Analysis of Integer Square Root Algorithms in Solidity[*]

Christopher H. Gorman

July 25, 2023

1 Introduction

We will look at various algorithms for computing integer square roots in Solidity and compare their efficiency (gas cost). This work extends [8] by adding more candidate algorithms and improving the analysis. Although the emphasis will be on efficiency and implementation details, we will also note when algorithms are *provably correct*, which is important for such a fundamental operation. A version of these results may be found online [7].

2 Integer Square Root Algorithm Overview

2.1 Definition

Given $a \in \mathbb{N}$, the *integer square root* of a is the unique integer m such that

$$m^2 \leq a < (m+1)^2 . \tag{2.1}$$

This will be denoted as

$$\textsc{Isqrt}(a) \coloneqq m \tag{2.2}$$

and is equivalent to

$$\textsc{Isqrt}(a) = \left\lfloor \sqrt{a} \right\rfloor , \tag{2.3}$$

the integer part of the square root of a.

2.2 Computation Methods

Most methods for computing integer square roots are based on Newton's method for computing square roots; see [3, Algorithm 1.7.1], [5, Algorithm 9.2.11], and [2, Algorithm 1.13]. Another method based on approximate square roots may be found in [6]. These algorithms usually come with a proof of correctness.

[*] Permanent ID of this document: `021e7116950e28e158ce35151b0dda3a`. Date: 2023.07.25.

The focus on this work is on computation within the Ethereum Virtual Machine. There is no native integer square root operation, so this operation must be performed using only basic integer operations. Given that this is a fundamental operation, it is important for this operation to be as efficient as possible *and provably correct*. The focus on this work is to look at various algorithms found online and compare them to see which is most efficient. Additional algorithms have been included which are *provably correct*; none of the referenced algorithms found online have a correctness proof.

3 Integer Square Root Algorithms

3.1 Algorithm Design Principles

The algorithm design philosophy is straightforward: provably correct algorithms which are efficient within the Ethereum Virtual Machine without any additional storage; this includes using no lookup values for initialization. It would be interesting to extend the results of [9] (which computes the integer square root of `uint64` and uses table lookups) to the `uint256` setting here.

3.2 Specific Algorithms in Solidity

3.2.1 Algorithms Found Online

The specific Isqrt algorithms we found online were `UniswapV2` (Listing 1) [14], `PRB` (Listing 2) [1], `OpenZeppelin` (Listing 3) [11], and `ABDK` (Listing 4) [4]. Two algorithms (`PRB` (Listing 2) and `ABDK` (Listing 4)) appear to be exactly the same aside from different integer notation and "unchecked" portions. The remaining algorithm `OpenZeppelin` (Listing 3) is equivalent to `PRB` (Listing 2) and `ABDK` (Listing 4) outside of separating the function calls.

None of these algorithms reference a proof of correctness. We note that `UniswapV2` (Listing 1) is provably correct as it implements [3, Algorithm 1.7.1] (although it doesn't follow the comment in [3, Chapter 1.7, Remarks] that "[w]hen actually implementing this algorithm, the initialization step must be modified"). Furthermore, `OpenZeppelin` (Listing 3) [11] says that we "need at most 7 iteration[s]" while `ABDK` (Listing 4) [4] states that "[s]even iterations should be enough"; `PRB` (Listing 2) [1] has similar language.

3.2.2 Provably Correct Algorithms

Due to the requirement of seeking *provably correct* algorithms for computing integer square roots, additional algorithms were written.

The first provably correct algorithm converts results from [6] into a Solidity algorithm; we denote this as `Python` (Listing 5). The next algorithms come in two groups of 3: one group is based on unrolled Newton iterations, and the second group is based on Newton iterations using a `while` loop. They are `Unrolled1` (Listing 6), `Unrolled2` (Listing 7), `Unrolled3` (Listing 8), `While1` (Listing 9), `While2` (Listing 10), and `While3` (Listing 11). We note that `Unrolled3` (Listing 8) was originally presented in [8].

Listing 1: Method for computing Integer Square Roots from `UniswapV2` [14]

```
// License: GPL-3.0
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;
    }
}
```

Here, `Unrolled` refers to the fact that the Newton iterations are unrolled: a fixed number of iterations is performed; `While` refers to the fact that the Newton iterations are performed within a `while` loop. Label 1 refers to the fact that the initialization begins at the largest power-of-2 less than or equal to integer square root value; Label 2 refers to the fact that the initialization begins at the smallest power-of-2 greater than the integer square root value; Label 3 refers to the fact that the initialization begins at the arithmetic mean of the power-of-2 above and below the integer square root value.

We note that in `Unrolled1` and `Unrolled2` (Listings 6 and 7), 7 Newton iterations are performed; in `Unrolled3` (Listings 8), only 6 Newton iterations are performed. The results from [8] show why the final check in `Unrolled` ensures a valid result. Also, in `While1` and `While3` (Listings 9 and 11), one Newton iteration is performed before entering the `while` loop (this is required for the stopping criterion); this is not present in `While2` (Listing 10). From [3, Algorithm 1.7.1], no additional check is needed before returning the result.

3.3 General Algorithm Discussion

When using Newton iterations, it is critical to use a good initial approximation to ensure a low overall cost. Most of the algorithms use the same initialization value: the largest power-of-2 less than or equal to the integer square root. These are `PRB` (Listing 2), `OpenZeppelin` (Listing 3), `ABDK` (Listing 4), `Unrolled1` (Listing 6), and `While1` (Listing 9). Other good initialization values are `Unrolled2` (Listing 7) and `While2` (Listing 10) which follow the initialization suggestions of [3, Chapter 1.7, Remarks (2)] and [5, Algorithm 9.2.11]) as well as `Unrolled3` (Listing 8) and `While3` (Listing 11) (the new initialization from [8]). `UniswapV2` (Listing 1) has the worst initialization value (the input value itself).

For a better understanding of approximate square roots, see [6]; the discussion in [8] on approximate square roots is based on [6].

