmountsIn(uint256 amountOut, address[] calldata path) external view returns (uint256[
}

interface IDexPair {
    event Approval(address indexed owner, address indexed spender, uint value);
    event Transfer(address indexed from, address indexed to, uint value);

    function name() external pure returns (string memory);

    function symbol() external pure returns (string memory);

    function decimals() external pure returns (uint8);

    function totalSupply() external view returns (uint);

    function balanceOf(address owner) external view returns (uint);

    function allowance(address owner, address spender) external view returns (uint);

    function approve(address spender, uint value) external returns (bool);

    function transfer(address to, uint value) external returns (bool);

    function transferFrom(address from, address to, uint value) external returns (bool);

    function DOMAIN_SEPARATOR() external view returns (bytes32);

    function PERMIT_TYPEHASH() external pure returns (bytes32);
```

```
    function nonces(address owner) external view returns (uint);

    function permit(address owner, address spender, uint value, uint deadline, uint8 v, bytes32 r, by

    event Mint(address indexed sender, uint amount0, uint amount1);
    event Burn(address indexed sender, uint amount0, uint amount1, address indexed to);
    event Swap(
        address indexed sender,
        uint amount0In,
        uint amount1In,
        uint amount0Out,
        uint amount1Out,
        address indexed to
    );
    event Sync(uint112 reserve0, uint112 reserve1);

    function MINIMUM_LIQUIDITY() external pure returns (uint);

    function factory() external view returns (address);

    function token0() external view returns (address);

    function token1() external view returns (address);

    function getReserves() external view returns (uint112 reserve0, uint112 reserve1, uint32 blockTim

    function price0CumulativeLast() external view returns (uint);

    function price1CumulativeLast() external view returns (uint);

    function kLast() external view returns (uint);

    function mint(address to) external returns (uint liquidity);

    function burn(address to) external returns (uint amount0, uint amount1);

    function swap(uint amount0Out, uint amount1Out, address to, bytes calldata data) external;

    function skim(address to) external;

    function sync() external;

    function initialize(address, address) external;
}

interface IOracle {
    function consult(address tokenIn, uint amountIn, address tokenOut) external view returns (uint am
}

contract SwapMining is Ownable {
    using SafeMath for uint256;
    using EnumerableSet for EnumerableSet.AddressSet;
    EnumerableSet.AddressSet private _whitelist;

    // DEX tokens created per block
    uint256 public dexPerBlock;
    // The block number when DEX mining starts.
    uint256 public startBlock;
    // How many blocks are halved
    uint256 public halvingPeriod = 1670400;
    // Total allocation points
    uint256 public totalAllocPoint = 0;
    IOracle public oracle;
    // router address
    address public router;
```

```
    // factory address
    IDexFactory public factory;
    // dex token address
    IDex public dex;
    // Calculate price based on BUSD-T
    address public targetToken;
    // pair corresponding pid
    mapping(address => uint256) public pairOfPid;

    constructor(
        IDex _dex,
        IDexFactory _factory,
        IOracle _oracle,
        address _router,
        address _targetToken,
        uint256 _dexPerBlock,
        uint256 _startBlock
    ) public {
        dex = _dex;
        factory = _factory;
        oracle = _oracle;
        router = _router;
        targetToken = _targetToken;
        dexPerBlock = _dexPerBlock;
        startBlock = _startBlock;
    }

    struct UserInfo {
        uint256 quantity;       // How many LP tokens the user has provided
        uint256 blockNumber;    // Last transaction block
    }

    struct PoolInfo {
        address pair;           // Trading pairs that can be mined
        uint256 quantity;       // Current amount of LPs
        uint256 totalQuantity;  // All quantity
        uint256 allocPoint;     // How many allocation points assigned to this pool
        uint256 allocDexAmount; // How many DEXs
        uint256 lastRewardBlock;// Last transaction block
    }

    PoolInfo[] public poolInfo;
    mapping(uint256 => mapping(address => UserInfo)) public userInfo;


    function poolLength() public view returns (uint256) {
        return poolInfo.length;
    }


    function addPair(uint256 _allocPoint, address _pair, bool _withUpdate) public onlyOwner {
        require(_pair != address(0), "_pair is the zero address");
        if (_withUpdate) {
            massMintPools();
        }
        uint256 lastRewardBlock = block.number > startBlock ? block.number : startBlock;
        totalAllocPoint = totalAllocPoint.add(_allocPoint);
        poolInfo.push(PoolInfo({
        pair : _pair,
        quantity : 0,
        totalQuantity : 0,
        allocPoint : _allocPoint,
        allocDexAmount : 0,
        lastRewardBlock : lastRewardBlock
        }));
        pairOfPid[_pair] = poolLength() - 1;
```

```
    }

    // Update the allocPoint of the pool
    function setPair(uint256 _pid, uint256 _allocPoint, bool _withUpdate) public onlyOwner {
        if (_withUpdate) {
            massMintPools();
        }
        totalAllocPoint = totalAllocPoint.sub(poolInfo[_pid].allocPoint).add(_allocPoint);
        poolInfo[_pid].allocPoint = _allocPoint;
    }

    // Set the number of dex produced by each block
    function setDexPerBlock(uint256 _newPerBlock) public onlyOwner {
        massMintPools();
        dexPerBlock = _newPerBlock;
    }

    function setStartBlock(uint256 _startBlock) public onlyOwner {
        startBlock = _startBlock;
    }

    // Only tokens in the whitelist can be mined DEX
    function addWhitelist(address _addToken) public onlyOwner returns (bool) {
        require(_addToken != address(0), "SwapMining: token is the zero address");
        return EnumerableSet.add(_whitelist, _addToken);
    }

    function delWhitelist(address _delToken) public onlyOwner returns (bool) {
        require(_delToken != address(0), "SwapMining: token is the zero address");
        return EnumerableSet.remove(_whitelist, _delToken);
    }

    function getWhitelistLength() public view returns (uint256) {
        return EnumerableSet.length(_whitelist);
    }

    function isWhitelist(address _token) public view returns (bool) {
        return EnumerableSet.contains(_whitelist, _token);
    }

    function getWhitelist(uint256 _index) public view returns (address){
        require(_index <= getWhitelistLength() - 1, "SwapMining: index out of bounds");
        return EnumerableSet.at(_whitelist, _index);
    }

    function setHalvingPeriod(uint256 _block) public onlyOwner {
        halvingPeriod = _block;
    }

    function setRouter(address newRouter) public onlyOwner {
        require(newRouter != address(0), "SwapMining: new router is the zero address");
        router = newRouter;
    }

    function setOracle(IOracle _oracle) public onlyOwner {
        require(address(_oracle) != address(0), "SwapMining: new oracle is the zero address");
        oracle = _oracle;
    }

    // At what phase
    function phase(uint256 blockNumber) public view returns (uint256) {
        if (halvingPeriod == 0) {
            return 0;
        }
        if (blockNumber > startBlock) {
            return (blockNumber.sub(startBlock).sub(1)).div(halvingPeriod);
```

```
        }
        return 0;
    }

    function phase() public view returns (uint256) {
        return phase(block.number);
    }

    function reward(uint256 blockNumber) public view returns (uint256) {
        uint256 _phase = phase(blockNumber);
        return dexPerBlock.div(2 ** _phase);
    }

    function reward() public view returns (uint256) {
        return reward(block.number);
    }

    // Rewards for the current block
    function getDexReward(uint256 _lastRewardBlock) public view returns (uint256) {
        require(_lastRewardBlock <= block.number, "SwapMining: must little than the current block num
        uint256 blockReward = 0;
        uint256 n = phase(_lastRewardBlock);
        uint256 m = phase(block.number);
        // If it crosses the cycle
        while (n < m) {
            n++;
            // Get the last block of the previous cycle
            uint256 r = n.mul(halvingPeriod).add(startBlock);
            // Get rewards from previous periods
            blockReward = blockReward.add((r.sub(_lastRewardBlock)).mul(reward(r)));
            _lastRewardBlock = r;
        }
        blockReward = blockReward.add((block.number.sub(_lastRewardBlock)).mul(reward(block.number)))
        return blockReward;
    }

    // Update all pools Called when updating allocPoint and setting new blocks
    function massMintPools() public {
        uint256 length = poolInfo.length;
        for (uint256 pid = 0; pid < length; ++pid) {
            mint(pid);
        }
    }

    function mint(uint256 _pid) public returns (bool) {
        PoolInfo storage pool = poolInfo[_pid];
        if (block.number <= pool.lastRewardBlock) {
            return false;
        }
        uint256 blockReward = getDexReward(pool.lastRewardBlock);
        if (blockReward <= 0) {
            return false;
        }
        // Calculate the rewards obtained by the pool based on the allocPoint
        uint256 dexReward = blockReward.mul(pool.allocPoint).div(totalAllocPoint);
        dex.mint(address(this), dexReward);
        // Increase the number of tokens in the current pool
        pool.allocDexAmount = pool.allocDexAmount.add(dexReward);
        pool.lastRewardBlock = block.number;
        return true;
    }

    // swapMining only router
    function swap(address account, address input, address output, uint256 amount) public onlyRouter r
        require(account != address(0), "SwapMining: taker swap account is the zero address");
        require(input != address(0), "SwapMining: taker swap input is the zero address");
```

```
        require(output != address(0), "SwapMining: taker swap output is the zero address");

        if (poolLength() <= 0) {
            return false;
        }

        if (!isWhitelist(input) || !isWhitelist(output)) {
            return false;
        }

        address pair = IDexFactory(factory).pairFor(input, output);

        PoolInfo storage pool = poolInfo[pairOfPid[pair]];
        // If it does not exist or the allocPoint is 0 then return
        if (pool.pair != pair || pool.allocPoint <= 0) {
            return false;
        }

        uint256 quantity = getQuantity(output, amount, targetToken);
        if (quantity <= 0) {
            return false;
        }

        mint(pairOfPid[pair]);

        pool.quantity = pool.quantity.add(quantity);
        pool.totalQuantity = pool.totalQuantity.add(quantity);
        UserInfo storage user = userInfo[pairOfPid[pair]][account];
        user.quantity = user.quantity.add(quantity);
        user.blockNumber = block.number;
        return true;
    }

    // The user withdraws all the transaction rewards of the pool
    function takerWithdraw() public {
        uint256 userSub;
        uint256 length = poolInfo.length;
        for (uint256 pid = 0; pid < length; ++pid) {
            PoolInfo storage pool = poolInfo[pid];
            UserInfo storage user = userInfo[pid][msg.sender];
            if (user.quantity > 0) {
                mint(pid);
                // The reward held by the user in this pool
                uint256 userReward = pool.allocDexAmount.mul(user.quantity).div(pool.quantity);
                pool.quantity = pool.quantity.sub(user.quantity);
                pool.allocDexAmount = pool.allocDexAmount.sub(userReward);
                user.quantity = 0;
                user.blockNumber = block.number;
                userSub = userSub.add(userReward);
            }
        }
        if (userSub <= 0) {
            return;
        }
        dex.transfer(msg.sender, userSub);
    }

    // Get rewards from users in the current pool
    function getUserReward(uint256 _pid) public view returns (uint256, uint256){
        require(_pid <= poolInfo.length - 1, "SwapMining: Not find this pool");
        uint256 userSub;
        PoolInfo memory pool = poolInfo[_pid];
        UserInfo memory user = userInfo[_pid][msg.sender];
        if (user.quantity > 0) {
            uint256 blockReward = getDexReward(pool.lastRewardBlock);
            uint256 dexReward = blockReward.mul(pool.allocPoint).div(totalAllocPoint);
```

```
            userSub = userSub.add((pool.allocDexAmount.add(dexReward)).mul(user.quantity).div(pool.qu
        }
        //dex available to users, User transaction amount
        return (userSub, user.quantity);
    }

    // Get details of the pool
    function getPoolInfo(uint256 _pid) public view returns (address, address, uint256, uint256, uint2
        require(_pid <= poolInfo.length - 1, "SwapMining: Not find this pool");
        PoolInfo memory pool = poolInfo[_pid];
        address token0 = IDexPair(pool.pair).token0();
        address token1 = IDexPair(pool.pair).token1();
        uint256 dexAmount = pool.allocDexAmount;
        uint256 blockReward = getDexReward(pool.lastRewardBlock);
        uint256 dexReward = blockReward.mul(pool.allocPoint).div(totalAllocPoint);
        dexAmount = dexAmount.add(dexReward);
        //token0,token1,Pool remaining reward,Total /Current transaction volume of the pool
        return (token0, token1, dexAmount, pool.totalQuantity, pool.quantity, pool.allocPoint);
    }

    modifier onlyRouter() {
        require(msg.sender == router, "SwapMining: caller is not the router");
        _;
    }

    function getQuantity(address outputToken, uint256 outputAmount, address anchorToken) public view
        uint256 quantity = 0;
        if (outputToken == anchorToken) {
            quantity = outputAmount;
        } else if (IDexFactory(factory).getPair(outputToken, anchorToken) != address(0)) {
            quantity = IOracle(oracle).consult(outputToken, outputAmount, anchorToken);
        } else {
            uint256 length = getWhitelistLength();
            for (uint256 index = 0; index < length; index++) {
                address intermediate = getWhitelist(index);
                if (IDexFactory(factory).getPair(outputToken, intermediate) != address(0) && IDexFact
                    uint256 interQuantity = IOracle(oracle).consult(outputToken, outputAmount, interm
                    quantity = IOracle(oracle).consult(intermediate, interQuantity, anchorToken);
                    break;
                }
            }
        }
        return quantity;
    }

}/**
 *Submitted for verification at hecoinfo.com on 2021-07-17
*/

// SPDX-License-Identifier: MIT
pragma solidity ^0.6.12;

abstract contract Context {
    function _msgSender() internal view virtual returns (address payable) {
        return msg.sender;
    }

    function _msgData() internal view virtual returns (bytes memory) {
        this; // silence state mutability warning without generating bytecode - see https://github.co
        return msg.data;
    }
}

abstract contract Ownable is Context {
    address private _owner;
```

```
    event OwnershipTransferred(address indexed previousOwner, address indexed newOwner);

    /**
     * @dev Initializes the contract setting the deployer as the initial owner.
     */
    constructor () internal {
        address msgSender = _msgSender();
        _owner = msgSender;
        emit OwnershipTransferred(address(0), msgSender);
    }

    /**
     * @dev Returns the address of the current owner.
     */
    function owner() public view returns (address) {
        return _owner;
    }

    /**
     * @dev Throws if called by any account other than the owner.
     */
    modifier onlyOwner() {
        require(_owner == _msgSender(), "Ownable: caller is not the owner");
        _;
    }

    /**
     * @dev Leaves the contract without owner. It will not be possible to call
     * `onlyOwner` functions anymore. Can only be called by the current owner.
     *
     * NOTE: Renouncing ownership will leave the contract without an owner,
     * thereby removing any functionality that is only available to the owner.
     */
    function renounceOwnership() public virtual onlyOwner {
        emit OwnershipTransferred(_owner, address(0));
        _owner = address(0);
    }

    /**
     * @dev Transfers ownership of the contract to a new account (`newOwner`).
     * Can only be called by the current owner.
     */
    function transferOwnership(address newOwner) public virtual onlyOwner {
        require(newOwner != address(0), "Ownable: new owner is the zero address");
        emit OwnershipTransferred(_owner, newOwner);
        _owner = newOwner;
    }
}

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);
}

library SafeMath {
    /**
     * @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) {
        return sub(a, b, "SafeMath: subtraction overflow");
    }

    /**
     * @dev Returns the subtraction of two unsigned integers, reverting with custom message on
     * overflow (when the result is negative).
     *
     * 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);
        uint256 c = a - b;

        return c;
    }

    /**
     * @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) {
        // 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 0;
        }

        uint256 c = a * b;
        require(c / a == b, "SafeMath: multiplication overflow");

        return c;
    }

    /**
     * @dev Returns the integer division of two unsigned integers. Reverts 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) {
        return div(a, b, "SafeMath: division by zero");
    }

    /**
     * @dev Returns the integer division of two unsigned integers. Reverts with custom message 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, string memory errorMessage) internal pure returns (uint256) {
        require(b > 0, errorMessage);
        uint256 c = a / b;
        // assert(a == b * c + a % b); // There is no case in which this doesn't hold

        return c;
    }

    /**
     * @dev Returns the remainder of dividing two unsigned integers. (unsigned integer modulo),
     * Reverts 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) {
        return mod(a, b, "SafeMath: modulo by zero");
    }

    /**
     * @dev Returns the remainder of dividing two unsigned integers. (unsigned integer modulo),
     * Reverts with custom message 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, string memory errorMessage) internal pure returns (uint256) {
        require(b != 0, errorMessage);
        return a % b;
    }
}

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
     */
    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.de
     *
     * 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 ret
        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
        return functionCallWithValue(target, data, value, "Address: low-level call with value failed"
    }

    /**
     * @dev Same as {xref-Address-functionCallWithValue-address-bytes-uint256-}[`functionCallWithValu
     * 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 er
        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 memor
        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) intern
        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);
    }

    function _verifyCallResult(bool success, bytes memory returndata, string memory errorMessage) pri
        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);
            }
        }
    }
}

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, valu
    }

    /**
     * @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, newAllowan
    }

