Efficient Integer Square Roots in Solidity

Christopher H. Gorman

October 19, 2022

1 Introduction

Mathematics in Solidity is based around operations on unsigned (nonnegative) integers. Although the usual operations of addition, subtraction, multiplication, (integer) division, and exponentiation are included, there is no operation for computing square roots. This missing operation is likely due to the fact that all operations in Solidity are restricted to integers. However, there are times when it is necessary to compute an *integer square root*. One example is when performing minting and burning calculations for algebraic sigmoid bonding curves [4]. It would be helpful to have an optimized function to perform this operation.

There are various examples online which discuss how to compute integer square roots; these include `Uniswap` [8]. `prb-math` [1], `Open-Zeppelin` [5], and `ABDK` [2]. All of these are based Newton's method for computing square roots (also called Babylonian or Heron's method). When using Newton's method, all operations involving real numbers are converted to operations involving integers; iteration is stopped after meeting the specified stopping criterion. No references mention proofs of correctness. A proof is preferred for such a fundamental operation.

We begin by reviewing integer square root algorithms based on Newton's method. After this, we give an improved version. Additionally, we present a method for computing integer square roots based on work in [3], which discusses the algorithm for computing integer square roots in Python. The two additional algorithms presented have the advantage of being *provably correct*. All methods are then compared to see which is most efficient. All proofs are relegated to appendices.

2 Definitions and Mathematical Review

We require the following definitions:

Definition 2.1 (Floor Function)**.** Given $x \in \mathbb{R}$, we set

$$\lfloor x \rfloor \coloneqq \max \{m \in \mathbb{Z} \mid m \leq x\}. \tag{2.1}$$

Thus, if $\lfloor x \rfloor = m$, then m is the unique integer such that

$$m \leq x < m + 1. \tag{2.2}$$

Definition 2.2 (Integer Square Root)**.** Given $n \in \mathbb{N}$, we say that a is the *integer square root* of n if

$$a^2 \leq n < (a+1)^2 . \tag{2.3}$$

This implies that

$$a = \left\lfloor \sqrt{n} \right\rfloor . \tag{2.4}$$

We will also write

$$\textsc{Isqrt}(n) \coloneqq \left\lfloor \sqrt{n} \right\rfloor . \tag{2.5}$$

Ideally, we want an efficient, generic algorithm which uses only integer operations. By *efficient*, we place particular emphasis on minimal gas cost when performed within the Ethereum Virtual Machine [9].

3 Iterative Algorithms based on Newton's Method

3.1 Algorithm for Real Numbers

Given $n > 0$, we can compute $\sqrt{n}$ using Newton's method for computing square roots. To do this, we define the function

$$f_n(x) \coloneqq \frac{1}{2}\left(x + \frac{n}{x}\right) . \tag{3.1}$$

Given an initial approximation $x_0 > 0$, we define the sequence

$$x_{k+1} \coloneqq f_n(x_k). \tag{3.2}$$

When all operations are performed using real numbers, it is not difficult to show that

$$\lim_{k\to\infty} x_k = \sqrt{n}. \tag{3.3}$$

Furthermore, this sequence converges quickly, and it can be shown that if

$$\varepsilon_k = x_k - \sqrt{n}, \tag{3.4}$$

then

$$\begin{aligned} \varepsilon_{k+1} &= \frac{\varepsilon_k^2}{2x_k} \\ &\leq \frac{\varepsilon_k^2}{2\sqrt{n}}. \end{aligned} \tag{3.5}$$

The inequality follows from the assumption $x_k \geq \sqrt{n}$. This implies that the sequence defined by Eq. (3.2) exhibits *quadratic convergence*; that is, the number of correct bits roughly doubles every iteration.

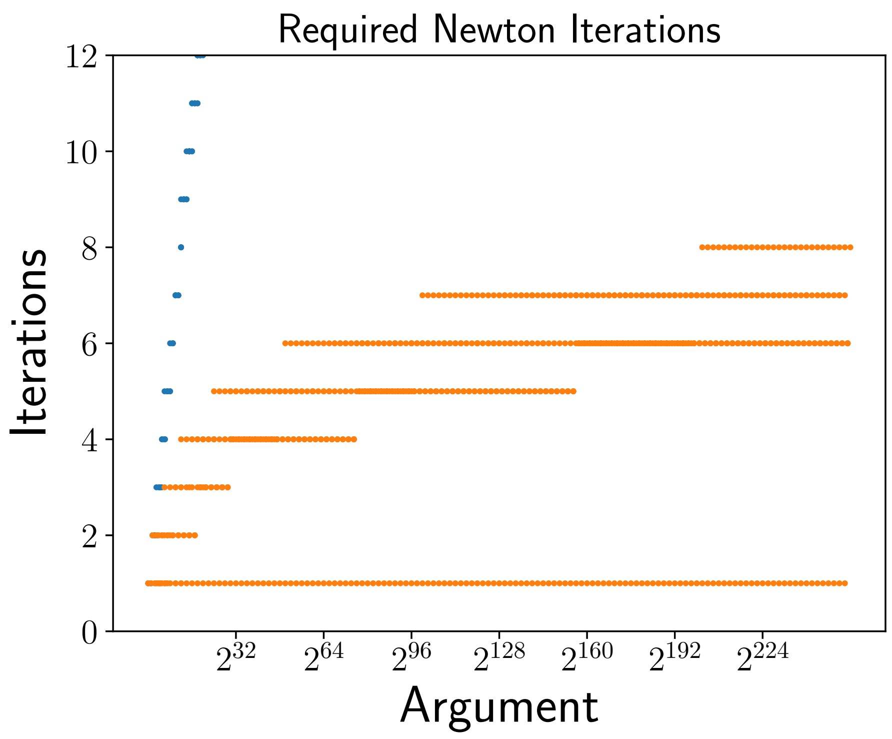


Figure 1: We plot the number of Newton iterations required to reach the integer square root. Blue uses the initialization $a_0 = n$. This is *very poor* when compared with the initialization in orange with $a_0 = 2^\ell$ for ℓ maximal such that $2^\ell \leq \textsc{Isqrt}(n)$. Precisely, given the initialization a_0 and $a_{k+1} = g_n(a_k)$, we are plotting i where a_i is the minimal index where $a_i = \textsc{Isqrt}(n)$.

3.2 Implementation using Computer Arithmetic

In practice, the iteration in Eq. (3.2) must be implemented on a computer. Using floating-point arithmetic, the convergence implies the method will converge to the correct solution rapidly given a good initial approximation.

Because we are interested in computing *integer square roots*, we could modify this to use only integer operations. Thus, the function f_n in Eq. (3.1) must become

$$g_n(a) \coloneqq \left\lfloor \frac{1}{2}\left(a + \frac{n}{a}\right)\right\rfloor. \tag{3.6}$$

Here, all divisions are integer divisions.

