- 0 Removals + 1 Addition

```
// SPDX-License-Identifier: AGPL-3.0-or-later
pragma solidity 0.7.5;

library FullMath {
    function fullMul(uint256 x, uint256 y) private pure returns (uint256 l, uint256 h) {
        uint256 mm = mulmod(x, y, uint256(-1));
        l = x * y;
        h = mm - l;
        if (mm < l) h -= 1;
    }

    function fullDiv(
        uint256 l,
        uint256 h,
        uint256 d
    ) private pure returns (uint256) {
        uint256 pow2 = d & -d;
        d /= pow2;
        l /= pow2;
        l += h * ((-pow2) / pow2 + 1);
        uint256 r = 1;
        r *= 2 - d * r;
        r *= 2 - d * r;
        r *= 2 - d * r;
        r *= 2 - d * r;
        r *= 2 - d * r;
        r *= 2 - d * r;
        r *= 2 - d * r;
        r *= 2 - d * r;
        return l * r;
    }

    function mulDiv(
        uint256 x,
        uint256 y,
        uint256 d
    ) internal pure returns (uint256) {
        (uint256 l, uint256 h) = fullMul(x, y);
        uint256 mm = mulmod(x, y, d);
        if (mm > l) h -= 1;
        l -= mm;
        require(h < d, 'FullMath::mulDiv: overflow');
        return fullDiv(l, h, d);
    }
}

library Babylonian {

    function sqrt(uint256 x) internal pure returns (uint256) {
        if (x == 0) return 0;

        uint256 xx = x;
        uint256 r = 1;
        if (xx >= 0x100000000000000000000000000000000) {
            xx >>= 128;
            r <<= 64;
        }
        if (xx >= 0x10000000000000000) {
            xx >>= 64;
```

```
// SPDX-License-Identifier: AGPL-3.0-or-later
pragma solidity 0.7.5;
//Only change to generate diff
library FullMath {
    function fullMul(uint256 x, uint256 y) private pure returns (uint256 l, uint256 h) {
        uint256 mm = mulmod(x, y, uint256(-1));
        l = x * y;
        h = mm - l;
        if (mm < l) h -= 1;
    }

    function fullDiv(
        uint256 l,
        uint256 h,
        uint256 d
    ) private pure returns (uint256) {
        uint256 pow2 = d & -d;
        d /= pow2;
        l /= pow2;
        l += h * ((-pow2) / pow2 + 1);
        uint256 r = 1;
        r *= 2 - d * r;
        r *= 2 - d * r;
        r *= 2 - d * r;
        r *= 2 - d * r;
        r *= 2 - d * r;
        r *= 2 - d * r;
        r *= 2 - d * r;
        r *= 2 - d * r;
        return l * r;
    }

    function mulDiv(
        uint256 x,
        uint256 y,
        uint256 d
    ) internal pure returns (uint256) {
        (uint256 l, uint256 h) = fullMul(x, y);
        uint256 mm = mulmod(x, y, d);
        if (mm > l) h -= 1;
        l -= mm;
        require(h < d, 'FullMath::mulDiv: overflow');
        return fullDiv(l, h, d);
    }
}

library Babylonian {

    function sqrt(uint256 x) internal pure returns (uint256) {
        if (x == 0) return 0;

        uint256 xx = x;
        uint256 r = 1;
        if (xx >= 0x100000000000000000000000000000000) {
            xx >>= 128;
            r <<= 64;
        }
        if (xx >= 0x10000000000000000) {
            xx >>= 64;
```

```
            r <<= 32;
        }
        if (xx >= 0x100000000) {
            xx >>= 32;
            r <<= 16;
        }
        if (xx >= 0x10000) {
            xx >>= 16;
            r <<= 8;
        }
        if (xx >= 0x100) {
            xx >>= 8;
            r <<= 4;
        }
        if (xx >= 0x10) {
            xx >>= 4;
            r <<= 2;
        }
        if (xx >= 0x8) {
            r <<= 1;
        }
        r = (r + x / r) >> 1;
        r = (r + x / r) >> 1;
        r = (r + x / r) >> 1;
        r = (r + x / r) >> 1;
        r = (r + x / r) >> 1;
        r = (r + x / r) >> 1;
        r = (r + x / r) >> 1; // Seven iterations s
hould be enough
        uint256 r1 = x / r;
        return (r < r1 ? r : r1);
    }
}

library BitMath {

    function mostSignificantBit(uint256 x) internal
 pure returns (uint8 r) {
        require(x > 0, 'BitMath::mostSignificantBi
t: zero');