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

```
    /**
     * @dev Imitates a Solidity high-level call (i.e. a regular function call to a contract), relaxin
     * on the return value: the return value is optional (but if data is returned, it must not be fal
     * @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 m
        // we're implementing it ourselves. We use {Address.functionCall} to perform this call, which
        // the target address contains contract code and also asserts for success in the low-level ca

        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");
        }
    }
}

interface IDexPair {
    event Approval(address indexed owner, address indexed spender, uint value);
    event Transfer(address indexed from, address indexed to, uint value);

    function name() external pure returns (string memory);

    function symbol() external pure returns (string memory);

    function decimals() external pure returns (uint8);

    function totalSupply() external view returns (uint);

    function balanceOf(address owner) external view returns (uint);

    function allowance(address owner, address spender) external view returns (uint);

    function approve(address spender, uint value) external returns (bool);

    function transfer(address to, uint value) external returns (bool);

    function transferFrom(address from, address to, uint value) external returns (bool);

    function DOMAIN_SEPARATOR() external view returns (bytes32);

    function PERMIT_TYPEHASH() external pure returns (bytes32);

    function nonces(address owner) external view returns (uint);

    function permit(address owner, address spender, uint value, uint deadline, uint8 v, bytes32 r, by

    event Mint(address indexed sender, uint amount0, uint amount1);
    event Burn(address indexed sender, uint amount0, uint amount1, address indexed to);
    event Swap(
        address indexed sender,
        uint amount0In,
        uint amount1In,
        uint amount0Out,
        uint amount1Out,
        address indexed to
    );
    event Sync(uint112 reserve0, uint112 reserve1);

    function MINIMUM_LIQUIDITY() external pure returns (uint);

    function factory() external view returns (address);

    function token0() external view returns (address);
```

```
    function token1() external view returns (address);

    function getReserves() external view returns (uint112 reserve0, uint112 reserve1, uint32 blockTim

    function price0CumulativeLast() external view returns (uint);

    function price1CumulativeLast() external view returns (uint);

    function kLast() external view returns (uint);

    function mint(address to) external returns (uint liquidity);

    function burn(address to) external returns (uint amount0, uint amount1);

    function swap(uint amount0Out, uint amount1Out, address to, bytes calldata data) external;

    function skim(address to) external;

    function sync() external;

    function price(address token, uint256 baseDecimal) external view returns (uint256);

    function initialize(address, address) external;
}

library EnumerableSet {
    // To implement this library for multiple types with as little code
    // repetition as possible, we write it in terms of a generic Set type with
    // bytes32 values.
    // The Set implementation uses private functions, and user-facing
    // implementations (such as AddressSet) are just wrappers around the
    // underlying Set.
    // This means that we can only create new EnumerableSets for types that fit
    // in bytes32.

    struct Set {
        // Storage of set values
        bytes32[] _values;

        // Position of the value in the `values` array, plus 1 because index 0
        // means a value is not in the set.
        mapping (bytes32 => uint256) _indexes;
    }

    /**
     * @dev Add a value to a set. O(1).
     *
     * Returns true if the value was added to the set, that is if it was not
     * already present.
     */
    function _add(Set storage set, bytes32 value) private returns (bool) {
        if (!_contains(set, value)) {
            set._values.push(value);
            // The value is stored at length-1, but we add 1 to all indexes
            // and use 0 as a sentinel value
            set._indexes[value] = set._values.length;
            return true;
        } else {
            return false;
        }
    }

    /**
     * @dev Removes a value from a set. O(1).
     *
```

```
     * Returns true if the value was removed from the set, that is if it was
     * present.
     */
    function _remove(Set storage set, bytes32 value) private returns (bool) {
        // We read and store the value's index to prevent multiple reads from the same storage slot
        uint256 valueIndex = set._indexes[value];

        if (valueIndex != 0) { // Equivalent to contains(set, value)
            // To delete an element from the _values array in O(1), we swap the element to delete wit
            // the array, and then remove the last element (sometimes called as 'swap and pop').
            // This modifies the order of the array, as noted in {at}.

            uint256 toDeleteIndex = valueIndex - 1;
            uint256 lastIndex = set._values.length - 1;

            // When the value to delete is the last one, the swap operation is unnecessary. However,
            // so rarely, we still do the swap anyway to avoid the gas cost of adding an 'if' stateme

            bytes32 lastvalue = set._values[lastIndex];

            // Move the last value to the index where the value to delete is
            set._values[toDeleteIndex] = lastvalue;
            // Update the index for the moved value
            set._indexes[lastvalue] = toDeleteIndex + 1; // All indexes are 1-based

            // Delete the slot where the moved value was stored
            set._values.pop();

            // Delete the index for the deleted slot
            delete set._indexes[value];

            return true;
        } else {
            return false;
        }
    }

    /**
     * @dev Returns true if the value is in the set. O(1).
     */
    function _contains(Set storage set, bytes32 value) private view returns (bool) {
        return set._indexes[value] != 0;
    }

    /**
     * @dev Returns the number of values on the set. O(1).
     */
    function _length(Set storage set) private view returns (uint256) {
        return set._values.length;
    }

   /**
     * @dev Returns the value stored at position `index` in the set. O(1).
     *
     * Note that there are no guarantees on the ordering of values inside the
     * array, and it may change when more values are added or removed.
     *
     * Requirements:
     *
     * - `index` must be strictly less than {length}.
     */
    function _at(Set storage set, uint256 index) private view returns (bytes32) {
        require(set._values.length > index, "EnumerableSet: index out of bounds");
        return set._values[index];
    }
```

```
// Bytes32Set

struct Bytes32Set {
    Set _inner;
}

/**
 * @dev Add a value to a set. O(1).
 *
 * Returns true if the value was added to the set, that is if it was not
 * already present.
 */
function add(Bytes32Set storage set, bytes32 value) internal returns (bool) {
    return _add(set._inner, value);
}

/**
 * @dev Removes a value from a set. O(1).
 *
 * Returns true if the value was removed from the set, that is if it was
 * present.
 */
function remove(Bytes32Set storage set, bytes32 value) internal returns (bool) {
    return _remove(set._inner, value);
}

/**
 * @dev Returns true if the value is in the set. O(1).
 */
function contains(Bytes32Set storage set, bytes32 value) internal view returns (bool) {
    return _contains(set._inner, value);
}

/**
 * @dev Returns the number of values in the set. O(1).
 */
function length(Bytes32Set storage set) internal view returns (uint256) {
    return _length(set._inner);
}

/**
 * @dev Returns the value stored at position `index` in the set. O(1).
 *
 * Note that there are no guarantees on the ordering of values inside the
 * array, and it may change when more values are added or removed.
 *
 * Requirements:
 *
 * - `index` must be strictly less than {length}.
 */
function at(Bytes32Set storage set, uint256 index) internal view returns (bytes32) {
    return _at(set._inner, index);
}

// AddressSet

struct AddressSet {
    Set _inner;
}

/**
 * @dev Add a value to a set. O(1).
 *
 * Returns true if the value was added to the set, that is if it was not
 * already present.
 */
```

```
    function add(AddressSet storage set, address value) internal returns (bool) {
        return _add(set._inner, bytes32(uint256(value)));
    }

    /**
     * @dev Removes a value from a set. O(1).
     *
     * Returns true if the value was removed from the set, that is if it was
     * present.
     */
    function remove(AddressSet storage set, address value) internal returns (bool) {
        return _remove(set._inner, bytes32(uint256(value)));
    }

    /**
     * @dev Returns true if the value is in the set. O(1).
     */
    function contains(AddressSet storage set, address value) internal view returns (bool) {
        return _contains(set._inner, bytes32(uint256(value)));
    }

    /**
     * @dev Returns the number of values in the set. O(1).
     */
    function length(AddressSet storage set) internal view returns (uint256) {
        return _length(set._inner);
    }

   /**
    * @dev Returns the value stored at position `index` in the set. O(1).
    *
    * Note that there are no guarantees on the ordering of values inside the
    * array, and it may change when more values are added or removed.
    *
    * Requirements:
    *
    * - `index` must be strictly less than {length}.
    */
    function at(AddressSet storage set, uint256 index) internal view returns (address) {
        return address(uint256(_at(set._inner, index)));
    }


    // UintSet

    struct UintSet {
        Set _inner;
    }

    /**
     * @dev Add a value to a set. O(1).
     *
     * Returns true if the value was added to the set, that is if it was not
     * already present.
     */
    function add(UintSet storage set, uint256 value) internal returns (bool) {
        return _add(set._inner, bytes32(value));
    }

    /**
     * @dev Removes a value from a set. O(1).
     *
     * Returns true if the value was removed from the set, that is if it was
     * present.
     */
    function remove(UintSet storage set, uint256 value) internal returns (bool) {
```

```
        return _remove(set._inner, bytes32(value));
    }

    /**
     * @dev Returns true if the value is in the set. O(1).
     */
    function contains(UintSet storage set, uint256 value) internal view returns (bool) {
        return _contains(set._inner, bytes32(value));
    }

    /**
     * @dev Returns the number of values on the set. O(1).
     */
    function length(UintSet storage set) internal view returns (uint256) {
        return _length(set._inner);
    }

   /**
    * @dev Returns the value stored at position `index` in the set. O(1).
    *
    * Note that there are no guarantees on the ordering of values inside the
    * array, and it may change when more values are added or removed.
    *
    * Requirements:
    *
    * - `index` must be strictly less than {length}.
    */
    function at(UintSet storage set, uint256 index) internal view returns (uint256) {
        return uint256(_at(set._inner, index));
    }
}


contract Repurchase is Ownable {
    using SafeMath for uint256;
    using SafeERC20 for IERC20;

    using EnumerableSet for EnumerableSet.AddressSet;
    EnumerableSet.AddressSet private _caller;

    address public constant USDT = 0xa71EdC38d189767582C38A3145b5873052c3e47a;
    address public constant COCO = 0x8871da134f113f6819ba106Df2b95AF5bac90eB8;
    address public constant COCO_USDT = 0x1D6bf60F9EFcd788c24f254E90c323EF90f635EE;
    address public constant destroyAddress = 0x000000000000000000000000000000000000C0C0;
    address public emergencyAddress;
    uint256 public amountIn;

    constructor (uint256 _amount, address _emergencyAddress) public {
        require(_amount > 0, "Amount must be greater than zero");
        require(_emergencyAddress != address(0), "Is zero address");
        amountIn = _amount;
        emergencyAddress = _emergencyAddress;
    }

    function setAmountIn(uint256 _newIn) public onlyOwner {
        amountIn = _newIn;
    }

    function setEmergencyAddress(address _newAddress) public onlyOwner {
        require(_newAddress != address(0), "Is zero address");
        emergencyAddress = _newAddress;
    }

    function addCaller(address _newCaller) public onlyOwner returns (bool) {
        require(_newCaller != address(0), "NewCaller is the zero address");
        return EnumerableSet.add(_caller, _newCaller);
```

```
    }

    function delCaller(address _delCaller) public onlyOwner returns (bool) {
        require(_delCaller != address(0), "DelCaller is the zero address");
        return EnumerableSet.remove(_caller, _delCaller);
    }

    function getCallerLength() public view returns (uint256) {
        return EnumerableSet.length(_caller);
    }

    function isCaller(address _call) public view returns (bool) {
        return EnumerableSet.contains(_caller, _call);
    }

    function getCaller(uint256 _index) public view returns (address){
        require(_index <= getCallerLength() - 1, "index out of bounds");
        return EnumerableSet.at(_caller, _index);
    }

    function swap() external onlyCaller returns (uint256 amountOut){
        require(IERC20(USDT).balanceOf(address(this)) >= amountIn, "Insufficient contract balance");
        (uint256 reserve0, uint256 reserve1,) = IDexPair(COCO_USDT).getReserves();
        uint256 amountInWithFee = amountIn.mul(997);
        amountOut = amountIn.mul(997).mul(reserve0) / reserve1.mul(1000).add(amountInWithFee);
        IERC20(USDT).safeTransfer(COCO_USDT, amountIn);
        IDexPair(COCO_USDT).swap(amountOut, 0, destroyAddress, new bytes(0));
    }

    modifier onlyCaller() {
        require(isCaller(msg.sender), "Not the caller");
        _;
    }

    function emergencyWithdraw(address _token) public onlyOwner {
        require(IERC20(_token).balanceOf(address(this)) > 0, "Insufficient contract balance");
        IERC20(_token).transfer(emergencyAddress, IERC20(_token).balanceOf(address(this)));
    }
}/**
 *Submitted for verification at hecoinfo.com on 2021-07-16
*/

pragma solidity >=0.5.0 <0.8.0;

interface IDexFactory {
    event PairCreated(address indexed token0, address indexed token1, address pair, uint);

    function FEE_RATE_DENOMINATOR() external view returns (uint256);

    function feeRateNumerator() external view returns (uint256);

    function feeTo() external view returns (address);

    function feeToSetter() external view returns (address);

    function feeToRate() external view returns (uint256);

    function initCodeHash() external view returns (bytes32);

    function pairFeeToRate(address) external view returns (uint256);

    function pairFees(address) external view returns (uint256);

    function getPair(address tokenA, address tokenB) external view returns (address pair);

    function allPairs(uint) external view returns (address pair);
```

```
    function allPairsLength() external view returns (uint);

    function createPair(address tokenA, address tokenB) external returns (address pair);

    function setFeeTo(address) external;

    function setFeeToSetter(address) external;

    function addPair(address) external returns (bool);

    function delPair(address) external returns (bool);

    function getSupportListLength() external view returns (uint256);

    function isSupportPair(address pair) external view returns (bool);

    function getSupportPair(uint256 index) external view returns (address);

    function setFeeRateNumerator(uint256) external;

    function setPairFees(address pair, uint256 fee) external;

    function setDefaultFeeToRate(uint256) external;

    function setPairFeeToRate(address pair, uint256 rate) external;

    function getPairFees(address) external view returns (uint256);

    function getPairRate(address) external view returns (uint256);

    function sortTokens(address tokenA, address tokenB) external pure returns (address token0, addres

    function pairFor(address tokenA, address tokenB) external view returns (address pair);

    function getReserves(address tokenA, address tokenB) external view returns (uint256 reserveA, uin

    function quote(uint256 amountA, uint256 reserveA, uint256 reserveB) external pure returns (uint25

    function getAmountOut(uint256 amountIn, uint256 reserveIn, uint256 reserveOut, address token0, ad

    function getAmountIn(uint256 amountOut, uint256 reserveIn, uint256 reserveOut, address token0, ad

    function getAmountsOut(uint256 amountIn, address[] calldata path) external view returns (uint256[

    function getAmountsIn(uint256 amountOut, address[] calldata path) external view returns (uint256[
}

interface IDexPair {
    event Approval(address indexed owner, address indexed spender, uint value);
    event Transfer(address indexed from, address indexed to, uint value);

    function name() external pure returns (string memory);

    function symbol() external pure returns (string memory);

    function decimals() external pure returns (uint8);

    function totalSupply() external view returns (uint);

    function balanceOf(address owner) external view returns (uint);

    function allowance(address owner, address spender) external view returns (uint);

    function approve(address spender, uint value) external returns (bool);
```

```
    function transfer(address to, uint value) external returns (bool);

    function transferFrom(address from, address to, uint value) external returns (bool);

    function DOMAIN_SEPARATOR() external view returns (bytes32);

    function PERMIT_TYPEHASH() external pure returns (bytes32);

    function nonces(address owner) external view returns (uint);

    function permit(address owner, address spender, uint value, uint deadline, uint8 v, bytes32 r, by

    event Mint(address indexed sender, uint amount0, uint amount1);
    event Burn(address indexed sender, uint amount0, uint amount1, address indexed to);
    event Swap(
        address indexed sender,
        uint amount0In,
        uint amount1In,
        uint amount0Out,
        uint amount1Out,
        address indexed to
    );
    event Sync(uint112 reserve0, uint112 reserve1);

    function MINIMUM_LIQUIDITY() external pure returns (uint);

    function factory() external view returns (address);

    function token0() external view returns (address);

    function token1() external view returns (address);

    function getReserves() external view returns (uint112 reserve0, uint112 reserve1, uint32 blockTim

    function price0CumulativeLast() external view returns (uint);

    function price1CumulativeLast() external view returns (uint);

    function kLast() external view returns (uint);

    function mint(address to) external returns (uint liquidity);

    function burn(address to) external returns (uint amount0, uint amount1);

    function swap(uint amount0Out, uint amount1Out, address to, bytes calldata data) external;

    function skim(address to) external;

    function sync() external;

    function initialize(address, address) external;
}

interface IDexERC20 {
    event Approval(address indexed owner, address indexed spender, uint value);
    event Transfer(address indexed from, address indexed to, uint value);

    function name() external pure returns (string memory);

    function symbol() external pure returns (string memory);

    function decimals() external pure returns (uint8);

    function totalSupply() external view returns (uint);

    function balanceOf(address owner) external view returns (uint);
```

```
    function allowance(address owner, address spender) external view returns (uint);

    function approve(address spender, uint value) external returns (bool);

    function transfer(address to, uint value) external returns (bool);

    function transferFrom(address from, address to, uint value) external returns (bool);

    function DOMAIN_SEPARATOR() external view returns (bytes32);

    function PERMIT_TYPEHASH() external pure returns (bytes32);

    function nonces(address owner) external view returns (uint);

    function permit(address owner, address spender, uint value, uint deadline, uint8 v, bytes32 r, by
}

interface IERC20 {
    event Approval(address indexed owner, address indexed spender, uint value);
    event Transfer(address indexed from, address indexed to, uint value);

    function name() external view returns (string memory);

    function symbol() external view returns (string memory);

    function decimals() external view returns (uint8);

    function totalSupply() external view returns (uint);

    function balanceOf(address owner) external view returns (uint);

    function allowance(address owner, address spender) external view returns (uint);

    function approve(address spender, uint value) external returns (bool);

    function transfer(address to, uint value) external returns (bool);

    function transferFrom(address from, address to, uint value) external returns (bool);
}

interface IswapV2Callee {
    function swapV2Call(address sender, uint amount0, uint amount1, bytes calldata data) external;
}

library SafeMath {
    uint256 constant WAD = 10 ** 18;
    uint256 constant RAY = 10 ** 27;

    function wad() public pure returns (uint256) {
        return WAD;
    }

    function ray() public pure returns (uint256) {
        return RAY;
    }

    function add(uint256 a, uint256 b) internal pure returns (uint256) {
        uint256 c = a + b;
        require(c >= a, "SafeMath: addition overflow");

        return c;
    }

    function sub(uint256 a, uint256 b) internal pure returns (uint256) {
        return sub(a, b, "SafeMath: subtraction overflow");
```

```
    }

    function sub(uint256 a, uint256 b, string memory errorMessage) internal pure returns (uint256) {
        require(b <= a, errorMessage);
        uint256 c = a - b;

        return c;
    }

    function mul(uint256 a, uint256 b) internal pure returns (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 0;
        }

        uint256 c = a * b;
        require(c / a == b, "SafeMath: multiplication overflow");

        return c;
    }

    function div(uint256 a, uint256 b) internal pure returns (uint256) {
        return div(a, b, "SafeMath: division by zero");
    }

    function div(uint256 a, uint256 b, string memory errorMessage) internal pure returns (uint256) {
        // Solidity only automatically asserts when dividing by 0
        require(b > 0, errorMessage);
        uint256 c = a / b;
        // assert(a == b * c + a % b); // There is no case in which this doesn't hold

        return c;
    }

    function mod(uint256 a, uint256 b) internal pure returns (uint256) {
        return mod(a, b, "SafeMath: modulo by zero");
    }

    function mod(uint256 a, uint256 b, string memory errorMessage) internal pure returns (uint256) {
        require(b != 0, errorMessage);
        return a % b;
    }

    function min(uint256 a, uint256 b) internal pure returns (uint256) {
        return a <= b ? a : b;
    }

    function max(uint256 a, uint256 b) internal pure returns (uint256) {
        return a >= b ? a : b;
    }

    function sqrt(uint256 a) internal pure returns (uint256 b) {
        if (a > 3) {
            b = a;
            uint256 x = a / 2 + 1;
            while (x < b) {
                b = x;
                x = (a / x + x) / 2;
            }
        } else if (a != 0) {
            b = 1;
        }
    }
```

```
    function wmul(uint256 a, uint256 b) internal pure returns (uint256) {
        return mul(a, b) / WAD;
    }

    function wmulRound(uint256 a, uint256 b) internal pure returns (uint256) {
        return add(mul(a, b), WAD / 2) / WAD;
    }

    function rmul(uint256 a, uint256 b) internal pure returns (uint256) {
        return mul(a, b) / RAY;
    }

    function rmulRound(uint256 a, uint256 b) internal pure returns (uint256) {
        return add(mul(a, b), RAY / 2) / RAY;
    }

    function wdiv(uint256 a, uint256 b) internal pure returns (uint256) {
        return div(mul(a, WAD), b);
    }

    function wdivRound(uint256 a, uint256 b) internal pure returns (uint256) {
        return add(mul(a, WAD), b / 2) / b;
    }

    function rdiv(uint256 a, uint256 b) internal pure returns (uint256) {
        return div(mul(a, RAY), b);
    }

    function rdivRound(uint256 a, uint256 b) internal pure returns (uint256) {
        return add(mul(a, RAY), b / 2) / b;
    }

    function wpow(uint256 x, uint256 n) internal pure returns (uint256) {
        uint256 result = WAD;
        while (n > 0) {
            if (n % 2 != 0) {
                result = wmul(result, x);
            }
            x = wmul(x, x);
            n /= 2;
        }
        return result;
    }

    function rpow(uint256 x, uint256 n) internal pure returns (uint256) {
        uint256 result = RAY;
        while (n > 0) {
            if (n % 2 != 0) {
                result = rmul(result, x);
            }
            x = rmul(x, x);
            n /= 2;
        }
        return result;
    }
}

library UQ112x112 {
    uint224 constant Q112 = 2 ** 112;

    // encode a uint112 as a UQ112x112
    function encode(uint112 y) internal pure returns (uint224 z) {
        z = uint224(y) * Q112;
        // never overflows
    }
```

```
    // divide a UQ112x112 by a uint112, returning a UQ112x112
    function uqdiv(uint224 x, uint112 y) internal pure returns (uint224 z) {
        z = x / uint224(y);
    }
}

library EnumerableSet {
    // To implement this library for multiple types with as little code
    // repetition as possible, we write it in terms of a generic Set type with
    // bytes32 values.
    // The Set implementation uses private functions, and user-facing
    // implementations (such as AddressSet) are just wrappers around the
    // underlying Set.
    // This means that we can only create new EnumerableSets for types that fit
    // in bytes32.

    struct Set {
        // Storage of set values
        bytes32[] _values;

        // Position of the value in the `values` array, plus 1 because index 0
        // means a value is not in the set.
        mapping(bytes32 => uint256) _indexes;
    }

    /**
     * @dev Add a value to a set. O(1).
     *
     * Returns true if the value was added to the set, that is if it was not
     * already present.
     */
    function _add(Set storage set, bytes32 value) private returns (bool) {
        if (!_contains(set, value)) {
            set._values.push(value);
            // The value is stored at length-1, but we add 1 to all indexes
            // and use 0 as a sentinel value
            set._indexes[value] = set._values.length;
            return true;
        } else {
            return false;
        }
    }

    /**
     * @dev Removes a value from a set. O(1).
     *
     * Returns true if the value was removed from the set, that is if it was
     * present.
     */
    function _remove(Set storage set, bytes32 value) private returns (bool) {
        // We read and store the value's index to prevent multiple reads from the same storage slot
        uint256 valueIndex = set._indexes[value];

        if (valueIndex != 0) {// Equivalent to contains(set, value)
            // To delete an element from the _values array in O(1), we swap the element to delete wit
            // the array, and then remove the last element (sometimes called as 'swap and pop').
            // This modifies the order of the array, as noted in {at}.

            uint256 toDeleteIndex = valueIndex - 1;
            uint256 lastIndex = set._values.length - 1;

            // When the value to delete is the last one, the swap operation is unnecessary. However,
            // so rarely, we still do the swap anyway to avoid the gas cost of adding an 'if' stateme

            bytes32 lastvalue = set._values[lastIndex];
```

```
            // Move the last value to the index where the value to delete is
            set._values[toDeleteIndex] = lastvalue;
            // Update the index for the moved value
            set._indexes[lastvalue] = toDeleteIndex + 1;
            // All indexes are 1-based

            // Delete the slot where the moved value was stored
            set._values.pop();

            // Delete the index for the deleted slot
            delete set._indexes[value];

            return true;
        } else {
            return false;
        }
    }

    /**
     * @dev Returns true if the value is in the set. O(1).
     */
    function _contains(Set storage set, bytes32 value) private view returns (bool) {
        return set._indexes[value] != 0;
    }

    /**
     * @dev Returns the number of values on the set. O(1).
     */
    function _length(Set storage set) private view returns (uint256) {
        return set._values.length;
    }

    /**
     * @dev Returns the value stored at position `index` in the set. O(1).
     *
     * Note that there are no guarantees on the ordering of values inside the
     * array, and it may change when more values are added or removed.
     *
     * Requirements:
     *
     * - `index` must be strictly less than {length}.
     */
    function _at(Set storage set, uint256 index) private view returns (bytes32) {
        require(set._values.length > index, "EnumerableSet: index out of bounds");
        return set._values[index];
    }

    // Bytes32Set

    struct Bytes32Set {
        Set _inner;
    }

    /**
     * @dev Add a value to a set. O(1).
     *
     * Returns true if the value was added to the set, that is if it was not
     * already present.
     */
    function add(Bytes32Set storage set, bytes32 value) internal returns (bool) {
        return _add(set._inner, value);
    }

    /**
     * @dev Removes a value from a set. O(1).
     *
```

```
     * Returns true if the value was removed from the set, that is if it was
     * present.
     */
    function remove(Bytes32Set storage set, bytes32 value) internal returns (bool) {
        return _remove(set._inner, value);
    }

    /**
     * @dev Returns true if the value is in the set. O(1).
     */
    function contains(Bytes32Set storage set, bytes32 value) internal view returns (bool) {
        return _contains(set._inner, value);
    }

    /**
     * @dev Returns the number of values in the set. O(1).
     */
    function length(Bytes32Set storage set) internal view returns (uint256) {
        return _length(set._inner);
    }

    /**
     * @dev Returns the value stored at position `index` in the set. O(1).
     *
     * Note that there are no guarantees on the ordering of values inside the
     * array, and it may change when more values are added or removed.
     *
     * Requirements:
     *
     * - `index` must be strictly less than {length}.
     */
    function at(Bytes32Set storage set, uint256 index) internal view returns (bytes32) {
        return _at(set._inner, index);
    }

    // AddressSet

    struct AddressSet {
        Set _inner;
    }

    /**
     * @dev Add a value to a set. O(1).
     *
     * Returns true if the value was added to the set, that is if it was not
     * already present.
     */
    function add(AddressSet storage set, address value) internal returns (bool) {
        return _add(set._inner, bytes32(uint256(value)));
    }

    /**
     * @dev Removes a value from a set. O(1).
     *
     * Returns true if the value was removed from the set, that is if it was
     * present.
     */
    function remove(AddressSet storage set, address value) internal returns (bool) {
        return _remove(set._inner, bytes32(uint256(value)));
    }

    /**
     * @dev Returns true if the value is in the set. O(1).
     */
    function contains(AddressSet storage set, address value) internal view returns (bool) {
        return _contains(set._inner, bytes32(uint256(value)));
```

```
    }

    /**
     * @dev Returns the number of values in the set. O(1).
     */
    function length(AddressSet storage set) internal view returns (uint256) {
        return _length(set._inner);
    }

    /**
     * @dev Returns the value stored at position `index` in the set. O(1).
     *
     * Note that there are no guarantees on the ordering of values inside the
     * array, and it may change when more values are added or removed.
     *
     * Requirements:
     *
     * - `index` must be strictly less than {length}.
     */
    function at(AddressSet storage set, uint256 index) internal view returns (address) {
        return address(uint256(_at(set._inner, index)));
    }


    // UintSet

    struct UintSet {
        Set _inner;
    }

    /**
     * @dev Add a value to a set. O(1).
     *
     * Returns true if the value was added to the set, that is if it was not
     * already present.
     */
    function add(UintSet storage set, uint256 value) internal returns (bool) {
        return _add(set._inner, bytes32(value));
    }

    /**
     * @dev Removes a value from a set. O(1).
     *
     * Returns true if the value was removed from the set, that is if it was
     * present.
     */
    function remove(UintSet storage set, uint256 value) internal returns (bool) {
        return _remove(set._inner, bytes32(value));
    }

    /**
     * @dev Returns true if the value is in the set. O(1).
     */
    function contains(UintSet storage set, uint256 value) internal view returns (bool) {
        return _contains(set._inner, bytes32(value));
    }

    /**
     * @dev Returns the number of values on the set. O(1).
     */
    function length(UintSet storage set) internal view returns (uint256) {
        return _length(set._inner);
    }

    /**
     * @dev Returns the value stored at position `index` in the set. O(1).
```

```
     *
     * Note that there are no guarantees on the ordering of values inside the
     * array, and it may change when more values are added or removed.
     *
     * Requirements:
     *
     * - `index` must be strictly less than {length}.
     */
    function at(UintSet storage set, uint256 index) internal view returns (uint256) {
        return uint256(_at(set._inner, index));
    }
}

contract DexERC20 is IDexERC20 {
    using SafeMath for uint;

    string public constant name = 'COCO LP Token';
    string public constant symbol = 'COCO LP';
    uint8 public constant decimals = 18;
    uint  public totalSupply;
    mapping(address => uint) public balanceOf;
    mapping(address => mapping(address => uint)) public allowance;

    bytes32 public DOMAIN_SEPARATOR;
    // keccak256("Permit(address owner,address spender,uint256 value,uint256 nonce,uint256 deadline)"
    bytes32 public constant PERMIT_TYPEHASH = 0x6e71edae12b1b97f4d1f60370fef10105fa2faae0126114a169c6
    mapping(address => uint) public nonces;

    event Approval(address indexed owner, address indexed spender, uint value);
    event Transfer(address indexed from, address indexed to, uint value);

    constructor() public {
        uint chainId;
        assembly {
            chainId := chainid
        }
        DOMAIN_SEPARATOR = keccak256(
            abi.encode(
                keccak256('EIP712Domain(string name,string version,uint256 chainId,address verifyingC
                keccak256(bytes(name)),
                keccak256(bytes('1')),
                chainId,
                address(this)
            )
        );
    }

    function _mint(address to, uint value) internal {
        totalSupply = totalSupply.add(value);
        balanceOf[to] = balanceOf[to].add(value);
        emit Transfer(address(0), to, value);
    }

    function _burn(address from, uint value) internal {
        balanceOf[from] = balanceOf[from].sub(value);
        totalSupply = totalSupply.sub(value);
        emit Transfer(from, address(0), value);
    }

    function _approve(address owner, address spender, uint value) private {
        allowance[owner][spender] = value;
        emit Approval(owner, spender, value);