Listing 2: Method for computing Integer Square Roots from PRB [1]

```
// SPDX-License-Identifier: MIT
function sqrt(uint256 x) internal pure returns (uint256 result) {
    if (x == 0) { return 0; }
    uint256 xAux = uint256(x);
    result = 1;
    if (xAux >= 2 ** 128) {
        xAux >>= 128;   result <<= 64;
    }
    if (xAux >= 2 ** 64) {
        xAux >>= 64;    result <<= 32;
    }
    if (xAux >= 2 ** 32) {
        xAux >>= 32;    result <<= 16;
    }
    if (xAux >= 2 ** 16) {
        xAux >>= 16;    result <<= 8;
    }
    if (xAux >= 2 ** 8) {
        xAux >>= 8;     result <<= 4;
    }
    if (xAux >= 2 ** 4) {
        xAux >>= 4;     result <<= 2;
    }
    if (xAux >= 2 ** 2) {
                        result <<= 1;
    }
    unchecked {
        result = (result + x / result) >> 1;
        result = (result + x / result) >> 1;
        result = (result + x / result) >> 1;
        result = (result + x / result) >> 1;
        result = (result + x / result) >> 1;
        result = (result + x / result) >> 1;
        result = (result + x / result) >> 1;
        uint256 roundedResult = x / result;
        if (result >= roundedResult) {
            result = roundedResult;
        }
    }
}
```

Listing 3: Method for computing Integer Square Roots from `OpenZeppelin` [11]

```
// SPDX-License-Identifier: MIT
function sqrt(uint256 a) internal pure returns (uint256) {
    if (a == 0) {
        return 0;
    }
    uint256 result = 1 << (log2(a) >> 1);
    // At this point `result` is an estimation with one bit of
    // precision. We know the true value is a uint128, since it is
    // the square root of a uint256. Newton's method converges
    // quadratically (precision doubles at every iteration). We
    // thus need at most 7 iteration to turn our partial result
    // with one bit of precision into the expected uint128 result.
    unchecked {
        result = (result + a / result) >> 1;
        result = (result + a / result) >> 1;
        result = (result + a / result) >> 1;
        result = (result + a / result) >> 1;
        result = (result + a / result) >> 1;
        result = (result + a / result) >> 1;
        result = (result + a / result) >> 1;
        return min(result, a / result);
    }
}

function log2(uint256 value) internal pure returns (uint256) {
    uint256 result = 0;
    unchecked {
        if (value >> 128 > 0) { value >>= 128; result += 128; }
        if (value >>  64 > 0) { value >>=  64; result +=  64; }
        if (value >>  32 > 0) { value >>=  32; result +=  32; }
        if (value >>  16 > 0) { value >>=  16; result +=  16; }
        if (value >>   8 > 0) { value >>=   8; result +=   8; }
        if (value >>   4 > 0) { value >>=   4; result +=   4; }
        if (value >>   2 > 0) { value >>=   2; result +=   2; }
        if (value >>   1 > 0) {                result +=   1; }
    }
    return result;
}

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

Listing 4: Method for computing Integer Square Roots from ABDK [4]

```
// SPDX-License-Identifier: BSD-4-Clause
function sqrt(uint256 x) private pure returns (uint128) {
    unchecked {
        if (x == 0) return 0;
        else {
            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 >= 0x4) {
                            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 should be enough
            uint256 r1 = x / r;
            return uint128 (r < r1 ? r : r1);
        }
    }
}
```

Listing 5: Provably correct method for computing Integer Square Roots based on [6]

```
// SPDX-License-Identifier: 0BSD
function sqrt(uint256 x) internal pure returns (uint256) {
    unchecked {
        if (x <= 1) { return x; }
        if (x >= ((1 << 128) - 1)**2) { return (1 << 128) - 1; }

        // Here, e represents the bit length
        uint256 e = 1;

        // Here, result is a copy of x to compute the bit length
        uint256 result = x;
        if (result >= (1 << 128)) { result >>= 128; e += 128; }
        if (result >= (1 <<  64)) { result >>=  64; e +=  64; }
        if (result >= (1 <<  32)) { result >>=  32; e +=  32; }
        if (result >= (1 <<  16)) { result >>=  16; e +=  16; }
        if (result >= (1 <<   8)) { result >>=   8; e +=   8; }
        if (result >= (1 <<   4)) { result >>=   4; e +=   4; }
        if (result >= (1 <<   2)) { result >>=   2; e +=   2; }
        if (result >= (1 <<   1)) {                  e +=   1; }

        // e is currently bit length; we overwrite it to scale x
        e = (256 - e) >> 1;

        // m now satisfies 2**254 <= m < 2**256
        uint256 m = x << (2 * e);

        // result now stores the result
        result = 1 + (m >> 254);
        result = (result <<  1) + (m >> 251) / result;
        result = (result <<  3) + (m >> 245) / result;
        result = (result <<  7) + (m >> 233) / result;
        result = (result << 15) + (m >> 209) / result;
        result = (result << 31) + (m >> 161) / result;
        result = (result << 63) + (m >>  65) / result;
        result >>= e;

        if (result * result <= x) {
            return result;
        }
        return result -1;
    }
}
```

Listing 6: Provably correct method for computing Integer Square Roots (`Unrolled1`)

```
// SPDX-License-Identifier: 0BSD
function sqrt(uint256 x) internal pure returns (uint256) {
    unchecked {
        if (x <= 1) { return x; }
        if (x >= ((1 << 128) - 1)**2) { return (1 << 128) - 1; }

        uint256 xAux = x;

        uint256 result = 1;

        if (xAux >= (1 << 128)) { xAux >>= 128; result <<= 64; }
        if (xAux >= (1 <<  64)) { xAux >>=  64; result <<= 32; }
        if (xAux >= (1 <<  32)) { xAux >>=  32; result <<= 16; }
        if (xAux >= (1 <<  16)) { xAux >>=  16; result <<=  8; }
        if (xAux >= (1 <<   8)) { xAux >>=   8; result <<=  4; }
        if (xAux >= (1 <<   4)) { xAux >>=   4; result <<=  2; }
        if (xAux >= (1 <<   2)) {               result <<=  1; }

        result = (result + x / result) >> 1;
        result = (result + x / result) >> 1;
        result = (result + x / result) >> 1;
        result = (result + x / result) >> 1;
        result = (result + x / result) >> 1;
        result = (result + x / result) >> 1;
        result = (result + x / result) >> 1;

        if (result * result <= x) {
            return result;
        }
        return result -1;
    }
}
```

Listing 7: Provably correct method for computing Integer Square Roots (`Unrolled2`)