        if (x >= 0x100000000000000000000000000000000
0) {
            x >>= 128;
            r += 128;
        }
        if (x >= 0x10000000000000000) {
            x >>= 64;
            r += 64;
        }
        if (x >= 0x100000000) {
            x >>= 32;
            r += 32;
        }
        if (x >= 0x10000) {
            x >>= 16;
            r += 16;
        }
        if (x >= 0x100) {
            x >>= 8;
            r += 8;
        }
        if (x >= 0x10) {
            x >>= 4;
            r += 4;
        }
        if (x >= 0x4) {
            x >>= 2;
```

```
            r <<= 32;
        }
        if (xx >= 0x100000000) {
            xx >>= 32;
            r <<= 16;
        }
        if (xx >= 0x10000) {
            xx >>= 16;
            r <<= 8;
        }
        if (xx >= 0x100) {
            xx >>= 8;
            r <<= 4;
        }
        if (xx >= 0x10) {
            xx >>= 4;
            r <<= 2;
        }
        if (xx >= 0x8) {
            r <<= 1;
        }
        r = (r + x / r) >> 1;
        r = (r + x / r) >> 1;
        r = (r + x / r) >> 1;
        r = (r + x / r) >> 1;
        r = (r + x / r) >> 1;
        r = (r + x / r) >> 1;
        r = (r + x / r) >> 1; // Seven iterations s
hould be enough
        uint256 r1 = x / r;
        return (r < r1 ? r : r1);
    }
}

library BitMath {

    function mostSignificantBit(uint256 x) internal
 pure returns (uint8 r) {
        require(x > 0, 'BitMath::mostSignificantBi
t: zero');

        if (x >= 0x100000000000000000000000000000000
0) {
            x >>= 128;
            r += 128;
        }
        if (x >= 0x10000000000000000) {
            x >>= 64;
            r += 64;
        }
        if (x >= 0x100000000) {
            x >>= 32;
            r += 32;
        }
        if (x >= 0x10000) {
            x >>= 16;
            r += 16;
        }
        if (x >= 0x100) {
            x >>= 8;
            r += 8;
        }
        if (x >= 0x10) {
            x >>= 4;
            r += 4;
        }
        if (x >= 0x4) {
            x >>= 2;
```

```
            r += 2;
        }
        if (x >= 0x2) r += 1;
    }
}

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 {
        uint256 _x;
    }

    uint8 private constant RESOLUTION = 112;
    uint256 private constant Q112 = 0x10000000000000000000000000000000;
    uint256 private constant Q224 = 0x100000000000000000000000000000000000000000000000;
    uint256 private constant LOWER_MASK = 0xffffffffffffffffffffffffffff; // decimal of UQ*x112 (lower 112 bits)

    // 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 uq112x112 into a uint with 18 decimals of precision
    function decode112with18(uq112x112 memory self) internal pure returns (uint) {
        return uint(self._x) / 5192296858534827;
    }

    function fraction(uint256 numerator, uint256 denominator) internal pure returns (uq112x112 memory) {
        require(denominator > 0, 'FixedPoint::fraction: division by zero');
        if (numerator == 0) return FixedPoint.uq112x112(0);

        if (numerator <= uint144(-1)) {
            uint256 result = (numerator << RESOLUTION) / denominator;
            require(result <= uint224(-1), 'FixedPoint::fraction: overflow');
            return uq112x112(uint224(result));
        } else {
            uint256 result = FullMath.mulDiv(numerator, Q112, denominator);
            require(result <= uint224(-1), 'FixedPoint::fraction: overflow');
            return uq112x112(uint224(result));
        }
    }

    // square root of a UQ112x112
    // lossy between 0/1 and 40 bits
    function sqrt(uq112x112 memory self) internal pure returns (uq112x112 memory) {
```

```
            r += 2;
        }
        if (x >= 0x2) r += 1;
    }
}

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 {
        uint256 _x;
    }

    uint8 private constant RESOLUTION = 112;