Given a positive integer a_0, we define

$$a_{k+1} \coloneqq g_n(a_k). \tag{3.7}$$

It can be shown that the sequence $\{a_k\}$ will eventually reach $\lfloor\sqrt{n}\rfloor$; see App. A for more information.

A poor choice of initial value will lead to a large number of Newton iterations; see Figure 1 for an example. The number of iterations required when choosing $a_0 = n$ grows quickly.

Listing 1: Method for computing Integer Square Roots from `Uniswap` [8]

```
// License: GPL-3.0; see https://github.com/Uniswap/v2-core
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;
    }
}
```

3.3 Specific Algorithms

We now compare the implementations from `Uniswap` (Listing 1), `prb-math` (Listing 2), `Open-Zeppelin` (Listing 3), and `ABDK` (Listing 4). Some comments have been removed due to space considerations.

Looking at `Uniswap` (Listing 1), we can see that it uses an initial approximation close to n and then performs Newton iterations until reaching its stopping criterion. Based on Figure 1, we might expect this might require a large number of iterations, leading to large gas costs; this is the case as discussed in Sec. 5 and seen in Table 1.

The other methods `prb-math` (Listing 2), `Open-Zeppelin` (Listing 3), and `ABDK` (Listing 4) all appear to be essentially the same. In particular, `prb-math` (Listing 2) and `ABDK` (Listing 4) appear to be essentially the same down even to the comment about only requiring 7 Newton iterations; compare Line 35 of `prb-math` (Listing 2) and Line 36 of `ABDK` (Listing 4). In `ABDK` (Listing 4), the initial approximation (`result`) is initialized slightly differently (compare `xAux >= 0x4` on Line 24 of `prb-math` (Listing 2) with `xx >= 0x8` on Line 26 of `ABDK` (Listing 4)); `prb-math` (Listing 2) previously used `0x8` but this was corrected (changed from `xAux >= 0x4` to `xAux >= 0x8`; see the file history in [1]). In `prb-math` (Listing 2), `result` is initialized to $2^k \leq \textsc{Isqrt}(x)$ (k maximal). Looking at `Open-Zeppelin` (Listing 3) and the definition of `log2`, we see that `Open-Zeppelin` (Listing 3) has the same initialization as `prb-math` (Listing 2); thus, these two methods will always produce the same result. `ABDK` (Listing 4) likely produces similar results, and the same results in certain situations. With the algorithms being essentially the same, the fact that `ABDK` (Listing 4) has the entire function “`unchecked`” would lead one to expect it may be more efficient; this is seen in Table 1.

The fact that none of these algorithms have proofs of correctness led the author of this work to search for provably correct algorithms. A *provably correct* algorithm based on Newton’s iteration is given in `Mod-Newton` (Listing 5); a prove of correctness is given in App. B. The main differences include a different initialization value and a final check

Listing 2: Method for computing Integer Square Roots from `prb-math` [1]

```
// SPDX-License-Identifier: Unlicense
function sqrt(uint256 x) internal pure returns (uint256 result) {
    if (x == 0) { return 0; }
    uint256 xAux = uint256(x);
    result = 1;
    if (xAux >= 0x100000000000000000000000000000000) {
        xAux >>= 128; result <<= 64;
    }
    if (xAux >= 0x10000000000000000) {
        xAux >>= 64;  result <<= 32;
    }
    if (xAux >= 0x100000000) {
        xAux >>= 32;  result <<= 16;
    }
    if (xAux >= 0x10000) {
        xAux >>= 16;  result <<= 8;
    }
    if (xAux >= 0x100) {
        xAux >>= 8;   result <<= 4;
    }
    if (xAux >= 0x10) {
        xAux >>= 4;   result <<= 2;
    }
    if (xAux >= 0x4) {
                      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;
        // Seven iterations should be enough
        uint256 roundedDown = x / result;
        return result >= roundedDown ? roundedDown : result;
    }
}
```

Listing 3: Method for computing Integer Square Roots from `Open-Zeppelin` [5]

```
// 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 [2]

