Python's integer square root algorithm

Mark Dickinson

May 5, 2024

Abstract

We present a novel adaptive-precision variant of Heron's method for computing integer square roots of arbitrary-precision integers. The algorithm is unusual in that it is almost branch-free, requiring only a single final correction step, and in that the number of required iterations is easily predicted in advance. The algorithm is efficient both at small scales and asymptotically, and represents an attractive compromise between speed and simplicity. Since Python 3.8, the algorithm is used by the CPython implementation of the Python programming language for its standard library integer square root function.

1 Introduction

For a nonnegative integer n, the *integer square root* of n is the unique nonnegative integer a satifying $a^2 \leq n < (a+1)^2$. Equivalently, it's the integer part of the nonnegative real square root of n, or $\lfloor \sqrt{n} \rfloor$. The integer square root is a basic building block of any arbitrary-precision arithmetic toolkit: many number-theoretic and cryptographic algorithms require either the ability to find integer square roots, or the slightly weaker ability to detect whether a given integer is a square, and if so to extract its square root.

The Python programming language has an arbitrary-precision integer type **int**, and since Python 3.8 the Python standard library implements an integer square root operation, math.isqrt. The implementation of this operation in the reference CPython implementation of the Python language was subject to competing concerns. The operation should be fast for small inputs (overwhelmingly the most common use-case) while remaining reasonably efficient even for large inputs. However, the CPython maintainers also value portability and maintainability of the codebase, which argues for not using a complex algorithm where a simpler one would be good enough. This paper describes the algorithm that was used in CPython, which (in the author's opinion) hits the sweet spot between the various concerns, giving an elegant compromise between efficiency and simplicity.

Before introducing the new algorithm, we review a common integer-based approach, Heron's method, in section 2. This forms the basis of the algorithm currently used for Java's BigInteger.isqrt method, for example. Section 2 provides much of the background we need to introduce our algorithm, which is presented in section 3.

2 Heron's method

In this section we describe a well-known approach to computing integer square roots, based on an adaptation of Heron's method to the domain of positive integers.

Heron's method, also known as the Babylonian method, and perhaps more recognisable in recent centuries as a special case of the Newton-Raphson method, is a procedure for approximating the square root of a positive real number t. The idea is that if x is a positive real approximation to $\sqrt{t}$, then $h(x)$, defined by

$$h(x) = \frac{x + t/x}{2},$$

is an improved approximation. Applying h to that improved approximation yields a still better approximation $h(h(x))$, and so on. It can be shown that the sequence of iterates of h on x converges towards $\sqrt{t}$ from any positive initial value x, and that once the sequence gets sufficiently close to $\sqrt{t}$, convergence is quadratic, so that the number of correct decimal places roughly doubles with each iteration.

Example 1. Suppose $t = 2$. Given an approximation $7/5 = 1.4$ to the square root $1.414213562\ldots$ of t, a single iteration of Heron's method produces a new approximation, accurate to 4 decimal places after the point:

$$h(7/5) = \frac{7/5 + 2/(7/5)}{2} = 99/70 = 1.414285714\ldots.$$

Applying a second iteration with $99/70$ as input gives

$$h(h(7/5)) = h(99/70) = 19601/13860 = 1.414213564\ldots,$$

which is accurate to 8 decimal places. A third iteration gives a value accurate to 17 decimal places.

Heron's method can be adapted to the domain of positive integers. For the remainder of this section we assume that n is a fixed positive integer, and we define a function g on the positive integers by

$$g(a) = \left\lfloor \frac{a + \lfloor n/a \rfloor}{2} \right\rfloor.$$

The hope is that, just like its real counterpart h, the function g will transform a poor approximation to the integer square root of n into an improved approximation, and so by applying g repeatedly we eventually reach the integer square root of n. This works, but we have to be a little careful with the details, and in particular the termination condition. The lemmas below make this precise.

Lemma 2. *For any positive integer a, $\lfloor \sqrt{n} \rfloor \leq g(a)$.*

Proof. Expanding and rearranging the inequality $0 \leq (a - \sqrt{n})^2$ gives

$$2a\sqrt{n} \leq a^2 + n.$$

Dividing through by $2a$ gives

$$\sqrt{n} \leq \frac{a + n/a}{2},$$

which can also be seen as the AM-GM inequality applied to a and n/a. Hence

$$\lfloor \sqrt{n} \rfloor \leq \left\lfloor \frac{a + n/a}{2} \right\rfloor = \left\lfloor \frac{\lfloor a + n/a \rfloor}{2} \right\rfloor = \left\lfloor \frac{a + \lfloor n/a \rfloor}{2} \right\rfloor = g(a).$$