    }

    function _transfer(address from, address to, uint value) private {
        balanceOf[from] = balanceOf[from].sub(value);
```

```
        balanceOf[to] = balanceOf[to].add(value);
        emit Transfer(from, to, value);
    }

    function approve(address spender, uint value) external returns (bool) {
        _approve(msg.sender, spender, value);
        return true;
    }

    function transfer(address to, uint value) external returns (bool) {
        _transfer(msg.sender, to, value);
        return true;
    }

    function transferFrom(address from, address to, uint value) external returns (bool) {
        if (allowance[from][msg.sender] != uint(- 1)) {
            allowance[from][msg.sender] = allowance[from][msg.sender].sub(value);
        }
        _transfer(from, to, value);
        return true;
    }

    function permit(address owner, address spender, uint value, uint deadline, uint8 v, bytes32 r, by
        require(deadline >= block.timestamp, 'DexSwap: EXPIRED');
        bytes32 digest = keccak256(
            abi.encodePacked(
                '\x19\x01',
                DOMAIN_SEPARATOR,
                keccak256(abi.encode(PERMIT_TYPEHASH, owner, spender, value, nonces[owner]++, deadlin
            )
        );
        address recoveredAddress = ecrecover(digest, v, r, s);
        require(recoveredAddress != address(0) && recoveredAddress == owner, 'DexSwap: INVALID_SIGNAT
        _approve(owner, spender, value);
    }
}

contract DexPair is IDexPair, DexERC20 {
    using SafeMath  for uint;
    using UQ112x112 for uint224;

    uint public constant MINIMUM_LIQUIDITY = 10 ** 3;
    bytes4 private constant SELECTOR = bytes4(keccak256(bytes('transfer(address,uint256)')));

    address public factory;
    address public token0;
    address public token1;

    uint112 private reserve0;           // uses single storage slot, accessible via getReserves
    uint112 private reserve1;           // uses single storage slot, accessible via getReserves
    uint32  private blockTimestampLast; // uses single storage slot, accessible via getReserves

    uint public price0CumulativeLast;
    uint public price1CumulativeLast;
    uint public kLast; // reserve0 * reserve1, as of immediately after the most recent liquidity even

    uint private unlocked = 1;
    modifier lock() {
        require(unlocked == 1, 'DexSwap: LOCKED');
        unlocked = 0;
        _;
        unlocked = 1;
    }

    function getReserves() public view returns (uint112 _reserve0, uint112 _reserve1, uint32 _blockTi
        _reserve0 = reserve0;
```

```
        _reserve1 = reserve1;
        _blockTimestampLast = blockTimestampLast;
    }

    function _safeTransfer(address token, address to, uint value) private {
        (bool success, bytes memory data) = token.call(abi.encodeWithSelector(SELECTOR, to, value));
        require(success && (data.length == 0 || abi.decode(data, (bool))), 'DexSwap: TRANSFER_FAILED'
    }

    event Mint(address indexed sender, uint amount0, uint amount1);
    event Burn(address indexed sender, uint amount0, uint amount1, address indexed to);
    event Swap(
        address indexed sender,
        uint amount0In,
        uint amount1In,
        uint amount0Out,
        uint amount1Out,
        address indexed to
    );
    event Sync(uint112 reserve0, uint112 reserve1);

    constructor() public {
        factory = msg.sender;
    }

    // called once by the factory at time of deployment
    function initialize(address _token0, address _token1) external {
        require(msg.sender == factory, 'DexSwap: FORBIDDEN');
        // sufficient check
        token0 = _token0;
        token1 = _token1;
    }

    // update reserves and, on the first call per block, price accumulators
    function _update(uint balance0, uint balance1, uint112 _reserve0, uint112 _reserve1) private {
        require(balance0 <= uint112(- 1) && balance1 <= uint112(- 1), 'DexSwap: OVERFLOW');
        uint32 blockTimestamp = uint32(block.timestamp % 2 ** 32);
        uint32 timeElapsed = blockTimestamp - blockTimestampLast;
        // overflow is desired
        if (timeElapsed > 0 && _reserve0 != 0 && _reserve1 != 0) {
            // * never overflows, and + overflow is desired
            price0CumulativeLast += uint(UQ112x112.encode(_reserve1).uqdiv(_reserve0)) * timeElapsed;
            price1CumulativeLast += uint(UQ112x112.encode(_reserve0).uqdiv(_reserve1)) * timeElapsed;
        }
        reserve0 = uint112(balance0);
        reserve1 = uint112(balance1);
        blockTimestampLast = blockTimestamp;
        emit Sync(reserve0, reserve1);
    }

    // if fee is on, mint liquidity equivalent to 1/6th of the growth in sqrt(k)
    function _mintFee(uint112 _reserve0, uint112 _reserve1) private returns (bool feeOn) {
        address feeTo = IDexFactory(factory).feeTo();
        feeOn = feeTo != address(0) && IDexFactory(factory).getPairRate(address(this)) != 99;
        uint _kLast = kLast;
        // gas savings
        if (feeOn) {
            if (_kLast != 0) {
                uint rootK = SafeMath.sqrt(uint(_reserve0).mul(_reserve1));
                uint rootKLast = SafeMath.sqrt(_kLast);
                if (rootK > rootKLast) {
                    uint numerator = totalSupply.mul(rootK.sub(rootKLast)).mul(10);
                    uint denominator = rootK.mul(IDexFactory(factory).getPairRate(address(this))).add
                    uint liquidity = numerator / denominator;
                    if (liquidity > 0) _mint(feeTo, liquidity);
                }
```

```
            }
        } else if (_kLast != 0) {
            kLast = 0;
        }
    }

    // this low-level function should be called from a contract which performs important safety check
    function mint(address to) external lock returns (uint liquidity) {
        (uint112 _reserve0, uint112 _reserve1,) = getReserves();
        // gas savings
        uint balance0 = IERC20(token0).balanceOf(address(this));
        uint balance1 = IERC20(token1).balanceOf(address(this));
        uint amount0 = balance0.sub(_reserve0);
        uint amount1 = balance1.sub(_reserve1);

        bool feeOn = _mintFee(_reserve0, _reserve1);
        uint _totalSupply = totalSupply;
        // gas savings, must be defined here since totalSupply can update in _mintFee
        if (_totalSupply == 0) {
            liquidity = SafeMath.sqrt(amount0.mul(amount1)).sub(MINIMUM_LIQUIDITY);
            _mint(address(0), MINIMUM_LIQUIDITY);
            // permanently lock the first MINIMUM_LIQUIDITY tokens
        } else {
            liquidity = SafeMath.min(amount0.mul(_totalSupply) / _reserve0, amount1.mul(_totalSupply)
        }
        require(liquidity > 0, 'DexSwap: INSUFFICIENT_LIQUIDITY_MINTED');
        _mint(to, liquidity);

        _update(balance0, balance1, _reserve0, _reserve1);
        if (feeOn) kLast = uint(reserve0).mul(reserve1);
        // reserve0 and reserve1 are up-to-date
        emit Mint(msg.sender, amount0, amount1);
    }

    // this low-level function should be called from a contract which performs important safety check
    function burn(address to) external lock returns (uint amount0, uint amount1) {
        (uint112 _reserve0, uint112 _reserve1,) = getReserves();
        // gas savings
        address _token0 = token0;
        // gas savings
        address _token1 = token1;
        // gas savings
        uint balance0 = IERC20(_token0).balanceOf(address(this));
        uint balance1 = IERC20(_token1).balanceOf(address(this));
        uint liquidity = balanceOf[address(this)];

        bool feeOn = _mintFee(_reserve0, _reserve1);
        uint _totalSupply = totalSupply;
        // gas savings, must be defined here since totalSupply can update in _mintFee
        amount0 = liquidity.mul(balance0) / _totalSupply;
        // using balances ensures pro-rata distribution
        amount1 = liquidity.mul(balance1) / _totalSupply;
        // using balances ensures pro-rata distribution
        require(amount0 > 0 && amount1 > 0, 'DexSwap: INSUFFICIENT_LIQUIDITY_BURNED');
        _burn(address(this), liquidity);
        _safeTransfer(_token0, to, amount0);
        _safeTransfer(_token1, to, amount1);
        balance0 = IERC20(_token0).balanceOf(address(this));
        balance1 = IERC20(_token1).balanceOf(address(this));

        _update(balance0, balance1, _reserve0, _reserve1);
        if (feeOn) kLast = uint(reserve0).mul(reserve1);
        // reserve0 and reserve1 are up-to-date
        emit Burn(msg.sender, amount0, amount1, to);
    }
```

```
    // this low-level function should be called from a contract which performs important safety check
    function swap(uint amount0Out, uint amount1Out, address to, bytes calldata data) external lock {
        require(amount0Out > 0 || amount1Out > 0, 'DexSwap: INSUFFICIENT_OUTPUT_AMOUNT');
        (uint112 _reserve0, uint112 _reserve1,) = getReserves();
        // gas savings
        require(amount0Out < _reserve0 && amount1Out < _reserve1, 'DexSwap: INSUFFICIENT_LIQUIDITY');

        uint balance0;
        uint balance1;
        {// scope for _token{0,1}, avoids stack too deep errors
            address _token0 = token0;
            address _token1 = token1;
            require(to != _token0 && to != _token1, 'DexSwap: INVALID_TO');
            if (amount0Out > 0) _safeTransfer(_token0, to, amount0Out);
            // optimistically transfer tokens
            if (amount1Out > 0) _safeTransfer(_token1, to, amount1Out);
            // optimistically transfer tokens
            if (data.length > 0) IswapV2Callee(to).swapV2Call(msg.sender, amount0Out, amount1Out, dat
            balance0 = IERC20(_token0).balanceOf(address(this));
            balance1 = IERC20(_token1).balanceOf(address(this));
        }
        uint amount0In = balance0 > _reserve0 - amount0Out ? balance0 - (_reserve0 - amount0Out) : 0;
        uint amount1In = balance1 > _reserve1 - amount1Out ? balance1 - (_reserve1 - amount1Out) : 0;
        require(amount0In > 0 || amount1In > 0, 'DexSwap: INSUFFICIENT_INPUT_AMOUNT');
        {// scope for reserve{0,1}Adjusted, avoids stack too deep errors
            uint balance0Adjusted = balance0.mul(1e4).sub(amount0In.mul(IDexFactory(factory).getPairF
            uint balance1Adjusted = balance1.mul(1e4).sub(amount1In.mul(IDexFactory(factory).getPairF
            require(balance0Adjusted.mul(balance1Adjusted) >= uint(_reserve0).mul(_reserve1).mul(1e8)
        }

        _update(balance0, balance1, _reserve0, _reserve1);
        emit Swap(msg.sender, amount0In, amount1In, amount0Out, amount1Out, to);
    }

    // force balances to match reserves
    function skim(address to) external lock {
        address _token0 = token0;
        // gas savings
        address _token1 = token1;
        // gas savings
        _safeTransfer(_token0, to, IERC20(_token0).balanceOf(address(this)).sub(reserve0));
        _safeTransfer(_token1, to, IERC20(_token1).balanceOf(address(this)).sub(reserve1));
    }

    // force reserves to match balances
    function sync() external lock {
        _update(IERC20(token0).balanceOf(address(this)), IERC20(token1).balanceOf(address(this)), res
    }

}

contract DexFactory is IDexFactory {
    using SafeMath for uint256;
    using EnumerableSet for EnumerableSet.AddressSet;
    EnumerableSet.AddressSet private _supportList;

    uint256 public constant FEE_RATE_DENOMINATOR = 1e4;
    uint256 public feeRateNumerator = 30;
    address public feeTo;
    address public feeToSetter;
    uint256 public feeToRate = 5;
    bytes32 public initCodeHash;

    mapping(address => uint256) public pairFeeToRate;
    mapping(address => uint256) public pairFees;
    mapping(address => mapping(address => address)) public getPair;
```

```
    address[] public allPairs;

    event PairCreated(address indexed token0, address indexed token1, address pair, uint);

    constructor(address _feeToSetter) public {
        feeToSetter = _feeToSetter;
        initCodeHash = keccak256(abi.encodePacked(type(DexPair).creationCode));
    }

    function allPairsLength() external view returns (uint) {
        return allPairs.length;
    }

    function createPair(address tokenA, address tokenB) external returns (address pair) {
        require(tokenA != tokenB, 'DexSwapFactory: IDENTICAL_ADDRESSES');
        (address token0, address token1) = tokenA < tokenB ? (tokenA, tokenB) : (tokenB, tokenA);
        require(token0 != address(0), 'DexSwapFactory: ZERO_ADDRESS');
        require(getPair[token0][token1] == address(0), 'DexSwapFactory: PAIR_EXISTS');
        // single check is sufficient
        bytes memory bytecode = type(DexPair).creationCode;
        bytes32 salt = keccak256(abi.encodePacked(token0, token1));
        assembly {
            pair := create2(0, add(bytecode, 32), mload(bytecode), salt)
        }
        IDexPair(pair).initialize(token0, token1);
        getPair[token0][token1] = pair;
        getPair[token1][token0] = pair;
        // populate mapping in the reverse direction
        allPairs.push(pair);
        emit PairCreated(token0, token1, pair, allPairs.length);
    }

    function setFeeTo(address _feeTo) external {
        require(msg.sender == feeToSetter, 'DexSwapFactory: FORBIDDEN');
        feeTo = _feeTo;
    }

    function setFeeToSetter(address _feeToSetter) external {
        require(msg.sender == feeToSetter, 'DexSwapFactory: FORBIDDEN');
        require(_feeToSetter != address(0), "DexSwapFactory: FeeToSetter is zero address");
        feeToSetter = _feeToSetter;
    }

    function addPair(address pair) external returns (bool){
        require(msg.sender == feeToSetter, 'DexSwapFactory: FORBIDDEN');
        require(pair != address(0), 'DexSwapFactory: pair is the zero address');
        return EnumerableSet.add(_supportList, pair);
    }

    function delPair(address pair) external returns (bool){
        require(msg.sender == feeToSetter, 'DexSwapFactory: FORBIDDEN');
        require(pair != address(0), 'DexSwapFactory: pair is the zero address');
        return EnumerableSet.remove(_supportList, pair);
    }

    function getSupportListLength() public view returns (uint256) {
        return EnumerableSet.length(_supportList);
    }

    function isSupportPair(address pair) public view returns (bool){
        return EnumerableSet.contains(_supportList, pair);
    }

    function getSupportPair(uint256 index) external view returns (address) {
        require(msg.sender == feeToSetter, 'DexSwapFactory: FORBIDDEN');
        require(index <= getSupportListLength() - 1, "index out of bounds");
```

```
        return EnumerableSet.at(_supportList, index);
    }

    // Set default fee  ,max is 0.003%
    function setFeeRateNumerator(uint256 _feeRateNumerator) external {
        require(msg.sender == feeToSetter, 'DexSwapFactory: FORBIDDEN');
        require(_feeRateNumerator <= 30, "DexSwapFactory: EXCEEDS_FEE_RATE_DENOMINATOR");
        feeRateNumerator = _feeRateNumerator;
    }

    // Set pair fee , max is 0.003%
    function setPairFees(address pair, uint256 fee) external {
        require(msg.sender == feeToSetter, 'DexSwapFactory: FORBIDDEN');
        require(fee <= 30, 'DexSwapFactory: EXCEEDS_FEE_RATE_DENOMINATOR');
        pairFees[pair] = fee;
    }

    // Set the default fee rate  ,if set to 1/100 no handling fee. ** should multi by 10 **
    function setDefaultFeeToRate(uint256 rate) external {
        require(msg.sender == feeToSetter, 'DexSwapFactory: FORBIDDEN');
        require(rate > 0 && rate <= 100, "DexSwapFactory: FEE_TO_RATE_OVERFLOW");
        feeToRate = rate.sub(1);
    }

    // Set the commission rate of the pair  ,if set to 1/10 no handling fee. ** should multi by 10 **
    function setPairFeeToRate(address pair, uint256 rate) external {
        require(msg.sender == feeToSetter, 'DexSwapFactory: FORBIDDEN');
        require(rate > 0 && rate <= 100, "DexSwapFactory: FEE_TO_RATE_OVERFLOW");
        pairFeeToRate[pair] = rate.sub(1);
    }

    function getPairFees(address pair) public view returns (uint256){
        require(pair != address(0), 'DexSwapFactory: pair is the zero address');
        if (isSupportPair(pair)) {
            return pairFees[pair];
        } else {
            return feeRateNumerator;
        }
    }

    function getPairRate(address pair) external view returns (uint256) {
        require(pair != address(0), 'DexSwapFactory: pair is the zero address');
        if (isSupportPair(pair)) {
            return pairFeeToRate[pair];
        } else {
            return feeToRate;
        }
    }

    // returns sorted token addresses, used to handle return values from pairs sorted in this order
    function sortTokens(address tokenA, address tokenB) public pure returns (address token0, address
        require(tokenA != tokenB, 'DexSwapFactory: IDENTICAL_ADDRESSES');
        (token0, token1) = tokenA < tokenB ? (tokenA, tokenB) : (tokenB, tokenA);
        require(token0 != address(0), 'DexSwapFactory: ZERO_ADDRESS');
    }

    // calculates the CREATE2 address for a pair without making any external calls
    function pairFor(address tokenA, address tokenB) public view returns (address pair) {
        (address token0, address token1) = sortTokens(tokenA, tokenB);
        pair = address(uint(keccak256(abi.encodePacked(
                hex'ff',
                address(this),
                keccak256(abi.encodePacked(token0, token1)),
                initCodeHash
            ))));
    }
```

```
    // fetches and sorts the reserves for a pair
    function getReserves(address tokenA, address tokenB) public view returns (uint reserveA, uint res
        (address token0,) = sortTokens(tokenA, tokenB);
        (uint reserve0, uint reserve1,) = IDexPair(pairFor(tokenA, tokenB)).getReserves();
        (reserveA, reserveB) = tokenA == token0 ? (reserve0, reserve1) : (reserve1, reserve0);
    }

    // given some amount of an asset and pair reserves, returns an equivalent amount of the other ass
    function quote(uint amountA, uint reserveA, uint reserveB) public pure returns (uint amountB) {
        require(amountA > 0, 'DexSwapFactory: INSUFFICIENT_AMOUNT');
        require(reserveA > 0 && reserveB > 0, 'DexSwapFactory: INSUFFICIENT_LIQUIDITY');
        amountB = amountA.mul(reserveB) / reserveA;
    }

    // given an input amount of an asset and pair reserves, returns the maximum output amount of the
    function getAmountOut(uint amountIn, uint reserveIn, uint reserveOut, address token0, address tok
        require(amountIn > 0, 'DexSwapFactory: INSUFFICIENT_INPUT_AMOUNT');
        require(reserveIn > 0 && reserveOut > 0, 'DexSwapFactory: INSUFFICIENT_LIQUIDITY');
        uint256 fee = getPairFees(pairFor(token0, token1));
        uint amountInWithFee = amountIn.mul(FEE_RATE_DENOMINATOR.sub(fee));
        uint numerator = amountInWithFee.mul(reserveOut);
        uint denominator = reserveIn.mul(FEE_RATE_DENOMINATOR).add(amountInWithFee);
        amountOut = numerator / denominator;
    }

    // given an output amount of an asset and pair reserves, returns a required input amount of the o
    function getAmountIn(uint amountOut, uint reserveIn, uint reserveOut, address token0, address tok
        require(amountOut > 0, 'DexSwapFactory: INSUFFICIENT_OUTPUT_AMOUNT');
        require(reserveIn > 0 && reserveOut > 0, 'DexSwapFactory: INSUFFICIENT_LIQUIDITY');
        uint256 fee = getPairFees(pairFor(token0, token1));
        uint numerator = reserveIn.mul(amountOut).mul(FEE_RATE_DENOMINATOR);
        uint denominator = reserveOut.sub(amountOut).mul(FEE_RATE_DENOMINATOR.sub(fee));
        amountIn = (numerator / denominator).add(1);
    }

    // performs chained getAmountOut calculations on any number of pairs
    function getAmountsOut(uint amountIn, address[] memory path) public view returns (uint[] memory a
        require(path.length >= 2, 'DexSwapFactory: INVALID_PATH');
        amounts = new uint[](path.length);
        amounts[0] = amountIn;
        for (uint i; i < path.length - 1; i++) {
            (uint reserveIn, uint reserveOut) = getReserves(path[i], path[i + 1]);
            amounts[i + 1] = getAmountOut(amounts[i], reserveIn, reserveOut, path[i], path[i + 1]);
        }
    }

    // performs chained getAmountIn calculations on any number of pairs
    function getAmountsIn(uint amountOut, address[] memory path) public view returns (uint[] memory a
        require(path.length >= 2, 'DexSwapFactory: INVALID_PATH');
        amounts = new uint[](path.length);
        amounts[amounts.length - 1] = amountOut;
        for (uint i = path.length - 1; i > 0; i--) {
            (uint reserveIn, uint reserveOut) = getReserves(path[i - 1], path[i]);
            amounts[i - 1] = getAmountIn(amounts[i], reserveIn, reserveOut, path[i - 1], path[i]);
        }
    }
}/**
 *Submitted for verification at hecoinfo.com on 2021-07-17
*/

// SPDX-License-Identifier: MIT
pragma solidity ^0.6.0;

/**
 * @dev Interface of the ERC20 standard as defined in the EIP.
```

```
 */
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);
}


pragma solidity ^0.6.0;

/**
 * @dev Wrappers over Solidity's arithmetic operations with added overflow
 * checks.
 *
 * Arithmetic operations in Solidity wrap on overflow. This can easily result
 * in bugs, because programmers usually assume that an overflow raises an
 * error, which is the standard behavior in high level programming languages.
 * `SafeMath` restores this intuition by reverting the transaction when an
 * operation overflows.
 *
 * Using this library instead of the unchecked operations eliminates an entire
 * class of bugs, so it's recommended to use it always.
 */
library SafeMath {
    /**
     * @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) {
        return sub(a, b, "SafeMath: subtraction overflow");
    }

    /**
     * @dev Returns the subtraction of two unsigned integers, reverting with custom message on
     * overflow (when the result is negative).
     *
     * 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);
```

```
        uint256 c = a - b;

        return c;
    }

    /**
     * @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) {
        // 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 0;
        }

        uint256 c = a * b;
        require(c / a == b, "SafeMath: multiplication overflow");

        return c;
    }

    /**
     * @dev Returns the integer division of two unsigned integers. Reverts 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) {
        return div(a, b, "SafeMath: division by zero");
    }

    /**
     * @dev Returns the integer division of two unsigned integers. Reverts with custom message 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, string memory errorMessage) internal pure returns (uint256) {
        require(b > 0, errorMessage);
        uint256 c = a / b;
        // assert(a == b * c + a % b); // There is no case in which this doesn't hold

        return c;
    }

    /**
```

```
     * @dev Returns the remainder of dividing two unsigned integers. (unsigned integer modulo),
     * Reverts 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) {
        return mod(a, b, "SafeMath: modulo by zero");
    }

    /**
     * @dev Returns the remainder of dividing two unsigned integers. (unsigned integer modulo),
     * Reverts with custom message 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, string memory errorMessage) internal pure returns (uint256) {
        require(b != 0, errorMessage);
        return a % b;
    }
}


pragma solidity ^0.6.2;

/**
 * @dev Collection of functions related to the address type
 */
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
     */
    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.de
     *
     * 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 ret
        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
        return functionCallWithValue(target, data, value, "Address: low-level call with value failed"
    }

    /**
     * @dev Same as {xref-Address-functionCallWithValue-address-bytes-uint256-}[`functionCallWithValu
     * 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 er
        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 memor
        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) intern
        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);
    }

    function _verifyCallResult(bool success, bytes memory returndata, string memory errorMessage) pri
        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);
            }
        }
    }
}
```

```
pragma solidity ^0.6.0;

/**
 * @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, valu
    }

    /**
     * @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, newAllowan
    }

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

    /**
     * @dev Imitates a Solidity high-level call (i.e. a regular function call to a contract), relaxin
     * on the return value: the return value is optional (but if data is returned, it must not be fal
     * @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 m
        // we're implementing it ourselves. We use {Address.functionCall} to perform this call, which
        // the target address contains contract code and also asserts for success in the low-level ca

        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");
        }
    }
}


pragma solidity ^0.6.0;

/*
 * @dev Provides information about the current execution context, including the
 * sender of the transaction and its data. While these are generally available
 * via msg.sender and msg.data, they should not be accessed in such a direct
 * manner, since when dealing with GSN meta-transactions the account sending and
 * paying for execution may not be the actual sender (as far as an application
 * is concerned).
 *
 * This contract is only required for intermediate, library-like contracts.
 */
abstract contract Context {
    function _msgSender() internal view virtual returns (address payable) {
        return msg.sender;
    }

    function _msgData() internal view virtual returns (bytes memory) {
        this;
        // silence state mutability warning without generating bytecode - see https://github.com/ethe
        return msg.data;
    }
}


pragma solidity ^0.6.0;

/**
 * @dev Contract module which provides a basic access control mechanism, where
 * there is an account (an owner) that can be granted exclusive access to
 * specific functions.
 *
 * By default, the owner account will be the one that deploys the contract. This
 * can later be changed with {transferOwnership}.
 *
 * This module is used through inheritance. It will make available the modifier
 * `onlyOwner`, which can be applied to your functions to restrict their use to
 * the owner.
 */
contract Ownable is Context {
    address private _owner;

    event OwnershipTransferred(address indexed previousOwner, address indexed newOwner);

    /**
     * @dev Initializes the contract setting the deployer as the initial owner.
     */
    constructor () internal {
        address msgSender = _msgSender();
        _owner = msgSender;
        emit OwnershipTransferred(address(0), msgSender);
    }

    /**
     * @dev Returns the address of the current owner.
     */
    function owner() public view returns (address) {
        return _owner;
    }
```

```
    /**
     * @dev Throws if called by any account other than the owner.
     */
    modifier onlyOwner() {
        require(_owner == _msgSender(), "Ownable: caller is not the owner");
        _;
    }

    /**
     * @dev Leaves the contract without owner. It will not be possible to call
     * `onlyOwner` functions anymore. Can only be called by the current owner.
     *
     * NOTE: Renouncing ownership will leave the contract without an owner,
     * thereby removing any functionality that is only available to the owner.
     */
    function renounceOwnership() public virtual onlyOwner {
        emit OwnershipTransferred(_owner, address(0));
        _owner = address(0);
    }

    /**
     * @dev Transfers ownership of the contract to a new account (`newOwner`).
     * Can only be called by the current owner.
     */
    function transferOwnership(address newOwner) public virtual onlyOwner {
        require(newOwner != address(0), "Ownable: new owner is the zero address");
        emit OwnershipTransferred(_owner, newOwner);
        _owner = newOwner;
    }
}


pragma solidity ^0.6.0;

/**
 * @dev Library for managing
 * https://en.wikipedia.org/wiki/Set_(abstract_data_type)[sets] of primitive
 * types.
 *
 * Sets have the following properties:
 *
 * - Elements are added, removed, and checked for existence in constant time
 * (O(1)).
 * - Elements are enumerated in O(n). No guarantees are made on the ordering.
 *
 * ```
 * contract Example {
 *     // Add the library methods
 *     using EnumerableSet for EnumerableSet.AddressSet;
 *
 *     // Declare a set state variable
 *     EnumerableSet.AddressSet private mySet;
 * }
 * ```
 *
 * As of v3.0.0, only sets of type `address` (`AddressSet`) and `uint256`
 * (`UintSet`) are supported.
 */
library EnumerableSet {
    // To implement this library for multiple types with as little code
    // repetition as possible, we write it in terms of a generic Set type with
    // bytes32 values.
    // The Set implementation uses private functions, and user-facing
    // implementations (such as AddressSet) are just wrappers around the
    // underlying Set.
```

```
    // This means that we can only create new EnumerableSets for types that fit
    // in bytes32.

    struct Set {
        // Storage of set values
        bytes32[] _values;

        // Position of the value in the `values` array, plus 1 because index 0
        // means a value is not in the set.
        mapping(bytes32 => uint256) _indexes;
    }

    /**
     * @dev Add a value to a set. O(1).
     *
     * Returns true if the value was added to the set, that is if it was not
     * already present.
     */
    function _add(Set storage set, bytes32 value) private returns (bool) {
        if (!_contains(set, value)) {
            set._values.push(value);
            // The value is stored at length-1, but we add 1 to all indexes
            // and use 0 as a sentinel value
            set._indexes[value] = set._values.length;
            return true;
        } else {
            return false;
        }
    }

    /**
     * @dev Removes a value from a set. O(1).
     *
     * Returns true if the value was removed from the set, that is if it was
     * present.
     */
    function _remove(Set storage set, bytes32 value) private returns (bool) {
        // We read and store the value's index to prevent multiple reads from the same storage slot
        uint256 valueIndex = set._indexes[value];

        if (valueIndex != 0) { // Equivalent to contains(set, value)
            // To delete an element from the _values array in O(1), we swap the element to delete wit
            // the array, and then remove the last element (sometimes called as 'swap and pop').
            // This modifies the order of the array, as noted in {at}.

            uint256 toDeleteIndex = valueIndex - 1;
            uint256 lastIndex = set._values.length - 1;

            // When the value to delete is the last one, the swap operation is unnecessary. However,
            // so rarely, we still do the swap anyway to avoid the gas cost of adding an 'if' stateme

            bytes32 lastvalue = set._values[lastIndex];

            // Move the last value to the index where the value to delete is
            set._values[toDeleteIndex] = lastvalue;
            // Update the index for the moved value
            set._indexes[lastvalue] = toDeleteIndex + 1;
            // All indexes are 1-based

            // Delete the slot where the moved value was stored
            set._values.pop();

            // Delete the index for the deleted slot
            delete set._indexes[value];

            return true;
```

```
        } else {
            return false;
        }
    }

    /**
     * @dev Returns true if the value is in the set. O(1).
     */
    function _contains(Set storage set, bytes32 value) private view returns (bool) {
        return set._indexes[value] != 0;
    }

    /**
     * @dev Returns the number of values on the set. O(1).
     */
    function _length(Set storage set) private view returns (uint256) {
        return set._values.length;
    }

    /**
     * @dev Returns the value stored at position `index` in the set. O(1).
     *
     * Note that there are no guarantees on the ordering of values inside the
     * array, and it may change when more values are added or removed.
     *
     * Requirements:
     *
     * - `index` must be strictly less than {length}.
     */
    function _at(Set storage set, uint256 index) private view returns (bytes32) {
        require(set._values.length > index, "EnumerableSet: index out of bounds");
        return set._values[index];
    }

    // AddressSet

    struct AddressSet {
        Set _inner;
    }

    /**
     * @dev Add a value to a set. O(1).
     *
     * Returns true if the value was added to the set, that is if it was not
     * already present.
     */
    function add(AddressSet storage set, address value) internal returns (bool) {
        return _add(set._inner, bytes32(uint256(value)));
    }

    /**
     * @dev Removes a value from a set. O(1).
     *
     * Returns true if the value was removed from the set, that is if it was
     * present.
     */
    function remove(AddressSet storage set, address value) internal returns (bool) {
        return _remove(set._inner, bytes32(uint256(value)));
    }

    /**
     * @dev Returns true if the value is in the set. O(1).
     */
    function contains(AddressSet storage set, address value) internal view returns (bool) {
        return _contains(set._inner, bytes32(uint256(value)));
    }
```

```
    /**
     * @dev Returns the number of values in the set. O(1).
     */
    function length(AddressSet storage set) internal view returns (uint256) {
        return _length(set._inner);
    }

    /**
     * @dev Returns the value stored at position `index` in the set. O(1).
     *
     * Note that there are no guarantees on the ordering of values inside the
     * array, and it may change when more values are added or removed.
     *
     * Requirements:
     *
     * - `index` must be strictly less than {length}.
     */
    function at(AddressSet storage set, uint256 index) internal view returns (address) {
        return address(uint256(_at(set._inner, index)));
    }


    // UintSet

    struct UintSet {
        Set _inner;
    }

    /**
     * @dev Add a value to a set. O(1).
     *
     * Returns true if the value was added to the set, that is if it was not
     * already present.
     */
    function add(UintSet storage set, uint256 value) internal returns (bool) {
        return _add(set._inner, bytes32(value));
    }

    /**
     * @dev Removes a value from a set. O(1).
     *
     * Returns true if the value was removed from the set, that is if it was
     * present.
     */
    function remove(UintSet storage set, uint256 value) internal returns (bool) {
        return _remove(set._inner, bytes32(value));
    }

    /**
     * @dev Returns true if the value is in the set. O(1).
     */
    function contains(UintSet storage set, uint256 value) internal view returns (bool) {
        return _contains(set._inner, bytes32(value));
    }

    /**
     * @dev Returns the number of values on the set. O(1).
     */
    function length(UintSet storage set) internal view returns (uint256) {
        return _length(set._inner);
    }

    /**
     * @dev Returns the value stored at position `index` in the set. O(1).
     *
```

```
     * Note that there are no guarantees on the ordering of values inside the
     * array, and it may change when more values are added or removed.
     *
     * Requirements:
     *
     * - `index` must be strictly less than {length}.
     */
    function at(UintSet storage set, uint256 index) internal view returns (uint256) {
        return uint256(_at(set._inner, index));
    }
}

interface IDex is IERC20 {