```
// 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 >= 0x8) {
                            r <<= 1;
            }
            r = (r + x / r) >> 1;
            r = (r + x / r) >> 1;
            r = (r + x / r) >> 1;
            r = (r + x / r) >> 1;
            r = (r + x / r) >> 1;
            r = (r + x / r) >> 1;
            r = (r + x / r) >> 1;
            // Seven iterations should be enough
            uint256 r1 = x / r;
            return uint128 (r < r1 ? r : r1);
        }
    }
}
```

Listing 5: Provably correct method for computing Integer Square Roots

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

ensuring Eq. (2.3) holds.

4 Computing Integer Square Roots in Python

While searching for provably correct algorithms for computing $\textsc{Isqrt}(n)$, the author came across [3]; this paper describes the method Python uses to compute integer square roots. The algorithm may be found in Alg. 1; the Solidity code may be found in Listing 6. A proof of correctness is relegated to App. C.

5 Gas Costs

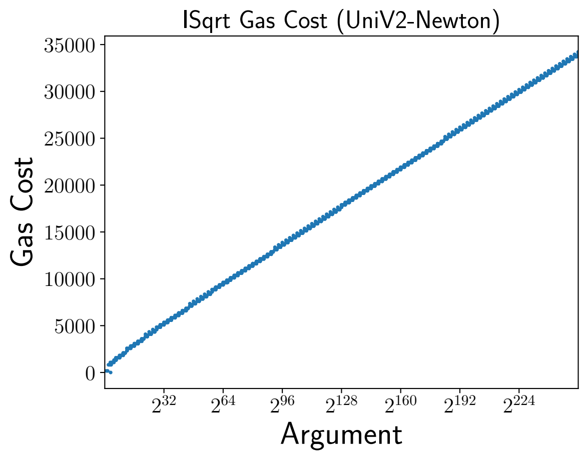
We now look at the associated gas costs of the algorithms we have discussed; this is the important metric when performing operations in Solidity. A plot showing gas cost data may be found in Fig. 2; see Table 1 for some summary statistics. While the exact values will depend on the choice of inputs, we expect the values we used provide a good characterization. No algorithm returned an incorrect result.

Algorithm 1 Provably correct algorithm for computing Integer Square Roots based on [3]

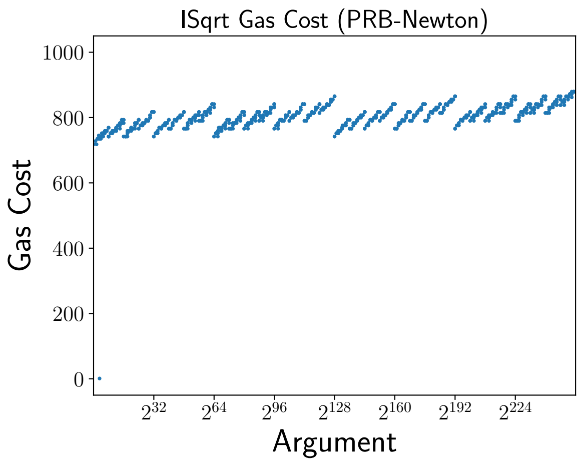
```
Require: $x \in \mathbb{N}$, $0 \leq x < 2^{256}$
function IntegerSquareRoot($x$)
    if $x \leq 1$ then
        return $x$
    end if
    if $x \geq (2^{128} - 1)^2$ then
        return $2^{128} - 1$
    end if
    $b \coloneqq$ BitLength($x$)
    $e \coloneqq (256 - b) \gg 1$
    $m \coloneqq x \ll 2e$                                ▷ We have $2^{254} \leq m < 2^{256}$
    $a \coloneqq 1 + (m \gg 254)$
    $a \coloneqq (a \ll 1) + (m \gg 251)/a$
    $a \coloneqq (a \ll 3) + (m \gg 245)/a$
    $a \coloneqq (a \ll 7) + (m \gg 233)/a$
    $a \coloneqq (a \ll 15) + (m \gg 209)/a$
    $a \coloneqq (a \ll 31) + (m \gg 161)/a$
    $a \coloneqq (a \ll 63) + (m \gg 65)/a$
    $a \coloneqq a \gg e$
    if $a^2 \leq x$ then
        return $a$
    else
        return $a - 1$
    end if
end function
```

Listing 6: Provably correct method for computing Integer Square Roots based on [3]

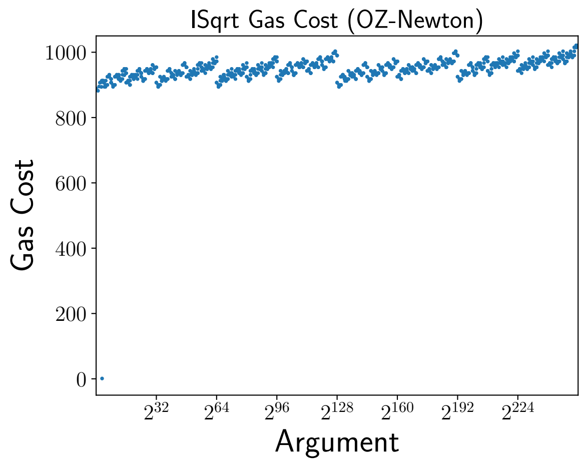
```
// SPDX-License-Identifier: MIT
function sqrt(uint256 x) internal pure returns (uint256) {
    unchecked {
        if (x <= 1) { return x; }
        if (x >= ((1 << 128) - 1)**2) { return (1 << 128) - 1; }
        // Here, e represents the bit length;
        uint256 e = 1;
        uint256 result = x;
        if (result >= (1 << 128)) { result >>= 128; e += 128; }
        if (result >= (1 << 64 )) { result >>= 64;  e += 64;  }
        if (result >= (1 << 32 )) { result >>= 32;  e += 32;  }
        if (result >= (1 << 16 )) { result >>= 16;  e += 16;  }
        if (result >= (1 << 8  )) { result >>= 8;   e += 8;   }
        if (result >= (1 << 4  )) { result >>= 4;   e += 4;   }
        if (result >= (1 << 2  )) { result >>= 2;   e += 2;   }
        if (result >= (1 << 1  )) {                 e += 1;   }
        // e is currently bit length; we overwrite it to scale x
        e = (256 - e) >> 1;
        // m now satisfies 2**254 <= m < 2**256
        uint256 m = x << (2 * e);
        // result now stores the result
        result = 1 + (m >> 254);
        result = (result << 1 ) + (m >> 251) / result;
        result = (result << 3 ) + (m >> 245) / result;
        result = (result << 7 ) + (m >> 233) / result;
        result = (result << 15) + (m >> 209) / result;
        result = (result << 31) + (m >> 161) / result;
        result = (result << 63) + (m >> 65 ) / result;
        result >>= e;
        return result * result <= x ? result : (result - 1);
    }
}
```



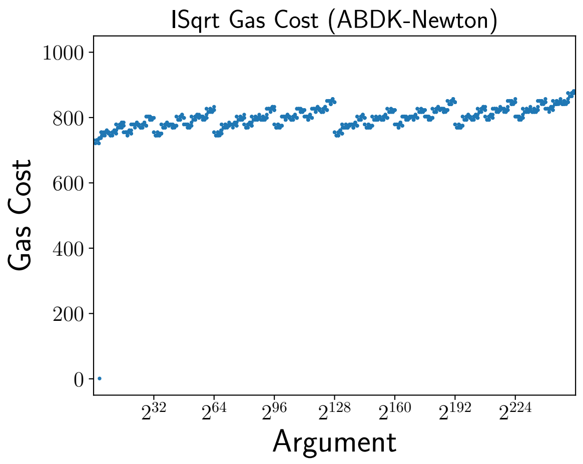
(a) Gas cost for Listing 1



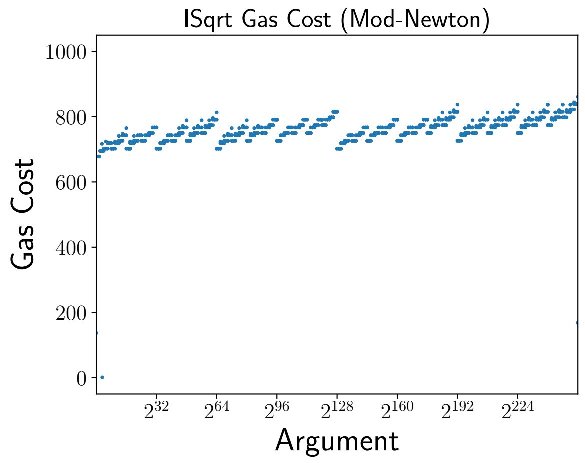
(b) Gas cost for Listing 2



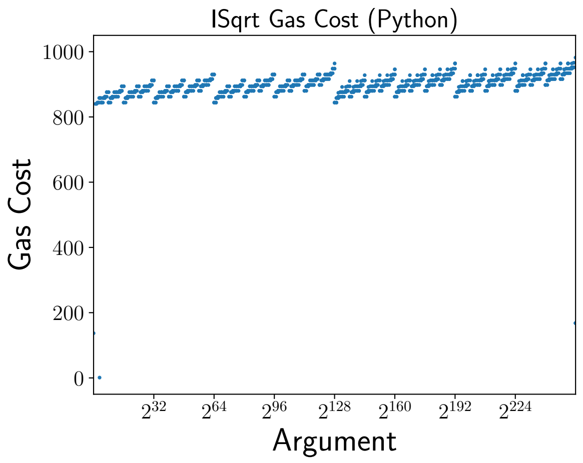
(c) Gas cost for Listing 3



(d) Gas cost for Listing 4



(e) Gas cost for Listing 5



(f) Gas cost for Listing 6

Figure 2: Here are plots of gas costs for computing Integer Square Roots in Solidity using a wide distribution of values. As we can see, the gas costs for `Uniswap` shows that it should not be used. We note the `Mod-Newton` and `Python` algorithms have proofs of correctness.

	`Uniswap`	`prb-math`	`Open-Zeppelin`	`ABDK`	`Mod-Newton`	`Python`
Max	34 205	879	1021	881	861	982
Mean	17 627	803	948	798	757	896
Median	17 890	807	949	799	767	898
Std	9549	43	44	43	55	62

Table 1: Here are some statistics related to the gas cost data from Fig. 2. We recall that the `Mod-Newton` and `Python` algorithms have proofs of correctness.

