1 Removal 1 Addition

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.11;

interface erc20 {
    function totalSupply() external view returns (uint256);
    function transfer(address recipient, uint amount) external returns (bool);
    function decimals() external view returns (uint8);
    function symbol() external view returns (string memory);
    function balanceOf(address) external view returns (uint);
    function transferFrom(address sender, address recipient, uint amount) external returns (bool);
    function approve(address spender, uint value) external returns (bool);
}

library Math {
    function min(uint a, uint b) internal pure returns (uint) {
        return a < b ? a : b;
    }
    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;
        }
    }
}

interface IBaseV1Callee {
    function hook(address sender, uint amount0, uint amount1, bytes calldata data) external;
}

// Base V1 Fees contract is used as a 1:1 pair relationship to split out fees, this ensures that the curve does not need to be modified for LP shares
contract BaseV1Fees {

    address internal immutable pair; // The pair it is bonded to
    address internal immutable token0; // token0 of pair, saved localy and statically for gas optimization
    address internal immutable token1; // Token1 of pair, saved localy and statically for gas optimization

    constructor(address _token0, address _token1) {
        pair = msg.sender;
        token0 = _token0;
        token1 = _token1;
```

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.11;

interface erc20 {
    function totalSupply() external view returns (uint256);
    function transfer(address recipient, uint amount) external returns (bool);
    function decimals() external view returns (uint8);
    function symbol() external view returns (string memory);
    function balanceOf(address) external view returns (uint);
    function transferFrom(address sender, address recipient, uint amount) external returns (bool);
    function approve(address spender, uint value) external returns (bool);
}

library Math {
    function min(uint a, uint b) internal pure returns (uint) {
        return a < b ? a : b;
    }
    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;
        }
    }
}

interface IBaseV1Callee {
    function hook(address sender, uint amount0, uint amount1, bytes calldata data) external;
}

// Base V1 Fees contract is used as a 1:1 pair relationship to split out fees, this ensures that the curve does not need to be modified for LP shares
contract BaseV1Fees {

    address internal immutable pair; // The pair it is bonded to
    address internal immutable token0; // token0 of pair, saved localy and statically for gas optimization
    address internal immutable token1; // Token1 of pair, saved localy and statically for gas optimization

    constructor(address _token0, address _token1) {
        pair = msg.sender;
        token0 = _token0;
        token1 = _token1;
```

```
    }

    function _safeTransfer(address token,address to,uint256 value) internal {
        require(token.code.length > 0);
        (bool success, bytes memory data) =
        token.call(abi.encodeWithSelector(erc20.transfer.selector, to, value));
        require(success && (data.length == 0 || abi.decode(data, (bool))));
    }

    // Allow the pair to transfer fees to users
    function claimFeesFor(address recipient, uint amount0, uint amount1) external {
        require(msg.sender == pair);
        if (amount0 > 0) _safeTransfer(token0, recipient, amount0);
        if (amount1 > 0) _safeTransfer(token1, recipient, amount1);
    }

}

// The base pair of pools, either stable or volatile
contract BaseV1Pair {

    string public name;
    string public symbol;
    uint8 public constant decimals = 18;

    // Used to denote stable or volatile pair, not immutable since construction happens in the initialize method for CREATE2 deterministic addresses
    bool public immutable stable;

    uint public totalSupply = 0;

    mapping(address => mapping (address => uint)) public allowance;
    mapping(address => uint) public balanceOf;

    bytes32 internal DOMAIN_SEPARATOR;
    // keccak256("Permit(address owner,address spender,uint256 value,uint256 nonce,uint256 deadline)");
    bytes32 internal constant PERMIT_TYPEHASH = 0x6e71edae12b1b97f4d1f60370fef10105fa2faae0126114a169c64845d6126c9;
    mapping(address => uint) public nonces;

    uint internal constant MINIMUM_LIQUIDITY = 10**3;

    address public immutable token0;
    address public immutable token1;
    address public immutable fees;
    address immutable factory;

    // Structure to capture time period obervations every 30 minutes, used for local oracles
    struct Observation {
        uint timestamp;
        uint reserve0Cumulative;
        uint reserve1Cumulative;
    }

    // Capture oracle reading every 30 minutes
    uint constant periodSize = 1800;

