Analysis of Integer Square Root Algorithms in Solidity[*]

Christopher H. Gorman

December 15, 2024

Abstract

We investigate methods for computing integer square roots within the Ethereum blockchain while developing new methods which are provably correct. After determining the cost of each algorithm at various inputs, we look at the summary statistics of each method to determine which procedure is most efficient.

1 Introduction

Basic integer arithmetic and bit operations are included within the Ethereum blockchain [30]; all other functions must be derived from those simple operations. We will look at various algorithms for computing integer square roots on the Ethereum blockchain (using the Solidity programming language [26]) and compare their efficiency. This work extends [12] by adding candidate algorithms and improving the analysis. The emphasis is on *provably correct* algorithms which are efficient. A version of these results may be found online [10].

2 Mathematical Definition of Integer Square Root

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.

[*]Permanent ID of this document: `021e7116950e28e158ce35151b0dda3a`. Date: 2024.12.15.

EVM Ops	Gas
$+, -$	3
$\times, \div$	5
and, or, xor, not	3
$<, >, =$	3
$\gg, \ll$	3
Transaction	$21 \cdot 10^3$
sstore (clean)	$20 \cdot 10^3$
sstore (dirty)	$5 \cdot 10^3$
Gas Target	$15 \cdot 10^6$
Gas Limit	$30 \cdot 10^6$

Table 1: Here are some of the gas costs for certain integer and bit operations within the Ethereum Virtual Machine [18, 30]. "Transaction" is the base gas cost of each transaction. "sstore (clean)" refers to writing a nonzero word (32 bytes) to the blockchain; "sstore (dirty)" refers to overwriting a nonzero word with another nonzero word. "Gas Target" is the target amount of gas included within a single Ethereum block; "Gas Limit" is the maximum amount of gas an Ethereum block may contain.

3 Ethereum and the Ethereum Virtual Machine

Ethereum is a blockchain designed to have the ability to perform arbitrary computations [30]. The Ethereum Virtual Machine (EVM) executes all transactions on Ethereum. While Turing-complete, all operations cost a certain amount of *gas* and there is a limit to the amount of gas included within each block. Thus, the efficiency of all operations on Ethereum are measured by their gas usage.

The fundamental type is a `uint256` object: a 256-bit unsigned integer. Table 1 lists the costs of various integer and bit operations within the EVM. The fact that basic integer and bit operations are around 5 gas while every transaction has a base cost of $21 \cdot 10^3$ gas means that integer and bit operations are inexpensive. The fact that clean (dirty) writes costs $20 \cdot 10^3$ ($5 \cdot 10^3$) gas means that storing data on Ethereum is significantly more expensive; thus, it is beneficial to minimize storage. Another notable feature is that multiplication and division have the same cost; this fact is nontrivial, and we note the following from a paper on computing integer square roots: "The absence of division primitives is the main reason why we look for an approximation of the inverse square root rather than the square root itself" [14, Section 2].

4 Generic Algorithm Based on Newton's Method

One method for computing integer square roots is based on Newton's method; see [3, Algorithm 1.7.1], [5, Algorithm 9.2.11], and [2, Algorithm 1.13]. While not the only method for computing integer square roots, the efficiency of Newton's method causes us to focus on these algorithms; additional algorithms are discussed in Appendix D. These algorithms usually come with a proof of correctness. For completeness, we reproduce [3, Algorithm 1.7.1]

Algorithm 1 Integer Square Root algorithm found in [3, Algorithm 1.7.1]. We include the following from [3, Chapter 1.7, Remarks]: "When actually implementing this algorithm, the initialization step must be modified."

```
Require: $n$ is a positive integer
1: procedure IntegerSquareRoot($n$)
2:     $x \coloneqq n$
3:     $y \coloneqq \lfloor (x + n/x)/2 \rfloor$          ▷ All operations are integer operations
4:     while $x > y$ do
5:         $x \coloneqq y$
6:         $y \coloneqq \lfloor (x + n/x)/2 \rfloor$
7:     end while
8:     return $x$
9: end procedure
```

in Algorithm 1, and include a modified version of the proof [3, Algorithm 1.7.1, Proof] in Appendix E.4.1.

5 Integer Square Root Algorithms

5.1 Algorithm Design Principle

The algorithm design philosophy is straightforward: provably correct algorithms which are efficient within the Ethereum Virtual Machine. The primary focus is on minimizing the per-call gas cost, but we also want to ensure low deployment cost as well.

5.2 Specific Algorithms in Solidity

5.2.1 Online Algorithms

The specific Isqrt algorithms we found online were `UniswapV2` (Listing 1) [27], `PRB` (Listing 2) [1], `OpenZeppelin` (Listing 3) [20], `ABDK` (Listing 4) [4], and `OpenZeppelinV2` (Listing 5) [21]. Two algorithms (`PRB` and `ABDK`) appear to be exactly the same aside from different integer notation and "unchecked" portions. The `OpenZeppelin` algorithm is mathematically equivalent to `PRB` and `ABDK`. `OpenZeppelinV2` was developed with input from the author, although in the end his contribution was only the choice of initialization value which lead to only needing six Newton iterations; the author did not write any of the code or comments [6].

None of these algorithms reference a proof of correctness with the exception of `OpenZeppelinV2` (which was developed after the initial version of the present work). We note that `UniswapV2` is provably correct as it essentially implements [3, Algorithm 1.7.1]; compare Listing 1 and Algorithm 1. Unfortunately, `UniswapV2` does not follow the comment in [3, Chapter 1.7, Remarks] which states that "[w]hen actually implementing this algorithm, the initialization step must be modified". `OpenZeppelin` [20] says that we "need at most 7 iteration[s]" while `ABDK` [4] states that "[s]even iterations should be enough"; `PRB` [1] has similar language.

Listing 1: Method for computing Integer Square Roots from `UniswapV2` [27]

```
// 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;
    }
}
```

5.2.2 Provably Correct Algorithms

Due to requiring *provably correct* methods for computing integer square roots, additional algorithms were written. All proofs are relegated to Appendix E.

The first algorithms come in two groups of 3. They are `Unrolled1` (Listing 6), `Unrolled2` (Listing 7), `Unrolled3` (Listing 8), `While1` (Listing 9), `While2` (Listing 10), and `While3` (Listing 11). A version of `Unrolled3` was previously presented in [12]; the *initialization value* is the same but the *initialization method* has been changed. Here, `Unrolled` refers to the fact that a fixed number of Newton iterations are performed; `While` refers to the fact that the Newton iterations are iterated within a `while` loop. Labels `1`, `2`, and `3` refer to the different initialization values: the largest power-of-two less than or equal to integer square root value; the smallest power-of-two greater than the integer square root value; the arithmetic mean of the two previous values.

Additional algorithms include `BitLength` (Listing 12), `Linear` (Listing 13), `Hyper4` (Listing 14), `Lookup4` (Listing 15), and `Lookup8` (Listing 16). `BitLength` uses the full bitlength of the number to produce an initial approximation. `Linear` uses a linear approximation based on the high bits of the input value. `Hyper4` uses a hyperbolic approximation based on the high bits of the input value; we investigated such approximations based on the discussion in [16]. `Lookup4` and `Lookup8` use lookup tables to obtain 4 bits and 8 bits of accuracy for the initial approximation; the idea to use a lookup table for initialization came from [14, 15], although [14] uses a different method for computing square roots (it computes square roots by first computing the *inverse square root*).

5.3 General Algorithm Discussion

When using Newton iterations, a good initial approximation is critical to ensure a low overall cost. The following algorithms use the largest power-of-two less than or equal to the integer square root: `PRB` (Listing 2), `OpenZeppelin` (Listing 3), `ABDK` (Listing 4), `Unrolled1`

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` [20]

```
// 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 from `OpenZeppelinV2` [21]. This method is based on `Unrolled3` (Listing 8).

```
// SPDX-License-Identifier: MIT
function sqrt(uint256 a) internal pure returns (uint256) {
    unchecked {
        if (a <= 1) {
            return a;
        }

        uint256 aa = a;
        uint256 xn = 1;

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

        xn = (3 * xn) >> 1;

        xn = (xn + a / xn) >> 1;
        xn = (xn + a / xn) >> 1;
        xn = (xn + a / xn) >> 1;
        xn = (xn + a / xn) >> 1;
        xn = (xn + a / xn) >> 1;
        xn = (xn + a / xn) >> 1;

        return xn - SafeCast.toUint(xn > a / xn);
    }
}
```

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; }

        uint256 xAux = x;

        uint256 result = 1;

        if (xAux >= (1 << 128)) { xAux >>= 128; result = 1 << 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 <= x/result) {
            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; }

        uint256 xAux = x;

        uint256 result = 2;

        if (xAux >= (1 << 128)) { xAux >>= 128; result = 2 << 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 <= x/result) {
            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; }

        uint256 result = x;

        uint256 e = 1;

        if (x      >= (1 << 128)) { result >>= 128; e = 129; }
        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)) {                 e +=  2; }
        result = (3 << (e/2)) >> 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 <= x/result) {
            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; }

        uint256 xAux = x;

        uint256 result = 1;

        if (xAux >= (1 << 128)) { xAux >>= 128; result = 1 << 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;
        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; }

        uint256 xAux = x;

        uint256 result = 2;

        if (xAux >= (1 << 128)) { xAux >>= 128; result = 2 << 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; }

        uint256 xAux = x;

        uint256 result = 1;

        if (xAux >= (1 << 128)) { xAux >>= 128; result = 1 << 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 = (3 * result) >> 1;

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

Listing 12: Provably correct method for computing Integer Square Roots (`BitLength`)

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

        // Here, e represents the bit length;
        // its value is at most 256, so it could fit in a uint16.
        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 =  129; }
        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)) {
            result = (27 << (e/2)) >> 4;
        } else {
            result = (39 << (e/2)) >> 5;
        }

        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 <= x/result) {
            return result;
        }
        return result-1;
    }
}
```

Listing 13: Provably correct method for computing Integer Square Roots (`Linear`)

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

        // Here, e represents the bit length;
        // its value is at most 256, so it could fit in a uint16.
        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 =  129; }
        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)) {                 e +=   2; }
        // 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

        // need to initialize result
        result = m >> 254;
        result = (4 + result) << 125;

        result = (result + m / result) >> 1;
        result = (result + m / result) >> 1;
        result = (result + m / result) >> 1;
        result = (result + m / result) >> 1;
        result = (result + m / result) >> 1;
        result = (result + m / result) >> 1;
        result >>= e;

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

Listing 14: Provably correct method for computing Integer Square Roots (`Hyper4`)

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

        // Here, e represents the bit length;
        // its value is at most 256, so it could fit in a uint16.
        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 =  129; }
        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)) {                 e +=   2; }
        // 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

        // need to initialize result
        result = m >> 252;
        result = (512/(31 - result)) << 123;

        result = (result + m / result) >> 1;
        result = (result + m / result) >> 1;
        result = (result + m / result) >> 1;
        result = (result + m / result) >> 1;
        result = (result + m / result) >> 1;
        result >>= e;

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

Listing 15: Provably-correct method for computing Integer Square Roots (Lookup4)

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

        // Here, e represents the "approximate" 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 =  129; }
        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)) {                 e +=   2; }

        // overwrite e for scaling parameter
        e = (256 - e) >> 1;
        // m now satisfies 2**254 <= m < 2**256
        uint256 m = x << (2 * e);

        // need to initialize result
        result = uint256(uint8(lookup_table_4[(m >> 252)])) << 123;

        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 >>= e;

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

bytes constant lookup_table_4 =
"\x00\x00\x00\x00\x11\x13\x15\x16\x17\x19\x1a\x1b\x1c\x1d\x1e\x1f";
```

Listing 16: Provably-correct method for computing Integer Square Roots (`Lookup8`)

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

        // Here, e represents the "approximate" 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 =  129; }
        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)) {                 e +=   2; }

        // overwrite e for scaling parameter
        e = (256 - e) >> 1;
        // m now satisfies 2**254 <= m < 2**256
        uint256 m = x << (2 * e);

        // need to initialize result
        result = 2**127 +
                (uint256(uint8(lookup_table_8[(m>>248)])) << 119);

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

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

bytes constant lookup_table_8 = "...";
```

(Listing 6), and `While1` (Listing 9). Another good initialization value is shared by `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]). `Unrolled3` (Listing 8), `While3` (Listing 11), and `OpenZeppelinV2` (Listing 5) use the new initialization from [12]. `BitLength` (Listing 12), `Linear` (Listing 13), `Hyper4` (Listing 14), `Lookup4` (Listing 15), and `Lookup8` (Listing 16) use initializations specific to each method. `UniswapV2` (Listing 1) has the worst initialization value (the input value itself).