We now look at the results from Table 1. First, it is clear that `Uniswap` (Listing 1) requires significantly more gas; as a result, it should not be used. The remaining algorithms have similar gas cost, especially `prb-math` (Listing 2) and `ABDK` (Listing 4). `ABDK` is likely more efficient due to using "`unchecked`" throughout the entire function. The fact that `Open-Zeppelin` (Listing 3) calls other functions results in increased gas costs.

Overall, we see that `Mod-Newton` (Listing 5) has the smallest maximum, mean, and median gas cost; the standard deviation is not much larger in comparison to the other methods. The minimal gas cost is due in part to using 6, rather than 7, Newton iterations. We see that `Python` (Listing 6) is efficient, though not most efficient; the provable nature of the algorithm is valuable, however.

It would be interesting to compare `Mod-Newton` (Listing 5) and `Python` (Listing 6) when both algorithms are written in assembly; this would likely result in more efficient operations.

6 Conclusion

Because none of the popular algorithms for computing Integer Square Roots in Solidity have a proof of correctness, two provably correct methods were presented. One of these provably correct methods was determined to be the most efficient in terms of gas cost based on a sampling of values.

References

[1] Paul Razvan Berg. *Integer Square Root Algorithm in Solidity*. June 2022. URL: https://github.com/paulrberg/prb-math/blob/main/contracts/PRBMath.sol#L599.

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

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

[4] Christopher H. Gorman. *Explicit Formulas for Algebraic Sigmoid Bonding Curves*. July 2022.

[5] OpenZeppelin. *Integer Square Root Algorithm in Solidity*. July 2022. URL: https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/utils/math/Math.sol#L158.

[6] Walter Rudin. *Principles of Mathematical Analysis*. 3rd ed. McGraw-Hill New York, 1976.

[7] Victor Shoup. *A Computational Introduction to Number Theory and Algebra*. Cambridge University Press, 2009. URL: https://shoup.net/ntb/.

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

[9] Gavin Wood. *Ethereum: A Secure Decentralised Generalised Transaction Ledger*. Yellowpaper. Ethereum Project, 2022. URL: https://github.com/ethereum/yellowpaper.

A Additional Mathematics

We will repeatedly use the following fact throughout this discussion:

Fact 1 (Exercise 1.5 in [7])
For $x \in \mathbb{R}$ and $n \in \mathbb{Z}$ with $n > 0$, we have

$$\left\lfloor \frac{\lfloor x \rfloor}{n} \right\rfloor = \left\lfloor \frac{x}{n} \right\rfloor . \tag{A.1}$$

This implies

$$\left\lfloor \frac{\lfloor a/b \rfloor}{c} \right\rfloor = \left\lfloor \frac{a}{bc} \right\rfloor \tag{A.2}$$

for all positive integers a, b, c.

A.1 Mathematics for Newton Iteration

Here we include additional mathematics needed for the appendices.

We restate f_n from Eq. (3.1) and g_n from Eq. (3.6) for convenience:

$$\begin{aligned} f_n(x) &\coloneqq \frac{1}{2}\left(x + \frac{n}{x}\right) \\ g_n(x) &\coloneqq \left\lfloor \frac{1}{2}\left(x + \frac{n}{x}\right) \right\rfloor . \end{aligned} \tag{A.3}$$

We recall that

$$f_n(x) \geq \sqrt{n}, \quad x > 0 \tag{A.4}$$

and

$$f_n(a) \geq g_n(a) \geq \left\lfloor \sqrt{n} \right\rfloor, \quad a \in \mathbb{N}. \tag{A.5}$$

We have equality in Eq. (A.4) if and only if $x = \sqrt{n}$.

The following mathematical facts are from [3]:

Fact 2 (Lemma 3 in [3])
For any positive integer a, we have

$$\left\lfloor \sqrt{n} \right\rfloor \leq g_n(a). \tag{A.6}$$

Fact 3 (Lemma 4 in [3])
Suppose a is a positive integer satisfying $\lfloor \sqrt{n} \rfloor < a$. Then

$$g_n(a) < a. \tag{A.7}$$

Fact 4 (Lemma 5 in [3])
If $a_0 \geq \lfloor \sqrt{n} \rfloor$, then for all $i \geq 0$, $\lfloor \sqrt{n} \rfloor \leq a_i$.

Fact 5 (Lemma 6 in [3])
If $a_0 \geq \lfloor \sqrt{n} \rfloor$, then for all $i \geq 0$, $\lfloor \sqrt{n} \rfloor = a_i$ if and only if $a_i \leq a_{i+1}$.

Fact 6 (Theorem 7 in [3])
If $a_0 \geq \lfloor \sqrt{n} \rfloor$, then there is $i \geq 0$ such that $\lfloor \sqrt{n} \rfloor = a_i$.

We also note the following:

Lemma 7 (Fixed points of g_n)
We have that $\textsc{Isqrt}(n)$ is a fixed point of g_n if and only if $n - 1$ is not a perfect square.

Proof. We let

$$x^2 \leq n < (x+1)^2; \tag{A.8}$$

that is, $x = \textsc{Isqrt}(n)$. It follows that

$$x^2 \leq n \leq (x+2)\,x. \tag{A.9}$$

This means that if

$$\left\lfloor \frac{n}{x} \right\rfloor = q, \tag{A.10}$$

then we have $q \in \{x, x+1, x+2\}$.

We look at those cases separately:

- $q = x$

  We see

$$\begin{aligned} g_n(x) &= \left\lfloor \frac{1}{2}\left(x + \frac{n}{x}\right) \right\rfloor \\ &= \left\lfloor \frac{1}{2}(x + x) \right\rfloor \\ &= x. \end{aligned} \tag{A.11}$$

In this case, x is a fixed point.

- $q = x + 1$

  We see

$$\begin{aligned} g_n(x) &= \left\lfloor \frac{1}{2}\left(x + \frac{n}{x}\right) \right\rfloor \\ &= \left\lfloor \frac{1}{2}(x + x + 1) \right\rfloor \\ &= x. \end{aligned} \tag{A.12}$$

  In this case, x is a fixed point.

- $q = x + 2$

  We see