    uint256 private constant Q112 = 0x10000000000000000000000000000000;
    uint256 private constant Q224 = 0x100000000000000000000000000000000000000000000000;
    uint256 private constant LOWER_MASK = 0xffffffffffffffffffffffffffff; // decimal of UQ*x112 (lower 112 bits)

    // 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 uq112x112 into a uint with 18 decimals of precision
    function decode112with18(uq112x112 memory self) internal pure returns (uint) {
        return uint(self._x) / 5192296858534827;
    }

    function fraction(uint256 numerator, uint256 denominator) internal pure returns (uq112x112 memory) {
        require(denominator > 0, 'FixedPoint::fraction: division by zero');
        if (numerator == 0) return FixedPoint.uq112x112(0);

        if (numerator <= uint144(-1)) {
            uint256 result = (numerator << RESOLUTION) / denominator;
            require(result <= uint224(-1), 'FixedPoint::fraction: overflow');
            return uq112x112(uint224(result));
        } else {
            uint256 result = FullMath.mulDiv(numerator, Q112, denominator);
            require(result <= uint224(-1), 'FixedPoint::fraction: overflow');
            return uq112x112(uint224(result));
        }
    }

    // square root of a UQ112x112
    // lossy between 0/1 and 40 bits
    function sqrt(uq112x112 memory self) internal pure returns (uq112x112 memory) {
```

```
        if (self._x <= uint144(-1)) {
            return uq112x112(uint224(Babylonian.sqrt(uint256(self._x) << 112)));
        }

        uint8 safeShiftBits = 255 - BitMath.mostSignificantBit(self._x);
        safeShiftBits -= safeShiftBits % 2;
        return uq112x112(uint224(Babylonian.sqrt(uint256(self._x) << safeShiftBits) << ((112 - safeShiftBits) / 2)));
    }
}

library LowGasSafeMath {
    /// @notice Returns x + y, reverts if sum overflows uint256
    /// @param x The augend
    /// @param y The addend
    /// @return z The sum of x and y
    function add(uint256 x, uint256 y) internal pure returns (uint256 z) {
        require((z = x + y) >= x);
    }

    function add32(uint32 x, uint32 y) internal pure returns (uint32 z) {
        require((z = x + y) >= x);
    }

    /// @notice Returns x - y, reverts if underflows
    /// @param x The minuend
    /// @param y The subtrahend
    /// @return z The difference of x and y
    function sub(uint256 x, uint256 y) internal pure returns (uint256 z) {
        require((z = x - y) <= x);
    }

    function sub32(uint32 x, uint32 y) internal pure returns (uint32 z) {
        require((z = x - y) <= x);
    }

    /// @notice Returns x * y, reverts if overflows
    /// @param x The multiplicand
    /// @param y The multiplier
    /// @return z The product of x and y
    function mul(uint256 x, uint256 y) internal pure returns (uint256 z) {
        require(x == 0 || (z = x * y) / x == y);
    }

    /// @notice Returns x + y, reverts if overflows or underflows
    /// @param x The augend
    /// @param y The addend
    /// @return z The sum of x and y
    function add(int256 x, int256 y) internal pure returns (int256 z) {
        require((z = x + y) >= x == (y >= 0));
    }

    /// @notice Returns x - y, reverts if overflows or underflows
    /// @param x The minuend
    /// @param y The subtrahend
    /// @return z The difference of x and y
```

```
        if (self._x <= uint144(-1)) {
            return uq112x112(uint224(Babylonian.sqrt(uint256(self._x) << 112)));
        }

        uint8 safeShiftBits = 255 - BitMath.mostSignificantBit(self._x);
        safeShiftBits -= safeShiftBits % 2;
        return uq112x112(uint224(Babylonian.sqrt(uint256(self._x) << safeShiftBits) << ((112 - safeShiftBits) / 2)));
    }
}

library LowGasSafeMath {
    /// @notice Returns x + y, reverts if sum overflows uint256
    /// @param x The augend
    /// @param y The addend
    /// @return z The sum of x and y