    function mint(address to, uint256 amount) external returns (bool);
}

interface IMasterChef {
    function pending(uint256 pid, address user) external view returns (uint256);

    function deposit(uint256 pid, uint256 amount) external;

    function withdraw(uint256 pid, uint256 amount) external;

    function emergencyWithdraw(uint256 pid) external;
}

contract HecoPool is Ownable {
    using SafeMath for uint256;
    using SafeERC20 for IERC20;

    using EnumerableSet for EnumerableSet.AddressSet;
    EnumerableSet.AddressSet private _multLP;
    EnumerableSet.AddressSet private _blackList;

    // Info of each user.
    struct UserInfo {
        uint256 amount;     // How many LP tokens the user has provided.
        uint256 rewardDebt; // Reward debt.
        uint256 multLpRewardDebt; //multLp Reward debt.
    }

    // Info of each pool.
    struct PoolInfo {
        IERC20 lpToken;           // Address of LP token contract.
        uint256 allocPoint;       // How many allocation points assigned to this pool. DEXs to distri
        uint256 lastRewardBlock;  // Last block number that DEXs distribution occurs.
        uint256 accDexPerShare; // Accumulated DEXs per share, times 1e12.
        uint256 accMultLpPerShare; //Accumulated multLp per share
        uint256 totalAmount;    // Total amount of current pool deposit.
    }

    // The DEX Token!
    IDex public dex;
    // DEX tokens created per block.
    uint256 public dexPerBlock;
    // Info of each pool.
    PoolInfo[] public poolInfo;
    // Info of each user that stakes LP tokens.
    mapping(uint256 => mapping(address => UserInfo)) public userInfo;
    // Corresponding to the pid of the multLP pool
    mapping(uint256 => uint256) public poolCorrespond;
    // pid corresponding address
    mapping(address => uint256) public LpOfPid;
    // Control mining
    bool public paused = false;
    // Total allocation points. Must be the sum of all allocation points in all pools.
```

```
    uint256 public totalAllocPoint = 0;
    // The block number when DEX mining starts.
    uint256 public startBlock;
    // multLP MasterChef
    address public multLpChef;
    // multLP Token
    address public multLpToken;
    // How many blocks are halved
    uint256 public halvingPeriod = 1670400;

    event Deposit(address indexed user, uint256 indexed pid, uint256 amount);
    event Withdraw(address indexed user, uint256 indexed pid, uint256 amount);
    event EmergencyWithdraw(address indexed user, uint256 indexed pid, uint256 amount);

    constructor(
        IDex _dex,
        uint256 _dexPerBlock,
        uint256 _startBlock
    ) public {
        dex = _dex;
        dexPerBlock = _dexPerBlock;
        startBlock = _startBlock;
    }

    function setHalvingPeriod(uint256 _block) public onlyOwner {
        halvingPeriod = _block;
    }

    // Set the number of dex produced by each block
    function setDexPerBlock(uint256 newPerBlock) public onlyOwner {
        massUpdatePools();
        dexPerBlock = newPerBlock;
    }

    function setStartBlock(uint256 _startBlock) public onlyOwner {
        startBlock = _startBlock;
    }

    function poolLength() public view returns (uint256) {
        return poolInfo.length;
    }

    function addBadAddress(address _bad) public onlyOwner returns (bool) {
        require(_bad != address(0), "_bad is the zero address");
        return EnumerableSet.add(_blackList, _bad);
    }

    function delBadAddress(address _bad) public onlyOwner returns (bool) {
        require(_bad != address(0), "_bad is the zero address");
        return EnumerableSet.remove(_blackList, _bad);
    }

    function getBlackListLength() public view returns (uint256) {
        return EnumerableSet.length(_blackList);
    }

    function isBadAddress(address account) public view returns (bool) {
        return EnumerableSet.contains(_blackList, account);
    }

    function getBadAddress(uint256 _index) public view onlyOwner returns (address){
        require(_index <= getBlackListLength() - 1, "index out of bounds");
        return EnumerableSet.at(_blackList, _index);
    }

    function addMultLP(address _addLP) public onlyOwner returns (bool) {
```

```
        require(_addLP != address(0), "LP is the zero address");
        IERC20(_addLP).approve(multLpChef, uint256(- 1));
        return EnumerableSet.add(_multLP, _addLP);
    }

    function isMultLP(address _LP) public view returns (bool) {
        return EnumerableSet.contains(_multLP, _LP);
    }

    function getMultLPLength() public view returns (uint256) {
        return EnumerableSet.length(_multLP);
    }

    function getMultLPAddress(uint256 _pid) public view returns (address){
        require(_pid <= getMultLPLength() - 1, "not find this multLP");
        return EnumerableSet.at(_multLP, _pid);
    }

    function setPause() public onlyOwner {
        paused = !paused;
    }

    function setMultLP(address _multLpToken, address _multLpChef) public onlyOwner {
        require(_multLpToken != address(0) && _multLpChef != address(0), "is the zero address");
        multLpToken = _multLpToken;
        multLpChef = _multLpChef;
    }

    function replaceMultLP(address _multLpToken, address _multLpChef) public onlyOwner {
        require(_multLpToken != address(0) && _multLpChef != address(0), "is the zero address");
        require(paused == true, "No mining suspension");
        multLpToken = _multLpToken;
        multLpChef = _multLpChef;
        uint256 length = getMultLPLength();
        while (length > 0) {
            address dAddress = EnumerableSet.at(_multLP, 0);
            uint256 pid = LpOfPid[dAddress];
            IMasterChef(multLpChef).emergencyWithdraw(poolCorrespond[pid]);
            EnumerableSet.remove(_multLP, dAddress);
            length--;
        }
    }

    // Add a new lp to the pool. Can only be called by the owner.
    // XXX DO NOT add the same LP token more than once. Rewards will be messed up if you do.
    function add(uint256 _allocPoint, IERC20 _lpToken, bool _withUpdate) public onlyOwner {
        require(address(_lpToken) != address(0), "_lpToken is the zero address");
        if (_withUpdate) {
            massUpdatePools();
        }
        uint256 lastRewardBlock = block.number > startBlock ? block.number : startBlock;
        totalAllocPoint = totalAllocPoint.add(_allocPoint);
        poolInfo.push(PoolInfo({
        lpToken : _lpToken,
        allocPoint : _allocPoint,
        lastRewardBlock : lastRewardBlock,
        accDexPerShare : 0,
        accMultLpPerShare : 0,
        totalAmount : 0
        }));
        LpOfPid[address(_lpToken)] = poolLength() - 1;
    }

    // Update the given pool's DEX allocation point. Can only be called by the owner.
    function set(uint256 _pid, uint256 _allocPoint, bool _withUpdate) public onlyOwner {
        if (_withUpdate) {
```

```
            massUpdatePools();
        }
        totalAllocPoint = totalAllocPoint.sub(poolInfo[_pid].allocPoint).add(_allocPoint);
        poolInfo[_pid].allocPoint = _allocPoint;
    }

    // The current pool corresponds to the pid of the multLP pool
    function setPoolCorr(uint256 _pid, uint256 _sid) public onlyOwner {
        require(_pid <= poolLength() - 1, "not find this pool");
        poolCorrespond[_pid] = _sid;
    }

    function phase(uint256 blockNumber) public view returns (uint256) {
        if (halvingPeriod == 0) {
            return 0;
        }
        if (blockNumber > startBlock) {
            return (blockNumber.sub(startBlock).sub(1)).div(halvingPeriod);
        }
        return 0;
    }

    function reward(uint256 blockNumber) public view returns (uint256) {
        uint256 _phase = phase(blockNumber);
        return dexPerBlock.div(2 ** _phase);
    }

    function getDexBlockReward(uint256 _lastRewardBlock) public view returns (uint256) {
        uint256 blockReward = 0;
        uint256 n = phase(_lastRewardBlock);
        uint256 m = phase(block.number);
        while (n < m) {
            n++;
            uint256 r = n.mul(halvingPeriod).add(startBlock);
            blockReward = blockReward.add((r.sub(_lastRewardBlock)).mul(reward(r)));
            _lastRewardBlock = r;
        }
        blockReward = blockReward.add((block.number.sub(_lastRewardBlock)).mul(reward(block.number)))
        return blockReward;
    }

    // Update reward variables for all pools. Be careful of gas spending!
    function massUpdatePools() public {
        uint256 length = poolInfo.length;
        for (uint256 pid = 0; pid < length; ++pid) {
            updatePool(pid);
        }
    }

    // Update reward variables of the given pool to be up-to-date.
    function updatePool(uint256 _pid) public {
        PoolInfo storage pool = poolInfo[_pid];
        if (block.number <= pool.lastRewardBlock) {
            return;
        }
        uint256 lpSupply;
        if (isMultLP(address(pool.lpToken))) {
            if (pool.totalAmount == 0) {
                pool.lastRewardBlock = block.number;
                return;
            }
            lpSupply = pool.totalAmount;
        } else {
            lpSupply = pool.lpToken.balanceOf(address(this));
            if (lpSupply == 0) {
                pool.lastRewardBlock = block.number;
```

```
                return;
            }
        }
        uint256 blockReward = getDexBlockReward(pool.lastRewardBlock);
        if (blockReward <= 0) {
            return;
        }
        uint256 dexReward = blockReward.mul(pool.allocPoint).div(totalAllocPoint);
        bool minRet = dex.mint(address(this), dexReward);
        if (minRet) {
            pool.accDexPerShare = pool.accDexPerShare.add(dexReward.mul(1e12).div(lpSupply));
        }
        pool.lastRewardBlock = block.number;
    }

    // View function to see pending DEXs on frontend.
    function pending(uint256 _pid, address _user) external view returns (uint256, uint256){
        PoolInfo storage pool = poolInfo[_pid];
        if (isMultLP(address(pool.lpToken))) {
            (uint256 dexAmount, uint256 tokenAmount) = pendingDexAndToken(_pid, _user);
            return (dexAmount, tokenAmount);
        } else {
            uint256 dexAmount = pendingDex(_pid, _user);
            return (dexAmount, 0);
        }
    }

    function pendingDexAndToken(uint256 _pid, address _user) private view returns (uint256, uint256){
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][_user];
        uint256 accDexPerShare = pool.accDexPerShare;
        uint256 accMultLpPerShare = pool.accMultLpPerShare;
        if (user.amount > 0) {
            uint256 TokenPending = IMasterChef(multLpChef).pending(poolCorrespond[_pid], address(this
            accMultLpPerShare = accMultLpPerShare.add(TokenPending.mul(1e12).div(pool.totalAmount));
            uint256 userPending = user.amount.mul(accMultLpPerShare).div(1e12).sub(user.multLpRewardD
            if (block.number > pool.lastRewardBlock) {
                uint256 blockReward = getDexBlockReward(pool.lastRewardBlock);
                uint256 dexReward = blockReward.mul(pool.allocPoint).div(totalAllocPoint);
                accDexPerShare = accDexPerShare.add(dexReward.mul(1e12).div(pool.totalAmount));
                return (user.amount.mul(accDexPerShare).div(1e12).sub(user.rewardDebt), userPending);
            }
            if (block.number == pool.lastRewardBlock) {
                return (user.amount.mul(accDexPerShare).div(1e12).sub(user.rewardDebt), userPending);
            }
        }
        return (0, 0);
    }

    function pendingDex(uint256 _pid, address _user) private view returns (uint256){
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][_user];
        uint256 accDexPerShare = pool.accDexPerShare;
        uint256 lpSupply = pool.lpToken.balanceOf(address(this));
        if (user.amount > 0) {
            if (block.number > pool.lastRewardBlock) {
                uint256 blockReward = getDexBlockReward(pool.lastRewardBlock);
                uint256 dexReward = blockReward.mul(pool.allocPoint).div(totalAllocPoint);
                accDexPerShare = accDexPerShare.add(dexReward.mul(1e12).div(lpSupply));
                return user.amount.mul(accDexPerShare).div(1e12).sub(user.rewardDebt);
            }
            if (block.number == pool.lastRewardBlock) {
                return user.amount.mul(accDexPerShare).div(1e12).sub(user.rewardDebt);
            }
        }
        return 0;
```

```
    }

    // Deposit LP tokens to HecoPool for DEX allocation.
    function deposit(uint256 _pid, uint256 _amount) public notPause {
        require(!isBadAddress(msg.sender), 'Illegal, rejected ');
        PoolInfo storage pool = poolInfo[_pid];
        if (isMultLP(address(pool.lpToken))) {
            depositDexAndToken(_pid, _amount, msg.sender);
        } else {
            depositDex(_pid, _amount, msg.sender);
        }
    }

    function depositDexAndToken(uint256 _pid, uint256 _amount, address _user) private {
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][_user];
        updatePool(_pid);
        if (user.amount > 0) {
            uint256 pendingAmount = user.amount.mul(pool.accDexPerShare).div(1e12).sub(user.rewardDeb
            if (pendingAmount > 0) {
                safeDexTransfer(_user, pendingAmount);
            }
            uint256 beforeToken = IERC20(multLpToken).balanceOf(address(this));
            IMasterChef(multLpChef).deposit(poolCorrespond[_pid], 0);
            uint256 afterToken = IERC20(multLpToken).balanceOf(address(this));
            pool.accMultLpPerShare = pool.accMultLpPerShare.add(afterToken.sub(beforeToken).mul(1e12)
            uint256 tokenPending = user.amount.mul(pool.accMultLpPerShare).div(1e12).sub(user.multLpR
            if (tokenPending > 0) {
                IERC20(multLpToken).safeTransfer(_user, tokenPending);
            }
        }
        if (_amount > 0) {
            pool.lpToken.safeTransferFrom(_user, address(this), _amount);
            if (pool.totalAmount == 0) {
                IMasterChef(multLpChef).deposit(poolCorrespond[_pid], _amount);
                user.amount = user.amount.add(_amount);
                pool.totalAmount = pool.totalAmount.add(_amount);
            } else {
                uint256 beforeToken = IERC20(multLpToken).balanceOf(address(this));
                IMasterChef(multLpChef).deposit(poolCorrespond[_pid], _amount);
                uint256 afterToken = IERC20(multLpToken).balanceOf(address(this));
                pool.accMultLpPerShare = pool.accMultLpPerShare.add(afterToken.sub(beforeToken).mul(1
                user.amount = user.amount.add(_amount);
                pool.totalAmount = pool.totalAmount.add(_amount);
            }
        }
        user.rewardDebt = user.amount.mul(pool.accDexPerShare).div(1e12);
        user.multLpRewardDebt = user.amount.mul(pool.accMultLpPerShare).div(1e12);
        emit Deposit(_user, _pid, _amount);
    }

    function depositDex(uint256 _pid, uint256 _amount, address _user) private {
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][_user];
        updatePool(_pid);
        if (user.amount > 0) {
            uint256 pendingAmount = user.amount.mul(pool.accDexPerShare).div(1e12).sub(user.rewardDeb
            if (pendingAmount > 0) {
                safeDexTransfer(_user, pendingAmount);
            }
        }
        if (_amount > 0) {
            pool.lpToken.safeTransferFrom(_user, address(this), _amount);
            user.amount = user.amount.add(_amount);
            pool.totalAmount = pool.totalAmount.add(_amount);
        }
```

```
        user.rewardDebt = user.amount.mul(pool.accDexPerShare).div(1e12);
        emit Deposit(_user, _pid, _amount);
    }

    // Withdraw LP tokens from HecoPool.
    function withdraw(uint256 _pid, uint256 _amount) public notPause {
        PoolInfo storage pool = poolInfo[_pid];
        if (isMultLP(address(pool.lpToken))) {
            withdrawDexAndToken(_pid, _amount, msg.sender);
        } else {
            withdrawDex(_pid, _amount, msg.sender);
        }
    }

    function withdrawDexAndToken(uint256 _pid, uint256 _amount, address _user) private {
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][_user];
        require(user.amount >= _amount, "withdrawDexAndToken: not good");
        updatePool(_pid);
        uint256 pendingAmount = user.amount.mul(pool.accDexPerShare).div(1e12).sub(user.rewardDebt);
        if (pendingAmount > 0) {
            safeDexTransfer(_user, pendingAmount);
        }
        if (_amount > 0) {
            uint256 beforeToken = IERC20(multLpToken).balanceOf(address(this));
            IMasterChef(multLpChef).withdraw(poolCorrespond[_pid], _amount);
            uint256 afterToken = IERC20(multLpToken).balanceOf(address(this));
            pool.accMultLpPerShare = pool.accMultLpPerShare.add(afterToken.sub(beforeToken).mul(1e12)
            uint256 tokenPending = user.amount.mul(pool.accMultLpPerShare).div(1e12).sub(user.multLpR
            if (tokenPending > 0) {
                IERC20(multLpToken).safeTransfer(_user, tokenPending);
            }
            user.amount = user.amount.sub(_amount);
            pool.totalAmount = pool.totalAmount.sub(_amount);
            pool.lpToken.safeTransfer(_user, _amount);
        }
        user.rewardDebt = user.amount.mul(pool.accDexPerShare).div(1e12);
        user.multLpRewardDebt = user.amount.mul(pool.accMultLpPerShare).div(1e12);
        emit Withdraw(_user, _pid, _amount);
    }

    function withdrawDex(uint256 _pid, uint256 _amount, address _user) private {
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][_user];
        require(user.amount >= _amount, "withdrawDex: not good");
        updatePool(_pid);
        uint256 pendingAmount = user.amount.mul(pool.accDexPerShare).div(1e12).sub(user.rewardDebt);
        if (pendingAmount > 0) {
            safeDexTransfer(_user, pendingAmount);
        }
        if (_amount > 0) {
            user.amount = user.amount.sub(_amount);
            pool.totalAmount = pool.totalAmount.sub(_amount);
            pool.lpToken.safeTransfer(_user, _amount);
        }
        user.rewardDebt = user.amount.mul(pool.accDexPerShare).div(1e12);
        emit Withdraw(_user, _pid, _amount);
    }

    // Withdraw without caring about rewards. EMERGENCY ONLY.
    function emergencyWithdraw(uint256 _pid) public notPause {
        PoolInfo storage pool = poolInfo[_pid];
        if (isMultLP(address(pool.lpToken))) {
            emergencyWithdrawDexAndToken(_pid, msg.sender);
        } else {
            emergencyWithdrawDex(_pid, msg.sender);
```

```
        }
    }

    function emergencyWithdrawDexAndToken(uint256 _pid, address _user) private {
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][_user];
        uint256 amount = user.amount;
        uint256 beforeToken = IERC20(multLpToken).balanceOf(address(this));
        IMasterChef(multLpChef).withdraw(poolCorrespond[_pid], amount);
        uint256 afterToken = IERC20(multLpToken).balanceOf(address(this));
        pool.accMultLpPerShare = pool.accMultLpPerShare.add(afterToken.sub(beforeToken).mul(1e12).div
        user.amount = 0;
        user.rewardDebt = 0;
        pool.lpToken.safeTransfer(_user, amount);
        pool.totalAmount = pool.totalAmount.sub(amount);
        emit EmergencyWithdraw(_user, _pid, amount);
    }

    function emergencyWithdrawDex(uint256 _pid, address _user) private {
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][_user];
        uint256 amount = user.amount;
        user.amount = 0;
        user.rewardDebt = 0;
        pool.lpToken.safeTransfer(_user, amount);
        pool.totalAmount = pool.totalAmount.sub(amount);
        emit EmergencyWithdraw(_user, _pid, amount);
    }

    // Safe DEX transfer function, just in case if rounding error causes pool to not have enough DEXs
    function safeDexTransfer(address _to, uint256 _amount) internal {
        uint256 dexBal = dex.balanceOf(address(this));
        if (_amount > dexBal) {
            dex.transfer(_to, dexBal);
        } else {
            dex.transfer(_to, _amount);
        }
    }

    modifier notPause() {
        require(paused == false, "Mining has been suspended");
        _;
    }

    // addresses not allowed to be represented to harvest
    mapping(address => bool) public notRepresents;

    function represent(bool _allow) public {
        if (!_allow) {
            notRepresents[msg.sender] = true;
        } else if (notRepresents[msg.sender]) {
            delete notRepresents[msg.sender];
        }
    }

    function harvest() public {
        uint256 length = poolInfo.length;
        for (uint256 pid = 0; pid < length; ++pid) {
            deposit(pid, 0);
        }
    }

    function harvestOf(address account) public {
        require(!isBadAddress(account), 'Illegal, rejected ');
        require(!notRepresents[account], 'not allowed');
        uint256 length = poolInfo.length;
```

```
        for (uint256 pid = 0; pid < length; ++pid) {
            PoolInfo storage pool = poolInfo[pid];
            if (isMultLP(address(pool.lpToken))) {
                depositDexAndToken(pid, 0, account);
            } else {
                depositDex(pid, 0, account);
            }
        }
    }
}/**
 *Submitted for verification at hecoinfo.com on 2021-07-15
*/

/**
 *Submitted for verification at BscScan.com on 2021-04-08
*/

// SPDX-License-Identifier: MIT
pragma solidity ^0.6.0;


/*
 * @dev Provides information about the current execution context, including the
 * sender of the transaction and its data. While these are generally available
 * via msg.sender and msg.data, they should not be accessed in such a direct
 * manner, since when dealing with GSN meta-transactions the account sending and
 * paying for execution may not be the actual sender (as far as an application
 * is concerned).
 *
 * This contract is only required for intermediate, library-like contracts.
 */
abstract contract Context {
    function _msgSender() internal view virtual returns (address payable) {
        return msg.sender;
    }

    function _msgData() internal view virtual returns (bytes memory) {
        this;
        // silence state mutability warning without generating bytecode - see https://github.com/ethe
        return msg.data;
    }
}


/**
 * @dev Contract module which provides a basic access control mechanism, where
 * there is an account (an owner) that can be granted exclusive access to
 * specific functions.
 *
 * By default, the owner account will be the one that deploys the contract. This
 * can later be changed with {transferOwnership}.
 *
 * This module is used through inheritance. It will make available the modifier
 * `onlyOwner`, which can be applied to your functions to restrict their use to
 * the owner.
 */
contract Ownable is Context {
    address private _owner;

    event OwnershipTransferred(address indexed previousOwner, address indexed newOwner);

    /**
     * @dev Initializes the contract setting the deployer as the initial owner.
     */
    constructor () internal {
        address msgSender = _msgSender();
```

```
        _owner = msgSender;
        emit OwnershipTransferred(address(0), msgSender);
    }

    /**
     * @dev Returns the address of the current owner.
     */
    function owner() public view returns (address) {
        return _owner;
    }

    /**
     * @dev Throws if called by any account other than the owner.
     */
    modifier onlyOwner() {
        require(_owner == _msgSender(), "Ownable: caller is not the owner");
        _;
    }

    /**
     * @dev Leaves the contract without owner. It will not be possible to call
     * `onlyOwner` functions anymore. Can only be called by the current owner.
     *
     * NOTE: Renouncing ownership will leave the contract without an owner,
     * thereby removing any functionality that is only available to the owner.
     */
    function renounceOwnership() public virtual onlyOwner {
        emit OwnershipTransferred(_owner, address(0));
        _owner = address(0);
    }

    /**
     * @dev Transfers ownership of the contract to a new account (`newOwner`).
     * Can only be called by the current owner.
     */
    function transferOwnership(address newOwner) public virtual onlyOwner {
        require(newOwner != address(0), "Ownable: new owner is the zero address");
        emit OwnershipTransferred(_owner, newOwner);
        _owner = newOwner;
    }
}


/**
 * @dev Interface of the ERC20 standard as defined in the EIP.
 */
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);
}


/**
 * @dev Wrappers over Solidity's arithmetic operations with added overflow
 * checks.
 *
 * Arithmetic operations in Solidity wrap on overflow. This can easily result
 * in bugs, because programmers usually assume that an overflow raises an
 * error, which is the standard behavior in high level programming languages.
 * `SafeMath` restores this intuition by reverting the transaction when an
 * operation overflows.
 *
 * Using this library instead of the unchecked operations eliminates an entire
 * class of bugs, so it's recommended to use it always.
 */
library SafeMath {
```

```
    /**
     * @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) {
        return sub(a, b, "SafeMath: subtraction overflow");
    }

    /**
     * @dev Returns the subtraction of two unsigned integers, reverting with custom message on
     * overflow (when the result is negative).
     *
     * 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);
        uint256 c = a - b;

        return c;
    }

    /**
     * @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) {
        // 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 0;
        }
```

```
        uint256 c = a * b;
        require(c / a == b, "SafeMath: multiplication overflow");

        return c;
    }

    /**
     * @dev Returns the integer division of two unsigned integers. Reverts 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) {
        return div(a, b, "SafeMath: division by zero");
    }

    /**
     * @dev Returns the integer division of two unsigned integers. Reverts with custom message 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, string memory errorMessage) internal pure returns (uint256) {
        require(b > 0, errorMessage);
        uint256 c = a / b;
        // assert(a == b * c + a % b); // There is no case in which this doesn't hold

        return c;
    }

    /**
     * @dev Returns the remainder of dividing two unsigned integers. (unsigned integer modulo),
     * Reverts 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) {
        return mod(a, b, "SafeMath: modulo by zero");
    }

    /**
     * @dev Returns the remainder of dividing two unsigned integers. (unsigned integer modulo),
     * Reverts with custom message 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, string memory errorMessage) internal pure returns (uint256) {
        require(b != 0, errorMessage);
        return a % b;
    }
}


/**
 * @dev Collection of functions related to the address type
 */
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) {
        // According to EIP-1052, 0x0 is the value returned for not-yet created accounts
        // and 0xc5d2460186f7233c927e7db2dcc703c0e500b653ca82273b7bfad8045d85a470 is returned
        // for accounts without code, i.e. `keccak256('')`
        bytes32 codehash;
        bytes32 accountHash = 0xc5d2460186f7233c927e7db2dcc703c0e500b653ca82273b7bfad8045d85a470;
        // solhint-disable-next-line no-inline-assembly
        assembly {codehash := extcodehash(account)}
        return (codehash != accountHash && codehash != 0x0);
    }

    /**
     * @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
     */
    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.de
     *
     * 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 ret
        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
        return functionCallWithValue(target, data, value, "Address: low-level call with value failed"
    }

    /**
     * @dev Same as {xref-Address-functionCallWithValue-address-bytes-uint256-}[`functionCallWithValu
     * 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 er
        require(address(this).balance >= value, "Address: insufficient balance for call");
        return _functionCallWithValue(target, data, value, errorMessage);
    }

    function _functionCallWithValue(address target, bytes memory data, uint256 weiValue, string memor
        require(isContract(target), "Address: call to non-contract");

        // solhint-disable-next-line avoid-low-level-calls
        (bool success, bytes memory returndata) = target.call{value : weiValue}(data);
        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 Implementation of the {IERC20} interface.
 *
 * This implementation is agnostic to the way tokens are created. This means
 * that a supply mechanism has to be added in a derived contract using {_mint}.
 * For a generic mechanism see {ERC20PresetMinterPauser}.
 *
 * TIP: For a detailed writeup see our guide
 * https://forum.zeppelin.solutions/t/how-to-implement-erc20-supply-mechanisms/226[How
 * to implement supply mechanisms].
 *
 * We have followed general OpenZeppelin guidelines: functions revert instead
 * of returning `false` on failure. This behavior is nonetheless conventional
 * and does not conflict with the expectations of ERC20 applications.
 *
 * Additionally, an {Approval} event is emitted on calls to {transferFrom}.
 * This allows applications to reconstruct the allowance for all accounts just
 * by listening to said events. Other implementations of the EIP may not emit
 * these events, as it isn't required by the specification.
 *
 * Finally, the non-standard {decreaseAllowance} and {increaseAllowance}
 * functions have been added to mitigate the well-known issues around setting
 * allowances. See {IERC20-approve}.
 */
contract ERC20 is Context, IERC20 {
    using SafeMath for uint256;
    using Address for address;

    mapping(address => uint256) private _balances;

    mapping(address => mapping(address => uint256)) private _allowances;

    uint256 private _totalSupply;

    string private _name;
    string private _symbol;
    uint8 private _decimals;

    /**
     * @dev Sets the values for {name} and {symbol}, initializes {decimals} with
     * a default value of 18.
     *
     * To select a different value for {decimals}, use {_setupDecimals}.
     *
     * All three of these values are immutable: they can only be set once during
     * construction.
     */
    constructor (string memory name, string memory symbol) public {
```

```
        _name = name;
        _symbol = symbol;
        _decimals = 18;
    }

    /**
     * @dev Returns the name of the token.
     */
    function name() public view returns (string memory) {
        return _name;
    }

    /**
     * @dev Returns the symbol of the token, usually a shorter version of the
     * name.
     */
    function symbol() public view returns (string memory) {
        return _symbol;
    }

    /**
     * @dev Returns the number of decimals used to get its user representation.
     * For example, if `decimals` equals `2`, a balance of `505` tokens should
     * be displayed to a user as `5,05` (`505 / 10 ** 2`).
     *
     * Tokens usually opt for a value of 18, imitating the relationship between
     * Ether and Wei. This is the value {ERC20} uses, unless {_setupDecimals} is
     * called.
     *
     * NOTE: This information is only used for _display_ purposes: it in
     * no way affects any of the arithmetic of the contract, including
     * {IERC20-balanceOf} and {IERC20-transfer}.
     */
    function decimals() public view returns (uint8) {
        return _decimals;
    }

    /**
     * @dev See {IERC20-totalSupply}.
     */
    function totalSupply() public view override returns (uint256) {
        return _totalSupply;
    }

    /**
     * @dev See {IERC20-balanceOf}.
     */
    function balanceOf(address account) public view override returns (uint256) {
        return _balances[account];
    }

    /**
     * @dev See {IERC20-transfer}.
     *
     * Requirements:
     *
     * - `recipient` cannot be the zero address.
     * - the caller must have a balance of at least `amount`.
     */
    function transfer(address recipient, uint256 amount) public virtual override returns (bool) {
        _transfer(_msgSender(), recipient, amount);
        return true;
    }

    /**
     * @dev See {IERC20-allowance}.
```

```
     */
    function allowance(address owner, address spender) public view virtual override returns (uint256)
        return _allowances[owner][spender];
    }

    /**
     * @dev See {IERC20-approve}.
     *
     * Requirements:
     *
     * - `spender` cannot be the zero address.
     */
    function approve(address spender, uint256 amount) public virtual override returns (bool) {
        _approve(_msgSender(), spender, amount);
        return true;
    }

    /**
     * @dev See {IERC20-transferFrom}.
     *
     * Emits an {Approval} event indicating the updated allowance. This is not
     * required by the EIP. See the note at the beginning of {ERC20};
     *
     * Requirements:
     * - `sender` and `recipient` cannot be the zero address.
     * - `sender` must have a balance of at least `amount`.
     * - the caller must have allowance for ``sender``'s tokens of at least
     * `amount`.
     */
    function transferFrom(address sender, address recipient, uint256 amount) public virtual override
        _transfer(sender, recipient, amount);
        _approve(sender, _msgSender(), _allowances[sender][_msgSender()].sub(amount, "ERC20: transfer
        return true;
    }

    /**
     * @dev Atomically increases the allowance granted to `spender` by the caller.
     *
     * This is an alternative to {approve} that can be used as a mitigation for
     * problems described in {IERC20-approve}.
     *
     * Emits an {Approval} event indicating the updated allowance.
     *
     * Requirements:
     *
     * - `spender` cannot be the zero address.
     */
    function increaseAllowance(address spender, uint256 addedValue) public virtual returns (bool) {
        _approve(_msgSender(), spender, _allowances[_msgSender()][spender].add(addedValue));
        return true;
    }

    /**
     * @dev Atomically decreases the allowance granted to `spender` by the caller.
     *
     * This is an alternative to {approve} that can be used as a mitigation for
     * problems described in {IERC20-approve}.
     *
     * Emits an {Approval} event indicating the updated allowance.
     *
     * Requirements:
     *
     * - `spender` cannot be the zero address.
     * - `spender` must have allowance for the caller of at least
     * `subtractedValue`.
     */
```

```
    function decreaseAllowance(address spender, uint256 subtractedValue) public virtual returns (bool
        _approve(_msgSender(), spender, _allowances[_msgSender()][spender].sub(subtractedValue, "ERC2
        return true;
    }

    /**
     * @dev Moves tokens `amount` from `sender` to `recipient`.
     *
     * This is internal function is equivalent to {transfer}, and can be used to
     * e.g. implement automatic token fees, slashing mechanisms, etc.
     *
     * Emits a {Transfer} event.
     *
     * Requirements:
     *
     * - `sender` cannot be the zero address.
     * - `recipient` cannot be the zero address.
     * - `sender` must have a balance of at least `amount`.
     */
    function _transfer(address sender, address recipient, uint256 amount) internal virtual {
        require(sender != address(0), "ERC20: transfer from the zero address");
        require(recipient != address(0), "ERC20: transfer to the zero address");

        _beforeTokenTransfer(sender, recipient, amount);

        _balances[sender] = _balances[sender].sub(amount, "ERC20: transfer amount exceeds balance");
        _balances[recipient] = _balances[recipient].add(amount);
        emit Transfer(sender, recipient, amount);
    }

    /** @dev Creates `amount` tokens and assigns them to `account`, increasing
     * the total supply.
     *
     * Emits a {Transfer} event with `from` set to the zero address.
     *
     * Requirements