6 Comparison of Gas Costs

Because all operations are performed within the EVM, 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 [12] for three reasons: first, some of the algorithms are different (and we chose the most recent version of those found online); second, we ensure that all input values are unique (certain values may have been double counted); third, different input values were chosen.

After analyzing the per-call gas cost distribution, we discuss the associated costs of deploying the algorithm to the Ethereum blockchain.

6.1 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 $2^k - 1$, 2^k, and $2^k + 1$ for positive values of k;
- $v - 1$, v, and $v + 1$ for $v = (2^{128} - 1)^2$ (these were chosen because of an edge case in an algorithm);
- random values chosen according to the loguniform distribution [24] on $[1, 2^{256}]$ when the random seed is initialized to 0 [19].

The number of random samples were increased until there were 2048 unique values. While it is clear that the particular distribution affects the results, we believe choosing the input values in this way reduces the risk of bias. There were a total of 768 deterministic values chosen and a total of 1303 random samples determined 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 [7, 22]. There was no instance in which an algorithm ever produced an incorrect result; this is, of course, not a proof that `PRB`, `OpenZeppelin`, and `ABDK` are correct, but rather that there are no known values where these algorithms fail.

	`UniswapV2`	`PRB`	`OpenZeppelin`	`ABDK`	`OpenZeppelinV2`
Max	33 931	874	1015	877	823
Mean	17 591	791	944	799	749
Median	17 497	794	943	799	751
Std	9482	34	30	33	35
Abs Cost	226 284	277 197	280 341	281 321	271 153
Rel Cost	44 133	95 046	98 190	99 170	89 002

Table 2: Here are statistics related to the gas cost data; we also include the absolute deployment gas cost and relative deployment gas cost (absolute gas cost less 182151 gas, the deployment cost of an empty smart contract). The `UniswapV2` and `OpenZeppelinV2` algorithms are provably correct. These results are for the tests in Section 6.

	`Unrolled1`	`Unrolled2`	`Unrolled3`	`While1`	`While2`	`While3`
Max	837	837	790	1200	1152	1130
Mean	762	762	730	815	872	831
Median	765	765	730	858	907	854
Std	33	33	28	176	155	135
Abs Cost	283 589	283 601	276 117	252 678	247 488	254 192
Rel Cost	101 438	101 450	93 966	70 527	65 337	72 041

Table 3: Here are statistics related to the gas cost data; we also include the absolute deployment gas cost and relative deployment gas cost (absolute gas cost less 182151 gas, the deployment cost of an empty smart contract). All of these algorithms are provably correct. These results are for the tests in Section 6.

	`BitLength`	`Linear`	`Hyper4`	`Lookup4`	`Lookup8`
Max	833	796	826	903	906
Mean	762	739	769	846	849
Median	762	739	769	846	849
Std	30	28	29	30	30
Abs Cost	282 605	276 105	280 407	301 301	349 214
Rel Cost	100 454	93 954	98 256	119 150	167 063

Table 4: Here are statistics related to the gas cost data; we also include the absolute deployment gas cost and relative deployment gas cost (absolute gas cost less 182151 gas, the deployment cost of an empty smart contract). All of these algorithms are provably correct. These results are for the tests in Section 6.

Total	2048
`UniswapV2`	2
`OpenZeppelinV2`	2
`Unrolled1`	2
`Unrolled2`	2
`Unrolled3`	1188
`While1`	383
`While2`	186
`While3`	295
`BitLength`	2
`Linear`	2
`Hyper4`	2
`Lookup4`	2
`Lookup8`	2

Table 5: Here are the number of times each method had minimal gas cost; methods not included were never minimal. These results are for the tests in Section 6.

6.2 Summary Statistics

A listing of the summary statistics describing the runs may be found in Tables 2, 3, and 4. From here, we see that the algorithm with the smallest mean, median, and maximum gas cost is `Unrolled3`; the standard deviation of `Unrolled3` is minimal as well. `Linear` finishes in a close second place. The fact that `OpenZeppelinV2` comes in third is not surprising given that it is based on `Unrolled3`.

6.3 Detailed Analysis

We now take a closer look at how the gas cost varies with the argument.

For each value we tested, we determined the minimal gas cost and then counted the number of times each algorithm was minimal; the results may be found in Table 5. Additionally, we show a histogram where certain algorithms were minimal in Figure 1. There does not appear to be a particular pattern in the value distribution. The main trend is that `Unrolled3` 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 those values require fewer Newton iterations.

Overall, `Unrolled3` has the lowest cost for a majority of points, so we declare this to be the algorithm of choice. A more extensive analysis (with similar results) may be found in Appendix A.

6.4 Deployment Gas Cost

While the primary design criterion is provably-correct algorithm with efficient operation, it is important to recognize the initial (one-time) cost of *deploying* the algorithm to the Ethereum blockchain.

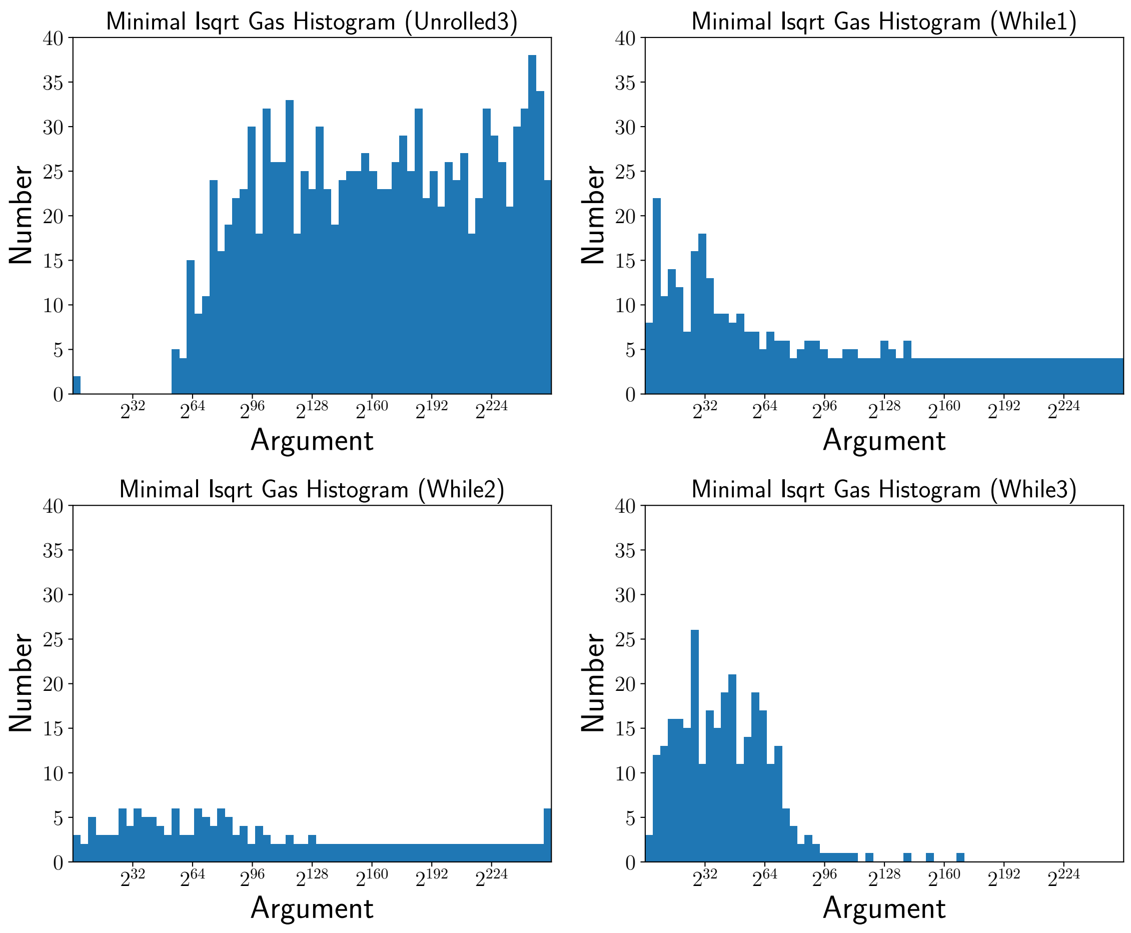

Figure 1: Here we plot a histogram showing where each algorithm is minimal and provides more detail to the results in Table 5. Each bin counts the total number of instances where the algorithm's gas cost is minimal. These results are for the tests in Section 6.

The deployment cost is related to the Ethereum bytecode; this is the compiled version of the algorithm which runs on the Ethereum Virtual Machine. The cost depends on both the size of the bytecode along with the specific instructions (different instructions have different gas costs).

In Tables 2, 3, and 4, we include the absolute and relative deployment costs: the absolute deployment cost is the gas cost given by the Hardhat Solidity compiler (using Solidity version `0.8.20` and the optimizer enabled for `1000000` runs) [8]; the relative deployment cost is obtained by subtracting the deployment cost of an empty smart contract (182151 gas). We highlight algorithms with relative deployment cost below 95000 gas; this value was chosen somewhat arbitrarily but makes it easier to distinguish algorithms by deployment cost.

Because the cost of deployment is a one-time cost, the data here is included to provide a larger picture. The expectation is that minimizing the per-operation cost is more important than a minimal deployment cost. Thus, even though `Unrolled3` has a slightly larger deployment cost than `OpenZeppelinV2`, it is still thought that `Unrolled3` will result in an overall lower cost.

7 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). A proof of correctness may be found Appendix E.

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 (cit. on pp. 3, 5, 58).

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

[3] Henri Cohen. *A Course in Computational Algebraic Number Theory.* Graduate Texts in Mathematics 138. Springer, 1993 (cit. on pp. 2, 3, 20, 59).

[4] ABDK Consulting. *Integer Square Root Algorithm in Solidity.* Oct. 2020. URL: https://github.com/abdk-consulting/abdk-libraries-solidity/blob/9e69beb255e2f87d67b44047d ABDKMath64x64.sol#L727 (cit. on pp. 3, 7, 58).

[5] Richard Crandall and Carl Pomerance. *Prime Numbers: A Computational Perspective.* 2nd ed. Springer, 2005 (cit. on pp. 2, 20).

[6] Hadrien Croubois, Ernesto García, and Christopher H. Gorman. *Improved integer square root (#4403).* Feb. 16, 2024. URL: https://github.com/OpenZeppelin/openzeppelin-contracts/commit/140d66fad8b3d8bf1fe793fc73d5e0a9f55f16e3 (visited on 12/11/2024) (cit. on p. 3).

[7] Mark Dickinson. *Python's integer square root algorithm*. Jan. 2022. URL: https://github.com/mdickinson/snippets/blob/master/papers/isqrt/isqrt.pdf (cit. on pp. 20, 37, 38).

[8] Nomic Foundation. *Hardhat*. Version 2.16.1. Dec. 10, 2024. URL: https://hardhat.org/ (cit. on p. 24).

[9] Gaston H. Gonnet, Lawrence D. Rogers, and J. Alan George. "An Algorithmic and Complexity Analysis of Interpolation Search". In: *Acta Informatica* 13 (1980), pp. 39–52. URL: https://www.researchgate.net/profile/Gaston-Gonnet/publication/220197937_An_Algorithmic_and_Complexity_Analysis_of_Interpolation_Search/links/00b7d5327f9304caef000000/An-Algorithmic-and-Complexity-Analysis-of-Interpolation-Search.pdf (cit. on p. 42).

[10] Christopher H. Gorman. *Analysis of Integer Square Root Algorithms in Solidity*. 2023. URL: https://github.com/chgorman/isqrt-gas (cit. on p. 1).

[11] Chun-Hua Guo. "On Newton's method and Halley's method for the principal pth root of a matrix". In: *Linear Algebra and its Applications* 432.8 (2010), pp. 1905–1922. DOI: https://doi.org/10.1016/j.laa.2009.02.030 (cit. on p. 31).

