/\*\*

\*Submitted for verification at Etherscan.io on 2020-05-05

\*/

// File: contracts/interfaces/IUniswapV2Pair.sol

pragma solidity >=0.5.0;

interface IUniswapV2Pair {

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, bytes32 s) external;

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 blockTimestampLast);

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;

}

// File: contracts/interfaces/IUniswapV2ERC20.sol

pragma solidity >=0.5.0;

interface IUniswapV2ERC20 {

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, bytes32 s) external;

}

// File: contracts/libraries/SafeMath.sol

pragma solidity =0.5.16;

// a library for performing overflow-safe math, courtesy of DappHub (https://github.com/dapphub/ds-math)

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');

}

}

// File: contracts/UniswapV2ERC20.sol

pragma solidity =0.5.16;

contract UniswapV2ERC20 is IUniswapV2ERC20 {

using SafeMath for uint;

string public constant name = 'Uniswap V2';

string public constant symbol = 'UNI-V2';

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 = 0x6e71edae12b1b97f4d1f60370fef10105fa2faae0126114a169c64845d6126c9;

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 verifyingContract)'),

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, bytes32 s) external {

require(deadline >= block.timestamp, 'UniswapV2: EXPIRED');

bytes32 digest = keccak256(

abi.encodePacked(

'\x19\x01',

DOMAIN\_SEPARATOR,

keccak256(abi.encode(PERMIT\_TYPEHASH, owner, spender, value, nonces[owner]++, deadline))

)

);

address recoveredAddress = ecrecover(digest, v, r, s);

require(recoveredAddress != address(0) && recoveredAddress == owner, 'UniswapV2: INVALID\_SIGNATURE');

\_approve(owner, spender, value);

}

}

// File: contracts/libraries/Math.sol

pragma solidity =0.5.16;

// a library for performing various math operations

library Math {

function min(uint x, uint y) internal pure returns (uint z) {

z = x < y ? x : y;

}

// babylonian method (https://en.wikipedia.org/wiki/Methods\_of\_computing\_square\_roots#Babylonian\_method)

function sqrt(uint y) internal pure returns (uint z) {

if (y > 3) {

z = y;

uint x = y / 2 + 1;

while (x < z) {

z = x;

x = (y / x + x) / 2;

}

} else if (y != 0) {

z = 1;

}

}

}

// File: contracts/libraries/UQ112x112.sol

pragma solidity =0.5.16;

// a library for handling binary fixed point numbers (https://en.wikipedia.org/wiki/Q\_(number\_format))

// range: [0, 2\*\*112 - 1]

// resolution: 1 / 2\*\*112

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);

}

}

// File: contracts/interfaces/IERC20.sol

pragma solidity >=0.5.0;

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);

}

// File: contracts/interfaces/IUniswapV2Factory.sol

pragma solidity >=0.5.0;

interface IUniswapV2Factory {

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

function feeTo() external view returns (address);

function feeToSetter() external view returns (address);

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;

}

// File: contracts/interfaces/IUniswapV2Callee.sol

pragma solidity >=0.5.0;

interface IUniswapV2Callee {

function uniswapV2Call(address sender, uint amount0, uint amount1, bytes calldata data) external;

}

// File: contracts/UniswapV2Pair.sol

pragma solidity =0.5.16;

contract UniswapV2Pair is IUniswapV2Pair, UniswapV2ERC20 {

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 event

uint private unlocked = 1;

modifier lock() {

require(unlocked == 1, 'UniswapV2: LOCKED');

unlocked = 0;

\_;

unlocked = 1;

}

function getReserves() public view returns (uint112 \_reserve0, uint112 \_reserve1, uint32 \_blockTimestampLast) {

\_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))), 'UniswapV2: 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, 'UniswapV2: 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), 'UniswapV2: 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 = IUniswapV2Factory(factory).feeTo();

feeOn = feeTo != address(0);

uint \_kLast = kLast; // gas savings

if (feeOn) {

if (\_kLast != 0) {

uint rootK = Math.sqrt(uint(\_reserve0).mul(\_reserve1));

uint rootKLast = Math.sqrt(\_kLast);

if (rootK > rootKLast) {

uint numerator = totalSupply.mul(rootK.sub(rootKLast));

uint denominator = rootK.mul(5).add(rootKLast);

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 checks

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 = Math.sqrt(amount0.mul(amount1)).sub(MINIMUM\_LIQUIDITY);

\_mint(address(0), MINIMUM\_LIQUIDITY); // permanently lock the first MINIMUM\_LIQUIDITY tokens

} else {

liquidity = Math.min(amount0.mul(\_totalSupply) / \_reserve0, amount1.mul(\_totalSupply) / \_reserve1);

}

require(liquidity > 0, 'UniswapV2: 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 checks

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, 'UniswapV2: 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 checks

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

require(amount0Out > 0 || amount1Out > 0, 'UniswapV2: INSUFFICIENT\_OUTPUT\_AMOUNT');

(uint112 \_reserve0, uint112 \_reserve1,) = getReserves(); // gas savings

require(amount0Out < \_reserve0 && amount1Out < \_reserve1, 'UniswapV2: 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, 'UniswapV2: 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) IUniswapV2Callee(to).uniswapV2Call(msg.sender, amount0Out, amount1Out, data);

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, 'UniswapV2: INSUFFICIENT\_INPUT\_AMOUNT');

{ // scope for reserve{0,1}Adjusted, avoids stack too deep errors

uint balance0Adjusted = balance0.mul(1000).sub(amount0In.mul(3));

uint balance1Adjusted = balance1.mul(1000).sub(amount1In.mul(3));

require(balance0Adjusted.mul(balance1Adjusted) >= uint(\_reserve0).mul(\_reserve1).mul(1000\*\*2), 'UniswapV2: K');

}

\_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)), reserve0, reserve1);

}

}