```

```
    }

    function _safeTransfer(address token,address to,uint256 value) internal {
        require(token.code.length > 0);
        (bool success, bytes memory data) =
        token.call(abi.encodeWithSelector(erc20.transfer.selector, to, value));
        require(success && (data.length == 0 || abi.decode(data, (bool))));
    }

    // Allow the pair to transfer fees to users
    function claimFeesFor(address recipient, uint amount0, uint amount1) external {
        require(msg.sender == pair);
        if (amount0 > 0) _safeTransfer(token0, recipient, amount0);
        if (amount1 > 0) _safeTransfer(token1, recipient, amount1);
    }

}

// The base pair of pools, either stable or volatile
contract BaseV1Pair {

    string public name;
    string public symbol;
    uint8 public constant decimals = 18;

    // Used to denote stable or volatile pair, not immutable since construction happens in the initialize method for CREATE2 deterministic addresses
    bool public immutable stable;

    uint public totalSupply = 0;

    mapping(address => mapping (address => uint)) public allowance;
    mapping(address => uint) public balanceOf;

    bytes32 internal DOMAIN_SEPARATOR;
    // keccak256("Permit(address owner,address spender,uint256 value,uint256 nonce,uint256 deadline)");
    bytes32 internal constant PERMIT_TYPEHASH = 0x6e71edae12b1b97f4d1f60370fef10105fa2faae0126114a169c64845d6126c9;
    mapping(address => uint) public nonces;

    uint internal constant MINIMUM_LIQUIDITY = 10**3;

    address public immutable token0;
    address public immutable token1;
    address public immutable fees;
    address immutable factory;

    // Structure to capture time period obervations every 30 minutes, used for local oracles
    struct Observation {
        uint timestamp;
        uint reserve0Cumulative;
        uint reserve1Cumulative;
    }

    // Capture oracle reading every 30 minutes
    uint constant periodSize = 1800;
```

```
    Observation[] public observations;

    uint internal immutable decimals0;
    uint internal immutable decimals1;

    uint public reserve0;
    uint public reserve1;
    uint public blockTimestampLast;

    uint public reserve0CumulativeLast;
    uint public reserve1CumulativeLast;

    // index0 and index1 are used to accumulate fees, this is split out from normal trades to keep the swap "clean"
    // this further allows LP holders to easily claim fees for tokens they have/staked
    uint public index0 = 0;
    uint public index1 = 0;

    // position assigned to each LP to track their current index0 & index1 vs the global position
    mapping(address => uint) public supplyIndex0;
    mapping(address => uint) public supplyIndex1;

    // tracks the amount of unclaimed, but claimable tokens off of fees for token0 and token1
    mapping(address => uint) public claimable0;
    mapping(address => uint) public claimable1;

    event Fees(address indexed sender, uint amount0, uint amount1);
    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(uint reserve0, uint reserve1);
    event Claim(address indexed sender, address indexed recipient, uint amount0, uint amount1);

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

    constructor() {
        factory = msg.sender;
        (address _token0, address _token1, bool _stable) = BaseV1Factory(msg.sender).getInitializable();
        (token0, token1, stable) = (_token0, _token1, _stable);
        fees = address(new BaseV1Fees(_token0, _token1));
        if (_stable) {
            name = string(abi.encodePacked("StableV1 AMM - ", erc20(_token0).symbol(), "/", erc20(_token1).symbol()));
            symbol = string(abi.encodePacked("sAMM-", erc20(_token0).symbol(), "/", erc20(_token1).symbol()));
```


```
    Observation[] public observations;

    uint internal immutable decimals0;
    uint internal immutable decimals1;

    uint public reserve0;
    uint public reserve1;
    uint public blockTimestampLast;

    uint public reserve0CumulativeLast;
    uint public reserve1CumulativeLast;

    // index0 and index1 are used to accumulate fees, this is split out from normal trades to keep the swap "clean"
    // this further allows LP holders to easily claim fees for tokens they have/staked
    uint public index0 = 0;
    uint public index1 = 0;

    // position assigned to each LP to track their current index0 & index1 vs the global position
    mapping(address => uint) public supplyIndex0;
    mapping(address => uint) public supplyIndex1;

    // tracks the amount of unclaimed, but claimable tokens off of fees for token0 and token1
    mapping(address => uint) public claimable0;
    mapping(address => uint) public claimable1;

    event Fees(address indexed sender, uint amount0, uint amount1);
    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(uint reserve0, uint reserve1);
    event Claim(address indexed sender, address indexed recipient, uint amount0, uint amount1);

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

    constructor() {
        factory = msg.sender;
        (address _token0, address _token1, bool _stable) = BaseV1Factory(msg.sender).getInitializable();
        (token0, token1, stable) = (_token0, _token1, _stable);
        fees = address(new BaseV1Fees(_token0, _token1));
        if (_stable) {
            name = string(abi.encodePacked("StableV1 AMM - ", erc20(_token0).symbol(), "/", erc20(_token1).symbol()));
            symbol = string(abi.encodePacked("sAMM-", erc20(_token0).symbol(), "/", erc20(_token1).symbol()));
```

```
        } else {
            name = string(abi.encodePacked("Volatil
eV1 AMM - ", erc20(_token0).symbol(), "/", erc20(_t
oken1).symbol()));
            symbol = string(abi.encodePacked("vAMM-
", erc20(_token0).symbol(), "/", erc20(_token1).sym
bol()));
        }

        decimals0 = 10**erc20(_token0).decimals();
        decimals1 = 10**erc20(_token1).decimals();

        observations.push(Observation(block.timesta
mp, 0, 0));
    }

    // simple re-entrancy check
    uint internal _unlocked = 1;
    modifier lock() {
        require(_unlocked == 1);
        _unlocked = 2;
        _;
        _unlocked = 1;
    }

    function observationLength() external view retu
rns (uint) {
        return observations.length;
    }

    function lastObservation() public view returns
 (Observation memory) {
        return observations[observations.length-1];
    }

    function metadata() external view returns (uint
dec0, uint dec1, uint r0, uint r1, bool st, address
t0, address t1) {
        return (decimals0, decimals1, reserve0, res
erve1, stable, token0, token1);
    }

    function tokens() external view returns (addres
s, address) {
        return (token0, token1);
    }

    // claim accumulated but unclaimed fees (viewab
le via claimable0 and claimable1)
    function claimFees() external returns (uint cla
imed0, uint claimed1) {
        return claimFeesFor(msg.sender);
    }

    function claimFeesFor(address recipient) public
lock returns (uint claimed0, uint claimed1) {
        _updateFor(recipient);

        claimed0 = claimable0[recipient];
        claimed1 = claimable1[recipient];

        claimable0[recipient] = 0;
        claimable1[recipient] = 0;

        BaseV1Fees(fees).claimFeesFor(recipient, cl
aimed0, claimed1);

        emit Claim(msg.sender, recipient, claimed0,
claimed1);
    }