[12] Christopher H. Gorman. *Efficient Integer Square Roots in Solidity*. Nov. 2022. URL: https://github.com/alicenet/.github/blob/main/docs/efficient_isqrt.pdf (cit. on pp. 1, 4, 20, 37, 38, 45).

[13] Donald Ervin Knuth. *The Art of Computer Programming*. 2nd ed. Vol. 3. Addison-Wesley, 1998 (cit. on p. 40).

[14] 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. DOI: https://doi.org/10.1109/ARITH.2019.00041. URL: https://inria.hal.science/hal-02092970/file/main.pdf (cit. on pp. 2, 4, 37).

[15] *Methods of computing square roots: Binary estimates*. URL: https://en.wikipedia.org/wiki/Methods_of_computing_square_roots#Binary_estimates (visited on 08/14/2023) (cit. on p. 4).

[16] *Methods of computing square roots: Hyperbolic estimates*. URL: https://en.wikipedia.org/wiki/Methods_of_computing_square_roots#Hyperbolic_estimates (visited on 08/14/2023) (cit. on p. 4).

[17] Adnan Saher Mohammed, Şahin Emrah Amrahov, and Fatih V Çelebi. "Interpolated binary search: An efficient hybrid search algorithm on ordered datasets". In: *Engineering Science and Technology, an International Journal* 24.5 (2021), pp. 1072–1079. URL: https://www.sciencedirect.com/science/article/pii/S221509862100046X (cit. on p. 42).

[18] Nero_eth. *On Block Sizes, Gas Limits and Scalability*. URL: https://ethresear.ch/t/on-block-sizes-gas-limits-and-scalability/18444 (visited on 11/11/2024) (cit. on p. 2).

[19] Numpy. *numpy.random.seed*. URL: https://numpy.org/doc/stable/reference/random/generated/numpy.random.seed.html (visited on 07/16/2023) (cit. on p. 20).

[20] OpenZeppelin. *Integer Square Root Algorithm in Solidity*. July 2023. URL: https://github.com/OpenZeppelin/openzeppelin-contracts/blob/6bf68a41d19f3d6e364d8c207cb7d1a contracts/utils/math/Math.sol#L220 (cit. on pp. 3, 6, 58).

[21] OpenZeppelin. *Integer Square Root Algorithm in Solidity*. Feb. 2024. URL: https://github.com/OpenZeppelin/openzeppelin-contracts/blob/140d66fad8b3d8bf1fe793fc73d5e0a contracts/utils/math/Math.sol#L391 (cit. on pp. 3, 8).

[22] Python. *Mathematical Functions*. URL: https://docs.python.org/3/library/math.html#math.isqrt (visited on 07/16/2023) (cit. on p. 20).

[23] Walter Rudin. *Principles of Mathematical Analysis*. 3rd ed. McGraw-Hill New York, 1976 (cit. on p. 30).

[24] Scipy. *scipy.stats.loguniform*. URL: https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.loguniform.html (visited on 07/16/2023) (cit. on pp. 20, 27).

[25] J. Michael Steele. *The Cauchy-Schwarz Master Class: An Introduction to the Art of Mathematical Inequalities*. Cambridge University Press, 2004 (cit. on p. 45).

[26] Solidity Development Team. *Solidity Language*. URL: https://soliditylang.org/ (visited on 12/15/2024) (cit. on p. 1).

[27] Uniswap. *Integer Square Root Algorithm in Solidity*. Jan. 2020. URL: https://github.com/Uniswap/v2-core/blob/585ee2ef824db671b71e351a91860a003c05431d/contracts/libraries/Math.sol#L11 (cit. on pp. 3, 4).

[28] Henry S. Warren Jr. *Hacker's Delight*. 2nd ed. Addison-Wesley, 2013 (cit. on pp. 37, 40, 42, 44).

[29] Henry S. Warren Jr. *Hacker's Delight Integer Square Root Code*. 2013. URL: https://web.archive.org/web/20190402153025/http://www.hackersdelight.org/hdcodetxt/isqrt.c.txt (visited on 03/16/2024) (cit. on pp. 40, 42).

[30] Gavin Wood. *Ethereum: A Secure Decentralised Generalised Transaction Ledger*. Yellowpaper. Ethereum Project, 2022. URL: https://github.com/ethereum/yellowpaper (cit. on pp. 1, 2).

[31] Andrew C. Yao and F. Francis Yao. “The complexity of searching an ordered random table”. In: *2013 IEEE 54th Annual Symposium on Foundations of Computer Science*. Los Alamitos, CA, USA: IEEE Computer Society, Oct. 1976, pp. 173–177. DOI: 10.1109/SFCS.1976.32. URL: https://doi.ieeecomputersociety.org/10.1109/SFCS.1976.32 (cit. on p. 40).

A Extended Analysis

While the results in Section 6 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),

Total	33 154
`Unrolled3`	15 766
`While1`	15 636
`While2`	133
`While3`	1625

Table 6: Here are the number of times each method had minimal gas cost; these are results for the additional deterministic values based around powers-of-two. These results are for the tests in Appendix A.1.

and `While3` (Listing 11), as these performed the best: they had a significant number of values which were minimal.

The deterministic and random values will be separated. All values tested in Section 6 are included in these tests.

A.1 Deterministic Tests

We test additional deterministic values. First, we include all deterministic values from Section 6. 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 6 and Figure 2.

The minimal values have a different trend than before: `Unrolled3` and `While1` both perform well. The fact that `While1` performs the well in Table 6 could be explained by noting that the initialization value in this test frequently starts *very close* to the correct value; see the discussion on gas costs of `while` loops in Appendix C.2.2. It is interesting that `Unrolled3` performed as well as it did in this situation which favors `While1`.

These results show 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.2 Loguniform Tests

The tests in Section 6 ran 1280 values which came from a loguniform distribution [24]. 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 7 and Figure 3. In this case, `Unrolled3` performs well.

There is no significant difference between the results in Tables 5 and 7 or Figures 1 and 3. `Unrolled3` has a majority of the minimal cost for random values.

A.3 Conclusion

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

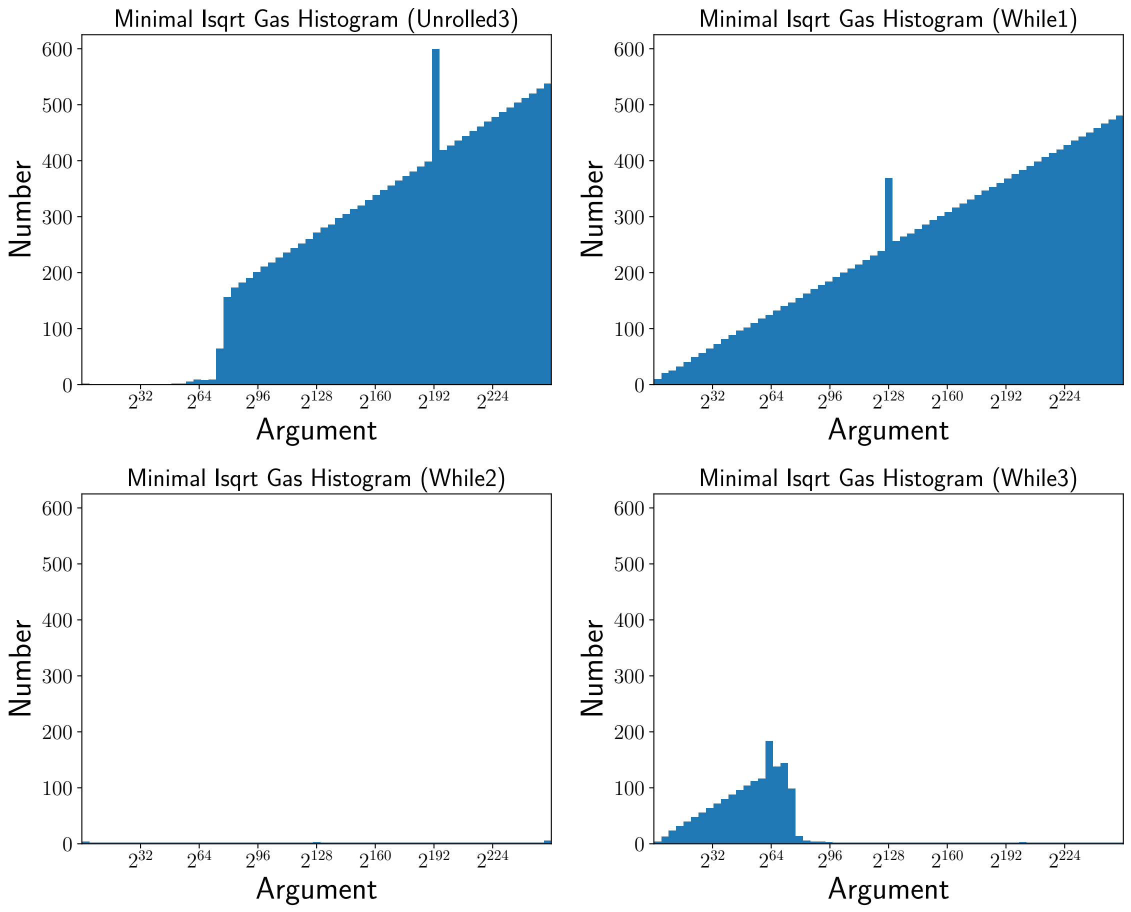

Figure 2: Here we plot a histogram showing where each algorithm is minimal when using a larger sample of deterministic values. Each bin counts the total number of instances where the algorithm's gas cost is minimal. These results are for the tests in Appendix A.1.

Total	16 384
`Unrolled3`	11 864
`While1`	1535
`While2`	645
`While3`	2343

Table 7: Here are the number of times each method had minimal gas cost; these are results for the additional loguniform random values. These results are for the tests in Appendix A.2.

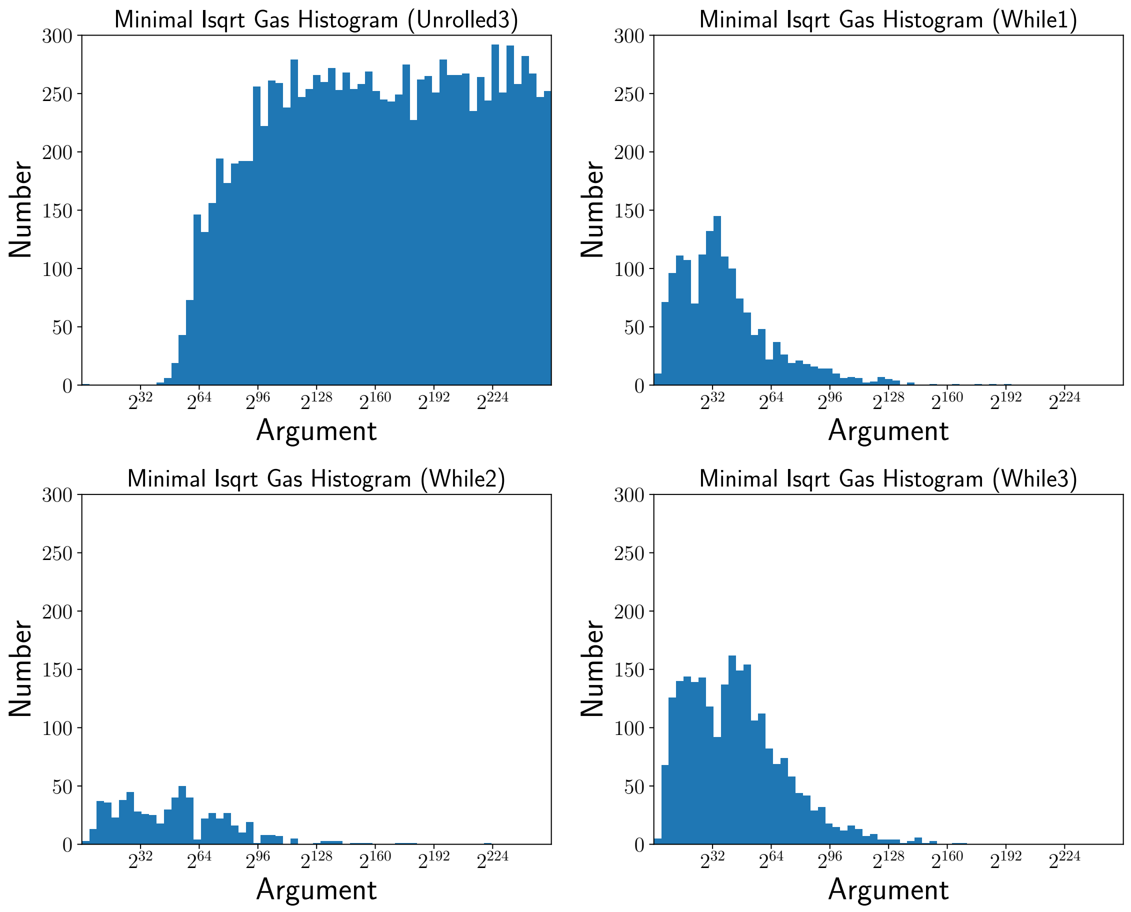

Figure 3: Here we plot a histogram showing where each algorithm is minimal when using a larger sample of loguniform random values. Each bin counts the total number of instances where the algorithm's gas cost is minimal. These results are for the tests in Appendix A.2.