```
// SPDX-License-Identifier: 0BSD
function sqrt(uint256 x) internal pure returns (uint256) {
    unchecked {
        if (x <= 1) { return x; }
        if (x >= ((1 << 128) - 1)**2) { return (1 << 128) - 1; }

        uint256 xAux = x;

        uint256 result = 2;

        if (xAux >= (1 << 128)) { xAux >>= 128; result <<= 64; }
        if (xAux >= (1 <<  64)) { xAux >>=  64; result <<= 32; }
        if (xAux >= (1 <<  32)) { xAux >>=  32; result <<= 16; }
        if (xAux >= (1 <<  16)) { xAux >>=  16; result <<=  8; }
        if (xAux >= (1 <<   8)) { xAux >>=   8; result <<=  4; }
        if (xAux >= (1 <<   4)) { xAux >>=   4; result <<=  2; }
        if (xAux >= (1 <<   2)) {               result <<=  1; }

        result = (result + x / result) >> 1;
        result = (result + x / result) >> 1;
        result = (result + x / result) >> 1;
        result = (result + x / result) >> 1;
        result = (result + x / result) >> 1;
        result = (result + x / result) >> 1;
        result = (result + x / result) >> 1;

        if (result * result <= x) {
            return result;
        }
        return result -1;
    }
}
```

Listing 8: Provably correct method for computing Integer Square Roots (`Unrolled3`)

```
// SPDX-License-Identifier: 0BSD
function sqrt(uint256 x) internal pure returns (uint256) {
    unchecked {
        if (x <= 1) { return x; }
        if (x >= ((1 << 128) - 1)**2) { return (1 << 128) - 1; }

        uint256 xAux = x;

        uint256 result = 1;

        if (xAux >= (1 << 128)) { xAux >>= 128; result <<= 64; }
        if (xAux >= (1 <<  64)) { xAux >>=  64; result <<= 32; }
        if (xAux >= (1 <<  32)) { xAux >>=  32; result <<= 16; }
        if (xAux >= (1 <<  16)) { xAux >>=  16; result <<=  8; }
        if (xAux >= (1 <<   8)) { xAux >>=   8; result <<=  4; }
        if (xAux >= (1 <<   4)) { xAux >>=   4; result <<=  2; }
        if (xAux >= (1 <<   2)) {               result <<=  1; }

        result += (result >> 1);

        result = (result + x / result) >> 1;
        result = (result + x / result) >> 1;
        result = (result + x / result) >> 1;
        result = (result + x / result) >> 1;
        result = (result + x / result) >> 1;
        result = (result + x / result) >> 1;

        if (result * result <= x) {
            return result;
        }
        return result -1;
    }
}
```

Listing 9: Provably correct method for computing Integer Square Roots (`While1`)

```
// SPDX-License-Identifier: 0BSD
function sqrt(uint256 x) internal pure returns (uint256) {
    unchecked {
        if (x <= 1) { return x; }
        if (x >= ((1 << 128) - 1)**2) { return (1 << 128) - 1; }

        uint256 xAux = x;

        uint256 result = 1;

        if (xAux >= (1 << 128)) { xAux >>= 128; result <<= 64; }
        if (xAux >= (1 <<  64)) { xAux >>=  64; result <<= 32; }
        if (xAux >= (1 <<  32)) { xAux >>=  32; result <<= 16; }
        if (xAux >= (1 <<  16)) { xAux >>=  16; result <<=  8; }
        if (xAux >= (1 <<   8)) { xAux >>=   8; result <<=  4; }
        if (xAux >= (1 <<   4)) { xAux >>=   4; result <<=  2; }
        if (xAux >= (1 <<   2)) {               result <<=  1; }

        xAux = (result + x / result) >> 1;
        result = xAux;
        xAux = (result + x / result) >> 1;
        while (xAux < result) {
            result = xAux;
            xAux = (result + x / result) >> 1;
        }
        return result;
    }
}
```

Listing 10: Provably correct method for computing Integer Square Roots (`While2`)

```
// SPDX-License-Identifier: 0BSD
function sqrt(uint256 x) internal pure returns (uint256) {
    unchecked {
        if (x <= 1) { return x; }
        if (x >= ((1 << 128) - 1)**2) { return (1 << 128) - 1; }

        uint256 xAux = x;

        uint256 result = 2;

        if (xAux >= (1 << 128)) { xAux >>= 128; result <<= 64; }
        if (xAux >= (1 <<  64)) { xAux >>=  64; result <<= 32; }
        if (xAux >= (1 <<  32)) { xAux >>=  32; result <<= 16; }
        if (xAux >= (1 <<  16)) { xAux >>=  16; result <<=  8; }
        if (xAux >= (1 <<   8)) { xAux >>=   8; result <<=  4; }
        if (xAux >= (1 <<   4)) { xAux >>=   4; result <<=  2; }
        if (xAux >= (1 <<   2)) {               result <<=  1; }

        xAux = (result + x / result) >> 1;
        while (xAux < result) {
            result = xAux;
            xAux = (result + x / result) >> 1;
        }
        return result;
    }
}
```

Listing 11: Provably correct method for computing Integer Square Roots (While3)

```
// SPDX-License-Identifier: 0BSD
function sqrt(uint256 x) internal pure returns (uint256) {
    unchecked {
        if (x <= 1) { return x; }
        if (x >= ((1 << 128) - 1)**2) { return (1 << 128) - 1; }

        uint256 xAux = x;

        uint256 result = 1;

        if (xAux >= (1 << 128)) { xAux >>= 128; result <<= 64; }
        if (xAux >= (1 <<  64)) { xAux >>=  64; result <<= 32; }
        if (xAux >= (1 <<  32)) { xAux >>=  32; result <<= 16; }
        if (xAux >= (1 <<  16)) { xAux >>=  16; result <<=  8; }
        if (xAux >= (1 <<   8)) { xAux >>=   8; result <<=  4; }
        if (xAux >= (1 <<   4)) { xAux >>=   4; result <<=  2; }
        if (xAux >= (1 <<   2)) {               result <<=  1; }

        result += (result >> 1);