```


```
        } else {
            name = string(abi.encodePacked("Volatil
eV1 AMM - ", erc20(_token0).symbol(), "/", erc20(_t
oken1).symbol()));
            symbol = string(abi.encodePacked("vAMM-
", erc20(_token0).symbol(), "/", erc20(_token1).sym
bol()));
        }

        decimals0 = 10**erc20(_token0).decimals();
        decimals1 = 10**erc20(_token1).decimals();

        observations.push(Observation(block.timesta
mp, 0, 0));
    }

    // simple re-entrancy check
    uint internal _unlocked = 1;
    modifier lock() {
        require(_unlocked == 1);
        _unlocked = 2;
        _;
        _unlocked = 1;
    }

    function observationLength() external view retu
rns (uint) {
        return observations.length;
    }

    function lastObservation() public view returns
 (Observation memory) {
        return observations[observations.length-1];
    }

    function metadata() external view returns (uint
dec0, uint dec1, uint r0, uint r1, bool st, address
t0, address t1) {
        return (decimals0, decimals1, reserve0, res
erve1, stable, token0, token1);
    }

    function tokens() external view returns (addres
s, address) {
        return (token0, token1);
    }

    // claim accumulated but unclaimed fees (viewab
le via claimable0 and claimable1)
    function claimFees() external returns (uint cla
imed0, uint claimed1) {
        return claimFeesFor(msg.sender);
    }

    function claimFeesFor(address recipient) public
lock returns (uint claimed0, uint claimed1) {
        _updateFor(recipient);

        claimed0 = claimable0[recipient];
        claimed1 = claimable1[recipient];

        claimable0[recipient] = 0;
        claimable1[recipient] = 0;

        BaseV1Fees(fees).claimFeesFor(recipient, cl
aimed0, claimed1);

        emit Claim(msg.sender, recipient, claimed0,
claimed1);
    }