B Asymptotic Analysis

We investigate the asymptotic error estimates of computing square roots. We focus on determining the number of iterations required based on Newton's method and its generalizations.

B.1 Error Estimates Based on Newton's Method

From [23, Chapter 3, Exercise 16], we recall that Newton's method for computing square roots has the form

$$x_{k+1} \coloneqq \frac{1}{2}\left(x_k + \frac{\alpha}{x_k}\right) \tag{B.1}$$

and that if

$$\begin{aligned} f_\alpha(t) &\coloneqq \frac{1}{2}\left(t + \frac{\alpha}{t}\right) \\ \varepsilon &\coloneqq t - \sqrt{\alpha} \end{aligned} \tag{B.2}$$

then we have

$$f_\alpha(t) - \sqrt{\alpha} = \frac{\varepsilon^2}{2t}. \tag{B.3}$$

We are particularly interested in asymptotic error bounds in order to determine the maximum required number of Newton iterations. To do this, we assume

$$2^{e-1} \leq \sqrt{\alpha} < 2^e \tag{B.4}$$

and

$$|\varepsilon_0| \leq 2^{e-b_0}. \tag{B.5}$$

This initial error bound allows us to compute

$$|\varepsilon_1| \leq 2^{e-2b_0}. \tag{B.6}$$

Iterating the error bound, we see that

$$|\varepsilon_k| \leq 2^{e-2^k b_0}. \tag{B.7}$$

The error decay rate in Eq. (B.7) is called *quadratic convergence*: the number of correct digits approximately *doubles* every iteration.

We want to find k such that

$$|\varepsilon_k| \leq 1, \tag{B.8}$$

and this is ensured when

$$2^{e-2^k b_0} \leq 1. \tag{B.9}$$

For this value of k, we have

$$1 < 2^{e-2^{k-1}b_0}. \tag{B.10}$$

Rearranging, we find

$$k < \lg e - \lg b_0 + 1 \tag{B.11}$$

and can be approximated as

$$k \simeq \lg\lg\sqrt{\alpha}. \tag{B.12}$$

B.2 Error Estimates Based on Halley's Method

The quadratic convergence (doubling of correct digits approximately every iteration) makes Newton's method very popular. This lead to the development of higher-order methods. In particular, Halley's method has *cubic convergence*: the number of correct digits approximately *triples* every iteration. This faster convergence comes at the cost of more expensive iterations.

Halley's method for computing square roots [11, Section 2] has the form

$$x_{k+1} = \frac{x_k^3 + 3\alpha x_k}{3x_k^2 + \alpha} \tag{B.13}$$

and that if

$$\begin{aligned} h_\alpha(t) &:= \frac{t^3 + 3\alpha t}{3t^2 + \alpha} \\ \varepsilon &:= t - \sqrt{\alpha}, \end{aligned} \tag{B.14}$$

then we have

$$h_\alpha(t) - \sqrt{\alpha} = \frac{\varepsilon^3}{3t^2 + \alpha}. \tag{B.15}$$

Follow the same procedure from Appendix B.1, we find k such that

$$0 \leq \varepsilon_k \leq 2^{e-3^k b_0} \leq 1. \tag{B.16}$$

This leads to

$$1 < 2^{e-3^{k-1}b_0} \tag{B.17}$$

and reduces to

$$k < \frac{1}{\lg 3}\lg e - \frac{1}{\lg 3}\lg b_0 + 1. \tag{B.18}$$

After noting $(\lg 3)^{-1} \simeq 0.63$, we see

$$k \simeq 0.63 \lg \lg \sqrt{\alpha}\,. \tag{B.19}$$

This leads to an asymptotic reduction in the required number of iteration steps at a greater per-iteration cost.

While Halley's method is better asymptotically, we see in Appendix D.4 that this method is not competitive against Newton's method for the values of interest (`uint256`).

B.3 Higher-order Iterative Methods

Higher-order generalizations are possible, but we do not pursue them here. The fact that Halley's method is not competitive with Newton's method in the region of interest suggests that better *practical* algorithms will not come from faster *asymptotic* methods. Furthermore, we note that deriving Newton's method for square roots involves computing the first derivative of $f(t) = t^2 - \alpha$ while Halley's method requires computing the second derivative of $f(t)$. Given that $f^{(\ell)}(t) = 0$ for $\ell \geq 3$, the author expects higher order methods will not converge faster than Halley.

C Gas Costs

We look at the specific gas costs of some operations; these will be useful in later discussions when making comparisons between different methods.

C.1 Initializations

We look at initialization costs associated with the algorithms; see Table 8 for the results.

Some of the initializations are self-explanatory. We note that `Newton1` is used by `Unrolled1` and `While1`; `Newton2` is used by `Unrolled2` and `While2`; `Newton3` is used by `While3`; and `Newton3-Var` is used by `Unrolled3`. The test used the value $2^{256} - 1$ to compute the gas cost. We ignore return logic but include gas costs associated with getting the input to and from the proper range; this is used in `Linear`, to give one example.

`Newton` and `Newton-Var` have similar gas values; a notable difference is seen in `Newton3` and `Newton3-Var`, and this is why `Unrolled3` uses the `Newton3-Var` initialization. `Linear` has the lowest initialization out of the non-`Newtons`; this (coupled with only needing 5 Newton iterations) shows why this method had such good results in Table 4. This also suggests that more complex initializations will not lead to overall improvements, as `Hyper4` gives the same accuracy and is only slightly more complicated yet costs significantly more gas. It is interesting to see that `Lookup4` and `Lookup8` have such expensive initializations; this suggests that no lookup-table-based method will be competitive.

C.2 Iterations

C.2.1 Newton Iteration

We look at the data in Table 9, which lists the gas costs associated with varying the number of Newton iterations; the `Newton3` initialization is used, although there is no return logic.

Method	Gas
`Newton1`	424
`Newton2`	417
`Newton3`	438
`Newton1-Var`	420
`Newton2-Var`	424
`Newton3-Var`	420
`BitLength`	468
`Linear`	454
`Hyper4`	500
`Lookup4`	577
`Lookup8`	628

Table 8: Here we give the gas cost for initialization methods; note that we ignored the initial check and did not include return logic. `Newton1` is used by `Unrolled1` and `While1`; `Newton2` is used by `Unrolled2` and `While2`; `Newton3` is used by `While3`; and `Newton3-Var` is used by `Unrolled3`. The value $2^{256} - 1$ was used. This clearly shows that some of the more complex initialization methods cost more gas.

Newton Iterations	Gas
0	336
1	391
2	439
3	487
4	535
5	583
6	631

Table 9: Here we give the gas cost for computing each Newton iteration. The difference between k and $k + 1$ iterations is 48 gas (notwithstanding the difference between 0 and 1, which is 55). The value $3 \cdot 2^{160}$ was used.

Value	While Iterations	Gas
$2^{256} - 2^{128}$	1	612
$2^{256} - 2^{224}$	2	702
$2^{256} - 2^{240}$	3	792
$2^{256} - 2^{248}$	4	882
$2^{256} - 2^{252}$	5	972
$2^{256} - 2^{254}$	6	1062

Table 10: Here we give the gas cost for computing each `While2`, which allows us to determine the cost per `while` loop. The difference between k and $k + 1$ iterations is 90 gas.

Halley Iterations	Gas
0	336
1	474
2	609
3	744
4	879

Table 11: Here we give the gas cost for computing each Halley iteration. As we can see, the difference between k and $k+1$ iterations is 135 gas (notwithstanding the difference between 0 and 1, which is 138). The value $3 \cdot 2^{160}$ was used.

Value	Binary Iterations	Total Gas	Total Diff	Init Gas	Init Diff	Comp Diff
2^3	2	595		293		
2^5	3	729	134	311	18	116
2^7	4	851	122	317	6	116
2^9	5	960	109	311	-6	115
2^{13}	6	1100	140	335	24	116
2^{15}	7	1221	121	341	6	115

Table 12: Here we give the gas cost for computing each iteration within Binary Search. Computing the difference in iterations for binary search is more involved because we must consider the difference in initialization cost. **Total Diff** gives the difference between the total gas between $k-1$ and k binary iterations; **Init Diff** gives the difference in initialization cost; **Comp Diff** gives the computed difference: **Comp Diff** = **Total Diff** – **Init Diff**. These results give an approximate iteration cost of 115 gas.

Value	Interp Iterations	Gas
$2^{256} - 2^{128}$	1	871
$2^{256} - 2^{180}$	2	1171
$2^{256} - 2^{196}$	3	1471
$2^{256} - 2^{224}$	4	1771
$2^{256} - 2^{230}$	5	2071
$2^{256} - 2^{236}$	6	2371

Table 13: Here we give the gas cost for computing each iteration within Interpolation Search, which allows us to determine the cost per `while` loop. The difference between k and $k+1$ iterations is 300 gas.

Each Newton iteration costs 48 gas, although the difference between 0 and 1 Newton iteration is 55 gas. These tests used the value $3 \cdot 2^{160}$ (to enable a direct comparison with the Halley iteration results in Appendix C.2.3).

C.2.2 While Iteration

It is not as straightforward to determine the cost per `while` loop within `While1`, `While2`, and `While3`, as the total number of iterations must be known (by running the analogous computation). To do so, we choose values of the same bitlength and determined the number of required iterations manually. The results may be found in Table 10; as we can see, each `while` loop costs 90 gas, which makes each iteration almost twice as expensive as the unrolled version.

From this we see that while it is clear that *certain* integer square root values may be computed more efficiently using `while` loops, in general this results in a more expensive operation.

C.2.3 Halley Iteration

We look at the data in Table 11, which lists the gas costs associated with varying the number of Halley iterations; the `Newton3` initialization is used, although there is no return logic. Each Halley iteration costs 135 gas, although the difference between 0 and 1 Halley iteration is 138 gas. These tests used the value $3 \cdot 2^{160}$ (so there is no overflow during the computation).

Thus, even though fewer Halley iterations are required to get to the same level of accuracy, each Halley iteration is almost three times more expensive than a Newton iteration.

C.2.4 Binary Search Iteration

We look at the data in Table 12, which lists the gas costs associated with varying the number of iterations within Binary search. The computation is more complicated because getting a different number of iterations requires numbers with different bit lengths, so a valid comparison should also compute (or estimate) the initialization cost (or rather, the initialization cost *difference*). In looking at the data, we estimate that each binary iteration costs 115 gas. This is more than twice as expensive as an unrolled Newton iteration and still more expensive than a Newton iteration within a `while` loop.

C.2.5 Interpolation Search Iteration

We look at the data in Table 13, which lists the gas costs associated with varying the number of iterations within Interpolation search. Each iteration costs 300 gas, so this method is not competitive.

D Additional Algorithms and Further Optimizations

We now investigate methods for potential improvements. We start by looking at improvements in initialization before turning to general methods. We include results from standard tests from Section 6 in Table 14, but none of the additional algorithms are competitive.

	`Python`	Binary Searh	Interpolation Searh	Hardware
Max	957	15 342	22 771	15 640
Mean	881	7846	6655	13 031
Median	882	7759	5636	12 994
Std	43	4283	5274	653

Table 14: Here are statistics related to the gas cost data. All of these algorithms are provably correct, but only `Python` is close to being competitive. These results are for the tests in Section 6.

D.1 Additional Initializations

Ceteris paribus, a more accurate initialization will require fewer Newton iterations than a less accurate initialization, so we begin by discussing initialization improvements.