        xAux = (result + x / result) >> 1;
        result = xAux;
        xAux = (result + x / result) >> 1;
        while (xAux < result) {
            result = xAux;
            xAux = (result + x / result) >> 1;
        }
        return result;
    }
}
```

4 Comparison of Gas Costs

Because all operations are performed within the Ethereum Virtual Machine, the primary cost metric is the overall gas cost. Thus, the focus is on minimizing the total gas used. Ideally, the best algorithm will have the lowest maximum, mean, and median gas costs. We note that the results here are different from those in [8] for three reasons: first, some of the algorithms are different (and we chose the most recent versions of those found online); second, we ensure that all input values are unique (there may have been double counting certain evaluations); third, different input values were chosen.

4.1 Data Point Selection

Any gas cost comparison requires the selection of `uint256` values for input. The values were chosen in this way:

- all numbers of the form 2^k, $2^k - 1$, and $2^k + 1$ for nonnegative values of k;
- $v - 1$, v, and $v + 1$ for $v = (2^{128} - 1)^2$ (these were chosen because of an edge case in the new algorithms);
- random values chosen according to the loguniform distribution [13] on $[1, 2^{256}]$ when the random seed is initialized to 0 [10].

The number of random samples were increased until there were 2048 unique values. While it is clear that the particular distribution affects to results, we believe choosing the input values in this way reduces the risk of potential bias. There were a total of 768 deterministic values chosen and a total of 1303 random samples were required to determine the remaining 1280 values.

In addition to measuring the gas cost, the result of each integer square root operation was validated using Python's integer square root [6, 12]. There was no instance in which an algorithm ever produced an incorrect result; this is, of course, not a proof that `PRB` (Listing 2), `OpenZeppelin` (Listing 3), and `ABDK` (Listing 4) are correct, but rather that there are no known values where these algorithms fail.

4.2 Summary Statistics

A listing of the summary statistics describing the runs may be found in Tables 1 and 2. From here, we see that the algorithm with the smallest maximum, mean, and median gas costs is `Unrolled3`. Although the standard deviation of `Unrolled3` is not the smallest, it is not particularly large. We note that standard deviations of `UniswapV2`, `While1`, `While2`, and `While3` are larger than the others, and we look at this more closely below.