$$\begin{aligned} g_n(x) &= \left\lfloor \frac{1}{2}\left(x + \frac{n}{x}\right) \right\rfloor \\ &= \left\lfloor \frac{1}{2}(x + x + 2) \right\rfloor \\ &= x + 1. \end{aligned} \tag{A.13}$$

  In this case, x is a *not* fixed point. This only happens when $n = (x+1)^2 - 1$.

  We also note that, in this case, we have

$$\begin{aligned} g_n(x+1) &= \left\lfloor \frac{1}{2}\left(x + \frac{n}{x+1}\right) \right\rfloor \\ &= \left\lfloor \frac{1}{2}(x + x) \right\rfloor \\ &= x. \end{aligned} \tag{A.14}$$

  Thus, in this case, iteration will lead to $x, x+1, x, x+1, \cdots$.

Thus, we have shown $\textsc{Isqrt}(n)$ is a fixed point of g_n if and only if $n-1$ is not a perfect square. $\square$

A.2 Mathematics for Python Integer Square Root

The mathematics required to understand [3] is different from that required for Newton iteration. We need the following definition:

Definition A.1 (Near Square Root)**.** Given $n \in \mathbb{N}$, we say that a is a *near square root* of n if we have

$$(a-1)^2 < n < (a+1)^2 . \tag{A.15}$$

If a is the integer square root of n, then we see that a is also a near square root of n. It turns out there are *at most 2* near square roots of n:

- If n is a perfect square, then $a^2 = n$ and a is the *only* near square root of n.
- If n is not a perfect square, then a and $a+1$ are both near square roots of n. This follows because we have

$$a^2 < n < (a+1)^2 . \tag{A.16}$$

It turns out that we can "lift" a near square root of one number to a near square root of another number. In particular, we have the following result:

Theorem 8 (Theorem 11 in [3])
Suppose that n and k are positive integers satisfying $4k^4 \leq n$. Let b be a near square root of $\lfloor n/4k^2 \rfloor$. Then a defined by

$$a = kb + \left\lfloor \frac{n}{4kb} \right\rfloor \tag{A.17}$$

is a near square root of n.

The main idea of [3] is to take an initial near square root and lift the near square root to another near square root using Thm. 8 until the final approximation. Then, only one check must be performed. In particular, if α is an integer with

$$1 \leq \alpha < 4, \tag{A.18}$$

then this implies that 1 is a near square root of α:

$$0 = (1-1)^2 < 1 \leq \alpha < 4 = (1+1)^2 . \tag{A.19}$$

B Proofs of Newton Iteration Errors

B.1 Sequence Setup and Definitions

We now look at proofs related to Newton iteration for f_n and g_n from Eq. (A.3).

Given an initial value $\hat{x}_0 \in \mathbb{N}$, We are interested in looking at the sequences defined by

$$\begin{aligned} x_1 &= f_n(\hat{x}_0) & \hat{x}_1 &= g_n(\hat{x}_0) \\ x_2 &= f_n(\hat{x}_1) & \hat{x}_2 &= g_n(\hat{x}_1) \\ &\vdots & &\vdots \end{aligned} \tag{B.1}$$

The definitions show that we always have $\hat{x}_k \in \mathbb{N}$. The need for these two iterations is based on the fact that the error for Newton iterations may be computed exactly (which requires f_n), while in practice we are interested in bounding the error from the integer operations (which requires g_n).

We assume that

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

This further implies that

$$2^{e-1} \leq \lfloor \sqrt{n} \rfloor < 2^e. \tag{B.3}$$

B.2 Newton Error Bounds

Given $x > 0$, if we have the initial error

$$x - \sqrt{n} = \varepsilon \tag{B.4}$$

then the error after one Newton iteration is

$$f_n(x) - \sqrt{n} = \frac{\varepsilon^2}{2x}. \tag{B.5}$$

This refined error critically depends on the initial error. A brief discussion on this error bound may be found in many places including [6, Exercise 3.16].

We will need the following: if

$$\sqrt{n} < y \leq f_n(x), \tag{B.6}$$

it follows that

$$\begin{aligned} y - \sqrt{n} &\leq f_n(x) - \sqrt{n} \\ &= \frac{\varepsilon^2}{2x}. \end{aligned} \tag{B.7}$$

This will be needed to compare exact Newton iteration with Newton iteration over the integers.

B.3 Error Iteration Setup

In App. B.4, we will work on bounding the error from Newton iterations. The work is a bit tedious but is included for completeness.

Throughout, we will have the following: We will have approximation $\hat{x}_k \in \mathbb{N}$ and $\hat{x}_k > \sqrt{n}$ with

$$\hat{x}_k - \sqrt{n} = \bar{\varepsilon}_k \tag{B.8}$$

From the previous work, we know

$$f_n(\hat{x}_k) - \sqrt{n} = \frac{\bar{\varepsilon}_k^2}{2\hat{x}_k}. \tag{B.9}$$

In *almost all cases* (the exceptional cases will be noted), we will bound this error using the previous bound for $\bar{\varepsilon}_k$ and $\hat{x}_k \geq 2^{e-1}$. By assuming

$$\hat{x}_{k+1} \coloneqq g_n(\hat{x}_k) > \sqrt{n}, \tag{B.10}$$

we will have

$$\hat{x}_{k+1} - \sqrt{n} \leq \frac{\bar{\varepsilon}_k^2}{2\hat{x}_k}. \tag{B.11}$$

When the inequality in Eq. (B.10) fails, it is because

$$\hat{x}_k = \textsc{Isqrt}(n); \tag{B.12}$$

this follows from Eqs. (A.4) and (A.5).

From Lemma 7, we know that once $\hat{x}_k = \textsc{Isqrt}(n)$, we will either remain at $\textsc{Isqrt}(n)$ (fixed point) or oscillate between $\textsc{Isqrt}(n)$ and $\textsc{Isqrt}(n) + 1$.

B.4 Error Iteration

B.4.1 Error Iteration 1

We look at the error when

$$\hat{x}_0 = 2^e. \tag{B.13}$$

From the assumption in Eq. (B.2), we have