```

```
    // Accrue fees on token0
    function _update0(uint amount) internal {
        _safeTransfer(token0, fees, amount); // transfer the fees out to BaseV1Fees
        uint256 _ratio = amount * 1e18 / totalSupply; // 1e18 adjustment is removed during claim
        if (_ratio > 0) {
            index0 += _ratio;
        }
        emit Fees(msg.sender, amount, 0);
    }

    // Accrue fees on token1
    function _update1(uint amount) internal {
        _safeTransfer(token1, fees, amount);
        uint256 _ratio = amount * 1e18 / totalSupply;
        if (_ratio > 0) {
            index1 += _ratio;
        }
        emit Fees(msg.sender, 0, amount);
    }

    // this function MUST be called on any balance changes, otherwise can be used to infinitely claim fees
    // Fees are segregated from core funds, so fees can never put liquidity at risk
    function _updateFor(address recipient) internal {
        uint _supplied = balanceOf[recipient]; // get LP balance of `recipient`
        if (_supplied > 0) {
            uint _supplyIndex0 = supplyIndex0[recipient]; // get last adjusted index0 for recipient
            uint _supplyIndex1 = supplyIndex1[recipient];
            uint _index0 = index0; // get global index0 for accumulated fees
            uint _index1 = index1;
            supplyIndex0[recipient] = _index0; // update user current position to global position
            supplyIndex1[recipient] = _index1;
            uint _delta0 = _index0 - _supplyIndex0; // see if there is any difference that need to be accrued
            uint _delta1 = _index1 - _supplyIndex1;
            if (_delta0 > 0) {
                uint _share = _supplied * _delta0 / 1e18; // add accrued difference for each supplied token
                claimable0[recipient] += _share;
            }
            if (_delta1 > 0) {
                uint _share = _supplied * _delta1 / 1e18;
                claimable1[recipient] += _share;
            }
        } else {
            supplyIndex0[recipient] = index0; // new users are set to the default global state
            supplyIndex1[recipient] = index1;
        }
    }

    function getReserves() public view returns (uint _reserve0, uint _reserve1, uint _blockTimestampLast) {
```


```
    // Accrue fees on token0
    function _update0(uint amount) internal {
        _safeTransfer(token0, fees, amount); // transfer the fees out to BaseV1Fees
        uint256 _ratio = amount * 1e18 / totalSupply; // 1e18 adjustment is removed during claim
        if (_ratio > 0) {
            index0 += _ratio;
        }
        emit Fees(msg.sender, amount, 0);
    }

    // Accrue fees on token1
    function _update1(uint amount) internal {
        _safeTransfer(token1, fees, amount);
        uint256 _ratio = amount * 1e18 / totalSupply;
        if (_ratio > 0) {
            index1 += _ratio;
        }
        emit Fees(msg.sender, 0, amount);
    }

    // this function MUST be called on any balance changes, otherwise can be used to infinitely claim fees
    // Fees are segregated from core funds, so fees can never put liquidity at risk
    function _updateFor(address recipient) internal {
        uint _supplied = balanceOf[recipient]; // get LP balance of `recipient`
        if (_supplied > 0) {
            uint _supplyIndex0 = supplyIndex0[recipient]; // get last adjusted index0 for recipient
            uint _supplyIndex1 = supplyIndex1[recipient];
            uint _index0 = index0; // get global index0 for accumulated fees
            uint _index1 = index1;
            supplyIndex0[recipient] = _index0; // update user current position to global position
            supplyIndex1[recipient] = _index1;
            uint _delta0 = _index0 - _supplyIndex0; // see if there is any difference that need to be accrued
            uint _delta1 = _index1 - _supplyIndex1;
            if (_delta0 > 0) {
                uint _share = _supplied * _delta0 / 1e18; // add accrued difference for each supplied token
                claimable0[recipient] += _share;
            }
            if (_delta1 > 0) {
                uint _share = _supplied * _delta1 / 1e18;
                claimable1[recipient] += _share;
            }
        } else {
            supplyIndex0[recipient] = index0; // new users are set to the default global state
            supplyIndex1[recipient] = index1;
        }
    }

    function getReserves() public view returns (uint _reserve0, uint _reserve1, uint _blockTimestampLast) {
```

```
        _reserve0 = reserve0;
        _reserve1 = reserve1;
        _blockTimestampLast = blockTimestampLast;
    }

    // update reserves and, on the first call per block, price accumulators
    function _update(uint balance0, uint balance1, uint _reserve0, uint _reserve1) internal {
        uint blockTimestamp = block.timestamp;
        uint timeElapsed = blockTimestamp - blockTimestampLast; // overflow is desired
        if (timeElapsed > 0 && _reserve0 != 0 && _reserve1 != 0) {
            reserve0CumulativeLast += _reserve0 * timeElapsed;
            reserve1CumulativeLast += _reserve1 * timeElapsed;
        }

        Observation memory _point = lastObservation();
        timeElapsed = blockTimestamp - _point.timestamp; // compare the last observation with current timestamp, if greater than 30 minutes, record a new event
        if (timeElapsed > periodSize) {
            observations.push(Observation(blockTimestamp, reserve0CumulativeLast, reserve1CumulativeLast));
        }
        reserve0 = balance0;
        reserve1 = balance1;
        blockTimestampLast = blockTimestamp;
        emit Sync(reserve0, reserve1);
    }

    // produces the cumulative price using counterfactuals to save gas and avoid a call to sync.
    function currentCumulativePrices() public view returns (uint reserve0Cumulative, uint reserve1Cumulative, uint blockTimestamp) {
        blockTimestamp = block.timestamp;
        reserve0Cumulative = reserve0CumulativeLast;
        reserve1Cumulative = reserve1CumulativeLast;

        // if time has elapsed since the last update on the pair, mock the accumulated price values
        (uint _reserve0, uint _reserve1, uint _blockTimestampLast) = getReserves();
        if (_blockTimestampLast != blockTimestamp) {
            // subtraction overflow is desired
            uint timeElapsed = blockTimestamp - _blockTimestampLast;
            reserve0Cumulative += _reserve0 * timeElapsed;
            reserve1Cumulative += _reserve1 * timeElapsed;
        }
    }

    // gives the current twap price measured from amountIn * tokenIn gives amountOut
    function current(address tokenIn, uint amountIn) external view returns (uint amountOut) {
```


```
        _reserve0 = reserve0;
        _reserve1 = reserve1;
        _blockTimestampLast = blockTimestampLast;
    }

    // update reserves and, on the first call per block, price accumulators
    function _update(uint balance0, uint balance1, uint _reserve0, uint _reserve1) internal {
        uint blockTimestamp = block.timestamp;
        uint timeElapsed = blockTimestamp - blockTimestampLast; // overflow is desired
        if (timeElapsed > 0 && _reserve0 != 0 && _reserve1 != 0) {
            reserve0CumulativeLast += _reserve0 * timeElapsed;
            reserve1CumulativeLast += _reserve1 * timeElapsed;
        }

        Observation memory _point = lastObservation();
        timeElapsed = blockTimestamp - _point.timestamp; // compare the last observation with current timestamp, if greater than 30 minutes, record a new event
        if (timeElapsed > periodSize) {
            observations.push(Observation(blockTimestamp, reserve0CumulativeLast, reserve1CumulativeLast));
        }
        reserve0 = balance0;
        reserve1 = balance1;
        blockTimestampLast = blockTimestamp;
        emit Sync(reserve0, reserve1);
    }

    // produces the cumulative price using counterfactuals to save gas and avoid a call to sync.
    function currentCumulativePrices() public view returns (uint reserve0Cumulative, uint reserve1Cumulative, uint blockTimestamp) {
        blockTimestamp = block.timestamp;
        reserve0Cumulative = reserve0CumulativeLast;
        reserve1Cumulative = reserve1CumulativeLast;

        // if time has elapsed since the last update on the pair, mock the accumulated price values
        (uint _reserve0, uint _reserve1, uint _blockTimestampLast) = getReserves();
        if (_blockTimestampLast != blockTimestamp) {
            // subtraction overflow is desired
            uint timeElapsed = blockTimestamp - _blockTimestampLast;
            reserve0Cumulative += _reserve0 * timeElapsed;
            reserve1Cumulative += _reserve1 * timeElapsed;
        }
    }

    // gives the current twap price measured from amountIn * tokenIn gives amountOut
    function current(address tokenIn, uint amountIn) external view returns (uint amountOut) {
```

```
        Observation memory _observation = lastObservation();
        (uint reserve0Cumulative, uint reserve1Cumulative,) = currentCumulativePrices();
        if (block.timestamp == _observation.timestamp) {
            _observation = observations[observations.length-2];
        }

        uint timeElapsed = block.timestamp - _observation.timestamp;
        uint _reserve0 = (reserve0Cumulative - _observation.reserve0Cumulative) / timeElapsed;
        uint _reserve1 = (reserve1Cumulative - _observation.reserve1Cumulative) / timeElapsed;
        amountOut = _getAmountOut(amountIn, tokenIn, _reserve0, _reserve1);
    }

    // as per `current`, however allows user configured granularity, up to the full window size
    function quote(address tokenIn, uint amountIn, uint granularity) external view returns (uint amountOut) {
        uint [] memory _prices = sample(tokenIn, amountIn, granularity, 1);
        uint priceAverageCumulative;
        for (uint i = 0; i < _prices.length; i++) {
            priceAverageCumulative += _prices[i];
        }
        return priceAverageCumulative / granularity;
    }

    // returns a memory set of twap prices
    function prices(address tokenIn, uint amountIn, uint points) external view returns (uint[] memory) {
        return sample(tokenIn, amountIn, points, 1);
    }

    function sample(address tokenIn, uint amountIn, uint points, uint window) public view returns (uint[] memory) {
        uint[] memory _prices = new uint[](points);

        uint length = observations.length-1;
        uint i = length - (points * window);
        uint nextIndex = 0;
        uint index = 0;

        for (; i < length; i+=window) {
            nextIndex = i + window;
            uint timeElapsed = observations[nextIndex].timestamp - observations[i].timestamp;
            uint _reserve0 = (observations[nextIndex].reserve0Cumulative - observations[i].reserve0Cumulative) / timeElapsed;
            uint _reserve1 = (observations[nextIndex].reserve1Cumulative - observations[i].reserve1Cumulative) / timeElapsed;
            _prices[index] = _getAmountOut(amountIn, tokenIn, _reserve0, _reserve1);
            index = index + 1;
        }
        return _prices;
    }
```


```
        Observation memory _observation = lastObservation();
        (uint reserve0Cumulative, uint reserve1Cumulative,) = currentCumulativePrices();
        if (block.timestamp == _observation.timestamp) {
            _observation = observations[observations.length-2];
        }

        uint timeElapsed = block.timestamp - _observation.timestamp;
        uint _reserve0 = (reserve0Cumulative - _observation.reserve0Cumulative) / timeElapsed;
        uint _reserve1 = (reserve1Cumulative - _observation.reserve1Cumulative) / timeElapsed;
        amountOut = _getAmountOut(amountIn, tokenIn, _reserve0, _reserve1);
    }

    // as per `current`, however allows user configured granularity, up to the full window size
    function quote(address tokenIn, uint amountIn, uint granularity) external view returns (uint amountOut) {
        uint [] memory _prices = sample(tokenIn, amountIn, granularity, 1);
        uint priceAverageCumulative;
        for (uint i = 0; i < _prices.length; i++) {
            priceAverageCumulative += _prices[i];
        }
        return priceAverageCumulative / granularity;
    }

    // returns a memory set of twap prices
    function prices(address tokenIn, uint amountIn, uint points) external view returns (uint[] memory) {
        return sample(tokenIn, amountIn, points, 1);
    }

    function sample(address tokenIn, uint amountIn, uint points, uint window) public view returns (uint[] memory) {
        uint[] memory _prices = new uint[](points);

        uint length = observations.length-1;
        uint i = length - (points * window);
        uint nextIndex = 0;
        uint index = 0;

        for (; i < length; i+=window) {
            nextIndex = i + window;
            uint timeElapsed = observations[nextIndex].timestamp - observations[i].timestamp;
            uint _reserve0 = (observations[nextIndex].reserve0Cumulative - observations[i].reserve0Cumulative) / timeElapsed;
            uint _reserve1 = (observations[nextIndex].reserve1Cumulative - observations[i].reserve1Cumulative) / timeElapsed;
            _prices[index] = _getAmountOut(amountIn, tokenIn, _reserve0, _reserve1);
            index = index + 1;
        }
        return _prices;
    }
```

```
343
344     // this low-level function should be called fro
    m a contract which performs important safety checks
345     // standard uniswap v2 implementation
346     function mint(address to) external lock returns
    (uint liquidity) {
347         (uint _reserve0, uint _reserve1) = (reserve
    0, reserve1);
348         uint _balance0 = erc20(token0).balanceOf(ad
    dress(this));
349         uint _balance1 = erc20(token1).balanceOf(ad
    dress(this));
350         uint _amount0 = _balance0 - _reserve0;
351         uint _amount1 = _balance1 - _reserve1;
352
353         uint _totalSupply = totalSupply; // gas sav
    ings, must be defined here since totalSupply can up
    date in _mintFee
354         if (_totalSupply == 0) {
355             liquidity = Math.sqrt(_amount0 * _amoun
    t1) - MINIMUM_LIQUIDITY;
356             _mint(address(0), MINIMUM_LIQUIDITY);
     // permanently lock the first MINIMUM_LIQUIDITY to
    kens
357         } else {
358             liquidity = Math.min(_amount0 * _totalS
    upply / _reserve0, _amount1 * _totalSupply / _reser
    ve1);
359         }
360         require(liquidity > 0, 'ILM'); // BaseV1: I
    NSUFFICIENT_LIQUIDITY_MINTED
361         _mint(to, liquidity);
362
363         _update(_balance0, _balance1, _reserve0, _r
    eserve1);
364         emit Mint(msg.sender, _amount0, _amount1);
365     }
366
367     // this low-level function should be called fro
    m a contract which performs important safety checks
368     // standard uniswap v2 implementation
369     function burn(address to) external lock returns
    (uint amount0, uint amount1) {
370         (uint _reserve0, uint _reserve1) = (reserve
    0, reserve1);
371         (address _token0, address _token1) = (token
    0, token1);
372         uint _balance0 = erc20(_token0).balanceOf(a
    ddress(this));
373         uint _balance1 = erc20(_token1).balanceOf(a
    ddress(this));
374         uint _liquidity = balanceOf[address(this)];
375
376         uint _totalSupply = totalSupply; // gas sav
    ings, must be defined here since totalSupply can up
    date in _mintFee
377         amount0 = _liquidity * _balance0 / _totalSu
    pply; // using balances ensures pro-rata distributi
    on
378         amount1 = _liquidity * _balance1 / _totalSu
    pply; // using balances ensures pro-rata distributi
    on
379         require(amount0 > 0 && amount1 > 0, 'ILB');
    // BaseV1: INSUFFICIENT_LIQUIDITY_BURNED
380         _burn(address(this), _liquidity);
381         _safeTransfer(_token0, to, amount0);
382         _safeTransfer(_token1, to, amount1);
```

```
343
344     // this low-level function should be called fro
    m a contract which performs important safety checks
345     // standard uniswap v2 implementation
346     function mint(address to) external lock returns
    (uint liquidity) {
347         (uint _reserve0, uint _reserve1) = (reserve
    0, reserve1);
348         uint _balance0 = erc20(token0).balanceOf(ad
    dress(this));
349         uint _balance1 = erc20(token1).balanceOf(ad
    dress(this));
350         uint _amount0 = _balance0 - _reserve0;
351         uint _amount1 = _balance1 - _reserve1;
352
353         uint _totalSupply = totalSupply; // gas sav
    ings, must be defined here since totalSupply can up
    date in _mintFee
354         if (_totalSupply == 0) {
355             liquidity = Math.sqrt(_amount0 * _amoun
    t1) - MINIMUM_LIQUIDITY;
356             _mint(address(0), MINIMUM_LIQUIDITY);
     // permanently lock the first MINIMUM_LIQUIDITY to
    kens
357         } else {
358             liquidity = Math.min(_amount0 * _totalS
    upply / _reserve0, _amount1 * _totalSupply / _reser
    ve1);
359         }
360         require(liquidity > 0, 'ILM'); // BaseV1: I
    NSUFFICIENT_LIQUIDITY_MINTED
361         _mint(to, liquidity);
362
363         _update(_balance0, _balance1, _reserve0, _r
    eserve1);
364         emit Mint(msg.sender, _amount0, _amount1);
365     }
366
367     // this low-level function should be called fro
    m a contract which performs important safety checks
368     // standard uniswap v2 implementation
369     function burn(address to) external lock returns
    (uint amount0, uint amount1) {
370         (uint _reserve0, uint _reserve1) = (reserve
    0, reserve1);
371         (address _token0, address _token1) = (token
    0, token1);
372         uint _balance0 = erc20(_token0).balanceOf(a
    ddress(this));
373         uint _balance1 = erc20(_token1).balanceOf(a
    ddress(this));
374         uint _liquidity = balanceOf[address(this)];
375
376         uint _totalSupply = totalSupply; // gas sav
    ings, must be defined here since totalSupply can up
    date in _mintFee
377         amount0 = _liquidity * _balance0 / _totalSu
    pply; // using balances ensures pro-rata distributi
    on
378         amount1 = _liquidity * _balance1 / _totalSu
    pply; // using balances ensures pro-rata distributi
    on
379         require(amount0 > 0 && amount1 > 0, 'ILB');
    // BaseV1: INSUFFICIENT_LIQUIDITY_BURNED
380         _burn(address(this), _liquidity);
381         _safeTransfer(_token0, to, amount0);
382         _safeTransfer(_token1, to, amount1);
```

```
        _balance0 = erc20(_token0).balanceOf(address(this));
        _balance1 = erc20(_token1).balanceOf(address(this));

        _update(_balance0, _balance1, _reserve0, _reserve1);
        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(!BaseV1Factory(factory).isPaused());
        require(amount0Out > 0 || amount1Out > 0, 'IOA'); // BaseV1: INSUFFICIENT_OUTPUT_AMOUNT
        (uint _reserve0, uint _reserve1) =  (reserve0, reserve1);
        require(amount0Out < _reserve0 && amount1Out < _reserve1, 'IL'); // BaseV1: INSUFFICIENT_LIQUIDITY

        uint _balance0;
        uint _balance1;
        { // scope for _token{0,1}, avoids stack too deep errors
        (address _token0, address _token1) = (token0, token1);
        require(to != _token0 && to != _token1, 'IT'); // BaseV1: 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) IBaseV1Callee(to).hook(msg.sender, amount0Out, amount1Out, data); // callback, used for flash loans
        _balance0 = erc20(_token0).balanceOf(address(this));
        _balance1 = erc20(_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, 'IIA'); // BaseV1: INSUFFICIENT_INPUT_AMOUNT
        { // scope for reserve{0,1}Adjusted, avoids stack too deep errors
        (address _token0, address _token1) = (token0, token1);
        if (amount0In > 0) _update0(amount0In / 10000); // accrue fees for token0 and move them out of pool
        if (amount1In > 0) _update1(amount1In / 10000); // accrue fees for token1 and move them out of pool
        _balance0 = erc20(_token0).balanceOf(address(this)); // since we removed tokens, we need to reconfirm balances, can also simply use previous balance - amountIn/ 10000, but doing balanceOf again as safety check
```

```
        _balance1 = erc20(_token1).balanceOf(address(this));
        // The curve, either x3y+y3x for stable pools, or x*y for volatile pools
        require(_k(_balance0, _balance1) >= _k(_reserve0, _reserve1), 'K'); // BaseV1: 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, address _token1) = (token0, token1);
        _safeTransfer(_token0, to, erc20(_token0).balanceOf(address(this)) - (reserve0));
        _safeTransfer(_token1, to, erc20(_token1).balanceOf(address(this)) - (reserve1));
    }

    // force reserves to match balances
    function sync() external lock {
        _update(erc20(token0).balanceOf(address(this)), erc20(token1).balanceOf(address(this)), reserve0, reserve1);
    }

    function _f(uint x0, uint y) internal pure returns (uint) {
        return x0*(y*y/1e18*y/1e18)/1e18+(x0*x0/1e18*x0/1e18)*y/1e18;
    }

    function _d(uint x0, uint y) internal pure returns (uint) {
        return 3*x0*(y*y/1e18)/1e18+(x0*x0/1e18*x0/1e18);
    }

    function _get_y(uint x0, uint xy, uint y) internal pure returns (uint) {
        for (uint i = 0; i < 255; i++) {
            uint y_prev = y;
            uint k = _f(x0, y);
            if (k < xy) {
                uint dy = (xy - k)*1e18/_d(x0, y);
                y = y + dy;
            } else {
                uint dy = (k - xy)*1e18/_d(x0, y);
                y = y - dy;
            }
            if (y > y_prev) {
                if (y - y_prev <= 1) {
                    return y;
                }
            } else {
                if (y_prev - y <= 1) {
                    return y;
                }
            }
        }
        return y;
    }