4.3 Detailed Analysis

We now plan to take a closer look at how the gas cost varies with the argument. Plots of the gas costs may be found in Figures 1 and 2.

	UniswapV2	PRB	OpenZeppelin	ABDK	Python
Max	34 205	878	1021	881	963
Mean	17 734	795	950	803	885
Median	17 639	798	949	803	888
Std	9558	34	30	33	44

Table 1: Here are some statistics related to the gas cost data from Figure 1. We recall that the UniswapV2 and Python algorithms are provably correct. These results are for the tests in Section 4.

	Unrolled1	Unrolled2	Unrolled3	While1	While2	While3
Max	858	851	831	1246	1190	1177
Mean	776	775	749	853	902	870
Median	779	777	752	898	932	895
Std	42	42	41	180	160	141

Table 2: Here are some statistics related to the gas cost data from Figure 2. We recall that all of these algorithms are provably correct. We see that Unrolled3 has the smallest gas costs in terms of maximum, mean, and median. These results are for the tests in Section 4.

Aside from UniswapV2, While1, While2, and While3, we see from Figures 1 and 2 that the gas costs are relatively constant across argument. In looking at the algorithms, this is not surprising: UniswapV2, While1, While2, and While3 all have while loops. In Figure 2, we can also see that small arguments in While algorithms have lower gas costs than the corresponding Unrolled algorithms. This is not surprising, given that fewer Newton iterations are required, and exiting before performing unnecessary operations results in a lower cost.

In order to determine when the Unrolled algorithms outperform the While algorithms, we determine which algorithms have the lowest gas cost for each argument evaluated. After looking at the gas costs from each algorithm, we counted the number of times each algorithm was minimal; the results may be found in Table 3. Additionally, we plot the instances where certain algorithms were minimal in Figure 3; a histogram may be found in Figure 4. There does not appear to be a particular pattern in the value distribution. The main trend is that Unrolled3 (Listing 8) consistently performs well across the range of arguments. The only region where this does not hold is for small arguments, where other algorithms (While) performed better; this makes sense because fewer Newton iterations are required.

A more extensive analysis may be found in Appendix A; the results are similar.

5 Conclusion