D.1.1 Polynomial Approximations

The results of `Linear` in Table 4 shows that a linear (affine) approximation provides a good initial approximation by providing essentially 4 correct bits. It is possible to extend the approximation to higher degree polynomials; however, the results in Appendix C.1 suggest the gas required to get a more accurate initialization will cost more than the potential savings in reduced Newton iterations (48 gas per iteration as noted in Appendix C.2.1).

Specifically, `Unrolled3`, with an initial accuracy of 2 bits, requires 6 Newton iterations; the results in Tables 3 and 4 show that `Linear`, with an initial accuracy of 4 bits, requires 5 Newton iterations but ends up being about 10 gas more expensive. In Table 8, we see the initialization of `Hyper4` costs 46 more gas than `Linear`. To reduce the number of Newton iterations to 4 would require than the initialization would be able to use more gas than `Linear` but less than `Hyper4`; the author is not convinced this is possible.

D.1.2 Rational Approximations

The results `Hyper4` in Table 4 shows that hyperbolas provide a good initial approximation; however, `Hyper4` is not more efficient than `Unrolled3` or `Linear`. The discussion in Appendix D.1.1 is worth reiterating: Newton iterations are cheap and good approximations are expensive. It seems unlikely that a rational function which gives 8 bits of accuracy will result in a lower overall gas cost.

D.1.3 Lookup Tables

The results in Table 4 show `Lookup4` and `Lookup8` produce decent results; however, the gas costs are essentially the same even though `Lookup8` uses one fewer Newton iteration than `Lookup4`. This leads us to expect that higher precision lookup tables (which would be "`Lookup16`") will not result in lower gas costs (not withstanding the cost of *storing* all the approximations, which would require a *much larger* deployment cost). Furthermore, in Table 8 we see `Lookup4` and `Lookup8` have the most expensive initialization costs.

Method	Gas
Halley's Method	952
`Unrolled3`	705

Table 15: Here we compare running Halley's method from Listing 18 with `Unrolled3` (Listing 8); the value $3 \cdot 2^{160}$ was used. Even if it were possible to remove one Halley iteration, the total gas would only decrease from 952 to 817 (which is still more expensive).

D.2 Integer Square Roots via Inverse Square Roots

In [14], the method for computing integer square roots is actually based around computing *inverse square roots*. We note, though, that the setting and constraints are different as [14] focuses on the inverse square root algorithm in order to not have to perform division. As noted in Table 1, within the Ethereum Virtual Machine the cost of integer multiplication and division is the same. Thus, although it would be an interesting exercise to implement the algorithm in Solidity, the author does not expect it to produce an efficient algorithm (that is, better than `Unrolled3`).

D.3 Approximate Square Roots

One method based on *approximate square roots* may be found in [7]. The algorithm is designated as `Python` (Listing 17); this algorithm was previously presented in [12]. The algorithm is provably correct but is not as efficient as the other Newton-based methods; see the results in Table 14.

D.4 Halley's Method and Higher-Order Approximations

We have primarily focused on variations of Newton's method. The asymptotic analysis in Appendix B showed that fewer Halley iterations are required in comparison to Newton iterations. Unfortunately, in Appendix C.2 we saw that Halley iterations are almost three times more expensive than Newton iterations (notwithstanding implementation issues around overflow).

For completeness, we include a version of Halley's method; see Listing 18. This method is of limited value as the code will overflow when the input is too large, and we have not proven the return logic is valid. The initialization is the same as `While3`, and this is required in order to only have to perform 4 Halley iterations; using the initializations of `Unrolled1` or `Unrolled2` would require 5 Halley iterations. See Appendix E.2.9 for the proof that 4 iterations is sufficient.

We include results of running Halley's method on $3 \cdot 2^{160}$ in Table 15; this is why the author does not expect higher-order approximations to produce practical improvements.

D.5 Search Algorithms

We turn our attention to search algorithms. The idea of using search algorithms comes from [28, Chapter 11-1 Integer Square Root, Binary Search]. At the end of the analysis, we

Listing 17: Provably correct method for computing Integer Square Roots based on [7]; this algorithm was previously presented in [12].

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

        // 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 =  129; }
        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 <= x/result) {
            return result;
        }
        return result-1;
    }
}
```

Listing 18: General form for computing Integer Square Roots based on Halley's method; the return logic of this method has not been verified and the values will overflow for large inputs.

```
// SPDX-License-Identifier: 0BSD
// NOTE: return logic has not been verified.
function sqrt(uint256 x) internal pure returns (uint256) {
    unchecked {
        if (x <= 1) { return x; }

        uint256 xAux = x;

        uint256 result = 1;

        if (xAux >= (1 << 128)) { xAux >>= 128; result = 1 << 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 = (3*result) >> 1;

        result = (result * (result**2 + 3*x))/(3*result**2 + x);
        result = (result * (result**2 + 3*x))/(3*result**2 + x);
        result = (result * (result**2 + 3*x))/(3*result**2 + x);
        result = (result * (result**2 + 3*x))/(3*result**2 + x);

        // TODO: determine correct return logic
        if (result <= x/result) {
            return result;
        }
        return result-1;
    }
}
```

Listing 19: Binary search method for computing Integer Square Roots in `C` from [28]

```
int isqrt3(unsigned x) {
   unsigned a, b, m;            // Limits and midpoint.

   a = 1;
   b = (x >> 5) + 8;            // See text.
   if (b > 65535) b = 65535;
   do {
      m = (a + b) >> 1;
      if (m*m > x) b = m - 1;
      else         a = m + 1;
   } while (b >= a);
   return a - 1;
}
```

will see that these methods are not competitive; this can be seen in the results in Table 14.

The noncompetitive nature appears to come from the inherent complexity of search algorithms: in [13, Page 425, Exercise 6.2.1.22], Knuth mentions that on average $\lg\lg N$ comparisons are required for interpolation sort and that "*all* search algorithms ... must make asymptotically $\lg\lg N$ comparisons, on the average" (emphasis in original). The required $\lg\lg N$ comparisons would appear to translate into $\lg\lg N$ loop iterations, and each loop iteration will presumably be more complex than a Newton iteration within the EVM; in Appendix C.2.4, we see the per-cost of each loop within binary search is approximately 115 gas (more than twice the cost of Newton). This would appear to rule out all generic search algorithms (even if better search algorithms only required $\lg\lg N$ iterations).

The paper referenced for the required $\lg\lg N$ comparisons on average is presumably [31], which the author of the present work is unable to access.

D.5.1 Binary Search

The idea of using *binary search* to compute the integer square root comes from its inclusion in [28]; see Listing 19 for the `C` code from [29] and Listing 20 for the Solidity code.

Looking at Listing 20, we see that the initial bounds may be sufficient when working with the `unsigned` type (usually `uint32`). The `uint256` types on the EVM encourage a more accurate bounds in order to reduce the number of loop iterations; thus, we use powers-of-2 for the left and right limits.

By bounding the square root and then choosing the midpoint, the error is halved at every iteration of the loop. This leads to requiring $\Theta(\lg(x))$ number of iterations; in particular, at most $\frac{1}{2}\lg(x)$ iterations. Unfortunately, this means that 128 loop iterations are required for the largest integer square root values we will compute, and this leads to the large associated gas costs. Thus, this method is not competitive and no optimization will make it competitive.

Listing 20: Provably correct method based on binary search

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

        uint256 xAux = x;
        uint256 left = 1;
        if (xAux >= (1 << 128)) { xAux >>= 128; left = 1 << 64; }
        if (xAux >= (1 << 64))  { xAux >>=  64; left <<= 32;    }
        if (xAux >= (1 << 32))  { xAux >>=  32; left <<= 16;    }
        if (xAux >= (1 << 16))  { xAux >>=  16; left <<=  8;    }
        if (xAux >= (1 << 8))   { xAux >>=   8; left <<=  4;    }
        if (xAux >= (1 << 4))   { xAux >>=   4; left <<=  2;    }
        if (xAux >= (1 << 2))   {               left <<=  1;    }

        uint256 right = left << 1;
        uint256 midpoint;

        while (left <= right) {
            midpoint = (left + right) >> 1;
            if (midpoint > x/midpoint) {
                right = midpoint - 1;
            } else {
                left = midpoint + 1;
            }
        }

        return left -1;
    }
}
```

Algorithm 2 Generic Interpolation Search from [9, Page 41] (modified)

```
Require: array is a sorted array of values; $\varphi(\alpha, n)$ is the probe position to inspect.
1: procedure InterpolationSearch($\alpha$)
2:     low := 0
3:     high := $n + 1$
4:     while high − low > 1 do
5:         $j$ := $\varphi((\alpha - \text{array}[\text{low}])/(\text{array}[\text{high}] - \text{array}[\text{low}]), \text{high} - \text{low} - 1) + \text{low}$
6:         if $\alpha = \text{array}[j]$ then
7:             return $j$
8:         else if $\alpha < \text{array}[j]$ then
9:             high := $j$
10:        else
11:            low := $j$
12:        end if
13:    end while
14:    return fail
15: end procedure
```

D.5.2 Interpolation Search

The $\Theta(\lg(x))$ number of iterations required by binary search significantly increases the gas cost. *Interpolation search* [9] seeks to reduce the required number of iterations by using a linear approximation to update where to search next. It is possible to show the required iterations is $\Theta(\lg\lg(x))$, but this *assumes a uniform search space* (which is not the case here). The results in Table 13 show the per-iteration cost of 300 gas per loop makes not a viable choice. The Solidity code may be found in Listing 21.

The general idea of interpolation search makes sense: you can search an ordered array more quickly (faster than binary search) by estimating how the values in the array change with a linear approximation. The problem is that we are not searching a generic array but looking to compute a square root, and we can compute the exact derivative (so no approximation is needed).

An analysis of interpolation search may be found in [9], and Algorithm 2 is based on the algorithm from [9, Page 41].

While there are more complicated search algorithms possible (like the *interpolated binary search* algorithm [17] which seeks to combine the best of both binary and interpolation search), the results here suggest that no search method will be competitive against Newton's method.

D.6 Hardware Algorithm

We include an additional algorithm based on one from [28, Page 285]; see Listing 22 for the `C` code from [29] and Listing 23 for the Solidity code. This is a "hardware" algorithm, and the only operations used are comparisons, bitwise-or, bit shifts, and subtraction. The 128 loop iterations, though, make this not competitive; see the results in Table 14.

Listing 21: Method based on interpolation search

```
// SPDX-License-Identifier: 0BSD
function sqrt(uint256 x) internal pure returns (uint256) {
    unchecked {
        if (x == 0) { return 0; }
        if (x < 16) {
            if (x < 4) {
                return 1;
            } else if (x < 9) {
                return 2;
            } else {
                return 3;
            }
        }

        uint256 xAux = x;
        uint256 left = 1;
        if (xAux >= (1 << 128)) { xAux >>= 128; left = 1 << 64; }
        if (xAux >= (1 <<  64)) { xAux >>=  64; left <<= 32; }
        if (xAux >= (1 <<  32)) { xAux >>=  32; left <<= 16; }
        if (xAux >= (1 <<  16)) { xAux >>=  16; left <<=  8; }
        if (xAux >= (1 <<   8)) { xAux >>=   8; left <<=  4; }
        if (xAux >= (1 <<   4)) { xAux >>=   4; left <<=  2; }
        if (xAux >= (1 <<   2)) {               left <<=  1; }

        uint256 right = (left << 1);
        uint256 interp;

        while ((left <= x/left) && (x/right < right)) {
            interp = left + (x - left**2)/(right + left);
            if (x/interp < interp) {
                right = interp;
            } else {
                left = interp + 1;
            }
        }
        return left -1;
    }
}
```

Listing 22: “Hardware” method for computing Integer Square Roots in `C` from [28]

```c
int isqrt4(unsigned x) {
   unsigned m, y, b;

   m = 0x40000000;
   y = 0;
   while(m != 0) {                    // Do 16 times.
      b = y | m;
      y = y >> 1;
      if (x >= b) {
         x = x - b;
         y = y | m;
      }
      m = m >> 2;
   }
   return y;
}
```

Listing 23: Provably correct method based on hardware-efficient operations

```solidity
// SPDX-License-Identifier: 0BSD
function sqrt(uint256 x) internal pure returns (uint256) {
    unchecked {
        uint256 m = 2**254;
        uint256 y = 0;
        uint256 b = 0;

        while (m != 0) {
            b = y | m;
            y = y >> 1;
            if (x >= b) {
                x = x - b;
                y = y | m;
            }
            m = m >> 2;
        }

        return y;
    }
}
```

D.7 Potential Optimizations

Based on the cost of Newton iterations (48 gas per unrolled iteration and 90 gas within a `while` loop; see Appendix C.2), reducing the number of Newton iterations only reduces the overall gas cost so much.