□

Lemma 3. *Suppose a is a positive integer satisfying $\lfloor \sqrt{n} \rfloor < a$. Then $g(a) < a$.*

Proof. We have the following chain of equivalences:

$$\begin{aligned}
\lfloor \sqrt{n} \rfloor < a &\iff \sqrt{n} < a \\
&\iff n < a^2 \\
&\iff n/a < a \\
&\iff \lfloor n/a \rfloor < a \\
&\iff a + \lfloor n/a \rfloor < 2a \\
&\iff \frac{a + \lfloor n/a \rfloor}{2} < a \\
&\iff \left\lfloor \frac{a + \lfloor n/a \rfloor}{2} \right\rfloor < a \\
&\iff g(a) < a.
\end{aligned}$$

This completes the proof. □

Now let a be any integer satisfying $\lfloor \sqrt{n} \rfloor \leq a$, and let

$$a_0, a_1, a_2, \ldots$$

be the sequence of iterates of g applied to a; that is, $a_0 = a$ and for all $i \geq 0$, $a_{i+1} = g(a_i)$. We'll show that this sequence must eventially reach $\lfloor \sqrt{n} \rfloor$.

Lemma 4. *For all $i \geq 0$, $\lfloor \sqrt{n} \rfloor \leq a_i$.*

Proof. This follows directly from Lemma 2 for $i \geq 1$, and from our assumption on a for $i = 0$. □

Lemma 5. *For all $i \geq 0$, $\lfloor \sqrt{n} \rfloor = a_i$ if and only if $a_i \leq a_{i+1}$.*

Proof. By Lemma 4, either $\lfloor \sqrt{n} \rfloor < a_i$ or $\lfloor \sqrt{n} \rfloor = a_i$. If $\lfloor \sqrt{n} \rfloor = a_i$ then by Lemma 4 again, $a_i \leq a_{i+1}$. If $\lfloor \sqrt{n} \rfloor < a_i$ then $a_{i+1} < a_i$ by Lemma 3. □

Theorem 6. *There exists an $i \geq 0$ such that a_i is the integer square root of n.*

Proof. By the well-ordering principle, there's an i for which a_i is minimal amongst all elements of the sequence. That minimality implies $a_i \leq a_{i+1}$, and so by Lemma 5, a_i is the integer square root of n. □

Listing 1: Integer square root via Heron's method, version 1

```
def isqrt(n):

    def g(a):
        return (a + n//a) // 2

    a = n
    while True:
        ga = g(a)
        if a <= ga:
            return a
        a = ga
```

Note that while the sequence is guaranteed to encounter $\lfloor\sqrt{n}\rfloor$ eventually, it's *not* necessarily true that the sequence *converges* to $\lfloor\sqrt{n}\rfloor$, in the sense that it's eventually constant. For example, if $n = 15$, the sequence of iterates of g eventually alternates between 3 and 4, and more generally a similar alternation will occur for any n of the form $a^2 - 1$ for some $a \geq 2$.

By the results above, we can find $\lfloor\sqrt{n}\rfloor$ by taking any starting point $a \geq \lfloor\sqrt{n}\rfloor$, and replacing a with $g(a)$ until $a \leq g(a)$. Listing 1 expresses this algorithm in Python, using $a = n$ as the starting point for the iteration. Note that the function isqrt assumes $n > 0$. Extending the code to return 0 for $n = 0$ and to raise an exception for negative n is left as an easy exercise for the reader.

A linguistic note for readers unfamiliar with Python: the // operator performs "floor division": n//a represents $\lfloor n/a\rfloor$. Indeed, from a computational perspective, an expression like $\lfloor n/a\rfloor$ is misleading: it resembles a composition of two operations, while most programming languages or arbitrary-precision integer-arithmetic packages will provide some form of integer division as an integer-to-integer primitive.

The use of n as an initial estimate is inefficient for large n: the initial iterations will roughly halve the estimate each time, and many iterations will be needed to get into the general neighborhood of the square root. It would be better to choose a starting value a that's closer to $\sqrt{n}$. If we can't guarantee that $a \geq \lfloor\sqrt{n}\rfloor$, we can replace a with $g(a)$ before running the main algorithm: by Lemma 2, $g(a)$ is guaranteed to satisfy $g(a) \geq \lfloor\sqrt{n}\rfloor$.

One simple and reasonably efficient possibility for the starting value is to use the smallest power of two exceeding $\lfloor\sqrt{n}\rfloor$. Before giving the code, we make a definition.