```

```
        _balance1 = erc20(_token1).balanceOf(address(this));
        // The curve, either x3y+y3x for stable pools, or x*y for volatile pools
        require(_k(_balance0, _balance1) >= _k(_reserve0, _reserve1), 'K'); // BaseV1: 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, address _token1) = (token0, token1);
        _safeTransfer(_token0, to, erc20(_token0).balanceOf(address(this)) - (reserve0));
        _safeTransfer(_token1, to, erc20(_token1).balanceOf(address(this)) - (reserve1));
    }

    // force reserves to match balances
    function sync() external lock {
        _update(erc20(token0).balanceOf(address(this)), erc20(token1).balanceOf(address(this)), reserve0, reserve1);
    }

    function _f(uint x0, uint y) internal pure returns (uint) {
        return x0*(y*y/1e18*y/1e18)/1e18+(x0*x0/1e18*x0/1e18)*y/1e18;
    }

    function _d(uint x0, uint y) internal pure returns (uint) {
        return 3*x0*(y*y/1e18)/1e18+(x0*x0/1e18*x0/1e18);
    }

    function _get_y(uint x0, uint xy, uint y) internal pure returns (uint) {
        for (uint i = 0; i < 255; i++) {
            uint y_prev = y;
            uint k = _f(x0, y);
            if (k < xy) {
                uint dy = (xy - k)*1e18/_d(x0, y);
                y = y + dy;
            } else {
                uint dy = (k - xy)*1e18/_d(x0, y);
                y = y - dy;
            }
            if (y > y_prev) {
                if (y - y_prev <= 1) {
                    return y;
                }
            } else {
                if (y_prev - y <= 1) {
                    return y;
                }
            }
        }
        return y;
    }

