This PDF represents the changes performed while forking mathematical libraries from **uniswap** library to make it compatible with **Fountain** project's `Oracle` contract

uniswap npm paths

- "@uniswap\lib\contracts\libraries\FullMath.sol"
- "@uniswap\lib\contracts\libraries\BitMath.sol"
- "@uniswap\lib\contracts\libraries\FixedPoint.sol"

Project's github repo

https://github.com/puls369ar/fountain-solicy-overview/Oracle

Changes Description

In Solidity, type(uint224).max returns the **maximum value** that a uint224 (an unsigned 224-bit integer) can hold. The same we've got in older version inside `FullMath`, `BitMath` and `FixedPoint` libraries, by using uint224(-1)

Also arithmetic operations changed with binary ones to prevent uint256 issues

When calculating `pow2`

below find code's *diff* between forked and original versions

Diffchecker

Untitled diff

− 8 removals 86 lines

```
// SPDX-License-Identifier: GPL-3.0-or-later
pragma solidity >=0.5.0;

library BitMath {
    // Returns the 0 indexed position of the most significant bit of the input x
    // s.t. x >= 2**msb and x < 2**(msb+1)
    function mostSignificantBit(uint256 x) internal pure returns (uint8 r) {
        require(x > 0, 'BitMath::mostSignificantBit: zero');

        if (x >= 0x100000000000000000000000000000000) {
            x >>= 128;
            r += 128;
        }
        if (x >= 0x10000000000000000) {
            x >>= 64;
            r += 64;
        }
        if (x >= 0x100000000) {
            x >>= 32;
            r += 32;
        }
        if (x >= 0x10000) {
            x >>= 16;
```

+ 8 additions 86 lines

```
// SPDX-License-Identifier: GPL-3.0-or-later
pragma solidity >=0.8.20;

library BitMath {
    // returns the 0 indexed position of the most significant bit of the input x
    // s.t. x >= 2**msb and x < 2**(msb+1)
    function mostSignificantBit(uint256 x) internal pure returns (uint8 r) {
        require(x > 0, 'BitMath::mostSignificantBit: zero');

        if (x >= 0x100000000000000000000000000000000) {
            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;
    }

    // Returns the 0 indexed position of the least significant bit of the input x
    // s.t. (x & 2**lsb) != 0 and (x & (2**(lsb) - 1)) == 0)
    // i.e. the bit at the index is set and the mask of all lower bits is 0
    function leastSignificantBit(uint256 x) internal pure returns (uint8 r) {
        require(x > 0, 'BitMath::leastSignificantBit: zero');

        r = 255;
        if (x & type(uint128).max > 0) {
            r -= 128;
        } else {
            x >>= 128;
        }
        if (x & type(uint64).max > 0) {
            r -= 64;
        } else {
            x >>= 64;
        }
        if (x & type(uint32).max > 0) {
            r -= 32;
        } else {
            x >>= 32;
        }
        if (x & type(uint16).max > 0) {
```

```
            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;
    }

    // returns the 0 indexed position of the least significant bit of the input x
    // s.t. (x & 2**lsb) != 0 and (x & (2**(lsb) - 1)) == 0)
    // i.e. the bit at the index is set and the mask of all lower bits is 0
    function leastSignificantBit(uint256 x) internal pure returns (uint8 r) {
        require(x > 0, 'BitMath::leastSignificantBit: zero');

        r = 255;
        if (x & uint128(-1) > 0) {
            r -= 128;
        } else {
            x >>= 128;
        }
        if (x & uint64(-1) > 0) {
            r -= 64;
        } else {
            x >>= 64;
        }
        if (x & uint32(-1) > 0) {
            r -= 32;
        } else {
            x >>= 32;
        }
        if (x & uint16(-1) > 0) {
            r -= 16;
        } else {
            x >>= 16;
        }
        if (x & type(uint8).max > 0) {
            r -= 8;
        } else {
            x >>= 8;
        }
        if (x & 0xf > 0) {
            r -= 4;
        } else {
            x >>= 4;
        }
        if (x & 0x3 > 0) {
            r -= 2;
        } else {
            x >>= 2;
        }
        if (x & 0x1 > 0) r -= 1;
    }
}