Definition 7. For a nonnegative integer n, the *bit length* of n, which we'll write simply as length(n), is the least nonnegative integer e for which $n < 2^e$.

For positive n, the bit length of n is $1+\lfloor\log_2(n)\rfloor$, while the bit length of zero is zero. In Python, the bit length of a nonnegative integer n can be computed as n.bit_length().

Given nonnegative integers n and e, we have the following chain of equiva-

Listing 2: Integer square root via Heron's method, streamlined

```
def isqrt(n):
    a = 1 << (n.bit_length() + 1) // 2
    while True:
        d = n // a
        if a <= d:
            return a
        a = (a + d) // 2
```

lences:

$$\begin{aligned}
\lfloor\sqrt{n}\rfloor < 2^e &\iff \sqrt{n} < 2^e \\
&\iff n < 2^{2e} \\
&\iff \text{length}(n) \le 2e \\
&\iff \text{length}(n) < 2e + 1 \\
&\iff \lfloor(\text{length}(n) - 1)/2\rfloor < e \\
&\iff \lfloor(\text{length}(n) + 1)/2\rfloor \le e
\end{aligned}$$

So taking $e = \lfloor(\text{length}(n) + 1)/2\rfloor$, 2^e is the smallest power of two that strictly exceeds the square root of n. Using that as our starting guess, and taking the opportunity to streamline the previous code a little at the same time, we get the algorithm in Listing 2, in which the termination condition $a \le g(a)$ has been replaced with the equivalent condition $a \le \lfloor n/a\rfloor$.

We chose e to satisfy $\lfloor\sqrt{n}\rfloor < 2^e$. For correctness, we only need to satisfy the weaker condition $\lfloor\sqrt{n}\rfloor \le 2^e$. Given that, we could tighten our initial bound slightly by using $\text{length}(n - 1)$ in place of $\text{length}(n)$. But that costs an extra operation for all inputs n, for a benefit only in the rare case that n is an exact power of four. As such, the change is probably not valuable.

Another simple and easily-computed starting guess is given by $a = \lfloor n/2^e\rfloor$ where $e = \lfloor(\text{length}(n) - 1)/2\rfloor$. We leave it to the reader to verify that this value for a also satisfies $\lfloor\sqrt{n}\rfloor \le a$. There's no obvious reason to prefer either form of starting guess over the other: in particular, neither value is consistently smaller than the other for all n.

The algorithm shown in Listing 2 is well-known and well-used. It has the virtues of simplicity and easy-to-establish correctness, and is reasonably efficient for inputs that aren't too large. Nevertheless, there are at least three areas in which there's room for improvement. First, for large inputs (say a few thousand bits or more), the first few iterations of the algorithm are performing expensive full-precision divisions to obtain only a handful of new correct bits at each iteration. Second, our particular choice of initial guess is somewhat ad hoc, and could probably be improved, possibly at the expense of some additional complexity: in that sense, the algorithm feels incomplete, or at least unsatisfyingly inelegant—it's really the combination of two separate algorithms: one to find an initial guess, and a second to improve that guess until we hit the integer square root. Finally, there's potential inefficiency towards the end of the algorithm, and in particular the fact that the termination condition requires that we perform an extra division *after* we've reached the desired result seems like

an unnecessary inefficiency. To illustrate the last point, consider the following example.

Example 8. Take $n = 16785408$, which is just a little larger than $2^{24} = 16777216$. Our starting guess for the square root is $2^{13} = 8192$. Successive iterations give 5120, 4199, 4098, 4097, and finally 4096, which is the integer square root. Each iteration requires one division, and a final division is needed to establish the termination condition, for a total of six divisions.

So even starting from 4098, just a distance of two away from the true integer square root, three more divisions are required before the correct integer square root can be identified and returned. Note that this example is not typical: it was deliberately chosen to show worst-case behaviour.

The algorithm introduced in the following section addresses all three of these deficiencies, at the expense of only a little extra complexity.

3 Variable-precision Heron's method

In this section we introduce a simple variant of Heron's method that's more efficient than the basic algorithm for large inputs. As in the previous section, we assume that n is a positive integer, and our aim is to compute the integer square root of n.

There are two key ideas. First, we vary the working precision as we go: our algorithm produces at each iteration an integer approximation to the square root of $\lfloor n/4^f \rfloor$ for some integer f, with f decreasing to 0 as the iterations progress. Second, we don't insist on obtaining the exact integer square root of $\lfloor n/4^f \rfloor$ at each iteration (which would require a per-iteration check-and-correct step), but instead allow the error to propagate, and we show that with careful control of the rate at which the precision increases, the error remains bounded throughout the algorithm. We then use a single check-and-correct step at the end of the algorithm.