```

```
    function getAmountOut(uint amountIn, address tokenIn) external view returns (uint) {
        (uint _reserve0, uint _reserve1) = (reserve0, reserve1);
        amountIn -= amountIn / 10000; // remove fee from amount received
        return _getAmountOut(amountIn, tokenIn, _reserve0, _reserve1);
    }

    function _getAmountOut(uint amountIn, address tokenIn, uint _reserve0, uint _reserve1) internal view returns (uint) {
        if (stable) {
            uint xy =  _k(_reserve0, _reserve1);
            _reserve0 = _reserve0 * 1e18 / decimals0;
            _reserve1 = _reserve1 * 1e18 / decimals1;
            (uint reserveA, uint reserveB) = tokenIn == token0 ? (_reserve0, _reserve1) : (_reserve1, _reserve0);
            amountIn = tokenIn == token0 ? amountIn * 1e18 / decimals0 : amountIn * 1e18 / decimals1;
            uint y = reserveB - _get_y(amountIn+reserveA, xy, reserveB);
            return y * (tokenIn == token0 ? decimals1 : decimals0) / 1e18;
        } else {
            (uint reserveA, uint reserveB) = tokenIn == token0 ? (_reserve0, _reserve1) : (_reserve1, _reserve0);
            return amountIn * reserveB / (reserveA + amountIn);
        }
    }

    function _k(uint x, uint y) internal view returns (uint) {
        if (stable) {
            uint _x = x * 1e18 / decimals0;
            uint _y = y * 1e18 / decimals1;
            uint _a = (_x * _y) / 1e18;
            uint _b = ((_x * _x) / 1e18 + (_y * _y) / 1e18);
            return _a * _b / 1e18;  // x3y+y3x >= k
        } else {
            return x * y; // xy >= k
        }
    }

    function _mint(address dst, uint amount) internal {
        _updateFor(dst); // balances must be updated on mint/burn/transfer
        totalSupply += amount;
        balanceOf[dst] += amount;
        emit Transfer(address(0), dst, amount);
    }

    function _burn(address dst, uint amount) internal {
        _updateFor(dst);
        totalSupply -= amount;
        balanceOf[dst] -= amount;
        emit Transfer(dst, address(0), amount);
    }