Additional Approximations As mentioned in Appendix D.1, it is expected that polynomial estimates, rational approximations, and lookup tables will not lead to an overall reduction in gas. It is unclear where additional approximation methods would come from. The best algorithms presented here (`Unrolled3` (Listing 8) followed closely by `Linear` (Listing 13)) produce 2 or 4 bits of initial precision.

Additional Algorithms We implemented algorithms based on Newton's method and its generalizations, search algorithms, and approximate square roots. It is unclear to the author what other methods could lead to a competitive algorithm. From the algorithms investigated here, the method would likely need to have quadratic convergence.

Assembly One way to reduce gas is to write all algorithms in assembly. This is not completely separate from algorithm development, as it may be the case that optimized (assembly) versions of algorithms have lower gas costs which result in a different ranking. This requires a thorough knowledge of Solidity assembly and the result should be audited to ensure the algorithm is properly implemented. We leave this task to the interested reader.

E Proofs

Here we include proofs for the algorithms. This includes some of the material from [12, Appendix B]. A notable omission is the proof of `Python` (Listing 17); this is due to the fact that the emphasis here is on algorithms based on Newton iteration. A proof of correctness for `Python` may be found in [12, Appendix C]. Some of the work here is similar to that found in [12, Appendix A] but is included for this document to be self-contained.

E.1 Mathematical Review

E.1.1 Newton Iteration over the Real Numbers

Fix $\alpha > 0$. For any $t > 0$, we define

$$f_\alpha(t) \coloneqq \frac{1}{2}\left(t + \frac{\alpha}{t}\right). \tag{E.1}$$

We always have

$$f_\alpha(t) = \frac{1}{2}\left(t + \frac{\alpha}{t}\right) \geq \sqrt{\alpha}. \tag{E.2}$$

This follows from the inequality of the arithmetic and geometric means (AM-GM inequality); one reference is [25, Page 20]. Given $t > 0$, we can look at the sequence $\{t_k\}_{k=0}^{\infty}$ defined by

$$t_k = \begin{cases} t & k = 0 \\ f_\alpha(t_{k-1}) & k > 0 \end{cases}. \tag{E.3}$$

Each application of $f_\alpha(\cdot)$ is called a *Newton iteration*. This sequence converges to $\sqrt{\alpha}$: $\lim_{k\to\infty} t_k = \sqrt{\alpha}$.

In order to see how the error changes with each iteration, we set

$$\varepsilon := t - \sqrt{\alpha}. \tag{E.4}$$

From here, we find

$$f_\alpha(t) - \sqrt{\alpha} = \frac{\varepsilon^2}{2t}. \tag{E.5}$$

The "ε^2" in the error term shows what we mean when we say the algorithm has quadratic convergence.

We make another important observation: when $t \geq \sqrt{\alpha}$, we have

$$\begin{aligned} f_\alpha(t) - \sqrt{\alpha} &= \frac{\varepsilon^2}{2t} \\ &\leq \frac{\varepsilon^2}{2\sqrt{\alpha}}. \end{aligned} \tag{E.6}$$

This fact allows us to simplify the error bounds we compute. We note that all values of the sequence in Eq. (E.3) satisfy $t_k \geq \sqrt{\alpha}$ with the possible exception of t_0.

E.1.2 Newton Iteration over the Integers

The fact that we care about *integer operations* within the Ethereum Virtual Machine means we focus on a slightly different function. Fix $n \in \mathbb{N}$ and let $x \in \mathbb{N}$. We define

$$g_n(x) := \left\lfloor \frac{1}{2}\left(x + \frac{n}{x}\right) \right\rfloor. \tag{E.7}$$

This represents the integer version of f_α.

E.2 Error Bounds

Based on the above work, we will now compute the error bounds for a number of algorithms. The computations are similar, and we primarily focus on the initial error and the first Newton iteration, as these are most important. Only when care is required to prove the desired error bounds do we go into greater detail.

Throughout this section, we are assuming that $\sqrt{n}$ is an e-bit number:

$$2^{e-1} \leq \sqrt{n} < 2^e. \tag{E.8}$$

This allows us to state the initialization value; different initialization *algorithms* may arrive at the same initialization *value*.

Throughout, we will specify the initialization x_0 and then look at

$$x_k = g_n(x_{k-1}), \quad k > 0. \tag{E.9}$$

We are interested in the values

$$\varepsilon_k \coloneqq x_k - \sqrt{n}. \tag{E.10}$$

These values will allow us to determine when x_k has reached $\textsc{Isqrt}(n)$.

Technically, we care about these error values:

$$\hat{\varepsilon}_k \coloneqq x_k - \left\lfloor \sqrt{n} \right\rfloor. \tag{E.11}$$

This is because we are computing *integer square roots*, not square roots. At the same time, we note that when $\varepsilon_k \leq 1$,

$$x_k \in \{q, q+1\}, \quad q = \left\lfloor \sqrt{n} \right\rfloor. \tag{E.12}$$

This follows because $x_k \in \mathbb{N}$ and $|\sqrt{n} - \lfloor\sqrt{n}\rfloor| < 1$. Thus, we focus on determining when $\varepsilon_k \leq 1$. Any attempt to get a smaller error bound is pointless because it is less expensive to check if x_k satisfies $x_k^2 \leq n$.

In each error bound calculation, we will assume that $x_k > \sqrt{n}$; if this ever fails, we know that $x_k = \lfloor\sqrt{n}\rfloor$.

E.2.1 `Unrolled1`

The initialization value is chosen to be the largest power-of-two less than or equal to $\sqrt{n}$:

$$x_0 = 2^{e-1}. \tag{E.13}$$

We have the error bound

$$|\varepsilon_0| \leq 2^{e-1}. \tag{E.14}$$

We now use this to bound ε_1:

$$\begin{aligned} \varepsilon_1 &= \frac{\varepsilon_0^2}{2x_0} \\ &\leq 2^{e-2}. \end{aligned} \tag{E.15}$$

Continuing, we find

$$\begin{aligned} \varepsilon_2 &\leq 2^{e-4} \\ \varepsilon_3 &\leq 2^{e-8} \\ &\vdots \end{aligned} \tag{E.16}$$

E.2.2 `Unrolled2`

The initialization value is chosen to be the smallest power-of-two greater than to $\sqrt{n}$:

$$x_0 = 2^e. \tag{E.17}$$

Note that strict inequality: we have $x_0 > \sqrt{n}$. We have the error bound

$$\varepsilon_0 \leq 2^{e-1}. \tag{E.18}$$

We now use this to bound ε_1:

$$\begin{aligned} \varepsilon_1 &= \frac{\varepsilon_0^2}{2x_0} \\ &\leq 2^{e-3}. \end{aligned} \tag{E.19}$$

Continuing, we find

$$\begin{aligned} \varepsilon_2 &\leq 2^{e-6} \\ \varepsilon_3 &\leq 2^{e-12} \\ &\vdots \end{aligned} \tag{E.20}$$

This shows that `Unrolled2` converges slightly faster (has smaller error) than `Unrolled1`. This is due solely to the larger initial starting value.

E.2.3 `Unrolled3`

The initialization value is chosen to be the arithmetic mean of the initializations for `Unrolled1` and `Unrolled2`:

$$x_0 = 2^{e-1} + 2^{e-2}. \tag{E.21}$$

We have the error bound

$$|\varepsilon_0| \leq 2^{e-2}. \tag{E.22}$$

We now use this to bound ε_1:

$$\begin{aligned} \varepsilon_1 &= \frac{\varepsilon_0^2}{2x_0} \\ &\leq 2^{e-4.5}. \end{aligned} \tag{E.23}$$

Here we are using the fact that $\log_2(1/3) < -1.5$. Continuing, we find

$$\begin{aligned} \varepsilon_2 &\leq 2^{e-9} \\ \varepsilon_3 &\leq 2^{e-18} \\ &\vdots \end{aligned} \tag{E.24}$$

E.2.4 `BitLength`

We use the following approximation

$$x_0 = \begin{cases} \lfloor (27 \cdot 2^{e-1})/2^4 \rfloor & \textsc{BitLength}(n) = 0 \mod 2 \\ \lfloor (39 \cdot 2^{e-1})/2^5 \rfloor & \textsc{BitLength}(n) = 1 \mod 2 \end{cases}. \tag{E.25}$$

We assume n is an f-bit number:

$$2^{f-1} \le n < 2^f, \quad f = 2e - k, \quad k \in \{0, 1\}. \tag{E.26}$$

- Case 0: $\textsc{BitLength}(n)$ is even or $k = 0$.

  In this case, we have

$$2^{2e-1} \le n < 2^{2e}, \tag{E.27}$$

  whence it follows that

$$\sqrt{2} \cdot 2^{e-1} \le \sqrt{n} < 2^e, \tag{E.28}$$

  Using the fact $45/32 < \sqrt{2}$, we have

$$45 \cdot 2^{e-6} \le \sqrt{n} < 64 \cdot 2^{e-6}. \tag{E.29}$$

  Because we are choosing

$$x_0 \coloneqq 27 \cdot 2^{e-5}, \tag{E.30}$$

  we have

$$|\varepsilon_0| \le 5 \cdot 2^{e-5}. \tag{E.31}$$

  It follows that

$$\begin{aligned} \varepsilon_1 &= \frac{\varepsilon_0^2}{2x_0} \\ &\le \frac{25}{27} 2^{e-6} \\ &\le 2^{e-6}. \end{aligned} \tag{E.32}$$

- Case 1: $\textsc{BitLength}(n)$ is odd or $k = 1$.

  In this case, we have

$$2^{2e-2} \leq n < 2^{2e-1}, \tag{E.33}$$

whence it follows that

$$\cdot 2^{e-1} \leq \sqrt{n} < \sqrt{2} \cdot 2^{e-1}, \tag{E.34}$$

Using the fact $\sqrt{2} < 23/16$, we have

$$16 \cdot 2^{e-5} \leq \sqrt{n} < 23 \cdot 2^{e-5}. \tag{E.35}$$

Because we are choosing

$$x_0 \coloneqq 39 \cdot 2^{e-6}, \tag{E.36}$$

we have

$$|\varepsilon_0| \leq 7 \cdot 2^{e-6}. \tag{E.37}$$

It follows that

$$\begin{aligned} \varepsilon_1 &= \frac{\varepsilon_0^2}{2x_0} \\ &\leq \frac{49}{39} 2^{e-7} \\ &\leq 2^{e-6.65}, \end{aligned} \tag{E.38}$$

where we are using the bound $\log_2(49/39) < 0.35$.

In the worst case, we have

$$\begin{aligned} \varepsilon_2 &\leq 2^{e-12} \\ \varepsilon_3 &\leq 2^{e-24} \\ &\vdots \end{aligned} \tag{E.39}$$

We note that it is possible to get better error bounds, but they are not worth the effort.