To describe the algorithm, it's convenient to introduce a new notion: that of a "near square root".

Definition 9. Suppose n is a positive integer. Call a positive integer a a *near square root* of n if $(a-1)^2 < n < (a+1)^2$.

In other words, a is a near square root of n if a is either $\lfloor \sqrt{n} \rfloor$ or $\lceil \sqrt{n} \rceil$. In particular, if $n = a^2$ is a perfect square then a is the *only* near square root of n; otherwise, n has two near square roots.

Given a near square root a of n, the integer square root of n is clearly either a or $a-1$, depending on whether $a^2 \le n$ or $a^2 > n$ (respectively). So an algorithm for computing near square roots provides us with a way to compute integer square roots, and in the remainder of this section we focus on finding near square roots.

Our algorithm is based on the idea of "lifting" a near square root of a quotient of n to a near square root of n. Suppose that j is a positive integer satisfying $j^2 \le n$ and b is a near square root of $\lfloor n/j^2 \rfloor$. Then b is an approximation to $\sqrt{n}/j$, and so jb is an approximation to $\sqrt{n}$. A single iteration of the integer

form of Heron's method applied to jb then gives an improved approximation

$$\left\lfloor \frac{jb + \lfloor n/jb \rfloor}{2} \right\rfloor \sim \sqrt{n}.$$

If j is not too large relative to n, this improved approximation will again be a near square root of n.

Here's a theorem that makes that "not too large" bound precise. For simplicity, we restrict j to be an even integer: write $j = 2k$, then in the above discussion, b is a near square root of $\lfloor n/4k^2 \rfloor$, $2kb$ is an approximation to $\sqrt{n}$, and our improved approximation is $kb + \lfloor n/4kb \rfloor$.

Theorem 10. *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 the positive integer a defined by*

$$a = kb + \left\lfloor \frac{n}{4kb} \right\rfloor$$

is a near square root of n.

Proof. By definition of near square root, we have

$$(b-1)^2 < \left\lfloor \frac{n}{4k^2} \right\rfloor < (b+1)^2. \tag{1}$$

Since $(b+1)^2$ is an integer, we can remove the floor brackets to obtain

$$(b-1)^2 < \frac{n}{4k^2} < (b+1)^2. \tag{2}$$

Multiplying by $4k^2$ throughout (2) and then taking square roots gives

$$2k(b-1) < \sqrt{n} < 2k(b+1) \tag{3}$$

which can be rearranged to the equivalent statement

$$|2kb - \sqrt{n}| < 2k. \tag{4}$$

Squaring and then dividing through by $4kb$ gives

$$0 \leq kb + \frac{n}{4kb} - \sqrt{n} < k/b, \tag{5}$$

which implies that

$$-1 < kb + \left\lfloor \frac{n}{4kb} \right\rfloor - \sqrt{n} < k/b. \tag{6}$$

Substituting the definition of a gives

$$-1 < a - \sqrt{n} < k/b. \tag{7}$$

To complete the proof, we need to know that $k/b \leq 1$. From our (so far unused) assumption that $4k^4 \leq n$ we have $k^2 \leq n/4k^2$, while from the right-hand side of (2) we have $n/4k^2 < (b+1)^2$. Combining these and taking square roots, $k < b+1$, hence $k \leq b$ and $k/b \leq 1$. So now combining this with (7) gives

$$-1 < a - \sqrt{n} < 1 \tag{8}$$

from which $(a-1)^2 < n < (a+1)^2$, so a is a near square of n, as required. □

Listing 3: Adaptive-precision Heron's method, recursive

```
def nsqrt(n):
    if n < 4:
        return 1
    else:
        e = (n.bit_length()-3) // 4
        a = nsqrt(n >> 2*e+2)
        return (a << e) + (n >> e+2) // a


def isqrt(n):
    a = nsqrt(n)
    return a if a*a <= n else a - 1
```

Example 11. Let $n = 46696$ and $k = 10$. Then $\lfloor n/k^2 \rfloor = 116$, so 10 and 11 are both near square roots of $\lfloor n/k^2 \rfloor$.

Taking $b = 10$, we get $a = 100 + \lfloor 46696/400 \rfloor = 216$. Since $\sqrt{46696} = 216.092572755\ldots$, 216 is indeed a near square root of n. If we take $b = 11$, we also get $a = 110 + \lfloor 46696/440 \rfloor = 216$.