```


```
    function getAmountOut(uint amountIn, address tokenIn) external view returns (uint) {
        (uint _reserve0, uint _reserve1) = (reserve0, reserve1);
        amountIn -= amountIn / 10000; // remove fee from amount received
        return _getAmountOut(amountIn, tokenIn, _reserve0, _reserve1);
    }

    function _getAmountOut(uint amountIn, address tokenIn, uint _reserve0, uint _reserve1) internal view returns (uint) {
        if (stable) {
            uint xy =  _k(_reserve0, _reserve1);
            _reserve0 = _reserve0 * 1e18 / decimals0;
            _reserve1 = _reserve1 * 1e18 / decimals1;
            (uint reserveA, uint reserveB) = tokenIn == token0 ? (_reserve0, _reserve1) : (_reserve1, _reserve0);
            amountIn = tokenIn == token0 ? amountIn * 1e18 / decimals0 : amountIn * 1e18 / decimals1;
            uint y = reserveB - _get_y(amountIn+reserveA, xy, reserveB);
            return y * (tokenIn == token0 ? decimals1 : decimals0) / 1e18;
        } else {
            (uint reserveA, uint reserveB) = tokenIn == token0 ? (_reserve0, _reserve1) : (_reserve1, _reserve0);
            return amountIn * reserveB / (reserveA + amountIn);
        }
    }

    function _k(uint x, uint y) internal view returns (uint) {
        if (stable) {
            uint _x = x * 1e18 / decimals0;
            uint _y = y * 1e18 / decimals1;
            uint _a = (_x * _y) / 1e18;
            uint _b = ((_x * _x) / 1e18 + (_y * _y) / 1e18);
            return _a * _b / 1e18;  // x3y+y3x >= k
        } else {
            return x * y; // xy >= k
        }
    }

    function _mint(address dst, uint amount) internal {
        _updateFor(dst); // balances must be updated on mint/burn/transfer
        totalSupply += amount;
        balanceOf[dst] += amount;
        emit Transfer(address(0), dst, amount);
    }

    function _burn(address dst, uint amount) internal {
        _updateFor(dst);
        totalSupply -= amount;
        balanceOf[dst] -= amount;
        emit Transfer(dst, address(0), amount);
    }