     *
     * - `to` cannot be the zero address.
     */
    function _mint(address account, uint256 amount) internal virtual {
        require(account != address(0), "ERC20: mint to the zero address");

        _beforeTokenTransfer(address(0), account, amount);

        _totalSupply = _totalSupply.add(amount);
        _balances[account] = _balances[account].add(amount);
        emit Transfer(address(0), account, amount);
    }

    /**
     * @dev Destroys `amount` tokens from `account`, reducing the
     * total supply.
     *
     * Emits a {Transfer} event with `to` set to the zero address.
     *
     * Requirements
     *
     * - `account` cannot be the zero address.
     * - `account` must have at least `amount` tokens.
     */
    function _burn(address account, uint256 amount) internal virtual {
        require(account != address(0), "ERC20: burn from the zero address");

        _beforeTokenTransfer(account, address(0), amount);

        _balances[account] = _balances[account].sub(amount, "ERC20: burn amount exceeds balance");
```

```
        _totalSupply = _totalSupply.sub(amount);
        emit Transfer(account, address(0), amount);
    }

    /**
     * @dev Sets `amount` as the allowance of `spender` over the `owner`s tokens.
     *
     * This is internal function is equivalent to `approve`, and can be used to
     * e.g. set automatic allowances for certain subsystems, etc.
     *
     * Emits an {Approval} event.
     *
     * Requirements:
     *
     * - `owner` cannot be the zero address.
     * - `spender` cannot be the zero address.
     */
    function _approve(address owner, address spender, uint256 amount) internal virtual {
        require(owner != address(0), "ERC20: approve from the zero address");
        require(spender != address(0), "ERC20: approve to the zero address");

        _allowances[owner][spender] = amount;
        emit Approval(owner, spender, amount);
    }

    /**
     * @dev Sets {decimals} to a value other than the default one of 18.
     *
     * WARNING: This function should only be called from the constructor. Most
     * applications that interact with token contracts will not expect
     * {decimals} to ever change, and may work incorrectly if it does.
     */
    function _setupDecimals(uint8 decimals_) internal {
        _decimals = decimals_;
    }

    /**
     * @dev Hook that is called before any transfer of tokens. This includes
     * minting and burning.
     *
     * Calling conditions:
     *
     * - when `from` and `to` are both non-zero, `amount` of ``from``'s tokens
     * will be to transferred to `to`.
     * - when `from` is zero, `amount` tokens will be minted for `to`.
     * - when `to` is zero, `amount` of ``from``'s tokens will be burned.
     * - `from` and `to` are never both zero.
     *
     * To learn more about hooks, head to xref:ROOT:extending-contracts.adoc#using-hooks[Using Hooks]
     */
    function _beforeTokenTransfer(address from, address to, uint256 amount) internal virtual {}
}


library EnumerableSet {
    // To implement this library for multiple types with as little code
    // repetition as possible, we write it in terms of a generic Set type with
    // bytes32 values.
    // The Set implementation uses private functions, and user-facing
    // implementations (such as AddressSet) are just wrappers around the
    // underlying Set.
    // This means that we can only create new EnumerableSets for types that fit
    // in bytes32.

    struct Set {
        // Storage of set values
```

```
        bytes32[] _values;

        // Position of the value in the `values` array, plus 1 because index 0
        // means a value is not in the set.
        mapping(bytes32 => uint256) _indexes;
    }

    /**
     * @dev Add a value to a set. O(1).
     *
     * Returns true if the value was added to the set, that is if it was not
     * already present.
     */
    function _add(Set storage set, bytes32 value) private returns (bool) {
        if (!_contains(set, value)) {
            set._values.push(value);
            // The value is stored at length-1, but we add 1 to all indexes
            // and use 0 as a sentinel value
            set._indexes[value] = set._values.length;
            return true;
        } else {
            return false;
        }
    }

    /**
     * @dev Removes a value from a set. O(1).
     *
     * Returns true if the value was removed from the set, that is if it was
     * present.
     */
    function _remove(Set storage set, bytes32 value) private returns (bool) {
        // We read and store the value's index to prevent multiple reads from the same storage slot
        uint256 valueIndex = set._indexes[value];

        if (valueIndex != 0) {// Equivalent to contains(set, value)
            // To delete an element from the _values array in O(1), we swap the element to delete wit
            // the array, and then remove the last element (sometimes called as 'swap and pop').
            // This modifies the order of the array, as noted in {at}.

            uint256 toDeleteIndex = valueIndex - 1;
            uint256 lastIndex = set._values.length - 1;

            // When the value to delete is the last one, the swap operation is unnecessary. However,
            // so rarely, we still do the swap anyway to avoid the gas cost of adding an 'if' stateme

            bytes32 lastvalue = set._values[lastIndex];

            // Move the last value to the index where the value to delete is
            set._values[toDeleteIndex] = lastvalue;
            // Update the index for the moved value
            set._indexes[lastvalue] = toDeleteIndex + 1;
            // All indexes are 1-based

            // Delete the slot where the moved value was stored
            set._values.pop();

            // Delete the index for the deleted slot
            delete set._indexes[value];

            return true;
        } else {
            return false;
        }
    }
```

```
    /**
     * @dev Returns true if the value is in the set. O(1).
     */
    function _contains(Set storage set, bytes32 value) private view returns (bool) {
        return set._indexes[value] != 0;
    }

    /**
     * @dev Returns the number of values on the set. O(1).
     */
    function _length(Set storage set) private view returns (uint256) {
        return set._values.length;
    }

    /**
     * @dev Returns the value stored at position `index` in the set. O(1).
     *
     * Note that there are no guarantees on the ordering of values inside the
     * array, and it may change when more values are added or removed.
     *
     * Requirements:
     *
     * - `index` must be strictly less than {length}.
     */
    function _at(Set storage set, uint256 index) private view returns (bytes32) {
        require(set._values.length > index, "EnumerableSet: index out of bounds");
        return set._values[index];
    }

    // Bytes32Set

    struct Bytes32Set {
        Set _inner;
    }

    /**
     * @dev Add a value to a set. O(1).
     *
     * Returns true if the value was added to the set, that is if it was not
     * already present.
     */
    function add(Bytes32Set storage set, bytes32 value) internal returns (bool) {
        return _add(set._inner, value);
    }

    /**
     * @dev Removes a value from a set. O(1).
     *
     * Returns true if the value was removed from the set, that is if it was
     * present.
     */
    function remove(Bytes32Set storage set, bytes32 value) internal returns (bool) {
        return _remove(set._inner, value);
    }

    /**
     * @dev Returns true if the value is in the set. O(1).
     */
    function contains(Bytes32Set storage set, bytes32 value) internal view returns (bool) {
        return _contains(set._inner, value);
    }

    /**
     * @dev Returns the number of values in the set. O(1).
     */
    function length(Bytes32Set storage set) internal view returns (uint256) {
```

```
        return _length(set._inner);
    }

    /**
     * @dev Returns the value stored at position `index` in the set. O(1).
     *
     * Note that there are no guarantees on the ordering of values inside the
     * array, and it may change when more values are added or removed.
     *
     * Requirements:
     *
     * - `index` must be strictly less than {length}.
     */
    function at(Bytes32Set storage set, uint256 index) internal view returns (bytes32) {
        return _at(set._inner, index);
    }

    // AddressSet

    struct AddressSet {
        Set _inner;
    }

    /**
     * @dev Add a value to a set. O(1).
     *
     * Returns true if the value was added to the set, that is if it was not
     * already present.
     */
    function add(AddressSet storage set, address value) internal returns (bool) {
        return _add(set._inner, bytes32(uint256(value)));
    }

    /**
     * @dev Removes a value from a set. O(1).
     *
     * Returns true if the value was removed from the set, that is if it was
     * present.
     */
    function remove(AddressSet storage set, address value) internal returns (bool) {
        return _remove(set._inner, bytes32(uint256(value)));
    }

    /**
     * @dev Returns true if the value is in the set. O(1).
     */
    function contains(AddressSet storage set, address value) internal view returns (bool) {
        return _contains(set._inner, bytes32(uint256(value)));
    }

    /**
     * @dev Returns the number of values in the set. O(1).
     */
    function length(AddressSet storage set) internal view returns (uint256) {
        return _length(set._inner);
    }

    /**
     * @dev Returns the value stored at position `index` in the set. O(1).
     *
     * Note that there are no guarantees on the ordering of values inside the
     * array, and it may change when more values are added or removed.
     *
     * Requirements:
     *
     * - `index` must be strictly less than {length}.
```

```
     */
    function at(AddressSet storage set, uint256 index) internal view returns (address) {
        return address(uint256(_at(set._inner, index)));
    }


    // UintSet

    struct UintSet {
        Set _inner;
    }

    /**
     * @dev Add a value to a set. O(1).
     *
     * Returns true if the value was added to the set, that is if it was not
     * already present.
     */
    function add(UintSet storage set, uint256 value) internal returns (bool) {
        return _add(set._inner, bytes32(value));
    }

    /**
     * @dev Removes a value from a set. O(1).
     *
     * Returns true if the value was removed from the set, that is if it was
     * present.
     */
    function remove(UintSet storage set, uint256 value) internal returns (bool) {
        return _remove(set._inner, bytes32(value));
    }

    /**
     * @dev Returns true if the value is in the set. O(1).
     */
    function contains(UintSet storage set, uint256 value) internal view returns (bool) {
        return _contains(set._inner, bytes32(value));
    }

    /**
     * @dev Returns the number of values on the set. O(1).
     */
    function length(UintSet storage set) internal view returns (uint256) {
        return _length(set._inner);
    }

    /**
     * @dev Returns the value stored at position `index` in the set. O(1).
     *
     * Note that there are no guarantees on the ordering of values inside the
     * array, and it may change when more values are added or removed.
     *
     * Requirements:
     *
     * - `index` must be strictly less than {length}.
     */
    function at(UintSet storage set, uint256 index) internal view returns (uint256) {
        return uint256(_at(set._inner, index));
    }
}

pragma solidity ^0.6.0;
pragma experimental ABIEncoderV2;

abstract contract DelegateERC20 is ERC20 {
    // @notice A record of each accounts delegate
```

```
    mapping(address => address) internal _delegates;

    /// @notice A checkpoint for marking number of votes from a given block
    struct Checkpoint {
        uint32 fromBlock;
        uint256 votes;
    }

    /// @notice A record of votes checkpoints for each account, by index
    mapping(address => mapping(uint32 => Checkpoint)) public checkpoints;

    /// @notice The number of checkpoints for each account
    mapping(address => uint32) public numCheckpoints;

    /// @notice The EIP-712 typehash for the contract's domain
    bytes32 public constant DOMAIN_TYPEHASH = keccak256("EIP712Domain(string name,uint256 chainId,add

    /// @notice The EIP-712 typehash for the delegation struct used by the contract
    bytes32 public constant DELEGATION_TYPEHASH = keccak256("Delegation(address delegatee,uint256 non

    /// @notice A record of states for signing / validating signatures
    mapping(address => uint) public nonces;


    // support delegates mint
    function _mint(address account, uint256 amount) internal override virtual {
        super._mint(account, amount);

        // add delegates to the minter
        _moveDelegates(address(0), _delegates[account], amount);
    }


    function _transfer(address sender, address recipient, uint256 amount) internal override virtual {
        super._transfer(sender, recipient, amount);
        _moveDelegates(_delegates[sender], _delegates[recipient], amount);
    }


    /**
    * @notice Delegate votes from `msg.sender` to `delegatee`
    * @param delegatee The address to delegate votes to
    */
    function delegate(address delegatee) external {
        return _delegate(msg.sender, delegatee);
    }

    /**
     * @notice Delegates votes from signatory to `delegatee`
     * @param delegatee The address to delegate votes to
     * @param nonce The contract state required to match the signature
     * @param expiry The time at which to expire the signature
     * @param v The recovery byte of the signature
     * @param r Half of the ECDSA signature pair
     * @param s Half of the ECDSA signature pair
     */
    function delegateBySig(
        address delegatee,
        uint nonce,
        uint expiry,
        uint8 v,
        bytes32 r,
        bytes32 s
    )
    external
    {
```

```
        bytes32 domainSeparator = keccak256(
            abi.encode(
                DOMAIN_TYPEHASH,
                keccak256(bytes(name())),
                getChainId(),
                address(this)
            )
        );

        bytes32 structHash = keccak256(
            abi.encode(
                DELEGATION_TYPEHASH,
                delegatee,
                nonce,
                expiry
            )
        );

        bytes32 digest = keccak256(
            abi.encodePacked(
                "\x19\x01",
                domainSeparator,
                structHash
            )
        );

        address signatory = ecrecover(digest, v, r, s);
        require(signatory != address(0), "DexToken::delegateBySig: invalid signature");
        require(nonce == nonces[signatory]++, "DexToken::delegateBySig: invalid nonce");
        require(now <= expiry, "DexToken::delegateBySig: signature expired");
        return _delegate(signatory, delegatee);
    }

    /**
     * @notice Gets the current votes balance for `account`
     * @param account The address to get votes balance
     * @return The number of current votes for `account`
     */
    function getCurrentVotes(address account)
    external
    view
    returns (uint256)
    {
        uint32 nCheckpoints = numCheckpoints[account];
        return nCheckpoints > 0 ? checkpoints[account][nCheckpoints - 1].votes : 0;
    }

    /**
     * @notice Determine the prior number of votes for an account as of a block number
     * @dev Block number must be a finalized block or else this function will revert to prevent misin
     * @param account The address of the account to check
     * @param blockNumber The block number to get the vote balance at
     * @return The number of votes the account had as of the given block
     */
    function getPriorVotes(address account, uint blockNumber)
    external
    view
    returns (uint256)
    {
        require(blockNumber < block.number, "DexToken::getPriorVotes: not yet determined");

        uint32 nCheckpoints = numCheckpoints[account];
        if (nCheckpoints == 0) {
            return 0;
        }
```

```
        // First check most recent balance
        if (checkpoints[account][nCheckpoints - 1].fromBlock <= blockNumber) {
            return checkpoints[account][nCheckpoints - 1].votes;
        }

        // Next check implicit zero balance
        if (checkpoints[account][0].fromBlock > blockNumber) {
            return 0;
        }

        uint32 lower = 0;
        uint32 upper = nCheckpoints - 1;
        while (upper > lower) {
            uint32 center = upper - (upper - lower) / 2;
            // ceil, avoiding overflow
            Checkpoint memory cp = checkpoints[account][center];
            if (cp.fromBlock == blockNumber) {
                return cp.votes;
            } else if (cp.fromBlock < blockNumber) {
                lower = center;
            } else {
                upper = center - 1;
            }
        }
        return checkpoints[account][lower].votes;
    }

    function _delegate(address delegator, address delegatee)
    internal
    {
        address currentDelegate = _delegates[delegator];
        uint256 delegatorBalance = balanceOf(delegator);
        // balance of underlying balances (not scaled);
        _delegates[delegator] = delegatee;

        _moveDelegates(currentDelegate, delegatee, delegatorBalance);

        emit DelegateChanged(delegator, currentDelegate, delegatee);
    }

    function _moveDelegates(address srcRep, address dstRep, uint256 amount) internal {
        if (srcRep != dstRep && amount > 0) {
            if (srcRep != address(0)) {
                // decrease old representative
                uint32 srcRepNum = numCheckpoints[srcRep];
                uint256 srcRepOld = srcRepNum > 0 ? checkpoints[srcRep][srcRepNum - 1].votes : 0;
                uint256 srcRepNew = srcRepOld.sub(amount);
                _writeCheckpoint(srcRep, srcRepNum, srcRepOld, srcRepNew);
            }

            if (dstRep != address(0)) {
                // increase new representative
                uint32 dstRepNum = numCheckpoints[dstRep];
                uint256 dstRepOld = dstRepNum > 0 ? checkpoints[dstRep][dstRepNum - 1].votes : 0;
                uint256 dstRepNew = dstRepOld.add(amount);
                _writeCheckpoint(dstRep, dstRepNum, dstRepOld, dstRepNew);
            }
        }
    }

    function _writeCheckpoint(
        address delegatee,
        uint32 nCheckpoints,
        uint256 oldVotes,
        uint256 newVotes
    )
```

```
    internal
    {
        uint32 blockNumber = safe32(block.number, "DexToken::_writeCheckpoint: block number exceeds 3

        if (nCheckpoints > 0 && checkpoints[delegatee][nCheckpoints - 1].fromBlock == blockNumber) {
            checkpoints[delegatee][nCheckpoints - 1].votes = newVotes;
        } else {
            checkpoints[delegatee][nCheckpoints] = Checkpoint(blockNumber, newVotes);
            numCheckpoints[delegatee] = nCheckpoints + 1;
        }

        emit DelegateVotesChanged(delegatee, oldVotes, newVotes);
    }

    function safe32(uint n, string memory errorMessage) internal pure returns (uint32) {
        require(n < 2 ** 32, errorMessage);
        return uint32(n);
    }

    function getChainId() internal pure returns (uint) {
        uint256 chainId;
        assembly {chainId := chainid()}

        return chainId;
    }

    /// @notice An event thats emitted when an account changes its delegate
    event DelegateChanged(address indexed delegator, address indexed fromDelegate, address indexed to

    /// @notice An event thats emitted when a delegate account's vote balance changes
    event DelegateVotesChanged(address indexed delegate, uint previousBalance, uint newBalance);

}

contract COCOToken is DelegateERC20, Ownable {
    uint256 private constant preMineSupply = 200000000 * 1e18;
    uint256 private constant maxSupply = 1000000000 * 1e18;     // the total supply

    using EnumerableSet for EnumerableSet.AddressSet;
    EnumerableSet.AddressSet private _minters;

    constructor() public ERC20("COCO Token", "COCO"){
        _mint(msg.sender, preMineSupply);
    }

    // mint with max supply
    function mint(address _to, uint256 _amount) public onlyMinter returns (bool) {
        if (_amount.add(totalSupply()) > maxSupply) {
            return false;
        }
        _mint(_to, _amount);
        return true;
    }

    function addMinter(address _addMinter) public onlyOwner returns (bool) {
        require(_addMinter != address(0), "DexToken: _addMinter is the zero address");
        return EnumerableSet.add(_minters, _addMinter);
    }

    function delMinter(address _delMinter) public onlyOwner returns (bool) {
        require(_delMinter != address(0), "DexToken: _delMinter is the zero address");
        return EnumerableSet.remove(_minters, _delMinter);
    }

    function getMinterLength() public view returns (uint256) {
        return EnumerableSet.length(_minters);
```

```
    }

    function isMinter(address account) public view returns (bool) {
        return EnumerableSet.contains(_minters, account);
    }

    function getMinter(uint256 _index) public view onlyOwner returns (address){
        require(_index <= getMinterLength() - 1, "DexToken: index out of bounds");
        return EnumerableSet.at(_minters, _index);
    }

    // modifier for mint function
    modifier onlyMinter() {
        require(isMinter(msg.sender), "caller is not the minter");
        _;
    }
}/**
 *Submitted for verification at hecoinfo.com on 2021-07-17
*/

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

pragma solidity >=0.4.0;

/*
 * @dev Provides information about the current execution context, including the
 * sender of the transaction and its data. While these are generally available
 * via msg.sender and msg.data, they should not be accessed in such a direct
 * manner, since when dealing with GSN meta-transactions the account sending and
 * paying for execution may not be the actual sender (as far as an application
 * is concerned).
 *
 * This contract is only required for intermediate, library-like contracts.
 */
contract Context {
    // Empty internal constructor, to prevent people from mistakenly deploying
    // an instance of this contract, which should be used via inheritance.
    constructor() internal {}

    function _msgSender() internal view returns (address payable) {
        return msg.sender;
    }

    function _msgData() internal view returns (bytes memory) {
        this; // silence state mutability warning without generating bytecode - see https://github.co
        return msg.data;
    }
}

pragma solidity >=0.4.0;

/**
 * @dev Contract module which provides a basic access control mechanism, where
 * there is an account (an owner) that can be granted exclusive access to
 * specific functions.
 *
 * By default, the owner account will be the one that deploys the contract. This
 * can later be changed with {transferOwnership}.
 *
 * This module is used through inheritance. It will make available the modifier
 * `onlyOwner`, which can be applied to your functions to restrict their use to
 * the owner.
 */
contract Ownable is Context {
    address private _owner;
```

```
    event OwnershipTransferred(address indexed previousOwner, address indexed newOwner);

    /**
     * @dev Initializes the contract setting the deployer as the initial owner.
     */
    constructor() internal {
        address msgSender = _msgSender();
        _owner = msgSender;
        emit OwnershipTransferred(address(0), msgSender);
    }

    /**
     * @dev Returns the address of the current owner.
     */
    function owner() public view returns (address) {
        return _owner;
    }

    /**
     * @dev Throws if called by any account other than the owner.
     */
    modifier onlyOwner() {
        require(_owner == _msgSender(), 'Ownable: caller is not the owner');
        _;
    }

    /**
     * @dev Leaves the contract without owner. It will not be possible to call
     * `onlyOwner` functions anymore. Can only be called by the current owner.
     *
     * NOTE: Renouncing ownership will leave the contract without an owner,
     * thereby removing any functionality that is only available to the owner.
     */
    function renounceOwnership() public onlyOwner {
        emit OwnershipTransferred(_owner, address(0));
        _owner = address(0);
    }

    /**
     * @dev Transfers ownership of the contract to a new account (`newOwner`).
     * Can only be called by the current owner.
     */
    function transferOwnership(address newOwner) public onlyOwner {
        _transferOwnership(newOwner);
    }

    /**
     * @dev Transfers ownership of the contract to a new account (`newOwner`).
     */
    function _transferOwnership(address newOwner) internal {
        require(newOwner != address(0), 'Ownable: new owner is the zero address');
        emit OwnershipTransferred(_owner, newOwner);
        _owner = newOwner;
    }
}

pragma solidity >=0.4.0;

/**
 * @dev Wrappers over Solidity's arithmetic operations with added overflow
 * checks.
 *
 * Arithmetic operations in Solidity wrap on overflow. This can easily result
 * in bugs, because programmers usually assume that an overflow raises an
 * error, which is the standard behavior in high level programming languages.
 * `SafeMath` restores this intuition by reverting the transaction when an
```

```
 * operation overflows.
 *
 * Using this library instead of the unchecked operations eliminates an entire
 * class of bugs, so it's recommended to use it always.
 */
library SafeMath {
    /**
     * @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) {
        return sub(a, b, 'SafeMath: subtraction overflow');
    }

    /**
     * @dev Returns the subtraction of two unsigned integers, reverting with custom message on
     * overflow (when the result is negative).
     *
     * 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);
        uint256 c = a - b;

        return c;
    }

    /**
     * @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) {
        // 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 0;
        }

        uint256 c = a * b;
        require(c / a == b, 'SafeMath: multiplication overflow');

        return c;
    }

    /**
     * @dev Returns the integer division of two unsigned integers. Reverts 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) {
        return div(a, b, 'SafeMath: division by zero');
    }

    /**
     * @dev Returns the integer division of two unsigned integers. Reverts with custom message 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,
        string memory errorMessage
    ) internal pure returns (uint256) {
        require(b > 0, errorMessage);
        uint256 c = a / b;
        // assert(a == b * c + a % b); // There is no case in which this doesn't hold

        return c;
    }

    /**
     * @dev Returns the remainder of dividing two unsigned integers. (unsigned integer modulo),
     * Reverts 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) {
        return mod(a, b, 'SafeMath: modulo by zero');
    }

    /**
     * @dev Returns the remainder of dividing two unsigned integers. (unsigned integer modulo),
     * Reverts with custom message 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,
        string memory errorMessage
    ) internal pure returns (uint256) {
        require(b != 0, errorMessage);
        return a % b;
    }

    function min(uint256 x, uint256 y) internal pure returns (uint256 z) {
        z = x < y ? x : y;
    }

    // babylonian method (https://en.wikipedia.org/wiki/Methods_of_computing_square_roots#Babylonian_
    function sqrt(uint256 y) internal pure returns (uint256 z) {
        if (y > 3) {
            z = y;
            uint256 x = y / 2 + 1;
            while (x < z) {
                z = x;
                x = (y / x + x) / 2;
            }
        } else if (y != 0) {
            z = 1;
        }
    }
}

pragma solidity >=0.4.0;

interface IHRC20 {
    /**
     * @dev Returns the amount of tokens in existence.
     */
    function totalSupply() external view returns (uint256);

    /**
     * @dev Returns the token decimals.
     */
    function decimals() external view returns (uint8);

    /**
     * @dev Returns the token symbol.
     */
    function symbol() external view returns (string memory);

    /**
     * @dev Returns the token name.
```

```
     */
    function name() external view returns (string memory);

    /**
     * @dev Returns the bep token owner.
     */
    function getOwner() external view returns (address);

    /**
     * @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);
}

pragma solidity ^0.6.2;

/**
 * @dev Collection of functions related to the address type
 */
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) {
        // According to EIP-1052, 0x0 is the value returned for not-yet created accounts
        // and 0xc5d2460186f7233c927e7db2dcc703c0e500b653ca82273b7bfad8045d85a470 is returned
        // for accounts without code, i.e. `keccak256('')`
        bytes32 codehash;
        bytes32 accountHash = 0xc5d2460186f7233c927e7db2dcc703c0e500b653ca82273b7bfad8045d85a470;
        // solhint-disable-next-line no-inline-assembly
        assembly {
            codehash := extcodehash(account)
        }
        return (codehash != accountHash && codehash != 0x0);
    }

    /**
     * @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
     */
    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.de
     *
     * 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-}[`functionCallWithValu
     * 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');
        return _functionCallWithValue(target, data, value, errorMessage);
    }

    function _functionCallWithValue(
        address target,
        bytes memory data,
        uint256 weiValue,
        string memory errorMessage
    ) private returns (bytes memory) {
        require(isContract(target), 'Address: call to non-contract');

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

pragma solidity ^0.6.0;

/**
 * @title SafeHRC20
 * @dev Wrappers around HRC20 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 SafeHRC20 for IHRC20;` statement to your contract,
 * which allows you to call the safe operations as `token.safeTransfer(...)`, etc.
 */
library SafeHRC20 {
    using SafeMath for uint256;
    using Address for address;

    function safeTransfer(
        IHRC20 token,
        address to,
        uint256 value
    ) internal {
        _callOptionalReturn(token, abi.encodeWithSelector(token.transfer.selector, to, value));
    }

    function safeTransferFrom(
        IHRC20 token,
        address from,
        address to,
```

```
        uint256 value
    ) internal {
        _callOptionalReturn(token, abi.encodeWithSelector(token.transferFrom.selector, from, to, valu
    }

    /**
     * @dev Deprecated. This function has issues similar to the ones found in
     * {IHRC20-approve}, and its usage is discouraged.
     *
     * Whenever possible, use {safeIncreaseAllowance} and
     * {safeDecreaseAllowance} instead.
     */
    function safeApprove(
        IHRC20 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),
            'SafeHRC20: approve from non-zero to non-zero allowance'
        );
        _callOptionalReturn(token, abi.encodeWithSelector(token.approve.selector, spender, value));
    }

    function safeIncreaseAllowance(
        IHRC20 token,
        address spender,
        uint256 value
    ) internal {
        uint256 newAllowance = token.allowance(address(this), spender).add(value);
        _callOptionalReturn(token, abi.encodeWithSelector(token.approve.selector, spender, newAllowan
    }

    function safeDecreaseAllowance(
        IHRC20 token,
        address spender,
        uint256 value
    ) internal {
        uint256 newAllowance = token.allowance(address(this), spender).sub(
            value,
            'SafeHRC20: decreased allowance below zero'
        );
        _callOptionalReturn(token, abi.encodeWithSelector(token.approve.selector, spender, newAllowan
    }

    /**
     * @dev Imitates a Solidity high-level call (i.e. a regular function call to a contract), relaxin
     * on the return value: the return value is optional (but if data is returned, it must not be fal
     * @param token The token targeted by the call.
     * @param data The call data (encoded using abi.encode or one of its variants).
     */
    function _callOptionalReturn(IHRC20 token, bytes memory data) private {
        // We need to perform a low level call here, to bypass Solidity's return data size checking m
        // we're implementing it ourselves. We use {Address.functionCall} to perform this call, which
        // the target address contains contract code and also asserts for success in the low-level ca

        bytes memory returndata = address(token).functionCall(data, 'SafeHRC20: low-level call failed
        if (returndata.length > 0) {
            // Return data is optional
            // solhint-disable-next-line max-line-length
            require(abi.decode(returndata, (bool)), 'SafeHRC20: HRC20 operation did not succeed');
        }
```

```
    }
}

pragma solidity 0.6.12;

contract xTokenPool is Ownable {
    using SafeMath for uint256;
    using SafeHRC20 for IHRC20;

    // Info of each user.
    struct UserInfo {
        uint256 amount;     // How many LP tokens the user has provided.
        uint256 rewardDebt; // Reward debt. See explanation below.
    }

    // Info of each pool.
    struct PoolInfo {
        IHRC20 lpToken;           // Address of LP token contract.
        uint256 allocPoint;       // How many allocation points assigned to this pool. reward tokens
        uint256 lastRewardBlock;  // Last block number that reward tokens distribution occurs.
        uint256 accRewardPerShare; // Accumulated reward tokens per share, times 1e12. See below.
    }

    // The x TOKEN!
    IHRC20 public xToken;
    IHRC20 public rewardToken;

    // uint256 public maxStaking;

    // reward tokens created per block.
    uint256 public rewardPerBlock;

    // Info of each pool.
    PoolInfo[] public poolInfo;
    // Info of each user that stakes LP tokens.
    mapping (address => UserInfo) public userInfo;
    // Total allocation poitns. Must be the sum of all allocation points in all pools.
    uint256 private totalAllocPoint = 0;
    // The block number when reward token mining starts.
    uint256 public startBlock;
    // The block number when reward token mining ends.
    uint256 public bonusEndBlock;

    event Deposit(address indexed user, uint256 amount);
    event Withdraw(address indexed user, uint256 amount);
    event EmergencyWithdraw(address indexed user, uint256 amount);

    constructor(
        IHRC20 _xToken,
        IHRC20 _rewardToken,
        uint256 _rewardPerBlock,
        uint256 _startBlock,
        uint256 _bonusEndBlock
    ) public {
        xToken = _xToken;
        rewardToken = _rewardToken;
        rewardPerBlock = _rewardPerBlock;
        startBlock = _startBlock;
        bonusEndBlock = _bonusEndBlock;

        // staking pool
        poolInfo.push(PoolInfo({
            lpToken: _xToken,
            allocPoint: 1000,
            lastRewardBlock: startBlock,
            accRewardPerShare: 0
```

```
        }));

        totalAllocPoint = 1000;
        // maxStaking = 5000000000000000000;

    }

    function stopReward() public onlyOwner {
        bonusEndBlock = block.number;
    }


    // Return reward multiplier over the given _from to _to block.
    function getMultiplier(uint256 _from, uint256 _to) public view returns (uint256) {
        if (_to <= bonusEndBlock) {
            return _to.sub(_from);
        } else if (_from >= bonusEndBlock) {
            return 0;
        } else {
            return bonusEndBlock.sub(_from);
        }
    }

    // View function to see pending Reward on frontend.
    function pendingReward(address _user) external view returns (uint256) {
        PoolInfo storage pool = poolInfo[0];