    function add(uint256 x, uint256 y) internal pure returns (uint256 z) {
        require((z = x + y) >= x);
    }

    function add32(uint32 x, uint32 y) internal pure returns (uint32 z) {
        require((z = x + y) >= x);
    }

    /// @notice Returns x - y, reverts if underflows
    /// @param x The minuend
    /// @param y The subtrahend
    /// @return z The difference of x and y
    function sub(uint256 x, uint256 y) internal pure returns (uint256 z) {
        require((z = x - y) <= x);
    }

    function sub32(uint32 x, uint32 y) internal pure returns (uint32 z) {
        require((z = x - y) <= x);
    }

    /// @notice Returns x * y, reverts if overflows
    /// @param x The multiplicand
    /// @param y The multiplier
    /// @return z The product of x and y
    function mul(uint256 x, uint256 y) internal pure returns (uint256 z) {
        require(x == 0 || (z = x * y) / x == y);
    }

    /// @notice Returns x + y, reverts if overflows or underflows
    /// @param x The augend
    /// @param y The addend
    /// @return z The sum of x and y
    function add(int256 x, int256 y) internal pure returns (int256 z) {
        require((z = x + y) >= x == (y >= 0));
    }

    /// @notice Returns x - y, reverts if overflows or underflows
    /// @param x The minuend
    /// @param y The subtrahend
    /// @return z The difference of x and y
```

```
    function sub(int256 x, int256 y) internal pure returns (int256 z) {
        require((z = x - y) <= x == (y >= 0));
    }

    function div(uint256 x, uint256 y) internal pure returns(uint256 z){
        require(y > 0);
        z=x/y;
    }

    function sqrrt(uint256 a) internal pure returns (uint c) {
        if (a > 3) {
            c = a;
            uint b = add( div( a, 2), 1 );
            while (b < c) {
                c = b;
                b = div( add( div( a, b ), b), 2 );
            }
        } else if (a != 0) {
            c = 1;
        }
    }
}

interface IERC20 {
    function decimals() external view returns (uint8);
}

interface IUniswapV2ERC20 {
    function totalSupply() external view returns (uint);
}

interface IUniswapV2Pair is IUniswapV2ERC20 {
    function getReserves() external view returns (uint112 reserve0, uint112 reserve1, uint32 blockTimestampLast);
    function token0() external view returns ( address );
    function token1() external view returns ( address );
}

interface IBondingCalculator {
  function valuation( address pair_, uint amount_ ) external view returns ( uint _value );
}

contract TimeBondingCalculator is IBondingCalculator {

    using FixedPoint for *;
    using LowGasSafeMath for uint;
    using LowGasSafeMath for uint112;

    IERC20 public immutable Time;

    constructor( address _Time ) {
        require( _Time != address(0) );
        Time = IERC20(_Time);
    }

    function getKValue( address _pair ) public view returns( uint k_ ) {
        uint token0 = IERC20( IUniswapV2Pair( _pair ).token0() ).decimals();
```

```
    function sub(int256 x, int256 y) internal pure returns (int256 z) {
        require((z = x - y) <= x == (y >= 0));
    }

    function div(uint256 x, uint256 y) internal pure returns(uint256 z){
        require(y > 0);
        z=x/y;
    }

    function sqrrt(uint256 a) internal pure returns (uint c) {
        if (a > 3) {
            c = a;
            uint b = add( div( a, 2), 1 );
            while (b < c) {
                c = b;
                b = div( add( div( a, b ), b), 2 );
            }
        } else if (a != 0) {
            c = 1;
        }
    }
}

interface IERC20 {
    function decimals() external view returns (uint8);
}

interface IUniswapV2ERC20 {
    function totalSupply() external view returns (uint);
}

interface IUniswapV2Pair is IUniswapV2ERC20 {
    function getReserves() external view returns (uint112 reserve0, uint112 reserve1, uint32 blockTimestampLast);