After comparing various algorithms for total gas cost, we found that the most efficient algorithm based on the sampled values is Unrolled3 (Listing 8), and a proof of correctness may be found in [8, Appendix B].

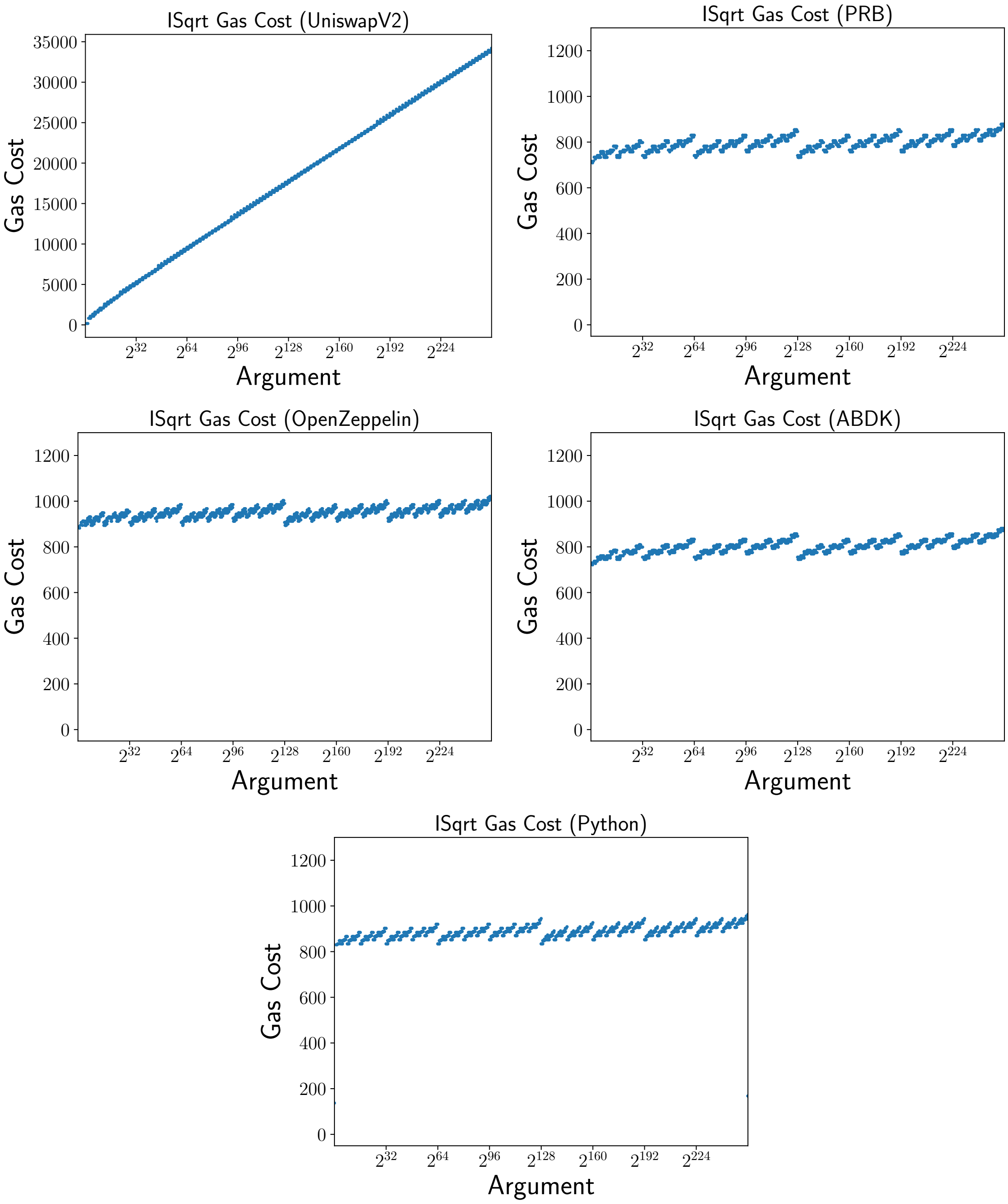

Figure 1: Here are plots of individual gas prices from Table 1. Note that all plots except for `UniswapV2` have the same y-axis. These results are for the tests in Section 4.

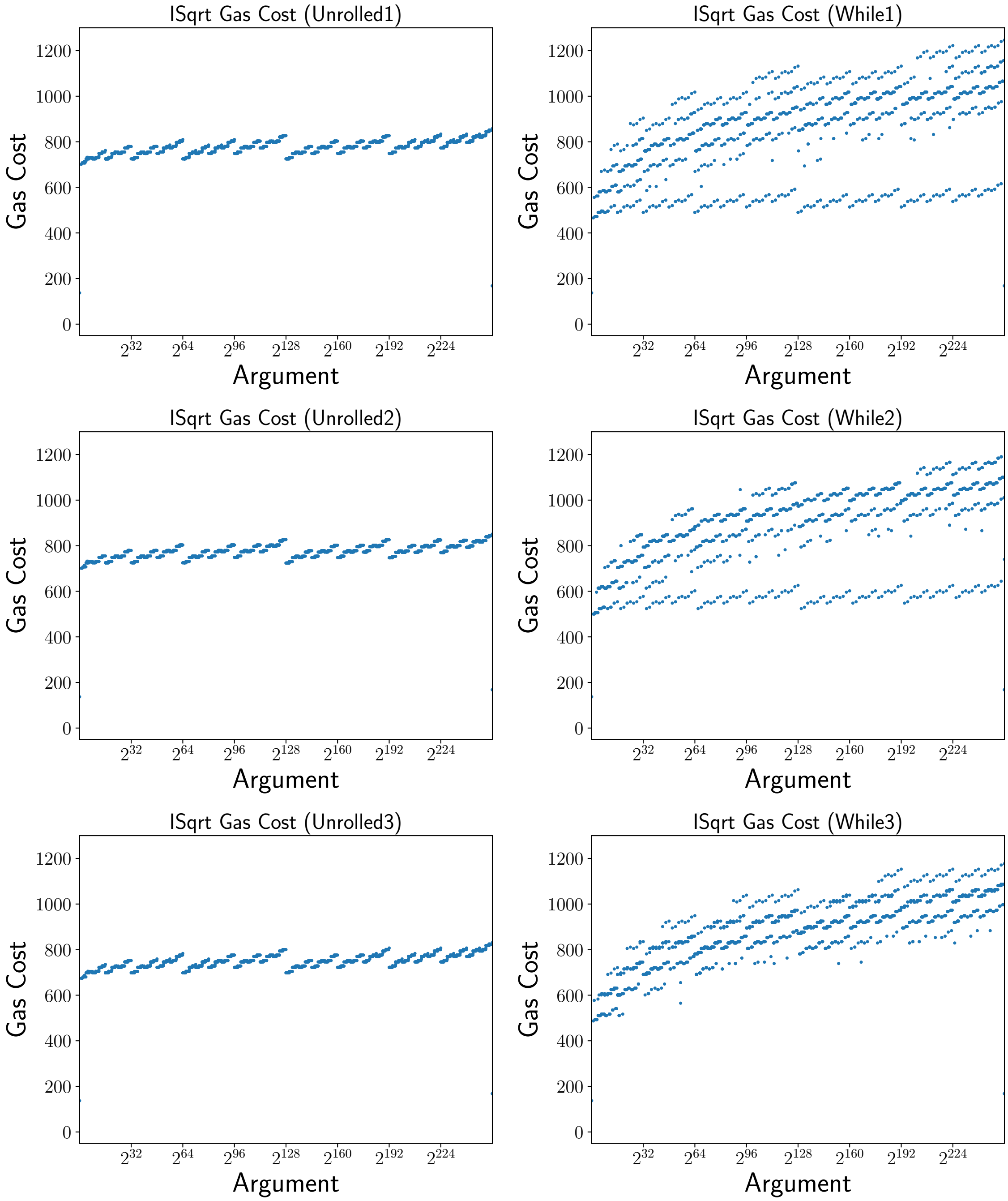

Figure 2: Here are plots of individual gas prices from Table 2. All plots have the same y-axis. These results are for the tests in Section 4.

Total	2048
UniswapV2	2
Python	5
Unrolled1	5
Unrolled2	5
Unrolled3	1222
While1	386
While2	156
While3	297

Table 3: Here are number of times each method had minimal gas costs; methods not included were never minimal. These results are for the tests in Section 4.

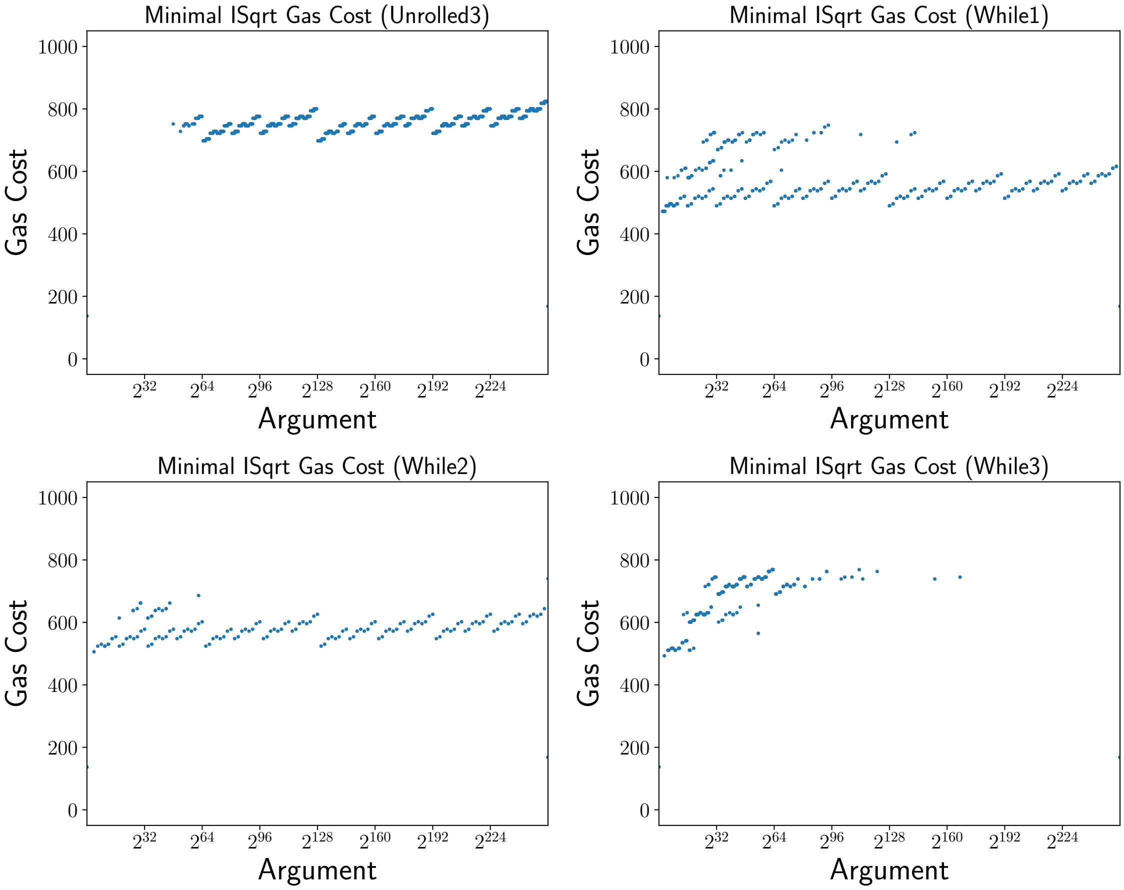

Figure 3: Here we plot the instances where each algorithm is minimal. A histogram of this data may be found in Figure 4. These results are the for the tests in Section 4.

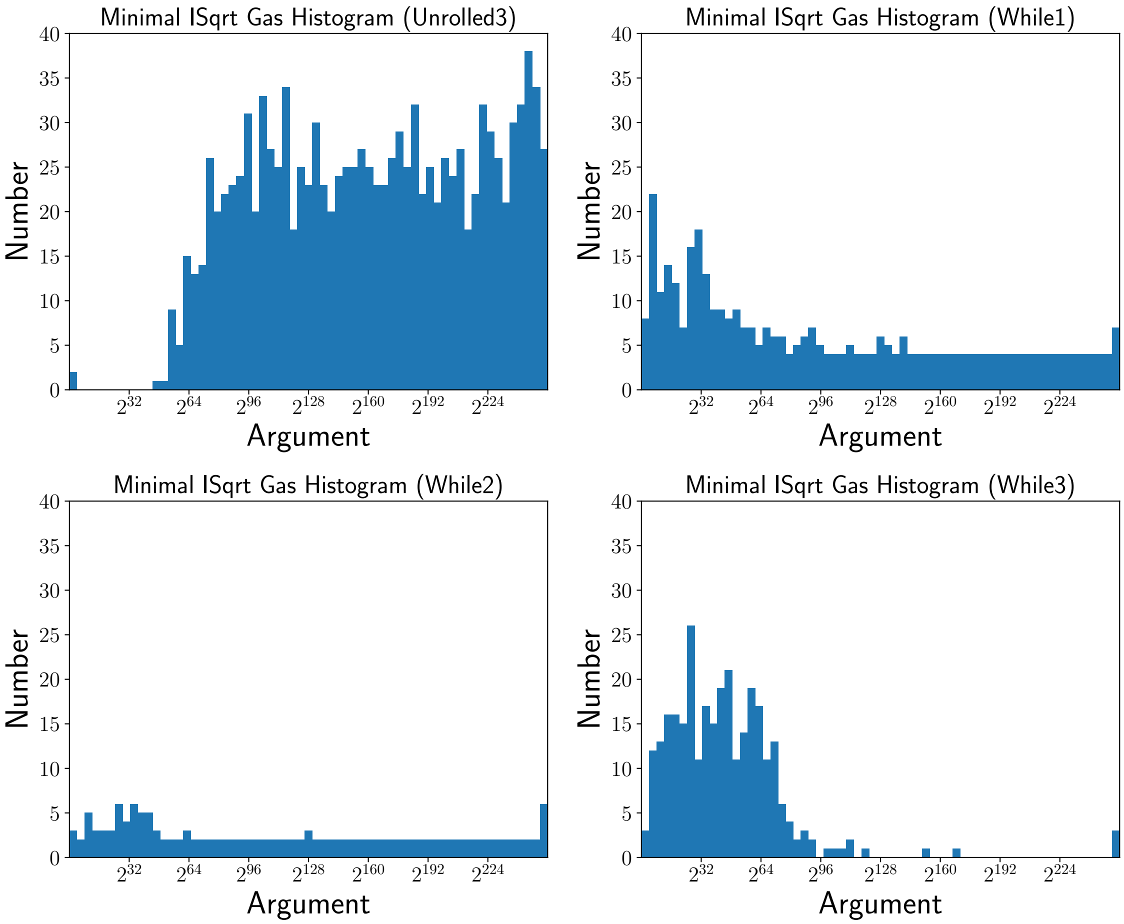

Figure 4: Here we plot a histogram showing where each algorithm is minimal and provides more detail to the results in Table 3. Each bin counts the total number instances where the algorithm's gas cost was minimal. These results are for the tests in Section 4.

References

[1] Paul Razvan Berg. *Integer Square Root Algorithm in Solidity*. June 2023. URL: https://github.com/PaulRBerg/prb-math/blob/28055f6cd9a2367f9ad7ab6c8e01c9ac8e9acc61/src/Common.sol#L595.

[2] Richard P. Brent and Paul Zimmermann. *Modern Computer Arithmetic*. Version 0.5.1. 2010. DOI: https://doi.org/10.48550/arXiv.1004.4710.

[3] Henri Cohen. *A Course in Computational Algebraic Number Theory*. Graduate Texts in Mathematics 138. Springer, 1993.

[4] ABDK Consulting. *Integer Square Root Algorithm in Solidity*. Sept. 2020. URL: https://github.com/abdk-consulting/abdk-libraries-solidity/blob/9e69beb255e2f87d67b44047d ABDKMath64x64.sol#L727.

[5] Richard Crandall and Carl Pomerance. *Prime Numbers: A Computational Perspective*. 2nd ed. Springer, 2005.

[6] Mark Dickinson. *Python's integer square root algorithm*. Jan. 2022. URL: https://github.com/mdickinson/snippets/blob/master/papers/isqrt/isqrt.pdf.

[7] Christopher H. Gorman. *Analysis of Integer Square Root Algorithms in Solidity*. 2023. URL: https://github.com/chgorman/isqrt-gas.

[8] Christopher H. Gorman. *Efficient Integer Square Roots in Solidity*. Nov. 2022. URL: https://github.com/alicenet/.github/blob/main/docs/efficient_isqrt.pdf.