E.2.5 `Linear`

We start by scaling n (relabeled as m) so that

$$2^{2e-2} \leq m < 2^{2e} \tag{E.40}$$

for $e = 128$. We then set

$$v := \left\lfloor \frac{m}{2^{2e-4}} \right\rfloor \tag{E.41}$$

so that

$$v \cdot 2^{2e-4} \leq m < (v+1) \cdot 2^{2e-4}, \quad v \in \{4, 5, \ldots, 15\}. \tag{E.42}$$

This allows us to make the reduction to

$$\sqrt{v} \cdot 2^{e-2} \leq \sqrt{m} < \sqrt{v+1} \cdot 2^{e-2}. \tag{E.43}$$

In order to determine the appropriate approximation, we first find bounds for $\sqrt{v}$ and $\sqrt{v+1}$. In particular, we computed values $\nu_{v,0}, \nu_{v,1} \in \mathbb{N}$ with

$$\begin{aligned} \frac{\nu_{v,0}}{16} &\leq \sqrt{v} \\ \frac{\nu_{v,1}}{16} &\geq \sqrt{v+1} \end{aligned} \tag{E.44}$$

Using these values, we let

$$\mu_v := 2 \cdot (14 + v). \tag{E.45}$$

The initial approximation is

$$x_0 = \mu_v \cdot 2^{e-6}. \tag{E.46}$$

We separate error calculations into two cases:

- Case 1: $v \in \{4, 5, 6, 7, 8\}$
  In this case, the initial error is

$$\varepsilon_0 \leq 2^{e-4} \tag{E.47}$$

  and we have the error bounds

$$\begin{aligned} \varepsilon_1 &\leq 2^{e-8} \\ \varepsilon_2 &\leq 2^{e-16} \\ &\vdots \end{aligned} \tag{E.48}$$

- Case 2: $v \in \{9, 10, 11, 12, 13, 14, 15\}$

  This case is more involved. We focus on the case $v = 9$. We have the bound

$$48 \cdot 2^{e-6} \leq \sqrt{m} < 51 \cdot 2^{e-6} \tag{E.49}$$

with

$$\mu_9 = 46 \cdot 2^{e-6}. \tag{E.50}$$

As we can see, $|\varepsilon_0| \leq 5 \cdot 2^{e-6}$. The error bound after the first Newton iteration is

$$\begin{aligned} \varepsilon_1 &= \frac{\varepsilon_0^2}{2x_0} \\ &\leq \frac{25}{23} 2^{e-8} \\ &\leq 2^{e-7.875}. \end{aligned} \tag{E.51}$$

Here, we are using the bound $\log_2(25/23) < 0.125$. The second iteration bound gives

$$\begin{aligned} \varepsilon_2 &= \frac{\varepsilon_1^2}{2x_1} \\ &\leq \frac{1}{24} 2^{e-11.75} \\ &\leq 2^{e-16.25}. \end{aligned} \tag{E.52}$$

Here, we are using the bound of $x_1 \geq 48 \cdot 2^{e-6} = 24 \cdot 2^{e-5}$ from Eq. (E.49); we also use the bound $\log_2(1/24) < -4.5$. This bound for x_k continues to be used for the next error bounds:

$$\begin{aligned} \varepsilon_3 &\leq 2^{e-33} \\ \varepsilon_4 &\leq 2^{e-66.5} \\ \varepsilon_5 &\leq 2^{e-133.5} \\ &\vdots \end{aligned} \tag{E.53}$$

Mutatis mutandis, the cases of $v \in \{10, 11, 12, 13, 14, 15\}$ follow; in these examples, we have $|\varepsilon_0| \leq 3 \cdot 2^{e-5}$. The important fact is that, in all cases, $\varepsilon_5 \leq 2^{e-128}$. We note that the case $v = 10$ is shown in Appendix E.2.6.

E.2.6 `Hyper4`

Note that the error bound derivation here is essentially the same as that of `Linear` in Appendix E.2.5. The analysis is the same; the difference is the initialization value.

We start by scaling n (relabeled as m) so that

$$2^{2e-2} \leq m < 2^{2e}. \tag{E.54}$$

for $e = 128$. We then set

$$v := \left\lfloor \frac{m}{2^{2e-4}} \right\rfloor \tag{E.55}$$

so that

$$v \cdot 2^{2e-4} \leq m < (v+1) \cdot 2^{2e-4}, \quad v \in \{4, 5, \ldots, 15\}. \tag{E.56}$$

This allows us to make the reduction to

$$\sqrt{v} \cdot 2^{e-2} \leq \sqrt{m} < \sqrt{v+1} \cdot 2^{e-2}. \tag{E.57}$$

In order to determine the appropriate approximation, we first find bounds for $\sqrt{v}$ and $\sqrt{v+1}$. In particular, we computed values $\nu_{v,0}, \nu_{v,1} \in \mathbb{N}$ with

$$\begin{aligned} \frac{\nu_{v,0}}{16} &\leq \sqrt{v} \\ \frac{\nu_{v,1}}{16} &\geq \sqrt{v+1} \end{aligned} \tag{E.58}$$

Using these values, we set $\mu_v \in \mathbb{N}$ with

$$\mu_v := 2 \left\lfloor \frac{512}{31 - v} \right\rfloor. \tag{E.59}$$

The initial approximation is

$$x_0 = \mu_v \cdot 2^{e-6}. \tag{E.60}$$

We separate error calculations into two cases:

- Case 1: $v \in \{4, 5, 6, 7, 8, 13, 14, 15\}$

  In this case, the initial error is

$$\varepsilon_0 \leq 2^{e-4} \tag{E.61}$$

  and we have the error bounds

$$\begin{aligned} \varepsilon_1 &\leq 2^{e-8} \\ \varepsilon_2 &\leq 2^{e-16} \\ &\vdots \end{aligned} \tag{E.62}$$

- Case 2: $v \in \{9, 10, 11, 12\}$

  This case is more involved. We focus on the case $v = 10$ so as to not repeat the work from `Linear` in Appendix E.2.5. We have the bound

$$25 \cdot 2^{e-5} \leq \sqrt{m} < 27 \cdot 2^{e-5} \tag{E.63}$$

  with

$$\mu_{10} = 24 \cdot 2^{e-5}. \tag{E.64}$$

  We have absorbed the common factor of 2 present in the previous two equations. As we can see, $|\varepsilon_0| \leq 3 \cdot 2^{e-5}$. The error bound after the first Newton iteration is

$$\begin{aligned}\varepsilon_1 &= \frac{\varepsilon_0^2}{2x_0} \\ &\leq \frac{9}{24} 2^{e-6} \\ &\leq 2^{e-7.4}.\end{aligned} \tag{E.65}$$

  Here, we are using the bound $\log_2(9/24) < -1.4$. The second iteration bound gives

$$\begin{aligned}\varepsilon_2 &= \frac{\varepsilon_1^2}{2x_1} \\ &\leq \frac{1}{25} 2^{e-10.8} \\ &\leq 2^{e-15.44}.\end{aligned} \tag{E.66}$$

  Here, we are using the bound of $x_1 \geq 25 \cdot 2^{e-5}$ from Eq. (E.63); we also use the bound $\log_2(1/25) < -4.64$. This bound for x_k continues to be used for the next error bounds:

$$\begin{aligned}\varepsilon_3 &\leq 2^{e-31.52} \\ \varepsilon_4 &\leq 2^{e-63.68} \\ \varepsilon_5 &\leq 2^{e-128} \\ &\vdots\end{aligned} \tag{E.67}$$

  The case of $v = 9$ was shown in Appendix E.2.5 and the case is similar for $v \in \{11, 12\}$ as $|\varepsilon_0| \leq 3 \cdot 2^{e-5}$; we do not include the details. The important fact is that, in all cases, $\varepsilon_5 \leq 2^{e-128}$.

E.2.7 `Lookup4`

We start by scaling n (relabeled as m) so that

$$2^{2e-2} \leq m < 2^{2e}. \tag{E.68}$$

for $e = 128$. We then set

$$v := \left\lfloor \frac{m}{2^{2e-4}} \right\rfloor \tag{E.69}$$

so that

$$v \cdot 2^{2e-4} \leq m < (v+1) \cdot 2^{2e-4}, \quad v \in \{4, 5, \dots, 15\}. \tag{E.70}$$

This allows us to make the reduction to

$$\sqrt{v} \cdot 2^{e-2} \leq \sqrt{m} < \sqrt{v+1} \cdot 2^{e-2}. \tag{E.71}$$

In order to determine the appropriate approximation, we first find bounds for $\sqrt{v}$ and $\sqrt{v+1}$. In particular, we computed values $\nu_{v,0}, \nu_{v,1} \in \mathbb{N}$ with

$$\begin{aligned} \frac{\nu_{v,0}}{8} &\leq \sqrt{v} \\ \frac{\nu_{v,1}}{8} &\geq \sqrt{v+1}. \end{aligned} \tag{E.72}$$

Combining Eqs. (E.71) and (E.72), we have

$$\nu_{v,0} \cdot 2^{e-5} \leq \sqrt{m} < \nu_{v,1} \cdot 2^{e-5}. \tag{E.73}$$

We can choose $\nu_{v,i}$ such that

$$\max_{v} |\nu_{v,1} - \nu_{v,0}| = 3. \tag{E.74}$$

This allows use to choose values $\mu_v \in \mathbb{N}$ so that

$$|\nu_{v,i} - \mu_v| \leq 2. \tag{E.75}$$

The initial approximation is

$$x_0 = \mu_v \cdot 2^{e-5} \tag{E.76}$$

and the initial error is

$$\varepsilon_0 \leq 2^{e-4}. \tag{E.77}$$

This gives the error bounds

$$\begin{aligned} \varepsilon_1 &\leq 2^{e-8} \\ \varepsilon_2 &\leq 2^{e-16} \\ &\vdots \end{aligned} \tag{E.78}$$

E.2.8 `Lookup8`

We start by scaling n (relabeled as m) so that

$$2^{2e-2} \leq m < 2^{2e}. \tag{E.79}$$

We then set for $e = 128$.

$$v := \left\lfloor \frac{m}{2^{2e-8}} \right\rfloor \tag{E.80}$$

so that

$$v \cdot 2^{2e-8} \leq m < (v+1) \cdot 2^{2e-8}, \quad v \in \{64, 65, \ldots, 255\}. \tag{E.81}$$

This allows us to make the reduction to

$$\sqrt{v} \cdot 2^{e-4} \leq \sqrt{m} < \sqrt{v+1} \cdot 2^{e-4}. \tag{E.82}$$

In order to determine the appropriate approximation, we first find bounds for $\sqrt{v}$ and $\sqrt{v+1}$. In particular, we computed values $\nu_{v,0}, \nu_{v,1} \in \mathbb{N}$ with

$$\begin{aligned} \frac{\nu_{v,0}}{32} &\leq \sqrt{v} \\ \frac{\nu_{v,1}}{32} &\geq \sqrt{v+1}. \end{aligned} \tag{E.83}$$

Combining Eqs. (E.82) and (E.83), we have

$$\nu_{v,0} \cdot 2^{e-9} \leq \sqrt{m} < \nu_{v,1} \cdot 2^{e-9}. \tag{E.84}$$

We can choose $\nu_{v,i}$ such that

$$\max_{v} |\nu_{v,1} - \nu_{v,0}| = 3. \tag{E.85}$$

This allows use to choose values $\mu_v \in \mathbb{N}$ so that

$$|\nu_{v,i} - \mu_v| \leq 2. \tag{E.86}$$

The initial approximation is

$$x_0 = \mu_v \cdot 2^{e-9} \tag{E.87}$$

and the initial error is

$$\varepsilon_0 \leq 2^{e-8}. \tag{E.88}$$

This gives the error bounds

$$\begin{aligned} \varepsilon_1 &\leq 2^{e-16} \\ \varepsilon_2 &\leq 2^{e-32} \\ &\vdots \end{aligned} \tag{E.89}$$

E.2.9 Halley Iterations

We briefly discuss the error bounds on Halley iterations required when using the `Newton3` initialization. We recall the Halley iteration error from Appendix B.2: if

$$\begin{aligned} h_\alpha(t) &:= \frac{t^3 + 3\alpha t}{3t^2 + \alpha} \\ \varepsilon &:= t - \sqrt{\alpha}, \end{aligned} \tag{E.90}$$

then we have

$$h_\alpha(t) - \sqrt{\alpha} = \frac{\varepsilon^3}{3t^2 + \alpha}. \tag{E.91}$$

The `Newton3` initialization gives $|\varepsilon_0| \leq 2^{e-2}$. We recall that $x_0 = 2^{e-1} + 2^{e-2}$ and compute the error bound

$$\begin{aligned} |\varepsilon_1| &= \frac{|\varepsilon_0|^3}{3x_0^2 + n} \\ &\leq \frac{2^{3e-6}}{27 \cdot 2^{2e-4} + n} \\ &\leq \frac{1}{31} 2^{e-2} \\ &\leq 2^{e-6.95}. \end{aligned} \tag{E.92}$$

In the last inequality, we are using the fact $\log_2(1/31) \leq -4.95$. Continuing the same process, we compute

$$\begin{aligned} |\varepsilon_2| &\leq 2^{e-20.85} \\ |\varepsilon_3| &\leq 2^{e-62.55} \\ |\varepsilon_4| &\leq 2^{e-187.65} \\ &\vdots \end{aligned} \tag{E.93}$$

This shows that 4 Halley iterations is sufficient *in exact arithmetic*, but we will not perform the additional work required to ensure valid return logic for `Halley` (Listing 18).