    function token0() external view returns ( address );
    function token1() external view returns ( address );
}

interface IBondingCalculator {
  function valuation( address pair_, uint amount_ ) external view returns ( uint _value );
}

contract TimeBondingCalculator is IBondingCalculator {

    using FixedPoint for *;
    using LowGasSafeMath for uint;
    using LowGasSafeMath for uint112;

    IERC20 public immutable Time;

    constructor( address _Time ) {
        require( _Time != address(0) );
        Time = IERC20(_Time);
    }

    function getKValue( address _pair ) public view returns( uint k_ ) {
        uint token0 = IERC20( IUniswapV2Pair( _pair ).token0() ).decimals();
```

```
        uint token1 = IERC20( IUniswapV2Pair( _pair ).token1() ).decimals();
        uint pairDecimals = IERC20( _pair ).decimals();

        (uint reserve0, uint reserve1, ) = IUniswapV2Pair( _pair ).getReserves();
        if (token0.add(token1) <  pairDecimals)
        {
            uint decimals = pairDecimals.sub(token0.add(token1));
            k_ = reserve0.mul(reserve1).mul( 10 ** decimals );
        }
        else {
            uint decimals = token0.add(token1).sub(pairDecimals);
            k_ = reserve0.mul(reserve1).div( 10 ** decimals );
        }

    }

    function getTotalValue( address _pair ) public view returns ( uint _value ) {
        _value = getKValue( _pair ).sqrrt().mul(2);
    }

    function valuation( address _pair, uint amount_ ) external view override returns ( uint _value ) {
        uint totalValue = getTotalValue( _pair );
        uint totalSupply = IUniswapV2Pair( _pair ).totalSupply();

        _value = totalValue.mul( FixedPoint.fraction( amount_, totalSupply ).decode112with18() ).div( 1e18 );
    }

    function markdown( address _pair ) external view returns ( uint ) {
        ( uint reserve0, uint reserve1, ) = IUniswapV2Pair( _pair ).getReserves();

        uint reserve;
        if ( IUniswapV2Pair( _pair ).token0() == address(Time) ) {
            reserve = reserve1;
        } else {
            require(IUniswapV2Pair( _pair ).token1() == address(Time), "not a Time lp pair");
            reserve = reserve0;
        }
        return reserve.mul( 2 * ( 10 ** Time.decimals() ) ).div( getTotalValue( _pair ) );
    }
}
```

```
        uint token1 = IERC20( IUniswapV2Pair( _pair ).token1() ).decimals();
        uint pairDecimals = IERC20( _pair ).decimals();

        (uint reserve0, uint reserve1, ) = IUniswapV2Pair( _pair ).getReserves();
        if (token0.add(token1) <  pairDecimals)
        {
            uint decimals = pairDecimals.sub(token0.add(token1));
            k_ = reserve0.mul(reserve1).mul( 10 ** decimals );
        }
        else {
            uint decimals = token0.add(token1).sub(pairDecimals);
            k_ = reserve0.mul(reserve1).div( 10 ** decimals );
        }

    }

    function getTotalValue( address _pair ) public view returns ( uint _value ) {
        _value = getKValue( _pair ).sqrrt().mul(2);
    }

    function valuation( address _pair, uint amount_ ) external view override returns ( uint _value ) {
        uint totalValue = getTotalValue( _pair );
        uint totalSupply = IUniswapV2Pair( _pair ).totalSupply();

        _value = totalValue.mul( FixedPoint.fraction( amount_, totalSupply ).decode112with18() ).div( 1e18 );
    }

    function markdown( address _pair ) external view returns ( uint ) {
        ( uint reserve0, uint reserve1, ) = IUniswapV2Pair( _pair ).getReserves();

        uint reserve;
        if ( IUniswapV2Pair( _pair ).token0() == address(Time) ) {
            reserve = reserve1;
        } else {
            require(IUniswapV2Pair( _pair ).token1() == address(Time), "not a Time lp pair");
            reserve = reserve0;
        }
        return reserve.mul( 2 * ( 10 ** Time.decimals() ) ).div( getTotalValue( _pair ) );
    }
}
```