```

```
            r -= 16;
        } else {
            x >>= 16;
        }
        if (x & uint8(-1) > 0) {
            r -= 8;
        } else {
            x >>= 8;
        }
        if (x & 0xf > 0) {
            r -= 4;
        } else {
            x >>= 4;
        }
        if (x & 0x3 > 0) {
            r -= 2;
        } else {
            x >>= 2;
        }
        if (x & 0x1 > 0) r -= 1;
    }
}

```

Diffchecker

Untitled diff

41 removals 147 lines

```
// SPDX-License-Identifier: GPL-3.0-or-later
pragma solidity >=0.8.20;

import '@uniswap/lib/contracts/libraries/FullMath.sol';
import '@uniswap/lib/contracts/libraries/Babylonian.sol';
import './BitMath.sol';

// A library for handling binary fixed-point numbers (https://en.wikipedia.org/wiki/Q_(number_format))
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;
    }
```

38 additions 147 lines

```
// SPDX-License-Identifier: GPL-3.0-or-later
pragma solidity >=0.8.20;

import '@uniswap/lib/contracts/libraries/FullMath.sol';
import '@uniswap/lib/contracts/libraries/Babylonian.sol';
import './BitMath.sol';

// a library for handling binary fixed point numbers (https://en.wikipedia.org/wiki/Q_(number_format))
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 public constant RESOLUTION = 112;
    uint256 public constant Q112 = 0x10000000000000000000000000000; // 2**112
    uint256 private constant Q224 = 0x100000000000000000000000000000000000000000000000000000000; // 2**224
    uint256 private constant LOWER_MASK = 0xffffffffffffffffffffffffffff; // Decimal of UQ*x112 (lower 112 bits)

    // 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);
    }

    // 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);
    }
```

```

    uint8 public constant RESOLUTION = 112;
    uint256 public constant Q112 = 0x10000000000000000000000000000; // 2**112
    uint256 private constant Q224 = 0x100000000000000000000000000000000000000000000000000000000; // 2**224
    uint256 private constant LOWER_MASK = 0xffffffffffffffffffffffffffff; // decimal of UQ*x112 (lower 112 bits)

    // 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);
    }

    // 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);
    }

    // Multiply a UQ112x112 by a uint, returning a UQ144x112
    // Reverts on overflow
    function mul(uq112x112 memory self, uint256 y) internal pure returns (uq144x112 memory) {
        uint256 z = 0;
        require(y == 0 || (z = self._x * y) / y == self._x, 'FixedPoint::mul: overflow');
        return uq144x112(z);
    }

    // Multiply a UQ112x112 by an int and decode, returning an int
    // Reverts on overflow
    function muli(uq112x112 memory self, int256 y) internal pure returns (int256) {
        uint256 z = FullMath.mulDiv(self._x, uint256(y < 0 ? -y : y), Q112);
        require(z < 2**255, 'FixedPoint::muli: overflow');
        return y < 0 ? -int256(z) : int256(z);
    }

    // Multiply a UQ112x112 by a UQ112x112, returning a UQ112x112
    // Lossy
    function muluq(uq112x112 memory self, uq112x112 memory other) internal pure returns (uq112x112 memory) {
        if (self._x == 0 || other._x == 0) {
            return uq112x112(0);
        }
        uint112 upper_self = uint112(self._x >> RESOLUTION); // * 2^0
        uint112 lower_self = uint112(self._x & LOWER_MASK); // * 2^-112
```

```

    // multiply a UQ112x112 by a uint, returning a UQ144x112
    // reverts on overflow
    function mul(uq112x112 memory self, uint256 y) internal pure returns (uq144x112 memory) {
        uint256 z = 0;
        require(y == 0 || (z = self._x * y) / y == self._x, 'FixedPoint::mul: overflow');
        return uq144x112(z);
    }