$$\begin{aligned}\bar{\varepsilon}_0 &\coloneqq \hat{x}_0 - \sqrt{n} \\ &\leq 2^{e-1}.\end{aligned} \tag{B.14}$$

This gives us the bound

$$\begin{aligned} f_n(\hat{x}_0) - \sqrt{n} &= \frac{\bar{\varepsilon}_0^2}{2\hat{x}_0} \\ &\leq 2^{e-3}. \end{aligned} \tag{B.15}$$

Note well: we are using $\hat{x}_0 = 2^e$ in the previous equation to produce an improved error bound.

We continue the process. We define

$$\hat{x}_{k+1} \coloneqq g_n(\hat{x}_k). \tag{B.16}$$

If at each stage we have

$$\hat{x}_k > \sqrt{n}, \tag{B.17}$$

then the iterations result in the following error bounds:

$$\begin{aligned} \bar{\varepsilon}_1 &\coloneqq \hat{x}_1 - \sqrt{n} \\ &\leq 2^{e-3} \\ \bar{\varepsilon}_2 &\coloneqq \hat{x}_2 - \sqrt{n} \\ &\leq 2^{e-6} \\ \bar{\varepsilon}_3 &\coloneqq \hat{x}_3 - \sqrt{n} \\ &\leq 2^{e-12} \\ \bar{\varepsilon}_4 &\coloneqq \hat{x}_4 - \sqrt{n} \\ &\leq 2^{e-24} \\ \bar{\varepsilon}_5 &\coloneqq \hat{x}_5 - \sqrt{n} \\ &\leq 2^{e-48} \\ \bar{\varepsilon}_6 &\coloneqq \hat{x}_6 - \sqrt{n} \\ &\leq 2^{e-96} \\ \bar{\varepsilon}_7 &\coloneqq \hat{x}_7 - \sqrt{n} \\ &\leq 2^{e-192}. \end{aligned} \tag{B.18}$$

In our case, we will be assuming that $e = 128$ is the largest value. This means that

$$\bar{\varepsilon}_7 \leq 2^{-64}. \tag{B.19}$$

In this case, it follows that

$$\begin{aligned} \hat{x}_7 &\leq \sqrt{n} + 2^{-64} \\ &< \left\lfloor \sqrt{n} \right\rfloor + 1 + 2^{-64}. \end{aligned} \tag{B.20}$$

Because we know have $\hat{x}_7 \geq \lfloor \sqrt{n} \rfloor$, we see that

$$\hat{x}_7 \in \left\{ \left\lfloor \sqrt{n} \right\rfloor, \left\lfloor \sqrt{n} \right\rfloor + 1 \right\}. \tag{B.21}$$

B.4.2 Error Iteration 2

We look at the error when

$$\hat{x}_0 = 2^{e-1}. \tag{B.22}$$

We note that

$$\begin{aligned} \bar{\varepsilon}_0 &\coloneqq \hat{x}_0 - \sqrt{n} \\ |\bar{\varepsilon}_0| &\leq 2^{e-1}. \end{aligned} \tag{B.23}$$

Under the same assumptions as before, we have the resulting error bounds:

$$\begin{aligned} \bar{\varepsilon}_1 &\coloneqq \hat{x}_1 - \sqrt{n} \\ &\leq 2^{e-2} \\ \bar{\varepsilon}_2 &\coloneqq \hat{x}_2 - \sqrt{n} \\ &\leq 2^{e-4} \\ \bar{\varepsilon}_3 &\coloneqq \hat{x}_3 - \sqrt{n} \\ &\leq 2^{e-8} \\ \bar{\varepsilon}_4 &\coloneqq \hat{x}_4 - \sqrt{n} \\ &\leq 2^{e-16} \\ \bar{\varepsilon}_5 &\coloneqq \hat{x}_5 - \sqrt{n} \\ &\leq 2^{e-32} \\ \bar{\varepsilon}_6 &\coloneqq \hat{x}_6 - \sqrt{n} \\ &\leq 2^{e-64} \\ \bar{\varepsilon}_7 &\coloneqq \hat{x}_7 - \sqrt{n} \\ &\leq 2^{e-128}. \end{aligned} \tag{B.24}$$

In our case, we will be assuming that $e = 128$ is the largest value. This means that

$$\bar{\varepsilon}_7 \leq 1. \tag{B.25}$$

In this case, it follows that

$$\begin{aligned} \hat{x}_7 &\leq \sqrt{n} + 1 \\ &< \left\lfloor \sqrt{n} \right\rfloor + 2. \end{aligned} \tag{B.26}$$

Because we know have $\hat{x}_7 \geq \lfloor \sqrt{n} \rfloor$, we see that

$$\hat{x}_7 \in \left\{ \left\lfloor \sqrt{n} \right\rfloor, \left\lfloor \sqrt{n} \right\rfloor + 1 \right\}. \tag{B.27}$$

Within the comments of Integer Square Root algorithms, there were suggestions that 7 iterations were sufficient; this shows those comments are correct. This *does not* ensure their final check is valid, though; see App. B.5.

B.4.3 Error Iteration 3

The previous initialization values required 7 Newton iterations. We can do better.

We look at the error when

$$\hat{x}_0 = 2^{e-1} + 2^{e-2}. \tag{B.28}$$

We note that

$$\begin{aligned}\bar{\varepsilon}_0 &\coloneqq \hat{x}_0 - \sqrt{n} \\ |\bar{\varepsilon}_0| &\leq 2^{e-2}.\end{aligned} \tag{B.29}$$

Under the same assumptions as before, we have the resulting error bounds:

$$\begin{aligned}\bar{\varepsilon}_1 &\coloneqq \hat{x}_1 - \sqrt{n} \\ &\leq 2^{e-4.5} \\ \bar{\varepsilon}_2 &\coloneqq \hat{x}_2 - \sqrt{n} \\ &\leq 2^{e-9} \\ \bar{\varepsilon}_3 &\coloneqq \hat{x}_3 - \sqrt{n} \\ &\leq 2^{e-18} \\ \bar{\varepsilon}_4 &\coloneqq \hat{x}_4 - \sqrt{n} \\ &\leq 2^{e-36} \\ \bar{\varepsilon}_5 &\coloneqq \hat{x}_5 - \sqrt{n} \\ &\leq 2^{e-72} \\ \bar{\varepsilon}_6 &\coloneqq \hat{x}_6 - \sqrt{n} \\ &\leq 2^{e-144}.\end{aligned} \tag{B.30}$$

When computing the bound on $\bar{\varepsilon}_1$, we used the exact value of $\hat{x}_0$ to produce a better error approximation.

In our case, we will be assuming that $e = 128$ is the largest value. This means that

$$\bar{\varepsilon}_6 \leq 1. \tag{B.31}$$

In this case, it follows that

$$\begin{aligned}\hat{x}_6 &\leq \sqrt{n} + 1 \\ &< \left\lfloor \sqrt{n} \right\rfloor + 2.\end{aligned} \tag{B.32}$$

Because we know have $\hat{x}_6 \geq \lfloor \sqrt{n} \rfloor$, we see that

$$\hat{x}_6 \in \left\{ \left\lfloor \sqrt{n} \right\rfloor, \left\lfloor \sqrt{n} \right\rfloor + 1 \right\}. \tag{B.33}$$

B.4.4 Further Refinement

It is possible to attempt to compute a better initial approximation. For instance, we could attempt to choose

$$\hat{x}_0 = 2^{e-1} + 2^{e-2} \pm 2^{e-3}, \tag{B.34}$$

where the sign is chosen appropriately; this can ensure that

$$|\bar{\varepsilon}_0| \leq 2^{e-3}. \tag{B.35}$$