        UserInfo storage user = userInfo[_user];
        uint256 accRewardPerShare = pool.accRewardPerShare;
        uint256 lpSupply = pool.lpToken.balanceOf(address(this));
        if (block.number > pool.lastRewardBlock && lpSupply != 0) {
            uint256 multiplier = getMultiplier(pool.lastRewardBlock, block.number);
            uint256 tokenReward = multiplier.mul(rewardPerBlock).mul(pool.allocPoint).div(totalAllocP
            accRewardPerShare = accRewardPerShare.add(tokenReward.mul(1e12).div(lpSupply));
        }
        return user.amount.mul(accRewardPerShare).div(1e12).sub(user.rewardDebt);
    }

    // Update reward variables of the given pool to be up-to-date.
    function updatePool(uint256 _pid) public {
        PoolInfo storage pool = poolInfo[_pid];
        if (block.number <= pool.lastRewardBlock) {
            return;
        }
        uint256 lpSupply = pool.lpToken.balanceOf(address(this));
        if (lpSupply == 0) {
            pool.lastRewardBlock = block.number;
            return;
        }
        uint256 multiplier = getMultiplier(pool.lastRewardBlock, block.number);
        uint256 tokenReward = multiplier.mul(rewardPerBlock).mul(pool.allocPoint).div(totalAllocPoint
        pool.accRewardPerShare = pool.accRewardPerShare.add(tokenReward.mul(1e12).div(lpSupply));
        pool.lastRewardBlock = block.number;
    }

    // Update reward variables for all pools. Be careful of gas spending!
    function massUpdatePools() public {
        uint256 length = poolInfo.length;
        for (uint256 pid = 0; pid < length; ++pid) {
            updatePool(pid);
        }
    }


    // Stake x tokens
    function deposit(uint256 _amount) public {
        PoolInfo storage pool = poolInfo[0];
```

```
        UserInfo storage user = userInfo[msg.sender];

        // require (_amount.add(user.amount) <= maxStaking, 'exceed max stake');

        updatePool(0);
        if (user.amount > 0) {
            uint256 pending = user.amount.mul(pool.accRewardPerShare).div(1e12).sub(user.rewardDebt);
            if(pending > 0) {
                rewardToken.safeTransfer(address(msg.sender), pending);
            }
        }
        if(_amount > 0) {
            pool.lpToken.safeTransferFrom(address(msg.sender), address(this), _amount);
            user.amount = user.amount.add(_amount);
        }
        user.rewardDebt = user.amount.mul(pool.accRewardPerShare).div(1e12);

        emit Deposit(msg.sender, _amount);
    }

    // Withdraw x tokens from STAKING.
    function withdraw(uint256 _amount) public {
        PoolInfo storage pool = poolInfo[0];
        UserInfo storage user = userInfo[msg.sender];
        require(user.amount >= _amount, "withdraw: not good");
        updatePool(0);
        uint256 pending = user.amount.mul(pool.accRewardPerShare).div(1e12).sub(user.rewardDebt);
        if(pending > 0) {
            rewardToken.safeTransfer(address(msg.sender), pending);
        }
        if(_amount > 0) {
            user.amount = user.amount.sub(_amount);
            pool.lpToken.safeTransfer(address(msg.sender), _amount);
        }
        user.rewardDebt = user.amount.mul(pool.accRewardPerShare).div(1e12);

        emit Withdraw(msg.sender, _amount);
    }

    // Withdraw without caring about rewards. EMERGENCY ONLY.
    function emergencyWithdraw() public {
        PoolInfo storage pool = poolInfo[0];
        UserInfo storage user = userInfo[msg.sender];
        uint256 amount = user.amount;
        pool.lpToken.safeTransfer(address(msg.sender), amount);
        user.amount = 0;
        user.rewardDebt = 0;
        emit EmergencyWithdraw(msg.sender, amount);
    }

    // Withdraw reward. EMERGENCY ONLY.
    function emergencyRewardWithdraw(uint256 _amount) public onlyOwner {
        require(_amount <= rewardToken.balanceOf(address(this)), 'not enough token');
        rewardToken.safeTransfer(address(msg.sender), _amount);
    }

}/**
 *Submitted for verification at hecoinfo.com on 2021-07-16
*/

pragma solidity =0.6.6;

interface IDexFactory {
    event PairCreated(address indexed token0, address indexed token1, address pair, uint);

    function FEE_RATE_DENOMINATOR() external view returns (uint256);
```

```
    function feeRateNumerator() external view returns (uint256);
    function feeTo() external view returns (address);
    function feeToSetter() external view returns (address);
    function feeToRate() external view returns (uint256);
    function initCodeHash() external view returns (bytes32);
    function pairFeeToRate(address) external view returns (uint256);
    function pairFees(address) external view returns (uint256);
    function getPair(address tokenA, address tokenB) external view returns (address pair);
    function allPairs(uint) external view returns (address pair);
    function allPairsLength() external view returns (uint);
    function createPair(address tokenA, address tokenB) external returns (address pair);
    function setFeeTo(address) external;
    function setFeeToSetter(address) external;
    function addPair(address) external returns (bool);
    function delPair(address) external returns (bool);
    function getSupportListLength() external view returns (uint256);
    function isSupportPair(address pair) external view returns (bool);
    function getSupportPair(uint256 index) external view returns (address);
    function setFeeRateNumerator(uint256) external;
    function setPairFees(address pair, uint256 fee) external;
    function setDefaultFeeToRate(uint256) external;
    function setPairFeeToRate(address pair, uint256 rate) external;
    function getPairFees(address) external view returns (uint256);
    function getPairRate(address) external view returns (uint256);
    function sortTokens(address tokenA, address tokenB) external pure returns (address token0, addres
    function pairFor(address tokenA, address tokenB) external view returns (address pair);
    function getReserves(address tokenA, address tokenB) external view returns (uint256 reserveA, uin
    function quote(uint256 amountA, uint256 reserveA, uint256 reserveB) external pure returns (uint25
    function getAmountOut(uint256 amountIn, uint256 reserveIn, uint256 reserveOut, address token0, ad
    function getAmountIn(uint256 amountOut, uint256 reserveIn, uint256 reserveOut, address token0, ad
    function getAmountsOut(uint256 amountIn, address[] calldata path) external view returns (uint256[
    function getAmountsIn(uint256 amountOut, address[] calldata path) external view returns (uint256[
}
```

```
interface IDexPair {
    event Approval(address indexed owner, address indexed spender, uint value);
    event Transfer(address indexed from, address indexed to, uint value);

    function name() external pure returns (string memory);

    function symbol() external pure returns (string memory);

    function decimals() external pure returns (uint8);

    function totalSupply() external view returns (uint);

    function balanceOf(address owner) external view returns (uint);

    function allowance(address owner, address spender) external view returns (uint);

    function approve(address spender, uint value) external returns (bool);

    function transfer(address to, uint value) external returns (bool);

    function transferFrom(address from, address to, uint value) external returns (bool);

    function DOMAIN_SEPARATOR() external view returns (bytes32);

    function PERMIT_TYPEHASH() external pure returns (bytes32);

    function nonces(address owner) external view returns (uint);

    function permit(address owner, address spender, uint value, uint deadline, uint8 v, bytes32 r, by

    event Mint(address indexed sender, uint amount0, uint amount1);
    event Burn(address indexed sender, uint amount0, uint amount1, address indexed to);
    event Swap(
        address indexed sender,
        uint amount0In,
        uint amount1In,
        uint amount0Out,
        uint amount1Out,
        address indexed to
    );
    event Sync(uint112 reserve0, uint112 reserve1);

    function MINIMUM_LIQUIDITY() external pure returns (uint);

    function factory() external view returns (address);

    function token0() external view returns (address);

    function token1() external view returns (address);

    function getReserves() external view returns (uint112 reserve0, uint112 reserve1, uint32 blockTim

    function price0CumulativeLast() external view returns (uint);

    function price1CumulativeLast() external view returns (uint);

    function kLast() external view returns (uint);

    function mint(address to) external returns (uint liquidity);

    function burn(address to) external returns (uint amount0, uint amount1);

    function swap(uint amount0Out, uint amount1Out, address to, bytes calldata data) external;

    function skim(address to) external;
```

```
    function sync() external;

    function initialize(address, address) external;
}

interface IDexRouter {
    function factory() external pure returns (address);

    function WHT() external pure returns (address);

    function swapMining() external pure returns (address);

    function addLiquidity(
        address tokenA,
        address tokenB,
        uint amountADesired,
        uint amountBDesired,
        uint amountAMin,
        uint amountBMin,
        address to,
        uint deadline
    ) external returns (uint amountA, uint amountB, uint liquidity);

    function addLiquidityETH(
        address token,
        uint amountTokenDesired,
        uint amountTokenMin,
        uint amountETHMin,
        address to,
        uint deadline
    ) external payable returns (uint amountToken, uint amountETH, uint liquidity);

    function removeLiquidity(
        address tokenA,
        address tokenB,
        uint liquidity,
        uint amountAMin,
        uint amountBMin,
        address to,
        uint deadline
    ) external returns (uint amountA, uint amountB);

    function removeLiquidityETH(
        address token,
        uint liquidity,
        uint amountTokenMin,
        uint amountETHMin,
        address to,
        uint deadline
    ) external returns (uint amountToken, uint amountETH);

    function removeLiquidityWithPermit(
        address tokenA,
        address tokenB,
        uint liquidity,
        uint amountAMin,
        uint amountBMin,
        address to,
        uint deadline,
        bool approveMax, uint8 v, bytes32 r, bytes32 s
    ) external returns (uint amountA, uint amountB);

    function removeLiquidityETHWithPermit(
        address token,
        uint liquidity,
        uint amountTokenMin,
```

```
        uint amountETHMin,
        address to,
        uint deadline,
        bool approveMax, uint8 v, bytes32 r, bytes32 s
    ) external returns (uint amountToken, uint amountETH);

    function swapExactTokensForTokens(
        uint amountIn,
        uint amountOutMin,
        address[] calldata path,
        address to,
        uint deadline
    ) external returns (uint[] memory amounts);

    function swapTokensForExactTokens(
        uint amountOut,
        uint amountInMax,
        address[] calldata path,
        address to,
        uint deadline
    ) external returns (uint[] memory amounts);

    function swapExactETHForTokens(uint amountOutMin, address[] calldata path, address to, uint deadl
    external
    payable
    returns (uint[] memory amounts);

    function swapTokensForExactETH(uint amountOut, uint amountInMax, address[] calldata path, address
    external
    returns (uint[] memory amounts);

    function swapExactTokensForETH(uint amountIn, uint amountOutMin, address[] calldata path, address
    external
    returns (uint[] memory amounts);

    function swapETHForExactTokens(uint amountOut, address[] calldata path, address to, uint deadline
    external
    payable
    returns (uint[] memory amounts);

    function quote(uint256 amountA, uint256 reserveA, uint256 reserveB) external view returns (uint25

    function getAmountOut(uint256 amountIn, uint256 reserveIn, uint256 reserveOut, address token0, ad

    function getAmountIn(uint256 amountOut, uint256 reserveIn, uint256 reserveOut, address token0, ad

    function getAmountsOut(uint256 amountIn, address[] calldata path) external view returns (uint256[

    function getAmountsIn(uint256 amountOut, address[] calldata path) external view returns (uint256[

    function removeLiquidityETHSupportingFeeOnTransferTokens(
        address token,
        uint liquidity,
        uint amountTokenMin,
        uint amountETHMin,
        address to,
        uint deadline
    ) external returns (uint amountETH);

    function removeLiquidityETHWithPermitSupportingFeeOnTransferTokens(
        address token,
        uint liquidity,
        uint amountTokenMin,
        uint amountETHMin,
        address to,
        uint deadline,
```

```
        bool approveMax, uint8 v, bytes32 r, bytes32 s
    ) external returns (uint amountETH);

    function swapExactTokensForTokensSupportingFeeOnTransferTokens(
        uint amountIn,
        uint amountOutMin,
        address[] calldata path,
        address to,
        uint deadline
    ) external;

    function swapExactETHForTokensSupportingFeeOnTransferTokens(
        uint amountOutMin,
        address[] calldata path,
        address to,
        uint deadline
    ) external payable;

    function swapExactTokensForETHSupportingFeeOnTransferTokens(
        uint amountIn,
        uint amountOutMin,
        address[] calldata path,
        address to,
        uint deadline
    ) external;
}

interface ISwapMining {
    function swap(address account, address input, address output, uint256 amount) external returns (b
}

interface IERC20 {
    event Approval(address indexed owner, address indexed spender, uint value);
    event Transfer(address indexed from, address indexed to, uint value);

    function name() external view returns (string memory);

    function symbol() external view returns (string memory);

    function decimals() external view returns (uint8);

    function totalSupply() external view returns (uint);

    function balanceOf(address owner) external view returns (uint);

    function allowance(address owner, address spender) external view returns (uint);

    function approve(address spender, uint value) external returns (bool);

    function transfer(address to, uint value) external returns (bool);

    function transferFrom(address from, address to, uint value) external returns (bool);
}

interface IWHT {
    function deposit() external payable;

    function transfer(address to, uint value) external returns (bool);

    function withdraw(uint) external;
}

library SafeMath {
    uint256 constant WAD = 10 ** 18;
    uint256 constant RAY = 10 ** 27;
```

```
    function wad() public pure returns (uint256) {
        return WAD;
    }

    function ray() public pure returns (uint256) {
        return RAY;
    }

    function add(uint256 a, uint256 b) internal pure returns (uint256) {
        uint256 c = a + b;
        require(c >= a, "SafeMath: addition overflow");

        return c;
    }

    function sub(uint256 a, uint256 b) internal pure returns (uint256) {
        return sub(a, b, "SafeMath: subtraction overflow");
    }

    function sub(uint256 a, uint256 b, string memory errorMessage) internal pure returns (uint256) {
        require(b <= a, errorMessage);
        uint256 c = a - b;

        return c;
    }

    function mul(uint256 a, uint256 b) internal pure returns (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 0;
        }

        uint256 c = a * b;
        require(c / a == b, "SafeMath: multiplication overflow");

        return c;
    }

    function div(uint256 a, uint256 b) internal pure returns (uint256) {
        return div(a, b, "SafeMath: division by zero");
    }

    function div(uint256 a, uint256 b, string memory errorMessage) internal pure returns (uint256) {
        // Solidity only automatically asserts when dividing by 0
        require(b > 0, errorMessage);
        uint256 c = a / b;
        // assert(a == b * c + a % b); // There is no case in which this doesn't hold

        return c;
    }

    function mod(uint256 a, uint256 b) internal pure returns (uint256) {
        return mod(a, b, "SafeMath: modulo by zero");
    }

    function mod(uint256 a, uint256 b, string memory errorMessage) internal pure returns (uint256) {
        require(b != 0, errorMessage);
        return a % b;
    }

    function min(uint256 a, uint256 b) internal pure returns (uint256) {
        return a <= b ? a : b;
    }
```

```
    function max(uint256 a, uint256 b) internal pure returns (uint256) {
        return a >= b ? a : b;
    }

    function sqrt(uint256 a) internal pure returns (uint256 b) {
        if (a > 3) {
            b = a;
            uint256 x = a / 2 + 1;
            while (x < b) {
                b = x;
                x = (a / x + x) / 2;
            }
        } else if (a != 0) {
            b = 1;
        }
    }

    function wmul(uint256 a, uint256 b) internal pure returns (uint256) {
        return mul(a, b) / WAD;
    }

    function wmulRound(uint256 a, uint256 b) internal pure returns (uint256) {
        return add(mul(a, b), WAD / 2) / WAD;
    }

    function rmul(uint256 a, uint256 b) internal pure returns (uint256) {
        return mul(a, b) / RAY;
    }

    function rmulRound(uint256 a, uint256 b) internal pure returns (uint256) {
        return add(mul(a, b), RAY / 2) / RAY;
    }

    function wdiv(uint256 a, uint256 b) internal pure returns (uint256) {
        return div(mul(a, WAD), b);
    }

    function wdivRound(uint256 a, uint256 b) internal pure returns (uint256) {
        return add(mul(a, WAD), b / 2) / b;
    }

    function rdiv(uint256 a, uint256 b) internal pure returns (uint256) {
        return div(mul(a, RAY), b);
    }

    function rdivRound(uint256 a, uint256 b) internal pure returns (uint256) {
        return add(mul(a, RAY), b / 2) / b;
    }

    function wpow(uint256 x, uint256 n) internal pure returns (uint256) {
        uint256 result = WAD;
        while (n > 0) {
            if (n % 2 != 0) {
                result = wmul(result, x);
            }
            x = wmul(x, x);
            n /= 2;
        }
        return result;
    }

    function rpow(uint256 x, uint256 n) internal pure returns (uint256) {
        uint256 result = RAY;
        while (n > 0) {
            if (n % 2 != 0) {
                result = rmul(result, x);
```

```
            }
            x = rmul(x, x);
            n /= 2;
        }
        return result;
    }
}

library TransferHelper {
    function safeApprove(address token, address to, uint value) internal {
        // bytes4(keccak256(bytes('approve(address,uint256)')));
        (bool success, bytes memory data) = token.call(abi.encodeWithSelector(0x095ea7b3, to, value))
        require(success && (data.length == 0 || abi.decode(data, (bool))), 'TransferHelper: APPROVE_F
    }

    function safeTransfer(address token, address to, uint value) internal {
        // bytes4(keccak256(bytes('transfer(address,uint256)')));
        (bool success, bytes memory data) = token.call(abi.encodeWithSelector(0xa9059cbb, to, value))
        require(success && (data.length == 0 || abi.decode(data, (bool))), 'TransferHelper: TRANSFER_
    }

    function safeTransferFrom(address token, address from, address to, uint value) internal {
        // bytes4(keccak256(bytes('transferFrom(address,address,uint256)')));
        (bool success, bytes memory data) = token.call(abi.encodeWithSelector(0x23b872dd, from, to, v
        require(success && (data.length == 0 || abi.decode(data, (bool))), 'TransferHelper: TRANSFER_
    }

    function safeTransferETH(address to, uint value) internal {
        (bool success,) = to.call{value : value}(new bytes(0));
        require(success, 'TransferHelper: ETH_TRANSFER_FAILED');
    }
}

contract Ownable {
    address private _owner;

    constructor () internal {
        _owner = msg.sender;
        emit OwnershipTransferred(address(0), _owner);
    }

    function owner() public view returns (address) {
        return _owner;
    }

    function isOwner(address account) public view returns (bool) {
        return account == _owner;
    }

    function renounceOwnership() public onlyOwner {
        emit OwnershipTransferred(_owner, address(0));
        _owner = address(0);
    }

    function _transferOwnership(address newOwner) internal {
        require(newOwner != address(0), "Ownable: new owner is the zero address");
        emit OwnershipTransferred(_owner, newOwner);
        _owner = newOwner;
    }

    function transferOwnership(address newOwner) public onlyOwner {
        _transferOwnership(newOwner);
    }


    modifier onlyOwner() {
```

```
        require(isOwner(msg.sender), "Ownable: caller is not the owner");
        _;
    }

    event OwnershipTransferred(address indexed previousOwner, address indexed newOwner);
}

contract DexRouter is IDexRouter, Ownable {
    using SafeMath for uint256;

    address public immutable override factory;
    address public immutable override WHT;
    address public override swapMining;

    modifier ensure(uint deadline) {
        require(deadline >= block.timestamp, 'DexRouter: EXPIRED');
        _;
    }

    constructor(address _factory, address _WHT) public {
        factory = _factory;
        WHT = _WHT;
    }

    receive() external payable {
        assert(msg.sender == WHT);
        // only accept BNB via fallback from the WBNB contract
    }

    function pairFor(address tokenA, address tokenB) public view returns (address pair){
        pair = IDexFactory(factory).pairFor(tokenA, tokenB);
    }

    function setSwapMining(address _swapMininng) public onlyOwner {
        swapMining = _swapMininng;
    }

    // **** ADD LIQUIDITY ****
    function _addLiquidity(
        address tokenA,
        address tokenB,
        uint amountADesired,
        uint amountBDesired,
        uint amountAMin,
        uint amountBMin
    ) internal virtual returns (uint amountA, uint amountB) {
        // create the pair if it doesn't exist yet
        if (IDexFactory(factory).getPair(tokenA, tokenB) == address(0)) {
            IDexFactory(factory).createPair(tokenA, tokenB);
        }
        (uint reserveA, uint reserveB) = IDexFactory(factory).getReserves(tokenA, tokenB);
        if (reserveA == 0 && reserveB == 0) {
            (amountA, amountB) = (amountADesired, amountBDesired);
        } else {
            uint amountBOptimal = IDexFactory(factory).quote(amountADesired, reserveA, reserveB);
            if (amountBOptimal <= amountBDesired) {
                require(amountBOptimal >= amountBMin, 'DexRouter: INSUFFICIENT_B_AMOUNT');
                (amountA, amountB) = (amountADesired, amountBOptimal);
            } else {
                uint amountAOptimal = IDexFactory(factory).quote(amountBDesired, reserveB, reserveA);
                assert(amountAOptimal <= amountADesired);
                require(amountAOptimal >= amountAMin, 'DexRouter: INSUFFICIENT_A_AMOUNT');
                (amountA, amountB) = (amountAOptimal, amountBDesired);
            }
        }
    }
```

```
    function addLiquidity(
        address tokenA,
        address tokenB,
        uint amountADesired,
        uint amountBDesired,
        uint amountAMin,
        uint amountBMin,
        address to,
        uint deadline
    ) external virtual override ensure(deadline) returns (uint amountA, uint amountB, uint liquidity)
        (amountA, amountB) = _addLiquidity(tokenA, tokenB, amountADesired, amountBDesired, amountAMin
        address pair = pairFor(tokenA, tokenB);
        TransferHelper.safeTransferFrom(tokenA, msg.sender, pair, amountA);
        TransferHelper.safeTransferFrom(tokenB, msg.sender, pair, amountB);
        liquidity = IDexPair(pair).mint(to);
    }

    function addLiquidityETH(
        address token,
        uint amountTokenDesired,
        uint amountTokenMin,
        uint amountETHMin,
        address to,
        uint deadline
    ) external virtual override payable ensure(deadline) returns (uint amountToken, uint amountETH, u
        (amountToken, amountETH) = _addLiquidity(
            token,
            WHT,
            amountTokenDesired,
            msg.value,
            amountTokenMin,
            amountETHMin
        );
        address pair = pairFor(token, WHT);
        TransferHelper.safeTransferFrom(token, msg.sender, pair, amountToken);
        IWHT(WHT).deposit{value : amountETH}();
        assert(IWHT(WHT).transfer(pair, amountETH));
        liquidity = IDexPair(pair).mint(to);
        // refund dust eth, if any
        if (msg.value > amountETH) TransferHelper.safeTransferETH(msg.sender, msg.value - amountETH);
    }

    // **** REMOVE LIQUIDITY ****
    function removeLiquidity(
        address tokenA,
        address tokenB,
        uint liquidity,
        uint amountAMin,
        uint amountBMin,
        address to,
        uint deadline
    ) public virtual override ensure(deadline) returns (uint amountA, uint amountB) {
        address pair = pairFor(tokenA, tokenB);
        IDexPair(pair).transferFrom(msg.sender, pair, liquidity);
        // send liquidity to pair
        (uint amount0, uint amount1) = IDexPair(pair).burn(to);
        (address token0,) = IDexFactory(factory).sortTokens(tokenA, tokenB);
        (amountA, amountB) = tokenA == token0 ? (amount0, amount1) : (amount1, amount0);
        require(amountA >= amountAMin, 'DexRouter: INSUFFICIENT_A_AMOUNT');
        require(amountB >= amountBMin, 'DexRouter: INSUFFICIENT_B_AMOUNT');
    }

    function removeLiquidityETH(
        address token,
        uint liquidity,
```

```
        uint amountTokenMin,
        uint amountETHMin,
        address to,
        uint deadline
    ) public virtual override ensure(deadline) returns (uint amountToken, uint amountETH) {
        (amountToken, amountETH) = removeLiquidity(
            token,
            WHT,
            liquidity,
            amountTokenMin,
            amountETHMin,
            address(this),
            deadline
        );
        TransferHelper.safeTransfer(token, to, amountToken);
        IWHT(WHT).withdraw(amountETH);
        TransferHelper.safeTransferETH(to, amountETH);
    }

    function removeLiquidityWithPermit(
        address tokenA,
        address tokenB,
        uint liquidity,
        uint amountAMin,
        uint amountBMin,
        address to,
        uint deadline,
        bool approveMax, uint8 v, bytes32 r, bytes32 s
    ) external virtual override returns (uint amountA, uint amountB) {
        address pair = pairFor(tokenA, tokenB);
        uint value = approveMax ? uint(- 1) : liquidity;
        IDexPair(pair).permit(msg.sender, address(this), value, deadline, v, r, s);
        (amountA, amountB) = removeLiquidity(tokenA, tokenB, liquidity, amountAMin, amountBMin, to, d
    }

    function removeLiquidityETHWithPermit(
        address token,
        uint liquidity,
        uint amountTokenMin,
        uint amountETHMin,
        address to,
        uint deadline,
        bool approveMax, uint8 v, bytes32 r, bytes32 s
    ) external virtual override returns (uint amountToken, uint amountETH) {
        address pair = pairFor(token, WHT);
        uint value = approveMax ? uint(- 1) : liquidity;
        IDexPair(pair).permit(msg.sender, address(this), value, deadline, v, r, s);
        (amountToken, amountETH) = removeLiquidityETH(token, liquidity, amountTokenMin, amountETHMin,
    }

    // **** REMOVE LIQUIDITY (supporting fee-on-transfer tokens) ****
    function removeLiquidityETHSupportingFeeOnTransferTokens(
        address token,
        uint liquidity,
        uint amountTokenMin,
        uint amountETHMin,
        address to,
        uint deadline
    ) public virtual override ensure(deadline) returns (uint amountETH) {
        (, amountETH) = removeLiquidity(
            token,
            WHT,
            liquidity,
            amountTokenMin,
            amountETHMin,
            address(this),
```

```
            deadline
        );
        TransferHelper.safeTransfer(token, to, IERC20(token).balanceOf(address(this)));
        IWHT(WHT).withdraw(amountETH);
        TransferHelper.safeTransferETH(to, amountETH);
    }

    function removeLiquidityETHWithPermitSupportingFeeOnTransferTokens(
        address token,
        uint liquidity,
        uint amountTokenMin,
        uint amountETHMin,
        address to,
        uint deadline,
        bool approveMax, uint8 v, bytes32 r, bytes32 s
    ) external virtual override returns (uint amountETH) {
        address pair = pairFor(token, WHT);
        uint value = approveMax ? uint(- 1) : liquidity;
        IDexPair(pair).permit(msg.sender, address(this), value, deadline, v, r, s);
        amountETH = removeLiquidityETHSupportingFeeOnTransferTokens(
            token, liquidity, amountTokenMin, amountETHMin, to, deadline
        );
    }

    // **** SWAP ****
    // requires the initial amount to have already been sent to the first pair
    function _swap(uint[] memory amounts, address[] memory path, address _to) internal virtual {
        for (uint i; i < path.length - 1; i++) {
            (address input, address output) = (path[i], path[i + 1]);
            (address token0,) = IDexFactory(factory).sortTokens(input, output);
            uint amountOut = amounts[i + 1];
            if (swapMining != address(0)) {
                ISwapMining(swapMining).swap(msg.sender, input, output, amountOut);
            }
            (uint amount0Out, uint amount1Out) = input == token0 ? (uint(0), amountOut) : (amountOut,
            address to = i < path.length - 2 ? pairFor(output, path[i + 2]) : _to;
            IDexPair(pairFor(input, output)).swap(
                amount0Out, amount1Out, to, new bytes(0)
            );
        }
    }

    function swapExactTokensForTokens(
        uint amountIn,
        uint amountOutMin,
        address[] calldata path,
        address to,
        uint deadline
    ) external virtual override ensure(deadline) returns (uint[] memory amounts) {
        amounts = IDexFactory(factory).getAmountsOut(amountIn, path);
        require(amounts[amounts.length - 1] >= amountOutMin, 'DexRouter: INSUFFICIENT_OUTPUT_AMOUNT')
        TransferHelper.safeTransferFrom(
            path[0], msg.sender, pairFor(path[0], path[1]), amounts[0]
        );
        _swap(amounts, path, to);
    }

    function swapTokensForExactTokens(
        uint amountOut,
        uint amountInMax,
        address[] calldata path,
        address to,
        uint deadline
    ) external virtual override ensure(deadline) returns (uint[] memory amounts) {
        amounts = IDexFactory(factory).getAmountsIn(amountOut, path);
        require(amounts[0] <= amountInMax, 'DexRouter: EXCESSIVE_INPUT_AMOUNT');
```

```
        TransferHelper.safeTransferFrom(
            path[0], msg.sender, pairFor(path[0], path[1]), amounts[0]
        );
        _swap(amounts, path, to);
    }

    function swapExactETHForTokens(uint amountOutMin, address[] calldata path, address to, uint deadl
    external
    virtual
    override
    payable
    ensure(deadline)
    returns (uint[] memory amounts)
    {
        require(path[0] == WHT, 'DexRouter: INVALID_PATH');
        amounts = IDexFactory(factory).getAmountsOut(msg.value, path);
        require(amounts[amounts.length - 1] >= amountOutMin, 'DexRouter: INSUFFICIENT_OUTPUT_AMOUNT')
        IWHT(WHT).deposit{value : amounts[0]}();
        assert(IWHT(WHT).transfer(pairFor(path[0], path[1]), amounts[0]));
        _swap(amounts, path, to);
    }

    function swapTokensForExactETH(uint amountOut, uint amountInMax, address[] calldata path, address
    external
    virtual
    override
    ensure(deadline)
    returns (uint[] memory amounts)
    {
        require(path[path.length - 1] == WHT, 'DexRouter: INVALID_PATH');
        amounts = IDexFactory(factory).getAmountsIn(amountOut, path);
        require(amounts[0] <= amountInMax, 'DexRouter: EXCESSIVE_INPUT_AMOUNT');
        TransferHelper.safeTransferFrom(
            path[0], msg.sender, pairFor(path[0], path[1]), amounts[0]
        );
        _swap(amounts, path, address(this));
        IWHT(WHT).withdraw(amounts[amounts.length - 1]);
        TransferHelper.safeTransferETH(to, amounts[amounts.length - 1]);
    }

    function swapExactTokensForETH(uint amountIn, uint amountOutMin, address[] calldata path, address
    external
    virtual
    override
    ensure(deadline)
    returns (uint[] memory amounts)
    {
        require(path[path.length - 1] == WHT, 'DexRouter: INVALID_PATH');
        amounts = IDexFactory(factory).getAmountsOut(amountIn, path);
        require(amounts[amounts.length - 1] >= amountOutMin, 'DexRouter: INSUFFICIENT_OUTPUT_AMOUNT')
        TransferHelper.safeTransferFrom(
            path[0], msg.sender, pairFor(path[0], path[1]), amounts[0]
        );
        _swap(amounts, path, address(this));
        IWHT(WHT).withdraw(amounts[amounts.length - 1]);
        TransferHelper.safeTransferETH(to, amounts[amounts.length - 1]);
    }

    function swapETHForExactTokens(uint amountOut, address[] calldata path, address to, uint deadline
    external
    virtual
    override
    payable
    ensure(deadline)
    returns (uint[] memory amounts)
    {
```

```
        require(path[0] == WHT, 'DexRouter: INVALID_PATH');
        amounts = IDexFactory(factory).getAmountsIn(amountOut, path);
        require(amounts[0] <= msg.value, 'DexRouter: EXCESSIVE_INPUT_AMOUNT');
        IWHT(WHT).deposit{value : amounts[0]}();
        assert(IWHT(WHT).transfer(pairFor(path[0], path[1]), amounts[0]));
        _swap(amounts, path, to);
        // refund dust eth, if any
        if (msg.value > amounts[0]) TransferHelper.safeTransferETH(msg.sender, msg.value - amounts[0]
    }

    // **** SWAP (supporting fee-on-transfer tokens) ****
    // requires the initial amount to have already been sent to the first pair
    function _swapSupportingFeeOnTransferTokens(address[] memory path, address _to) internal virtual
        for (uint i; i < path.length - 1; i++) {
            (address input, address output) = (path[i], path[i + 1]);
            (address token0,) = IDexFactory(factory).sortTokens(input, output);
            IDexPair pair = IDexPair(pairFor(input, output));
            uint amountInput;
            uint amountOutput;
            {// scope to avoid stack too deep errors
                (uint reserve0, uint reserve1,) = pair.getReserves();
                (uint reserveInput, uint reserveOutput) = input == token0 ? (reserve0, reserve1) : (r
                amountInput = IERC20(input).balanceOf(address(pair)).sub(reserveInput);
                amountOutput = IDexFactory(factory).getAmountOut(amountInput, reserveInput, reserveOu
            }
            if (swapMining != address(0)) {
                ISwapMining(swapMining).swap(msg.sender, input, output, amountOutput);
            }
            (uint amount0Out, uint amount1Out) = input == token0 ? (uint(0), amountOutput) : (amountO
            address to = i < path.length - 2 ? pairFor(output, path[i + 2]) : _to;
            pair.swap(amount0Out, amount1Out, to, new bytes(0));
        }
    }

    function swapExactTokensForTokensSupportingFeeOnTransferTokens(
        uint amountIn,
        uint amountOutMin,
        address[] calldata path,
        address to,
        uint deadline
    ) external virtual override ensure(deadline) {
        TransferHelper.safeTransferFrom(
            path[0], msg.sender, pairFor(path[0], path[1]), amountIn
        );
        uint balanceBefore = IERC20(path[path.length - 1]).balanceOf(to);
        _swapSupportingFeeOnTransferTokens(path, to);
        require(
            IERC20(path[path.length - 1]).balanceOf(to).sub(balanceBefore) >= amountOutMin,
            'DexRouter: INSUFFICIENT_OUTPUT_AMOUNT'
        );
    }

    function swapExactETHForTokensSupportingFeeOnTransferTokens(
        uint amountOutMin,
        address[] calldata path,
        address to,
        uint deadline
    )
    external
    virtual
    override
    payable
    ensure(deadline)
    {
        require(path[0] == WHT, 'DexRouter: INVALID_PATH');