    // multiply a UQ112x112 by an int and decode, returning an int
    // reverts on overflow
    function muli(uq112x112 memory self, int256 y) internal pure returns (int256) {
        uint256 z = FullMath.mulDiv(self._x, uint256(y < 0 ? -y : y), Q112);
        require(z < 2**255, 'FixedPoint::muli: overflow');
        return y < 0 ? -int256(z) : int256(z);
    }

    // multiply a UQ112x112 by a UQ112x112, returning a UQ112x112
    // lossy
    function muluq(uq112x112 memory self, uq112x112 memory other) internal pure returns (uq112x112 memory) {
        if (self._x == 0 || other._x == 0) {
            return uq112x112(0);
        }
        uint112 upper_self = uint112(self._x >> RESOLUTION); // * 2^0
        uint112 lower_self = uint112(self._x & LOWER_MASK); // * 2^-112
        uint112 upper_other = uint112(other._x >> RESOLUTION); // * 2^0
        uint112 lower_other = uint112(other._x & LOWER_MASK); // * 2^-112

        // Partial products
        uint224 upper = uint224(upper_self) * upper_other; // * 2^0
        uint224 lower = uint224(lower_self) * lower_other; // * 2^-224
        uint224 uppers_lowero = uint224(upper_self) * lower_other; // * 2^-112
        uint224 uppero_lowers = uint224(upper_other) * lower_self; // * 2^-112

        // So the bit shift does not overflow
        require(upper <= type(uint112).max, 'FixedPoint::muluq: upper overflow');

        // This cannot exceed 256 bits, all values are 224 bits
        uint256 sum = uint256(upper << RESOLUTION) + uppers_lowero + uppero_lowers + (lower >> RESOLUTION);

        // So the cast does not overflow
        require(sum <= type(uint224).max, 'FixedPoint::muluq: sum overflow');

        return uq112x112(uint224(sum));
    }

    // Divide a UQ112x112 by a UQ112x112, returning a UQ112x112
```

```
        uint112 upper_other = uint112(other._x >> RESOLUTION); // * 2^0
        uint112 lower_other = uint112(other._x & LOWER_MASK); // * 2^-112

        // partial products
        uint224 upper = uint224(upper_self) * upper_other; // * 2^0
        uint224 lower = uint224(lower_self) * lower_other; // * 2^-224
        uint224 uppers_lowero = uint224(upper_self) * lower_other; // * 2^-112
        uint224 uppero_lowers = uint224(upper_other) * lower_self; // * 2^-112

        // so the bit shift does not overflow
        require(upper <= uint112(-1), 'FixedPoint::muluq: upper overflow');

        // this cannot exceed 256 bits, all values are 224 bits
        uint256 sum = uint256(upper << RESOLUTION) + uppers_lowero + uppero_lowers + (lower >> RESOLUTION);

        // so the cast does not overflow
        require(sum <= uint224(-1), 'FixedPoint::muluq: sum overflow');

        return uq112x112(uint224(sum));
    }

    // divide a UQ112x112 by a UQ112x112, returning a UQ112x112
    function divuq(uq112x112 memory self, uq112x112 memory other) internal pure returns (uq112x112 memory) {
        require(other._x > 0, 'FixedPoint::divuq: division by zero');
        if (self._x == other._x) {
            return uq112x112(uint224(Q112));
        }
        if (self._x <= type(uint144).max) {  // Changed here
            uint256 value = (uint256(self._x) << RESOLUTION) / other._x;
            require(value <= type(uint224).max, 'FixedPoint::divuq: overflow');
            return uq112x112(uint224(value));
        }

        uint256 result = FullMath.mulDiv(Q112, self._x, other._x);
        require(result <= type(uint224).max, 'FixedPoint::divuq: overflow');
        return uq112x112(uint224(result));
    }

    // Returns a UQ112x112 which represents the ratio of the numerator to the denominator
    // Can be lossy
    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);