Going through the same error analysis will show that this initialization still requires 6 Newton iterations. This means the additional setup cost does not result in an overall cost reduction.

The more refined value of

$$\hat{x}_0 = 2^{e-1} + 2^{e-2} \pm 2^{e-3} \pm 2^{e-4} \tag{B.36}$$

will require only 5 Newton iterations due to the initial error

$$|\bar{\varepsilon}_0| \leq 2^{e-4}. \tag{B.37}$$

With that said, the additional cost of computing the appropriate signs appears to produce a more expensive overall result. Thus, in `Mod-Newton` (Listing 5) we use the initialization from App. B.4.3.

B.5 Final Division and Return

From Eq. (B.27), we set

$$\begin{aligned} r &:= \hat{x}_7 \\ r' &:= \left\lfloor \frac{n}{r} \right\rfloor. \end{aligned} \tag{B.38}$$

From previous work, we can see that

$$\begin{aligned} r &\in \{\textsc{Isqrt}(n), \textsc{Isqrt}(n)+1\} \\ r' &\in \{\textsc{Isqrt}(n)-1, \textsc{Isqrt}(n)\}. \end{aligned} \tag{B.39}$$

With the current analysis, Listings 2, 3, and 4 cannot ensure a valid final result. At the same time, we note that no explicit example has been found to show that Listings 2, 3, and 4 produce an incorrect result.

To ensure a valid final result, we check $r^2 \leq n$ and return r or $r-1$. In Listing 5, we took care of the edge case when $\textsc{Isqrt}(n) = 2^{128}-1$; thus, the check of $r^2 \leq n$ will not cause an overflow.

C Proofs of Python Integer Square Root

We now prove Alg. 1 returns the correct value for integer square root and ensure that no overflow occurs.

Proof of Alg. 1. By assumption, x is an integer with $x \in \{0, 1, \cdots, 2^{256} - 1\}$.

Edge Cases

First, we see that when $x \in \{0, 1\}$, we have $\textsc{Isqrt}(x) = x$, and we return the correct result.

Similarly, when $x \geq (2^{128} - 1)^2$, we have

$$\left(2^{128} - 1\right)^2 \leq x < 2^{256} = \left(2^{128}\right)^2. \tag{C.1}$$

Thus, $\textsc{Isqrt}(x) = 2^{128} - 1$, and we return the correct result.

Standard Case

Convert General x to Specific m

We now focus on the general situation when $2 \leq x < (2^{128} - 1)^2$. The first part here will prove that $2^{254} \leq m < 2^{256}$, where

$$m \coloneqq x \cdot 2^{2e}. \tag{C.2}$$

If $b = \textsc{BitLength}(x)$, then we have

$$2^{b-1} \leq x < 2^b. \tag{C.3}$$

In this case, we know that $b \geq 2$. We set

$$\begin{aligned} e &\coloneqq \left\lfloor \frac{256 - b}{2} \right\rfloor \\ &= \begin{cases} \frac{256-b}{2} & b \text{ even} \\ \frac{255-b}{2} & b \text{ odd} \end{cases}. \end{aligned} \tag{C.4}$$

We separate the two cases:

- b even:

  We see

$$\begin{aligned} m &= x \cdot 2^{2e} \\ &= x \cdot 2^{256-b}. \end{aligned} \tag{C.5}$$

  From here, we see

$$\begin{aligned} m &= x \cdot 2^{256-b} \\ &< 2^b \cdot 2^{256-b} \\ &= 2^{256} \end{aligned} \tag{C.6}$$

and

$$\begin{aligned} m &= x \cdot 2^{256-b} \\ &\geq 2^{b-1} \cdot 2^{256-b} \\ &= 2^{255}. \end{aligned} \tag{C.7}$$

- b odd:

$$\begin{aligned} m &= x \cdot 2^{2e} \\ &= x \cdot 2^{255-b}. \end{aligned} \tag{C.8}$$

From here, we see

$$\begin{aligned} m &= x \cdot 2^{255-b} \\ &< 2^b \cdot 2^{255-b} \\ &= 2^{255} \end{aligned} \tag{C.9}$$

and

$$\begin{aligned} m &= x \cdot 2^{255-b} \\ &\geq 2^{b-1} \cdot 2^{255-b} \\ &= 2^{254}. \end{aligned} \tag{C.10}$$

This shows that we have

$$2^{254} \leq m < 2^{256}. \tag{C.11}$$

Compute Near Square Root of m

The next part of the proof focuses on lifting a sequence of near square roots to a near square root of m. We will repeatedly use Thm. 8 for various values of k, b, and n. We rewrite the previous restriction on m from Eq. (C.11) with a base of 4:

$$4^{127} \leq m < 4^{128}. \tag{C.12}$$

This implies

$$1 \leq \frac{m}{4^{127}} < 4 \tag{C.13}$$

and

$$1 \leq \left\lfloor \frac{m}{4^{127}} \right\rfloor < 4 \tag{C.14}$$

Lift 1 From Eq. (C.14), we see that $b = 1$ is a near square root of

$$\left\lfloor \frac{m}{4^{127}} \right\rfloor = \left\lfloor \frac{\lfloor m/4^{126} \rfloor}{4 \cdot 1^2} \right\rfloor. \tag{C.15}$$

We set $k = 1$ and see

$$4 \cdot 1^4 \leq \left\lfloor \frac{m}{4^{126}} \right\rfloor. \tag{C.16}$$

Using Thm. 8 with $b = 1$, $k = 1$, and $n = \lfloor m/4^{126} \rfloor$, we see that

$$\begin{aligned} a &:= kb + \left\lfloor \frac{n}{4kb} \right\rfloor \\ &:= 1 + \left\lfloor \frac{m}{4^{127}} \right\rfloor \\ &:= 1 + (m \gg 254) \end{aligned} \tag{C.17}$$

is a near square root of $n = \lfloor m/4^{126} \rfloor$.

Lift 2 We see that $b = a$ from Eq. (C.17) is a near square root of

$$\left\lfloor \frac{m}{4^{126}} \right\rfloor = \left\lfloor \frac{\lfloor m/4^{124} \rfloor}{4 \cdot 2^2} \right\rfloor. \tag{C.18}$$

We set $k = 2$ and see

$$4 \cdot 2^4 = 4^3 \leq \left\lfloor \frac{m}{4^{124}} \right\rfloor. \tag{C.19}$$

Using Thm. 8 with b as defined, $k = 2$, and $n = \lfloor m/4^{124} \rfloor$, we see that

$$\begin{aligned} a &:= kb + \left\lfloor \frac{n}{4kb} \right\rfloor \\ &:= 2a + \left\lfloor \frac{m}{4^{125} \cdot 2 \cdot a} \right\rfloor \\ &:= \left(a \ll 1\right) + \left(m \gg 251\right)/a \end{aligned} \tag{C.20}$$

is a near square root of $n = \lfloor m/4^{124} \rfloor$.