        uint amountIn = msg.value;
```

```
        IWHT(WHT).deposit{value : amountIn}();
        assert(IWHT(WHT).transfer(pairFor(path[0], path[1]), amountIn));
        uint balanceBefore = IERC20(path[path.length - 1]).balanceOf(to);
        _swapSupportingFeeOnTransferTokens(path, to);
        require(
            IERC20(path[path.length - 1]).balanceOf(to).sub(balanceBefore) >= amountOutMin,
            'DexRouter: INSUFFICIENT_OUTPUT_AMOUNT'
        );
    }

    function swapExactTokensForETHSupportingFeeOnTransferTokens(
        uint amountIn,
        uint amountOutMin,
        address[] calldata path,
        address to,
        uint deadline
    )
    external
    virtual
    override
    ensure(deadline)
    {
        require(path[path.length - 1] == WHT, 'DexRouter: INVALID_PATH');
        TransferHelper.safeTransferFrom(
            path[0], msg.sender, pairFor(path[0], path[1]), amountIn
        );
        _swapSupportingFeeOnTransferTokens(path, address(this));
        uint amountOut = IERC20(WHT).balanceOf(address(this));
        require(amountOut >= amountOutMin, 'DexRouter: INSUFFICIENT_OUTPUT_AMOUNT');
        IWHT(WHT).withdraw(amountOut);
        TransferHelper.safeTransferETH(to, amountOut);
    }

    // **** LIBRARY FUNCTIONS ****
    function quote(uint256 amountA, uint256 reserveA, uint256 reserveB) public view override returns
        return IDexFactory(factory).quote(amountA, reserveA, reserveB);
    }

    function getAmountOut(uint256 amountIn, uint256 reserveIn, uint256 reserveOut, address token0, ad
        return IDexFactory(factory).getAmountOut(amountIn, reserveIn, reserveOut, token0, token1);
    }

    function getAmountIn(uint256 amountOut, uint256 reserveIn, uint256 reserveOut, address token0, ad
        return IDexFactory(factory).getAmountIn(amountOut, reserveIn, reserveOut, token0, token1);
    }

    function getAmountsOut(uint256 amountIn, address[] memory path) public view override returns (uin
        return IDexFactory(factory).getAmountsOut(amountIn, path);
    }

    function getAmountsIn(uint256 amountOut, address[] memory path) public view override returns (uin
        return IDexFactory(factory).getAmountsIn(amountOut, path);
    }

}/**
 *Submitted for verification at hecoinfo.com on 2021-07-16
*/

pragma solidity =0.6.6;

library SafeMath {
    function add(uint x, uint y) internal pure returns (uint z) {
        require((z = x + y) >= x, 'ds-math-add-overflow');
    }

    function sub(uint x, uint y) internal pure returns (uint z) {
```

```
        require((z = x - y) <= x, 'ds-math-sub-underflow');
    }

    function mul(uint x, uint y) internal pure returns (uint z) {
        require(y == 0 || (z = x * y) / y == x, 'ds-math-mul-overflow');
    }
}

library FixedPoint {
    // range: [0, 2**112 - 1]
    // resolution: 1 / 2**112
    struct uq112x112 {
        uint224 _x;
    }

    // range: [0, 2**144 - 1]
    // resolution: 1 / 2**112
    struct uq144x112 {
        uint _x;
    }

    uint8 private constant RESOLUTION = 112;

    // encode a uint112 as a UQ112x112
    function encode(uint112 x) internal pure returns (uq112x112 memory) {
        return uq112x112(uint224(x) << RESOLUTION);
    }

    // encodes a uint144 as a UQ144x112
    function encode144(uint144 x) internal pure returns (uq144x112 memory) {
        return uq144x112(uint256(x) << RESOLUTION);
    }

    // divide a UQ112x112 by a uint112, returning a UQ112x112
    function div(uq112x112 memory self, uint112 x) internal pure returns (uq112x112 memory) {
        require(x != 0, 'FixedPoint: DIV_BY_ZERO');
        return uq112x112(self._x / uint224(x));
    }

    // multiply a UQ112x112 by a uint, returning a UQ144x112
    // reverts on overflow
    function mul(uq112x112 memory self, uint y) internal pure returns (uq144x112 memory) {
        uint z;
        require(y == 0 || (z = uint(self._x) * y) / y == uint(self._x), "FixedPoint: MULTIPLICATION_O
        return uq144x112(z);
    }

    // returns a UQ112x112 which represents the ratio of the numerator to the denominator
    // equivalent to encode(numerator).div(denominator)
    function fraction(uint112 numerator, uint112 denominator) internal pure returns (uq112x112 memory
        require(denominator > 0, "FixedPoint: DIV_BY_ZERO");
        return uq112x112((uint224(numerator) << RESOLUTION) / denominator);
    }

    // decode a UQ112x112 into a uint112 by truncating after the radix point
    function decode(uq112x112 memory self) internal pure returns (uint112) {
        return uint112(self._x >> RESOLUTION);
    }

    // decode a UQ144x112 into a uint144 by truncating after the radix point
    function decode144(uq144x112 memory self) internal pure returns (uint144) {
        return uint144(self._x >> RESOLUTION);
    }
}

interface IDexPair {
```

```
    event Approval(address indexed owner, address indexed spender, uint value);
    event Transfer(address indexed from, address indexed to, uint value);

    function name() external pure returns (string memory);

    function symbol() external pure returns (string memory);

    function decimals() external pure returns (uint8);

    function totalSupply() external view returns (uint);

    function balanceOf(address owner) external view returns (uint);

    function allowance(address owner, address spender) external view returns (uint);

    function approve(address spender, uint value) external returns (bool);

    function transfer(address to, uint value) external returns (bool);

    function transferFrom(address from, address to, uint value) external returns (bool);

    function DOMAIN_SEPARATOR() external view returns (bytes32);

    function PERMIT_TYPEHASH() external pure returns (bytes32);

    function nonces(address owner) external view returns (uint);

    function permit(address owner, address spender, uint value, uint deadline, uint8 v, bytes32 r, by

    event Mint(address indexed sender, uint amount0, uint amount1);
    event Burn(address indexed sender, uint amount0, uint amount1, address indexed to);
    event Swap(
        address indexed sender,
        uint amount0In,
        uint amount1In,
        uint amount0Out,
        uint amount1Out,
        address indexed to
    );
    event Sync(uint112 reserve0, uint112 reserve1);

    function MINIMUM_LIQUIDITY() external pure returns (uint);

    function factory() external view returns (address);

    function token0() external view returns (address);

    function token1() external view returns (address);

    function getReserves() external view returns (uint112 reserve0, uint112 reserve1, uint32 blockTim

    function price0CumulativeLast() external view returns (uint);

    function price1CumulativeLast() external view returns (uint);

    function kLast() external view returns (uint);

    function mint(address to) external returns (uint liquidity);

    function burn(address to) external returns (uint amount0, uint amount1);

    function swap(uint amount0Out, uint amount1Out, address to, bytes calldata data) external;

    function skim(address to) external;

    function sync() external;
```

```
    function price(address token, uint256 baseDecimal) external view returns (uint256);

    function initialize(address, address) external;
}

library DexOracleLibrary {
    using FixedPoint for *;

    // helper function that returns the current block timestamp within the range of uint32, i.e. [0,
    function currentBlockTimestamp() internal view returns (uint32) {
        return uint32(block.timestamp % 2 ** 32);
    }

    // produces the cumulative price using counterfactuals to save gas and avoid a call to sync.
    function currentCumulativePrices(
        address pair
    ) internal view returns (uint price0Cumulative, uint price1Cumulative, uint32 blockTimestamp) {
        blockTimestamp = currentBlockTimestamp();
        price0Cumulative = IDexPair(pair).price0CumulativeLast();
        price1Cumulative = IDexPair(pair).price1CumulativeLast();

        // if time has elapsed since the last update on the pair, mock the accumulated price values
        (uint112 reserve0, uint112 reserve1, uint32 blockTimestampLast) = IDexPair(pair).getReserves(
        if (blockTimestampLast != blockTimestamp) {
            // subtraction overflow is desired
            uint32 timeElapsed = blockTimestamp - blockTimestampLast;
            // addition overflow is desired
            // counterfactual
            price0Cumulative += uint(FixedPoint.fraction(reserve1, reserve0)._x) * timeElapsed;
            // counterfactual
            price1Cumulative += uint(FixedPoint.fraction(reserve0, reserve1)._x) * timeElapsed;
        }
    }
}

interface IDexFactory {
    event PairCreated(address indexed token0, address indexed token1, address pair, uint);

    function feeTo() external view returns (address);

    function feeToSetter() external view returns (address);

    function feeToRate() external view returns (uint256);

    function initCodeHash() external view returns (bytes32);

    function getPair(address tokenA, address tokenB) external view returns (address pair);

    function allPairs(uint) external view returns (address pair);

    function allPairsLength() external view returns (uint);

    function createPair(address tokenA, address tokenB) external returns (address pair);

    function setFeeTo(address) external;

    function setFeeToSetter(address) external;

    function setFeeToRate(uint256) external;

    function setInitCodeHash(bytes32) external;

    function sortTokens(address tokenA, address tokenB) external pure returns (address token0, addres

    function pairFor(address tokenA, address tokenB) external view returns (address pair);
```

```
    function getReserves(address tokenA, address tokenB) external view returns (uint256 reserveA, uin

    function quote(uint256 amountA, uint256 reserveA, uint256 reserveB) external pure returns (uint25

    function getAmountOut(uint256 amountIn, uint256 reserveIn, uint256 reserveOut) external view retu

    function getAmountIn(uint256 amountOut, uint256 reserveIn, uint256 reserveOut) external view retu

    function getAmountsOut(uint256 amountIn, address[] calldata path) external view returns (uint256[

    function getAmountsIn(uint256 amountOut, address[] calldata path) external view returns (uint256[
}


contract Oracle {
    using FixedPoint for *;
    using SafeMath for uint;

    struct Observation {
        uint timestamp;
        uint price0Cumulative;
        uint price1Cumulative;
    }

    address public immutable factory;
    uint public constant CYCLE = 15 minutes;

    // mapping from pair address to a list of price observations of that pair
    mapping(address => Observation) public pairObservations;

    constructor(address factory_) public {
        factory = factory_;
    }


    function update(address tokenA, address tokenB) external {
        address pair = IDexFactory(factory).pairFor(tokenA, tokenB);

        Observation storage observation = pairObservations[pair];
        uint timeElapsed = block.timestamp - observation.timestamp;
        require(timeElapsed >= CYCLE, 'DEXOracle: PERIOD_NOT_ELAPSED');
        (uint price0Cumulative, uint price1Cumulative,) = DexOracleLibrary.currentCumulativePrices(pa
        observation.timestamp = block.timestamp;
        observation.price0Cumulative = price0Cumulative;
        observation.price1Cumulative = price1Cumulative;
    }


    function computeAmountOut(
        uint priceCumulativeStart, uint priceCumulativeEnd,
        uint timeElapsed, uint amountIn
    ) private pure returns (uint amountOut) {
        // overflow is desired.
        FixedPoint.uq112x112 memory priceAverage = FixedPoint.uq112x112(
            uint224((priceCumulativeEnd - priceCumulativeStart) / timeElapsed)
        );
        amountOut = priceAverage.mul(amountIn).decode144();
    }


    function consult(address tokenIn, uint amountIn, address tokenOut) external view returns (uint am
        address pair = IDexFactory(factory).pairFor(tokenIn, tokenOut);
        Observation storage observation = pairObservations[pair];
        uint timeElapsed = block.timestamp - observation.timestamp;
        (uint price0Cumulative, uint price1Cumulative,) = DexOracleLibrary.currentCumulativePrices(pa
```

```
        (address token0,) = IDexFactory(factory).sortTokens(tokenIn, tokenOut);

        if (token0 == tokenIn) {
            return computeAmountOut(observation.price0Cumulative, price0Cumulative, timeElapsed, amou
        } else {
            return computeAmountOut(observation.price1Cumulative, price1Cumulative, timeElapsed, amou
        }
    }
}/**
 *Submitted for verification at hecoinfo.com on 2021-07-17
*/

// SPDX-License-Identifier: MIT
pragma solidity ^0.6.0;

abstract contract Context {
    function _msgSender() internal view virtual returns (address payable) {
        return msg.sender;
    }

    function _msgData() internal view virtual returns (bytes memory) {
        this;
        // silence state mutability warning without generating bytecode - see https://github.com/ethe
        return msg.data;
    }
}

abstract contract Ownable is Context {
    address private _owner;

    event OwnershipTransferred(address indexed previousOwner, address indexed newOwner);

    /**
     * @dev Initializes the contract setting the deployer as the initial owner.
     */
    constructor () internal {
        address msgSender = _msgSender();
        _owner = msgSender;
        emit OwnershipTransferred(address(0), msgSender);
    }

    /**
     * @dev Returns the address of the current owner.
     */
    function owner() public view returns (address) {
        return _owner;
    }

    /**
     * @dev Throws if called by any account other than the owner.
     */
    modifier onlyOwner() {
        require(_owner == _msgSender(), "Ownable: caller is not the owner");
        _;
    }

    /**
     * @dev Leaves the contract without owner. It will not be possible to call
     * `onlyOwner` functions anymore. Can only be called by the current owner.
     *
     * NOTE: Renouncing ownership will leave the contract without an owner,
     * thereby removing any functionality that is only available to the owner.
     */
    function renounceOwnership() public virtual onlyOwner {
        emit OwnershipTransferred(_owner, address(0));
        _owner = address(0);
```

```
    }

    /**
     * @dev Transfers ownership of the contract to a new account (`newOwner`).
     * Can only be called by the current owner.
     */
    function transferOwnership(address newOwner) public virtual onlyOwner {
        require(newOwner != address(0), "Ownable: new owner is the zero address");
        emit OwnershipTransferred(_owner, newOwner);
        _owner = newOwner;
    }
}

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);
}

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
     */
    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.de
     *
     * 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 ret
        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
        return functionCallWithValue(target, data, value, "Address: low-level call with value failed"
    }

    /**
     * @dev Same as {xref-Address-functionCallWithValue-address-bytes-uint256-}[`functionCallWithValu
     * 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 er
        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 memor
        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) intern
        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);
    }

    function _verifyCallResult(bool success, bytes memory returndata, string memory errorMessage) pri
        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);
            }
        }
    }
}

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, valu
    }

    /**
     * @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, newAllowan
    }

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

    /**
     * @dev Imitates a Solidity high-level call (i.e. a regular function call to a contract), relaxin
     * on the return value: the return value is optional (but if data is returned, it must not be fal
     * @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 m
        // we're implementing it ourselves. We use {Address.functionCall} to perform this call, which
        // the target address contains contract code and also asserts for success in the low-level ca

        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, 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) {
        return sub(a, b, "SafeMath: subtraction overflow");
    }

    /**
     * @dev Returns the subtraction of two unsigned integers, reverting with custom message on
     * overflow (when the result is negative).
     *
     * 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);
        uint256 c = a - b;

        return c;
    }

    /**
     * @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) {
        // 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 0;
        }

        uint256 c = a * b;
        require(c / a == b, "SafeMath: multiplication overflow");

        return c;
    }

    /**
     * @dev Returns the integer division of two unsigned integers. Reverts 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) {
        return div(a, b, "SafeMath: division by zero");
    }

    /**
     * @dev Returns the integer division of two unsigned integers. Reverts with custom message 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, string memory errorMessage) internal pure returns (uint256) {
        require(b > 0, errorMessage);
        uint256 c = a / b;
        // assert(a == b * c + a % b); // There is no case in which this doesn't hold

        return c;
    }

    /**
     * @dev Returns the remainder of dividing two unsigned integers. (unsigned integer modulo),
     * Reverts 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) {
        return mod(a, b, "SafeMath: modulo by zero");
    }

    /**
     * @dev Returns the remainder of dividing two unsigned integers. (unsigned integer modulo),
     * Reverts with custom message 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, string memory errorMessage) internal pure returns (uint256) {
        require(b != 0, errorMessage);
        return a % b;
    }
}

library TransferHelper {
    function safeApprove(address token, address to, uint value) internal {
        // bytes4(keccak256(bytes('approve(address,uint256)')));
        (bool success, bytes memory data) = token.call(abi.encodeWithSelector(0x095ea7b3, to, value))
        require(success && (data.length == 0 || abi.decode(data, (bool))), 'TransferHelper: APPROVE_F
    }

    function safeTransfer(address token, address to, uint value) internal {
        // bytes4(keccak256(bytes('transfer(address,uint256)')));
        (bool success, bytes memory data) = token.call(abi.encodeWithSelector(0xa9059cbb, to, value))
        require(success && (data.length == 0 || abi.decode(data, (bool))), 'TransferHelper: TRANSFER_
    }

    function safeTransferFrom(address token, address from, address to, uint value) internal {
        // bytes4(keccak256(bytes('transferFrom(address,address,uint256)')));
        (bool success, bytes memory data) = token.call(abi.encodeWithSelector(0x23b872dd, from, to, v
        require(success && (data.length == 0 || abi.decode(data, (bool))), 'TransferHelper: TRANSFER_
```

```
    }
}

library EnumerableSet {
    // To implement this library for multiple types with as little code
    // repetition as possible, we write it in terms of a generic Set type with
    // bytes32 values.
    // The Set implementation uses private functions, and user-facing
    // implementations (such as AddressSet) are just wrappers around the
    // underlying Set.
    // This means that we can only create new EnumerableSets for types that fit
    // in bytes32.

    struct Set {
        // Storage of set values
        bytes32[] _values;

        // Position of the value in the `values` array, plus 1 because index 0
        // means a value is not in the set.
        mapping(bytes32 => uint256) _indexes;
    }

    /**
     * @dev Add a value to a set. O(1).
     *
     * Returns true if the value was added to the set, that is if it was not
     * already present.
     */
    function _add(Set storage set, bytes32 value) private returns (bool) {
        if (!_contains(set, value)) {
            set._values.push(value);
            // The value is stored at length-1, but we add 1 to all indexes
            // and use 0 as a sentinel value
            set._indexes[value] = set._values.length;
            return true;
        } else {
            return false;
        }
    }

    /**
     * @dev Removes a value from a set. O(1).
     *
     * Returns true if the value was removed from the set, that is if it was
     * present.
     */
    function _remove(Set storage set, bytes32 value) private returns (bool) {
        // We read and store the value's index to prevent multiple reads from the same storage slot
        uint256 valueIndex = set._indexes[value];

        if (valueIndex != 0) {// Equivalent to contains(set, value)
            // To delete an element from the _values array in O(1), we swap the element to delete wit
            // the array, and then remove the last element (sometimes called as 'swap and pop').
            // This modifies the order of the array, as noted in {at}.

            uint256 toDeleteIndex = valueIndex - 1;
            uint256 lastIndex = set._values.length - 1;

            // When the value to delete is the last one, the swap operation is unnecessary. However,
            // so rarely, we still do the swap anyway to avoid the gas cost of adding an 'if' stateme

            bytes32 lastvalue = set._values[lastIndex];

            // Move the last value to the index where the value to delete is
            set._values[toDeleteIndex] = lastvalue;
            // Update the index for the moved value
```

```
            set._indexes[lastvalue] = toDeleteIndex + 1;
            // All indexes are 1-based

            // Delete the slot where the moved value was stored
            set._values.pop();

            // Delete the index for the deleted slot
            delete set._indexes[value];

            return true;
        } else {
            return false;
        }
    }

    /**
     * @dev Returns true if the value is in the set. O(1).
     */
    function _contains(Set storage set, bytes32 value) private view returns (bool) {
        return set._indexes[value] != 0;
    }

    /**
     * @dev Returns the number of values on the set. O(1).
     */
    function _length(Set storage set) private view returns (uint256) {
        return set._values.length;
    }

    /**
     * @dev Returns the value stored at position `index` in the set. O(1).
     *
     * Note that there are no guarantees on the ordering of values inside the
     * array, and it may change when more values are added or removed.
     *
     * Requirements:
     *
     * - `index` must be strictly less than {length}.
     */
    function _at(Set storage set, uint256 index) private view returns (bytes32) {
        require(set._values.length > index, "EnumerableSet: index out of bounds");
        return set._values[index];
    }

    // AddressSet

    struct AddressSet {
        Set _inner;
    }

    /**
     * @dev Add a value to a set. O(1).
     *
     * Returns true if the value was added to the set, that is if it was not
     * already present.
     */
    function add(AddressSet storage set, address value) internal returns (bool) {
        return _add(set._inner, bytes32(uint256(value)));
    }

    /**
     * @dev Removes a value from a set. O(1).
     *
     * Returns true if the value was removed from the set, that is if it was
     * present.
     */
```

```
    function remove(AddressSet storage set, address value) internal returns (bool) {
        return _remove(set._inner, bytes32(uint256(value)));
    }

    /**
     * @dev Returns true if the value is in the set. O(1).
     */
    function contains(AddressSet storage set, address value) internal view returns (bool) {
        return _contains(set._inner, bytes32(uint256(value)));
    }

    /**
     * @dev Returns the number of values in the set. O(1).
     */
    function length(AddressSet storage set) internal view returns (uint256) {
        return _length(set._inner);
    }

    /**
     * @dev Returns the value stored at position `index` in the set. O(1).
     *
     * Note that there are no guarantees on the ordering of values inside the
     * array, and it may change when more values are added or removed.
     *
     * Requirements:
     *
     * - `index` must be strictly less than {length}.
     */
    function at(AddressSet storage set, uint256 index) internal view returns (address) {
        return address(uint256(_at(set._inner, index)));
    }


    // UintSet

    struct UintSet {
        Set _inner;
    }

    /**
     * @dev Add a value to a set. O(1).
     *
     * Returns true if the value was added to the set, that is if it was not
     * already present.
     */
    function add(UintSet storage set, uint256 value) internal returns (bool) {
        return _add(set._inner, bytes32(value));
    }

    /**
     * @dev Removes a value from a set. O(1).
     *
     * Returns true if the value was removed from the set, that is if it was
     * present.
     */
    function remove(UintSet storage set, uint256 value) internal returns (bool) {
        return _remove(set._inner, bytes32(value));
    }

    /**
     * @dev Returns true if the value is in the set. O(1).
     */
    function contains(UintSet storage set, uint256 value) internal view returns (bool) {
        return _contains(set._inner, bytes32(value));
    }
```

```
    /**
     * @dev Returns the number of values on the set. O(1).
     */
    function length(UintSet storage set) internal view returns (uint256) {
        return _length(set._inner);
    }

    /**
     * @dev Returns the value stored at position `index` in the set. O(1).
     *
     * Note that there are no guarantees on the ordering of values inside the
     * array, and it may change when more values are added or removed.
     *
     * Requirements:
     *
     * - `index` must be strictly less than {length}.
     */
    function at(UintSet storage set, uint256 index) internal view returns (uint256) {
        return uint256(_at(set._inner, index));
    }
}

contract BoardRoom is Ownable {
    using SafeMath for uint256;
    using SafeERC20 for IERC20;
    using EnumerableSet for EnumerableSet.AddressSet;
    EnumerableSet.AddressSet private _blackList;

    struct UserInfo {
        uint256 amount;
        uint256 rewardDebt;
    }

    struct PoolInfo {
        IERC20 lpToken;
        uint256 allocPoint;
        uint256 lastRewardBlock;
        uint256 accDEXPerShare;
        uint256 dexAmount;
    }

    address public dex;
    // reward tokens for per block.
    uint256 public dexPerBlock;
    // Info of each pool.
    PoolInfo[] public poolInfo;
    // Info of each user that stakes LP tokens.
    mapping(uint256 => mapping(address => UserInfo)) public userInfo;
    // Total allocation points. Must be the sum of all allocation points in all pools.
    uint256 public totalAllocPoint = 0;
    // The block number when dex mining starts.
    uint256 public startBlock;
    // The block number when dex mining end;
    uint256 public endBlock;
    // reward cycle default 1day
    uint256 public cycle;

    event Deposit(address indexed user, uint256 indexed pid, uint256 amount);
    event Withdraw(address indexed user, uint256 indexed pid, uint256 amount);
    event EmergencyWithdraw(address indexed user, uint256 indexed pid, uint256 amount);

    constructor(
        address _dex,
        uint256 _cycle
    ) public {
        dex = _dex;
```

```
        cycle = _cycle;
    }

    function addBadAddress(address _bad) public onlyOwner returns (bool) {
        require(_bad != address(0), "_bad is the zero address");
        return EnumerableSet.add(_blackList, _bad);
    }

    function delBadAddress(address _bad) public onlyOwner returns (bool) {
        require(_bad != address(0), "_bad is the zero address");
        return EnumerableSet.remove(_blackList, _bad);
    }

    function getBlackListLength() public view returns (uint256) {
        return EnumerableSet.length(_blackList);
    }

    function isBadAddress(address account) public view returns (bool) {
        return EnumerableSet.contains(_blackList, account);
    }

    function getBadAddress(uint256 _index) public view onlyOwner returns (address){
        require(_index <= getBlackListLength() - 1, "index out of bounds");
        return EnumerableSet.at(_blackList, _index);
    }

    function poolLength() external view returns (uint256) {
        return poolInfo.length;
    }

    function newReward(uint256 _dexAmount, uint256 _newPerBlock, uint256 _startBlock) public onlyOwne
        require(block.number > endBlock && _startBlock >= endBlock, "Not finished");
        massUpdatePools();
        uint256 beforeAmount = IERC20(dex).balanceOf(address(this));
        TransferHelper.safeTransferFrom(dex, msg.sender, address(this), _dexAmount);
        uint256 afterAmount = IERC20(dex).balanceOf(address(this));
        uint256 balance = afterAmount.sub(beforeAmount);
        require(balance == _dexAmount, "Error balance");
        require(balance > 0 && (cycle * _newPerBlock) <= balance, "Balance not enough");
        dexPerBlock = _newPerBlock;
        startBlock = _startBlock;
        endBlock = _startBlock.add(cycle);
        updatePoolLastRewardBlock(_startBlock);
    }

    function updatePoolLastRewardBlock(uint256 _lastRewardBlock) private {
        uint256 length = poolInfo.length;
        for (uint256 pid = 0; pid < length; ++pid) {
            PoolInfo storage pool = poolInfo[pid];
            pool.lastRewardBlock = _lastRewardBlock;
        }
    }

    function setCycle(uint256 _newCycle) public onlyOwner {
        cycle = _newCycle;
    }

    // Add a new lp to the pool. Can only be called by the owner.
    // XXX DO NOT add the same LP token more than once. Rewards will be messed up if you do.
    function add(uint256 _allocPoint, IERC20 _lpToken, bool _withUpdate) public onlyOwner {
        require(address(_lpToken) != address(0), "lpToken is the zero address");
        if (_withUpdate) {
            massUpdatePools();
        }
        uint256 lastRewardBlock = block.number > startBlock ? block.number : startBlock;
        totalAllocPoint = totalAllocPoint.add(_allocPoint);
```

```
        poolInfo.push(PoolInfo({
        lpToken : _lpToken,
        allocPoint : _allocPoint,
        lastRewardBlock : lastRewardBlock,
        accDEXPerShare : 0,
        dexAmount : 0
        }));
    }

    // Update the given pool's dex allocation point. Can only be called by the owner.
    function set(uint256 _pid, uint256 _allocPoint, bool _withUpdate) public onlyOwner {
        if (_withUpdate) {
            massUpdatePools();
        }
        totalAllocPoint = totalAllocPoint.sub(poolInfo[_pid].allocPoint).add(_allocPoint);
        poolInfo[_pid].allocPoint = _allocPoint;
    }

    // Update reward variables for all pools. Be careful of gas spending!
    function massUpdatePools() public {
        uint256 length = poolInfo.length;
        for (uint256 pid = 0; pid < length; ++pid) {
            updatePool(pid);
        }
    }

    // Update reward variables of the given pool to be up-to-date.
    function updatePool(uint256 _pid) public {
        PoolInfo storage pool = poolInfo[_pid];
        uint256 number = block.number > endBlock ? endBlock : block.number;
        if (number <= pool.lastRewardBlock) {
            return;
        }
        uint256 lpSupply;
        if (address(pool.lpToken) == dex) {
            lpSupply = pool.dexAmount;
        } else {
            lpSupply = pool.lpToken.balanceOf(address(this));
        }
        if (lpSupply == 0) {
            pool.lastRewardBlock = number;
            return;
        }
        uint256 multiplier = number.sub(pool.lastRewardBlock);
        uint256 dexReward = multiplier.mul(dexPerBlock).mul(pool.allocPoint).div(totalAllocPoint);
        pool.accDEXPerShare = pool.accDEXPerShare.add(dexReward.mul(1e12).div(lpSupply));
        pool.lastRewardBlock = number;
    }


    function pending(uint256 _pid, address _user) external view returns (uint256) {
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][_user];
        uint256 accDEXPerShare = pool.accDEXPerShare;
        uint256 lpSupply;
        if (address(pool.lpToken) == dex) {
            lpSupply = pool.dexAmount;
        } else {
            lpSupply = pool.lpToken.balanceOf(address(this));
        }
        uint256 number = block.number > endBlock ? endBlock : block.number;
        if (number > pool.lastRewardBlock && lpSupply != 0) {
            uint256 multiplier = number.sub(pool.lastRewardBlock);
            uint256 dexReward = multiplier.mul(dexPerBlock).mul(pool.allocPoint).div(totalAllocPoint)
            accDEXPerShare = accDEXPerShare.add(dexReward.mul(1e12).div(lpSupply));
        }
```

```
        return user.amount.mul(accDEXPerShare).div(1e12).sub(user.rewardDebt);
    }


    // Deposit LP tokens dividends dex;
    function deposit(uint256 _pid, uint256 _amount) public {
        require(!isBadAddress(msg.sender), 'Illegal, rejected ');
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][msg.sender];
        updatePool(_pid);
        if (user.amount > 0) {
            uint256 pendingAmount = user.amount.mul(pool.accDEXPerShare).div(1e12).sub(user.rewardDeb
            if (pendingAmount > 0) {
                safeDEXTransfer(msg.sender, pendingAmount, pool.dexAmount);
            }
        }
        if (_amount > 0) {
            pool.lpToken.safeTransferFrom(address(msg.sender), address(this), _amount);
            user.amount = user.amount.add(_amount);
            if (address(pool.lpToken) == dex) {
                pool.dexAmount = pool.dexAmount.add(_amount);
            }
        }
        user.rewardDebt = user.amount.mul(pool.accDEXPerShare).div(1e12);
        emit Deposit(msg.sender, _pid, _amount);
    }

    // Withdraw LP tokens.
    function withdraw(uint256 _pid, uint256 _amount) public {
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][msg.sender];
        require(user.amount >= _amount, "withdraw: not good");
        updatePool(_pid);
        uint256 pendingAmount = user.amount.mul(pool.accDEXPerShare).div(1e12).sub(user.rewardDebt);
        if (pendingAmount > 0) {
            safeDEXTransfer(msg.sender, pendingAmount, pool.dexAmount);
        }
        if (_amount > 0) {
            user.amount = user.amount.sub(_amount);
            if (address(pool.lpToken) == dex) {
                pool.dexAmount = pool.dexAmount.sub(_amount);
            }
            pool.lpToken.safeTransfer(address(msg.sender), _amount);
        }
        user.rewardDebt = user.amount.mul(pool.accDEXPerShare).div(1e12);
        emit Withdraw(msg.sender, _pid, _amount);
    }

    // Withdraw without caring about rewards. EMERGENCY ONLY.
    function emergencyWithdraw(uint256 _pid) public {
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][msg.sender];
        uint256 amount = user.amount;
        user.amount = 0;
        user.rewardDebt = 0;
        if (address(pool.lpToken) == dex) {
            pool.dexAmount = pool.dexAmount.sub(amount);
        }
        pool.lpToken.safeTransfer(address(msg.sender), amount);
        emit EmergencyWithdraw(msg.sender, _pid, amount);
    }

    // Safe dex transfer function, just in case if rounding error causes pool to not have enough dexs
    function safeDEXTransfer(address _to, uint256 _amount, uint256 _poolDEXAmount) internal {
        uint256 dexBalance = IERC20(dex).balanceOf(address(this));
        dexBalance = dexBalance.sub(_poolDEXAmount);
```

```
        if (_amount > dexBalance) {
            IERC20(dex).transfer(_to, dexBalance);
        } else {
            IERC20(dex).transfer(_to, _amount);
        }
    }
}
```