[9] Guillaume Melquiond and Raphaël Rieu-Helft. "Formal Verification of a State-of-the-Art Integer Square Root". In: *2019 IEEE 26th Symposium on Computer Arithmetic (ARITH)*. IEEE. 2019, pp. 183–186. URL: https://inria.hal.science/hal-02092970/file/main.pdf.

[10] Numpy. ***numpy.random.seed***. URL: https://numpy.org/doc/stable/reference/random/generated/numpy.random.seed.html (visited on 07/16/2023).

[11] OpenZeppelin. *Integer Square Root Algorithm in Solidity*. July 2023. URL: https://github.com/OpenZeppelin/openzeppelin-contracts/blob/6bf68a41d19f3d6e364d8c207cb7d1a contracts/utils/math/Math.sol#L220.

[12] Python. *Mathematical Functions*. URL: https://docs.python.org/3/library/math.html#math.isqrt (visited on 07/16/2023).

[13] Scipy. ***scipy.stats.loguniform***. URL: https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.loguniform.html (visited on 07/16/2023).

[14] Uniswap. *Integer Square Root Algorithm in Solidity*. Jan. 2020. URL: https://github.com/Uniswap/v2-core/blob/585ee2ef824db671b71e351a91860a003c05431d/contracts/libraries/Math.sol#L11.

A Extended Analysis

While the results in Section 4 are good, it is desirable to have a more extensive analysis. The only algorithms tested are `Unrolled3` (Listing 8), `While1` (Listing 9), `While2` (Listing 10), and `While3` (Listing 11), as these performed the best. The deterministic and random values will be separated. All values tested in Section 4 are included in these tests.

A.1 Deterministic Tests

A.1.1 Overview

We test additional deterministic values. First, we include all deterministic values from Section 4. Additional values are of the form $2^k + 2^j$ for $j, k \in \mathbb{N}$ and $j < k$. The total number of values were 33154. The results are shown in Table 4 and Figure 5.

The minimal values have a somewhat different trend than before: `Unrolled3` still performs well for larger numbers, but `While1` does very good, too. This may be explained by the initialization value in this instance starting *very close* to the actual value; see the discussion in Appendix A.1.2. It is interesting that `Unrolled3` performed as well as it did in this situation which favors `While1`.

The results here show that it is easy to (unwittingly) test values which are biased toward one particular algorithm. This is the reason for testing a large collection of random values in Appendix A.2.

A.1.2 Detailed Look