Lift 3 We see that $b = a$ from Eq. (C.20) is a near square root of

$$\left\lfloor \frac{m}{4^{124}} \right\rfloor = \left\lfloor \frac{\lfloor m/4^{120} \rfloor}{4 \cdot \left(2^3\right)^2} \right\rfloor. \tag{C.21}$$

We set $k = 2^3$ and see

$$4 \cdot \left(2^3\right)^4 = 4^7 \leq \left\lfloor \frac{m}{4^{120}} \right\rfloor. \tag{C.22}$$

Using Thm. 8 with b as defined, $k = 2^3$, and $n = \lfloor m/4^{120} \rfloor$, we see that

$$\begin{aligned} a &:= kb + \left\lfloor \frac{n}{4kb} \right\rfloor \\ &:= 2^3 a + \left\lfloor \frac{m}{4^{121} \cdot 2^3 \cdot a} \right\rfloor \\ &:= \left(a \ll 3\right) + \left(m \gg 245\right)/a \end{aligned} \tag{C.23}$$

is a near square root of $n = \lfloor m/4^{120} \rfloor$.

Lift 4 We see that $b = a$ from Eq. (C.23) is a near square root of

$$\left\lfloor \frac{m}{4^{120}} \right\rfloor = \left\lfloor \frac{\lfloor m/4^{112} \rfloor}{4 \cdot \left(2^7\right)^2} \right\rfloor. \tag{C.24}$$

We set $k = 2^7$ and see

$$4 \cdot \left(2^7\right)^4 = 4^{15} \leq \left\lfloor \frac{m}{4^{112}} \right\rfloor. \tag{C.25}$$

Using Thm. 8 with b as defined, $k = 2^7$, and $n = \lfloor m/4^{112} \rfloor$, we see that

$$\begin{aligned} a &:= kb + \left\lfloor \frac{n}{4kb} \right\rfloor \\ &:= 2^7 a + \left\lfloor \frac{m}{4^{113} \cdot 2^7 \cdot a} \right\rfloor \\ &:= \left(a \ll 7\right) + \left(m \gg 233\right)/a \end{aligned} \tag{C.26}$$

is a near square root of $n = \lfloor m/4^{112} \rfloor$.

Lift 5 We see that $b = a$ from Eq. (C.26) is a near square root of

$$\left\lfloor \frac{m}{4^{112}} \right\rfloor = \left\lfloor \frac{\lfloor m/4^{96} \rfloor}{4 \cdot \left(2^{15}\right)^2} \right\rfloor . \tag{C.27}$$

We set $k = 2^{15}$ and see

$$4 \cdot \left(2^{15}\right)^4 = 4^{31} \leq \left\lfloor \frac{m}{4^{96}} \right\rfloor . \tag{C.28}$$

Using Thm. 8 with b as defined, $k = 2^{15}$, and $n = \lfloor m/4^{96} \rfloor$, we see that

$$\begin{aligned} a &:= kb + \left\lfloor \frac{n}{4kb} \right\rfloor \\ &:= 2^{15}a + \left\lfloor \frac{m}{4^{97} \cdot 2^{15} \cdot a} \right\rfloor \\ &:= \left(a \ll 15\right) + \left(m \gg 209\right)/a \end{aligned} \tag{C.29}$$

is a near square root of $n = \lfloor m/4^{96} \rfloor$.

Lift 6 We see that $b = a$ from Eq. (C.29) is a near square root of

$$\left\lfloor \frac{m}{4^{96}} \right\rfloor = \left\lfloor \frac{\lfloor m/4^{64} \rfloor}{4 \cdot \left(2^{31}\right)^2} \right\rfloor . \tag{C.30}$$

We set $k = 2^{31}$ and see

$$4 \cdot \left(2^{31}\right)^4 = 4^{63} \leq \left\lfloor \frac{m}{4^{64}} \right\rfloor . \tag{C.31}$$

Using Thm. 8 with b as defined, $k = 2^{31}$, and $n = \lfloor m/4^{64} \rfloor$, we see that

$$\begin{aligned} a &:= kb + \left\lfloor \frac{n}{4kb} \right\rfloor \\ &:= 2^{31}a + \left\lfloor \frac{m}{4^{65} \cdot 2^{31} \cdot a} \right\rfloor \\ &:= \left(a \ll 31\right) + \left(m \gg 161\right)/a \end{aligned} \tag{C.32}$$

is a near square root of $n = \lfloor m/4^{64} \rfloor$.

Lift 7 We see that $b = a$ from Eq. (C.32) is a near square root of

$$\left\lfloor \frac{m}{4^{64}} \right\rfloor = \left\lfloor \frac{m}{4 \cdot \left(2^{63}\right)^2} \right\rfloor . \tag{C.33}$$

We set $k = 2^{63}$ and see

$$4 \cdot \left(2^{63}\right)^4 = 4^{127} \leq m. \tag{C.34}$$

Using Thm. 8 with b as defined, $k = 2^{63}$, and $n = m$, we see that

$$\begin{aligned} a &:= kb + \left\lfloor \frac{n}{4kb} \right\rfloor \\ &:= 2^{63}a + \left\lfloor \frac{m}{4 \cdot 2^{63} \cdot a} \right\rfloor \\ &:= \ (a \ll 63) + (m \gg 65)\,/a \end{aligned} \tag{C.35}$$

is a near square root of m.

Convert Near Square Root of m to Near Square Root of x

At this point, a is a near square root of m; that is, we have

$$(a-1)^2 < m < (a+1)^2\,. \tag{C.36}$$

We recall that $m = 2^{2e}x$. We *want* a near square root of x.

Case 1: $e = 0$ In this case, $m = x$, so a is a near square root of x. Nothing else is required.

Case 2: $e > 0$ In this case, $m = 2^{2e}x$ and a is a near square root of m with $e \geq 1$. That is, we have

$$(a-1)^2 < 2^{2e}x < (a+1)^2\,. \tag{C.37}$$

We set

$$a = q2^e + r \tag{C.38}$$

with $q \geq 0$ and $0 \leq r < 2^e$.

We will show that q is a near square root of x. First, we see

$$2^{2e}(q-1)^2 \leq (2^e q + r - 1)^2 = (a-1)^2 < 2^{2e}x. \tag{C.39}$$

This reduces to

$$(q-1)^2 < x. \tag{C.40}$$

We also see

$$2^{2e}x < (a+1)^2 = (2^e q + r + 1)^2 \leq 2^{2e}(q+1)^2\,. \tag{C.41}$$

It follows that

$$x < (q+1)^2\,. \tag{C.42}$$

This shows that q is a near square root of x.

Convert Near Square Root to Integer Square Root

Now that we have a is near square root of x, we want to compute $\textsc{Isqrt}(x)$.

To check, all we need to do is determine if $a^2 \leq x$. There is the potential that the evaluation a^2 may overflow; this is not possible because we already know $x < (2^{128-1} - 1)^2$. Thus, $\textsc{Isqrt}(x) \leq 2^{128} - 2$, so its near square root is at most $2^{128} - 1$. This is the reason why a special case is required for x close to 2^{256}.

Thus, if $a^2 \leq x$, then $\textsc{Isqrt}(x) = a$; otherwise, $\textsc{Isqrt}(x) = a - 1$. $\square$