E.3 Return Logic

We now discuss the final check performed in the new algorithms and discuss a potential issue with checks performed in other algorithms found online.

E.3.1 Possible Values at the end of Newton Iterations

In the case of algorithms performing unrolled Newton iterations, all algorithms perform iterations until $\varepsilon_k \leq 1$. This ensures that $r \in \{q, q+1\}$ with respect to error bounds. This is also ensured even if the iteration oscillates. Thus, we must ensure that we return the correct value.

E.3.2 Ensuring Proper Return Logic

We want to ensure $r^2 \leq n$. This can be done in at least two different ways.

Equivalent Forms For $n, r \in \mathbb{N}^*$ and $q \coloneqq \textsc{Isqrt}(n)$, the following are equivalent:

1. $r \leq q$
2. $r^2 \leq n$
3. $r \leq \lfloor n/r \rfloor$.

We have 1 implies 2 as $r \leq q$ implies $r^2 \leq q^2 \leq n$. We have 2 implies 3 as $r^2 \leq n$ implies $r \leq n/r$, and $r \in \mathbb{N}$ so $r \leq \lfloor n/r \rfloor$. Finally, 3 implies 1 as $r \leq \lfloor n/r \rfloor \leq n/r$ and so $r^2 \leq n$; q is the largest integer with $q^2 \leq n$, so $r \leq q$.

Using the Equivalence Given that we have $r \in \{q, q+1\}$, we can check if $r^2 \leq n$: if this is true, then $r = q$; otherwise, $r = q + 1$. While valid, this return logic may cause problems due to overflow; thus, in some of the newly developed algorithms, the case $n \geq (2^{128} - 1)^2$ is handled separately with an `if` statement at the beginning. We may also check $r \leq \lfloor n/r \rfloor$, which will not overflow.

E.3.3 Return Minimum

We note that in some of the algorithms found online (`PRB` (Listing 2) [1], `OpenZeppelin` (Listing 3) [20], and `ABDK` (Listing 4) [4]) the final check is of the form

$$\textbf{return} \quad \min\left(r, \lfloor n/r \rfloor\right). \tag{E.94}$$

This is done after applying 7 Newton iterations.

It is not clear that this logic ensures the correct value is returned. These algorithms have the same initialization as in `Unrolled1` and perform 7 Newton iterations, so we know $r \in \{q, q+1\}$.

Suppose we have

$$n = q^2 + \ell, \quad \ell \in \{0, \ldots, q-1\}. \tag{E.95}$$

Then we see

$$n = (q-1)(q+1) + (1+\ell) \tag{E.96}$$

so that

$$\left\lfloor \frac{n}{q+1} \right\rfloor = q - 1. \tag{E.97}$$

This shows that if n is as specified and we have $r = q + 1$ after the final Newton iteration (which is possible with the current analysis), then $\min(r, \lfloor n/r \rfloor) = \lfloor n/r \rfloor = q - 1$, which is incorrect. Thus, the current analysis will not ensure that this return logic will always give the correct result.

We note that although we have shown that it is *possible* for `PRB`, `OpenZeppelin`, and `ABDK` to return incorrect results (that is, results which are off by 1), we must mention that there is no known value where this error occurs. That is to say, the current analysis is *unable to prove the algorithms are correct*, but that is not the same as *proving they are incorrect*. With that said, these algorithms operate on `uint256` values and approximately half of all integers are of the form described in Eq. (E.95), so it is not possible to manually check every value to ensure a valid result is returned. Thus, if `PRB`, `OpenZeppelin`, and `ABDK` are to be proven correct, additional analysis is required; we leave this extended analysis to the interested reader.

E.4 Convergence Proofs

We now combine the results from the previous sections to give proofs of convergence.

E.4.1 Proof of General Newton Iteration

We start by proving Algorithm 1 (from [3, Algorithm 1.7.1]) produces the correct result. This is included for completeness, and the proof provided here is essentially that of [3, Algorithm 1.7.1, Proof].

First, x decreases at every iteration. Because n is finite, the algorithm must eventually terminate. We set $q = \lfloor \sqrt{n} \rfloor$. From the discussion above, we always have $(t + n/t)/2 \geq \sqrt{n}$ for all $t > 0$, so we always have $x \geq q$. We see

$$y - x = \left\lfloor \frac{n - x^2}{2x} \right\rfloor. \tag{E.98}$$

It follows that $x > \sqrt{n}$ iff $n - x^2 < 0$ iff $y < x$; therefore, $y \geq x$ implies $x \leq \sqrt{n}$. Because $x \in \mathbb{N}$ and $q \leq x \leq \sqrt{n}$, we surmise $x = q$. Thus, the algorithm returns the correct result.

Note: the only place we used the initialization value is to ensure that we initially have $x \geq \lfloor \sqrt{n} \rfloor$. Thus, provided the initialization value always satisfies this, this proof may be applied.

E.4.2 Proof of Newton Iteration Fixed Points

We now prove that for $q = \lfloor \sqrt{n} \rfloor$ we have q is a fixed point of g_n (that is, $g_n(q) = q$) iff $n+1$ is not a square.

We let $y = g_n(x)$ as in Appendix E.4.1. The work there shows that $x > \sqrt{n}$ implies that $y < x$; additionally, for $x < q$ we have $y > x$. This holds for all n. So, we only need to investigate the case $x = q$.

From the definition of $q = \lfloor \sqrt{n} \rfloor$ we have

$$q^2 \leq n < (q+1)^2 . \tag{E.99}$$

This implies

$$q^2 \leq n \leq (q+2)\, q \tag{E.100}$$

so that

$$\left\lfloor \frac{n}{q} \right\rfloor \in \{q, q+1, q+2\} . \tag{E.101}$$

It follows that

$$g_n(q) = \begin{cases} q & \lfloor n/q \rfloor = q \\ q & \lfloor n/q \rfloor = q+1 \\ q+1 & \lfloor n/q \rfloor = q+2 \end{cases} . \tag{E.102}$$

We note that when $n = a^2 - 1$ (that is, when $n+1$ is a square) q is not a fixed point for g_n. We note that the result will oscillate as $q, q+1, q, q+1, \ldots$.

E.4.3 `Unrolled1`

From the error bounds in Appendix E.2.1, we see that $\varepsilon_7 \leq 1$ (we require 7 Newton iterations). Thus, $r \in \{q, q+1\}$, and the return logic discussed in Appendix E.3 ensures the correct result.

E.4.4 `Unrolled2`

From the error bounds in Appendix E.2.2, we see that $\varepsilon_7 \leq 1$ (we require 7 Newton iterations). Thus, $r \in \{q, q+1\}$, and the return logic discussed in Appendix E.3 ensures the correct result.

E.4.5 `Unrolled3`

From the error bounds in Appendix E.2.3, we see that $\varepsilon_6 \leq 1$ (we require 6 Newton iterations). Thus, $r \in \{q, q+1\}$, and the return logic discussed in Appendix E.3 ensures the correct result.

E.4.6 `While1`

After the first Newton iteration, we have $r \geq q$; the proof in Appendix E.4.1 then applies. The first Newton iteration *is required* for this proof to be valid.

E.4.7 `While2`

The proof in Appendix E.4.1 applies because $r > \sqrt{n}$.

E.4.8 `While3`

After the first Newton iteration, we have $r \geq q$; the proof in Appendix E.4.1 then applies. The first Newton iteration *is required* for this proof to be valid.

E.4.9 `BitLength`

We let $k \in \mathbb{N}$ and suppose we have

$$2^{2k-2} \leq n < 2^{2k} \tag{E.103}$$

After the `if (result >= (1 << 2))` statement, we will have $e = 2k - 1$.

If the statement `if (result >= (1 << 1))` is True, then this means that we actually have the bitlength is equal to $2k$. We have

$$r = 27 \cdot 2^{k-5}. \tag{E.104}$$

Otherwise, `if (result >= (1 << 1))` is False and we have the bitlength is equal to $2k-1$, We have

$$r = 39 \cdot 2^{k-6}. \tag{E.105}$$

From the error bounds in Appendix E.2.4, we see that $\varepsilon_6 \leq 1$ (we require 6 Newton iterations). Thus, $r \in \{q, q+1\}$, and the return logic discussed in Appendix E.3 ensures the correct result.

E.4.10 `Linear`

Initial Scaling We show that we properly scale n so that $2^{254} \leq m < 2^{256}$.

We let $k \in \mathbb{N}$ such that

$$2^{2k-2} \leq n < 2^{2k}. \tag{E.106}$$

In this case, we see that the "bitlength" b is

$$b := 2k - 1. \tag{E.107}$$

The "bitlength" b is the scaling factor e before it is overwritten. From here, we compute the scaling factor

$$e := 128 - k. \tag{E.108}$$

We define

$$m := n \cdot 2^{2e}. \tag{E.109}$$

From Eq. (E.106), we see

$$\begin{aligned} m &= 2^{2e} n \\ &\geq 2^{256-2k} \cdot 2^{2k-2} \\ &= 2^{254} \end{aligned} \tag{E.110}$$

and

$$\begin{aligned} m &= 2^{2e} n \\ &< 2^{256-2k} \cdot 2^{2k} \\ &= 2^{256}. \end{aligned} \tag{E.111}$$

Together, these imply

$$2^{254} \leq m < 2^{256}, \tag{E.112}$$

as desired.

Error Bounds From the error bounds in Appendix E.2.5, we see that $\varepsilon_5 \leq 1$ (we require 5 Newton iterations). Thus, $r \in \{q, q+1\}$ for $q = \lfloor \sqrt{m} \rfloor$.

Final Scaling and Check At this point, we have $r \in \{q, q+1\}$ for $q = \lfloor \sqrt{m} \rfloor$. This means

$$(r-1)^2 \leq m < (r+1)^2. \tag{E.113}$$

We set

$$r = s2^e + t, \quad t \in \left\{0, 1, \cdots, 2^{e-1}\right\}. \tag{E.114}$$

This implies that

$$s = \left\lfloor \frac{r}{2^e} \right\rfloor. \tag{E.115}$$

We see

$$\begin{aligned} (r-1)^2 = (s2^e + t - 1)^2 \geq 2^{2e}(s-1)^2 \\ (r+1)^2 = (s2^e + t + 1)^2 \leq 2^{2e}(s+1)^2. \end{aligned} \tag{E.116}$$

After substituting $m = 2^{2e} n$ and using the previous inequalities, Eq. (E.113) reduces to

$$2^{2e}(s-1)^2 \leq 2^{2e} n < 2^{2e}(s+1)^2 \tag{E.117}$$

or

$$(s-1)^2 \leq n < (s+1)^2 . \tag{E.118}$$

Thus, $s \in \{z, z+1\}$ for $z = \lfloor \sqrt{n} \rfloor$. The final check ensures we return the correct result. There is no overflow because we check if $s^2 \leq n$ by checking if $s \leq \lfloor n/s \rfloor$.

E.4.11 `Hyper4`

Initial Scaling See the discussion in Appendix E.4.10.

Error Bounds From the error bounds in Appendix E.2.6, we see that $\varepsilon_5 \leq 1$ (we require 5 Newton iterations). Thus, $r \in \{q, q+1\}$ for $q = \lfloor \sqrt{m} \rfloor$.

Final Scaling and Check See the discussion in Appendix E.4.10. This shows the correct result is returned.

E.4.12 `Lookup4`

Initial Scaling See the discussion in Appendix E.4.10.

Error Bounds From the error bounds in Appendix E.2.7, we see that $\varepsilon_5 \leq 1$ (we require 5 Newton iterations). Thus, $r \in \{q, q+1\}$ for $q = \lfloor \sqrt{m} \rfloor$.

Final Scaling and Check See the discussion in Appendix E.4.10. This shows the correct result is returned.

E.4.13 `Lookup8`

Initial Scaling See the discussion in Appendix E.4.10.

Error Bounds From the error bounds in Appendix E.2.8, we see that $\varepsilon_4 \leq 1$ (we require 4 Newton iterations). Thus, $r \in \{q, q+1\}$ for $q = \lfloor \sqrt{m} \rfloor$.

Final Scaling and Check See the discussion in Appendix E.4.10. This shows the correct result is returned.