We now take a closer look at particular values: we look at all values $2^{254} \leq v < 2^{255}$ and then $2^{255} \leq v < 2^{256}$. In this instance, we are just comparing the gas cost of `Unrolled3` and `While1`. We expect these two collections to be characteristic of the remaining gas values.

Values $2^{254} \leq v < 2^{255}$ For `Unrolled3`, there are only two gas costs in this range: 824 and 831. `While1` starts off at a low gas cost of 616; it then jumps to 706, 796, 886, 976, and 1066. We estimate that each Newton iteration costs 90 gas.

Values $2^{255} \leq v < 2^{256}$ For `Unrolled3`, the gas cost is predominantly 824; there is one instance of 831, and three values of 168 (from early exit). Almost all gas cost values for `While1` are 1066, with some larger values and a few smaller (early exit).

A.2 Loguniform Tests

The tests in Section 4 ran 1280 values which came from a loguniform distribution [13]. Here, we perform the same test using more samples; in particular, we use $16384 = 2^{14}$ random values from 16808 samples. The results are shown in Table 5 and Figure 6.

There is no significant difference between the results in Tables 3 and 5 or Figures 4 and 6. The results are clear: `Unrolled3` has almost 75% of the minimal cost for random values.

A.3 Conclusion

This extended analysis gives additional insight into how the algorithms perform but with the same result: `Unrolled3` is the best algorithm for a generic `uint256` value.

Total	33 154
Unrolled3	15 769
While1	15 639
While2	133
While3	1628

Table 4: Here are number of times each method had minimal gas costs; these are results for the additional deterministic values based around powers-of-2. These results are for the tests in Appendix A.1.

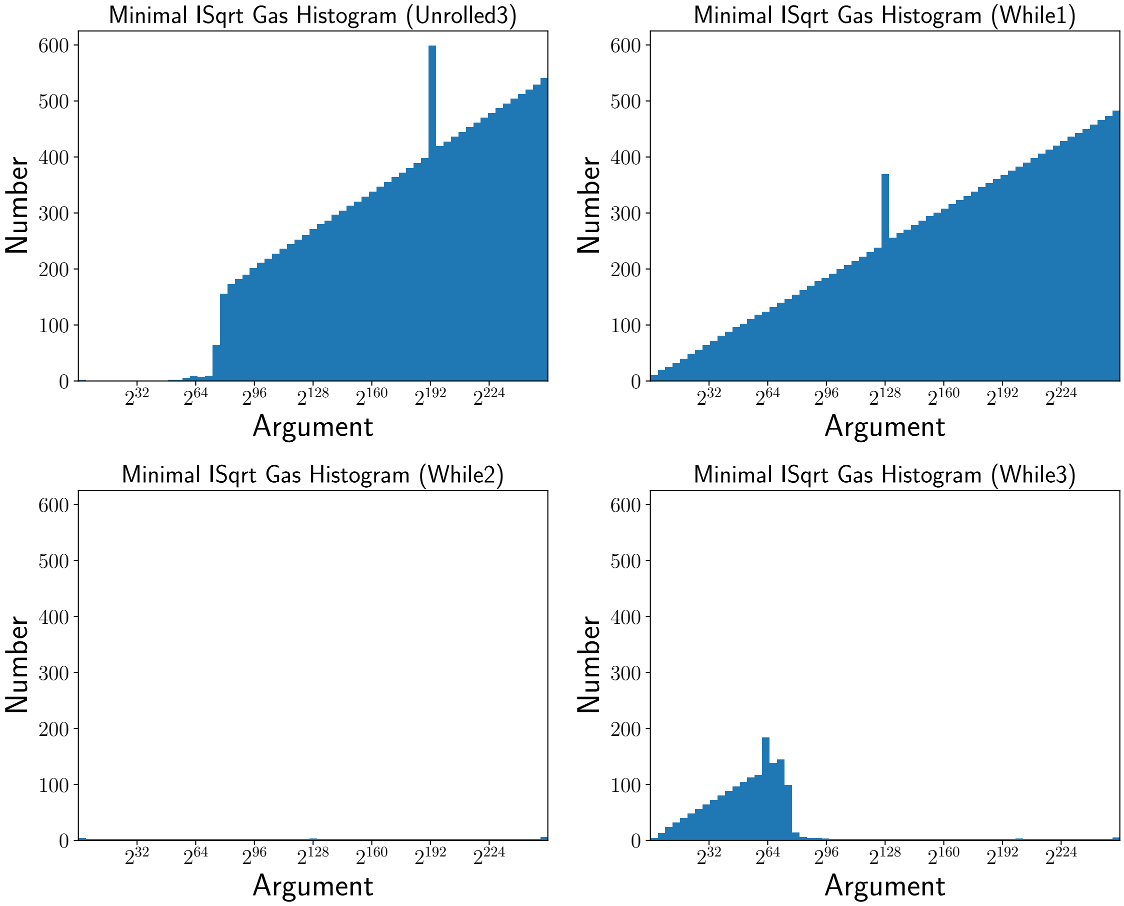

Figure 5: Here we plot a histogram showing where each algorithm is minimal when using a larger sample of deterministic values. Each bin counts the total number instances where the algorithm's gas cost was minimal. These results are for the tests in Appendix A.1.

Total	16 384
`Unrolled3`	12 183
`While1`	1537
`While2`	320
`While3`	2347

Table 5: Here are number of times each method had minimal gas costs; these are results for the additional loguniform random values. These results are for the tests in Appendix A.2.

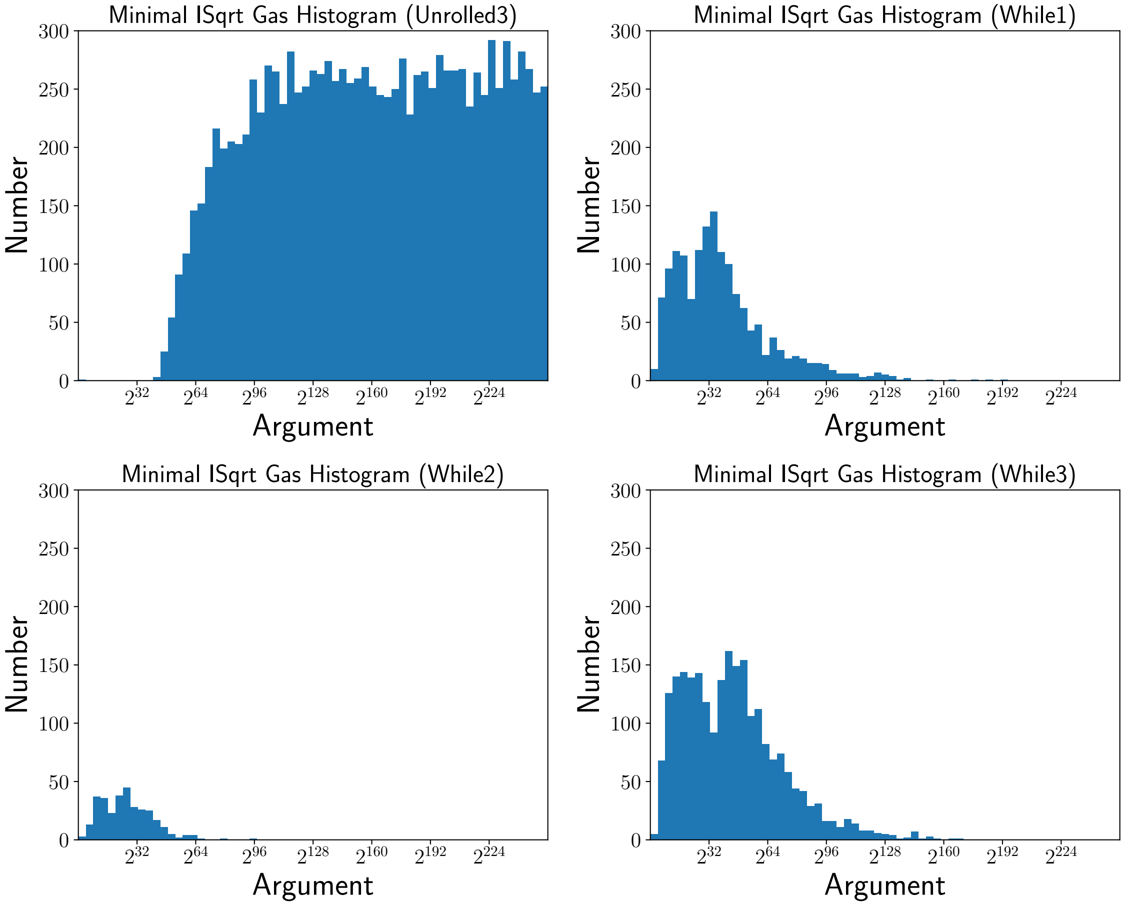

Figure 6: Here we plot a histogram showing where each algorithm is minimal when using a larger sample of loguniform random values. Each bin counts the total number instances where the algorithm's gas cost was minimal. These results are for the tests in Appendix A.2.