```

```
    function divuq(uq112x112 memory self, uq112x112 memory other) internal pure returns (uq112x112 memory) {
        require(other._x > 0, 'FixedPoint::divuq: division by zero');
        if (self._x == other._x) {
            return uq112x112(uint224(Q112));
        }
        if (self._x <= uint144(-1)) {
            uint256 value = (uint256(self._x) << RESOLUTION) / other._x;
            require(value <= uint224(-1), 'FixedPoint::divuq: overflow');
            return uq112x112(uint224(value));
        }

        uint256 result = FullMath.mulDiv(Q112, self._x, other._x);
        require(result <= uint224(-1), 'FixedPoint::divuq: overflow');
        return uq112x112(uint224(result));
    }

    // returns a UQ112x112 which represents the ratio of the numerator to the denominator
    // can be lossy
    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 <= type(uint144).max) { // Changed here
            uint256 result = (numerator << RESOLUTION) / denominator;
            require(result <= type(uint224).max, 'FixedPoint::fraction: overflow');
            return uq112x112(uint224(result));
        } else {
            uint256 result = FullMath.mulDiv(numerator, Q112, denominator);
            require(result <= type(uint224).max, 'FixedPoint::fraction: overflow');
            return uq112x112(uint224(result));
        }
    }

    // Take the reciprocal of a UQ112x112
    // Reverts on overflow
    // Lossy
    function reciprocal(uq112x112 memory self) internal pure returns (uq112x112 memory) {
        require(self._x != 0, 'FixedPoint::reciprocal: reciprocal of zero');
        require(self._x != 1, 'FixedPoint::reciprocal: overflow');
        return uq112x112(uint224(Q224 / self._x));
    }

    // 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 <= type(uint144).max) { // Changed here
```

```
        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));
        }
    }

    // take the reciprocal of a UQ112x112
    // reverts on overflow
    // lossy
    function reciprocal(uq112x112 memory self) internal pure returns (uq112x112 memory) {
        require(self._x != 0, 'FixedPoint::reciprocal: reciprocal of zero');
        require(self._x != 1, 'FixedPoint::reciprocal: overflow');
        return uq112x112(uint224(Q224 / self._x));
    }

    // 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)));
    }
}

```

```
            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)));
    }
}

```

Diffchecker

Untitled diff

− 8 removals 52 lines

```
// SPDX-License-Identifier: CC-BY-4.0
pragma solidity >=0.4.0;

// Taken from https://medium.com/coinmonks/math-in-solidity-part-3-percents-and-proportions-4db014e080b1
// License is CC-BY-4.0
library FullMath {
    function fullMul(uint256 x, uint256 y) internal pure returns (uint256 l, uint256 h) {
        uint256 mm = mulmod(x, y, type(uint256).max);
        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 + 1);  // Changed here
        d /= pow2;
        l /= pow2;
```

+ 6 additions 52 lines

```
// SPDX-License-Identifier: CC-BY-4.0
pragma solidity >=0.4.0;

// taken from https://medium.com/coinmonks/math-in-solidity-part-3-percents-and-proportions-4db014e080b1
// license is CC-BY-4.0
library FullMath {
    function fullMul(uint256 x, uint256 y) internal 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); // Changed here, Just commenting it is useless and out of our use
        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;

        if (h == 0) return l / d;

        require(h < d, 'FullMath: FULLDIV_OVERFLOW');
        return fullDiv(l, h, d);
    }
}

```

```
        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;

        if (h == 0) return l / d;

        require(h < d, 'FullMath: FULLDIV_OVERFLOW');
        return fullDiv(l, h, d);
    }
}

```