```

```
    function approve(address spender, uint amount) external returns (bool) {
        allowance[msg.sender][spender] = amount;

        emit Approval(msg.sender, spender, amount);
        return true;
    }

    function permit(address owner, address spender, uint value, uint deadline, uint8 v, bytes32 r, bytes32 s) external {
        require(deadline >= block.timestamp, 'BaseV1: EXPIRED');
        DOMAIN_SEPARATOR = keccak256(
            abi.encode(
                keccak256('EIP712Domain(string name,string version,uint256 chainId,address verifyingContract)'),
                keccak256(bytes(name)),
                keccak256('1'),
                block.chainid,
                address(this)
            )
        );
        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, 'BaseV1: INVALID_SIGNATURE');
        allowance[owner][spender] = value;

        emit Approval(owner, spender, value);
    }

    function transfer(address dst, uint amount) external returns (bool) {
        _transferTokens(msg.sender, dst, amount);
        return true;
    }

    function transferFrom(address src, address dst, uint amount) external returns (bool) {
        address spender = msg.sender;
        uint spenderAllowance = allowance[src][spender];

        if (spender != src && spenderAllowance != type(uint).max) {
            uint newAllowance = spenderAllowance - amount;
            allowance[src][spender] = newAllowance;

            emit Approval(src, spender, newAllowance);
        }

        _transferTokens(src, dst, amount);
        return true;
    }
```

```
    function approve(address spender, uint amount) external returns (bool) {
        allowance[msg.sender][spender] = amount;

        emit Approval(msg.sender, spender, amount);
        return true;
    }

    function permit(address owner, address spender, uint value, uint deadline, uint8 v, bytes32 r, bytes32 s) external {
        require(deadline >= block.timestamp, 'BaseV1: EXPIRED');
        DOMAIN_SEPARATOR = keccak256(
            abi.encode(
                keccak256('EIP712Domain(string name,string version,uint256 chainId,address verifyingContract)'),
                keccak256(bytes(name)),
                keccak256(bytes('1')),
                block.chainid,
                address(this)
            )
        );
        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, 'BaseV1: INVALID_SIGNATURE');
        allowance[owner][spender] = value;

        emit Approval(owner, spender, value);
    }

    function transfer(address dst, uint amount) external returns (bool) {
        _transferTokens(msg.sender, dst, amount);
        return true;
    }

    function transferFrom(address src, address dst, uint amount) external returns (bool) {
        address spender = msg.sender;
        uint spenderAllowance = allowance[src][spender];

        if (spender != src && spenderAllowance != type(uint).max) {
            uint newAllowance = spenderAllowance - amount;
            allowance[src][spender] = newAllowance;

            emit Approval(src, spender, newAllowance);
        }

        _transferTokens(src, dst, amount);
        return true;
    }
```

```
    function _transferTokens(address src, address dst, uint amount) internal {
        _updateFor(src); // update fee position for src
        _updateFor(dst); // update fee position for dst

        balanceOf[src] -= amount;
        balanceOf[dst] += amount;

        emit Transfer(src, dst, amount);
    }

    function _safeTransfer(address token,address to,uint256 value) internal {
        require(token.code.length > 0);
        (bool success, bytes memory data) =
        token.call(abi.encodeWithSelector(erc20.transfer.selector, to, value));
        require(success && (data.length == 0 || abi.decode(data, (bool))));
    }
}

contract BaseV1Factory {

    bool public isPaused;
    address public pauser;
    address public pendingPauser;

    mapping(address => mapping(address => mapping(bool => address))) public getPair;
    address[] public allPairs;
    mapping(address => bool) public isPair; // simplified check if its a pair, given that `stable` flag might not be available in peripherals

    address internal _temp0;
    address internal _temp1;
    bool internal _temp;

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

    constructor() {
        pauser = msg.sender;
        isPaused = false;
    }

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

    function setPauser(address _pauser) external {
        require(msg.sender == pauser);
        pendingPauser = _pauser;
    }

    function acceptPauser() external {
        require(msg.sender == pendingPauser);
        pauser = pendingPauser;
    }

    function setPause(bool _state) external {
        require(msg.sender == pauser);
        isPaused = _state;
    }

```

```
    function pairCodeHash() external pure returns (bytes32) {
        return keccak256(type(BaseV1Pair).creationCode);
    }

    function getInitializable() external view returns (address, address, bool) {
        return (_temp0, _temp1, _temp);
    }

    function createPair(address tokenA, address tokenB, bool stable) external returns (address pair) {
        require(tokenA != tokenB, 'IA'); // BaseV1: IDENTICAL_ADDRESSES
        (address token0, address token1) = tokenA < tokenB ? (tokenA, tokenB) : (tokenB, tokenA);
        require(token0 != address(0), 'ZA'); // BaseV1: ZERO_ADDRESS
        require(getPair[token0][token1][stable] == address(0), 'PE'); // BaseV1: PAIR_EXISTS - single check is sufficient
        bytes32 salt = keccak256(abi.encodePacked(token0, token1, stable)); // notice salt includes stable as well, 3 parameters
        (_temp0, _temp1, _temp) = (token0, token1, stable);
        pair = address(new BaseV1Pair{salt:salt}());
        getPair[token0][token1][stable] = pair;
        getPair[token1][token0][stable] = pair; // populate mapping in the reverse direction
        allPairs.push(pair);
        isPair[pair] = true;
        emit PairCreated(token0, token1, stable, pair, allPairs.length);
    }
}
```


```
    function pairCodeHash() external pure returns (bytes32) {
        return keccak256(type(BaseV1Pair).creationCode);
    }

    function getInitializable() external view returns (address, address, bool) {
        return (_temp0, _temp1, _temp);
    }

    function createPair(address tokenA, address tokenB, bool stable) external returns (address pair) {
        require(tokenA != tokenB, 'IA'); // BaseV1: IDENTICAL_ADDRESSES
        (address token0, address token1) = tokenA < tokenB ? (tokenA, tokenB) : (tokenB, tokenA);
        require(token0 != address(0), 'ZA'); // BaseV1: ZERO_ADDRESS
        require(getPair[token0][token1][stable] == address(0), 'PE'); // BaseV1: PAIR_EXISTS - single check is sufficient
        bytes32 salt = keccak256(abi.encodePacked(token0, token1, stable)); // notice salt includes stable as well, 3 parameters
        (_temp0, _temp1, _temp) = (token0, token1, stable);
        pair = address(new BaseV1Pair{salt:salt}());
        getPair[token0][token1][stable] = pair;
        getPair[token1][token0][stable] = pair; // populate mapping in the reverse direction
        allPairs.push(pair);
        isPair[pair] = true;
        emit PairCreated(token0, token1, stable, pair, allPairs.length);
    }
}
```