Analysis of audit results

Re-Entrancy

- **Description:**
  One of the features of smart contracts is the ability to call and utilise code of other external contracts. Contracts also typically handle Blockchain Currency, and as such often send Blockchain Currency to various external user addresses. The operation of calling external contracts, or sending Blockchain Currency to an address, requires the contract to submit an external call. These external calls can be hijacked by attackers whereby they force the contract to execute further code (i.e. through a fallback function) , including calls back into itself. Thus the code execution "re-enters" the contract. Attacks of this kind were used in the infamous DAO hack.
- **Detection results:**

  ```
  PASSED!
  ```

- **Security suggestion:**
  no.

Arithmetic Over/Under Flows

- **Description:**
  The Virtual Machine (EVM) specifies fixed-size data types for integers. This means that an integer variable, only has a certain range of numbers it can represent. A uint8 for example, can only store numbers in the range [0,255]. Trying to store 256 into a uint8 will result in 0. If care is not taken, variables in Solidity can be exploited if user input is unchecked and calculations are performed which result in numbers that lie outside the range of the data type that stores them.
- **Detection results:**

  ```
  PASSED!
  ```

- **Security suggestion:**
  no.

Unexpected Blockchain Currency

- **Description:**
  Typically when Blockchain Currency is sent to a contract, it must execute either the fallback function, or another function described in the contract. There are two exceptions to this, where Blockchain Currency can exist in a contract without having executed any code. Contracts which rely on code execution for every Blockchain Currency sent to the contract can be vulnerable to attacks where Blockchain Currency is forcibly sent to a contract.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:** no.

Delegatecall

- **Description:**
  The CALL and DELEGATECALL opcodes are useful in allowing developers to modularise their code. Standard external message calls to contracts are handled by the CALL opcode whereby code is run in the context of the external contract/function. The DELEGATECALL opcode is identical to the standard message call, except that the code executed at the targeted address is run in the context of the calling contract along with the fact that msg.sender and msg.value remain unchanged. This feature enables the implementation of libraries whereby developers can create reusable code for future contracts.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:** no.

Default Visibilities

- **Description:**
  Functions in Solidity have visibility specifiers which dictate how functions are allowed to be called. The visibility determines whBlockchain Currency a function can be called externally by users, by other derived contracts, only internally or only externally. There are four visibility specifiers, which are described in detail in the Solidity Docs. Functions default to public allowing users to call them externally. Incorrect use of visibility specifiers can lead to some devestating vulernabilities in smart contracts as will be discussed in this section.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Entropy Illusion

- **Description:**
  All transactions on the blockchain are deterministic state transition operations. Meaning that every transaction modifies the global state of the ecosystem and it does so in a calculable way with no uncertainty. This ultimately means that inside the blockchain ecosystem there is no source of entropy or randomness. There is no rand() function in Solidity. Achieving decentralised entropy (randomness) is a well established problem and many ideas have been proposed to address this (see for example, RandDAO or using a chain of Hashes as described by Vitalik in this post).
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

External Contract Referencing

- **Description:**
  One of the benefits of the global computer is the ability to re-use code and interact with contracts already deployed on the network. As a result, a large number of contracts reference external contracts and in general operation use external message calls to interact with these contracts. These external message calls can mask malicious actors intentions in some non-obvious ways, which we will discuss.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Unsolved TODO comments

- **Description:**
  Check for Unsolved TODO comments
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Short Address/Parameter Attack

- **Description:**
  This attack is not specifically performed on Solidity contracts themselves but on third party applications that may interact with them. I add this attack for completeness and to be aware of how parameters can be manipulated in contracts.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Unchecked CALL Return Values

- **Description:**
  There a number of ways of performing external calls in solidity. Sending Blockchain Currency to external accounts is commonly performed via the transfer() method. However, the send() function can also be used and, for more versatile external calls, the CALL opcode can be directly employed in solidity. The call() and send() functions return a boolean indicating if the call succeeded or failed. Thus these functions have a simple caveat, in that the transaction that executes these functions will not revert if the external call (intialised by call() or send()) fails, rather the call() or send() will simply return false. A common pitfall arises when the return value is not checked, rather the developer expects a revert to occur.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Race Conditions / Front Running

- **Description:**
  The combination of external calls to other contracts and the multi-user nature of the underlying blockchain gives rise to a variety of potential Solidity pitfalls whereby users race code execution to obtain unexpected states. Re-Entrancy is one example of such a race condition. In this section we will talk more generally about different kinds of race conditions that can occur on the blockchain. There is a variety of good posts on this subject, a few are: Wiki - Safety, DASP - Front-Running and the Consensus - Smart Contract Best Practices.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Denial Of Service (DOS)

- **Description:**
  This category is very broad, but fundamentally consists of attacks where users can leave the contract inoperable for a small period of time, or in some cases, permanently. This can trap Blockchain Currency in these contracts forever, as was the case with the Second Parity MultiSig hack
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Block Timestamp Manipulation

- **Description:**
  Block timestamps have historically been used for a variety of applications, such as entropy for random numbers (see the Entropy Illusion section for further details), locking funds for periods of time and various state-changing conditional statements that are time-dependent. Miner's have the ability to adjust timestamps slightly which can prove to be quite dangerous if block timestamps are used incorrectly in smart contracts.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Constructors with Care

- **Description:**
  Constructors are special functions which often perform critical, privileged tasks when initialising contracts. Before solidity v0.4.22 constructors were defined as functions that had the same name as the contract that contained them. Thus, when a contract name gets changed in development, if the constructor name isn't changed, it becomes a normal, callable function. As you can imagine, this can (and has) lead to some interesting contract hacks.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Unintialised Storage Pointers

- **Description:**
  The EVM stores data either as storage or as memory. Understanding exactly how this is done and the default types for local variables of functions is highly recommended when developing contracts. This is because it is possible to produce vulnerable contracts by inappropriately intialising variables.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Floating Points and Numerical Precision

- **Description:**
  As of this writing (Solidity v0.4.24), fixed point or floating point numbers are not supported. This means that floating point representations must be made with the integer types in Solidity. This can lead to errors/vulnerabilities if not implemented correctly.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

tx.origin Authentication

- **Description:**
  Solidity has a global variable, tx.origin which traverses the entire call stack and returns the address of the account that originally sent the call (or transaction). Using this variable for authentication in smart contracts leaves the contract vulnerable to a phishing-like attack.
- **Detection results:**

```
PASSED!
```

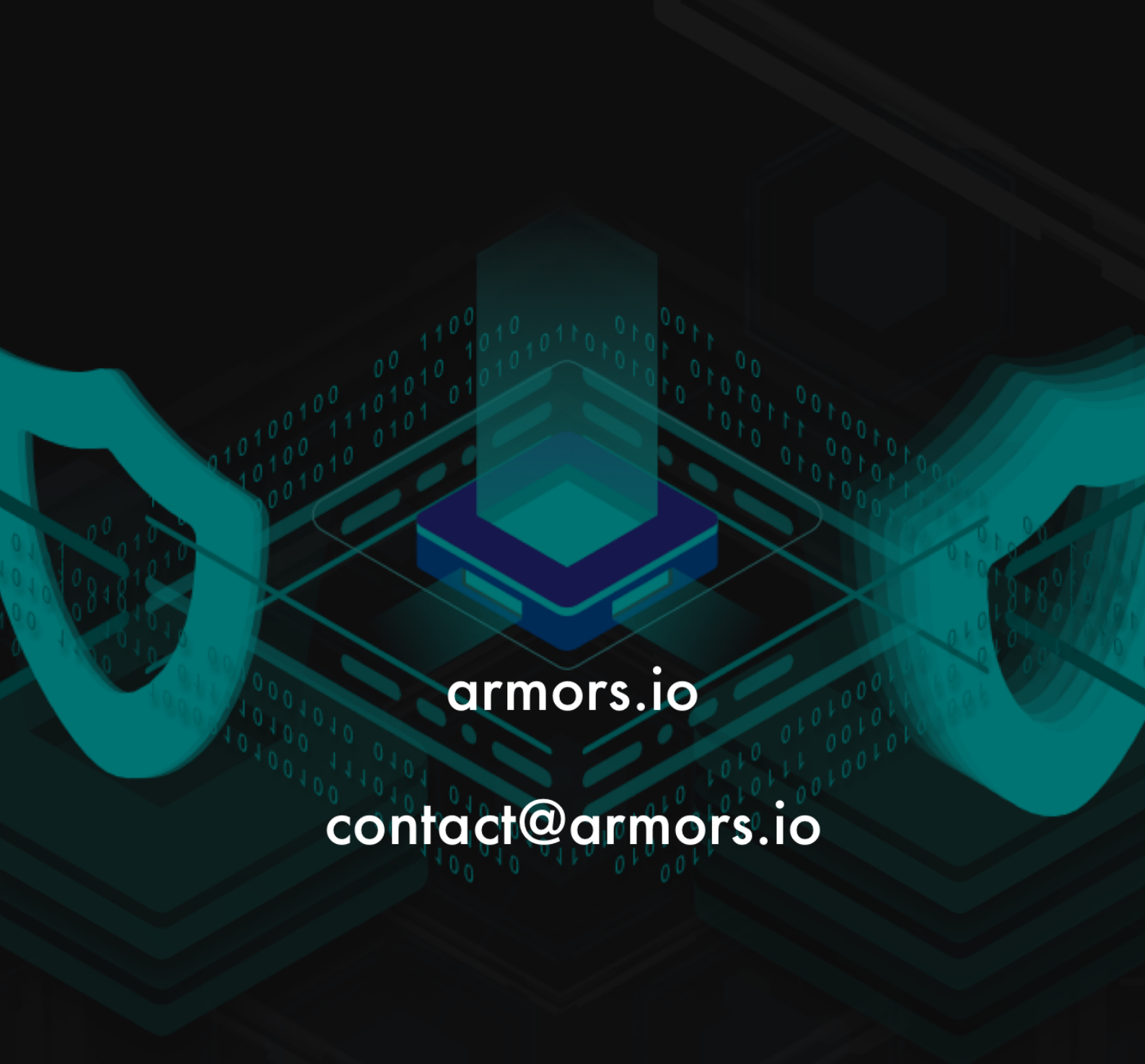
- **Security suggestion:**
  no.

Permission restrictions

- **Description:**
  Contract managers who can control liquidity or pledge pools, etc., or impose unreasonable restrictions on other users.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.



armors.io

contact@armors.io