Now we turn to implementation. Like many arbitrary-precision integer implementations, Python's integer implementation is binary-based, so multiplications and floor divisions by powers of two can be performed efficiently by bit-shifting. So in applying Theorem 10, we'd like to choose k to be a power of two satisfying $4k^4 \le n$, and in order for the algorithm to proceed as fast as possible we'd like k to be the largest such power. Writing $k = 2^e$, we have:

$$\begin{aligned} 4k^4 \le n &\iff 2^{2+4e} \le n \\ &\iff 2 + 4e < \operatorname{length} n \\ &\iff 3 + 4e \le \operatorname{length} n \\ &\iff e \le \left\lfloor \frac{\operatorname{length} n - 3}{4} \right\rfloor \end{aligned}$$

So we take $k = 2^e$, where $e = \lfloor (\operatorname{length} n - 3)/4 \rfloor$. This gives the recursive near square root implementation shown in Listing 3, where the multiplication by k and the divisions by $4k$ and $4k^2$ are replaced by the corresponding bit-shift operations.

Each step of the algorithm shown in Listing 3 involves three big-integer shifts, one big-integer addition, one bit-length computation, and one big-integer division, along with a handful of operations that only involve small integers. We can improve this slightly: one of the two right-shifts can be eliminated, by keeping track of the amount by which n should be shifted in the recursive call instead of actually shifting. With a little more bookkeeping, we can also replace the per-iteration bit-length computation with a single initial bit-length computation. These changes represent minor efficiency improvements at the expense of some small loss of clarity; we leave it to the interested reader to pursue them further.

Listing 4: Adaptive-precision Heron's method, iterative

```
def isqrt(n):
    c = (n.bit_length() - 1) // 2
    s = c.bit_length()
    d = 0
    a = 1
    while s > 0:
        e = d
        s = s - 1
        d = c >> s
        a = (a << d-e-1) + (n >> 2*c-d-e+1) // a
    return a if a*a <= n else a - 1
```

The math.isqrt implementation in CPython 3.8 doesn't use the recursive version shown in Listing 3. Instead, we unwind the recursion to obtain an iterative version of the algorithm. This version is presented in Listing 4. It may not be immediately obvious that Listing 4 is equivalent to Listing 3. Rather than describing the equivalence and establishing the correctness of the iterative version via that equivalence, we give a direct proof of correctness for the iterative version.

We establish two loop invariants on the variables s, d and a. These invariants hold just before entering the **while** loop and at the end of every iteration of that loop, and hence also the beginning of any subsequent iteration. The first loop invariant is $d = \lfloor c/2^s \rfloor$; this should be clear from examining the code. The second loop invariant states that at every step, a is a near square root of $\lfloor n/4^{c-d} \rfloor$. In particular, on exit from the **while** loop, $s = 0$, $d = \lfloor c/2^0 \rfloor = c$ and so a is a near square root of $\lfloor n/4^{c-c} \rfloor = n$.

To establish the second invariant, note first that the invariant holds on the first entry to the **while** loop: we have $c = \lfloor \log_4 n \rfloor$, so $4^c \le n < 4^{c+1}$, $1 \le \lfloor n/4^c \rfloor < 4$, and $a = 1$ is a near square root of $\lfloor n/4^c \rfloor$. We then need to establish that if the invariant holds at the beginning of any **while** loop iteration, it also applies at the end of that iteration. We prove this via the following lemma, in which b and e represent the values of a and d at the start of the iteration.

Lemma 12. *Suppose that* $0 \le s < \text{length}(c)$*, that* $e = \lfloor c/2^{s+1} \rfloor$*, that* $d = \lfloor c/2^s \rfloor$*, and that* b *is a near square root of* $\lfloor n/4^{c-e} \rfloor$*. Let*

$$a = 2^{d-e-1}b + \left\lfloor \frac{n}{2^{2c-d-e+1}b} \right\rfloor.$$

Then a *is a near square root of* $\lfloor n/4^{c-d} \rfloor$*.*

Proof. Let $m = \lfloor n/4^{c-d} \rfloor$. Then b is a near square root of $\lfloor m/4^{d-e} \rfloor$ and we can rewrite a as

$$a = 2^{d-e-1}b + \left\lfloor \frac{m}{4 \cdot 2^{d-e-1}b} \right\rfloor.$$

Now we can apply the main theorem: if we can show that 2^{d-e-1} is an integer and that $4(2^{d-e-1})^4 \le m$, it follows from Theorem 10 that a is a near square

root of m. But from the definitions of d and e, $1 \le d$ and $e = \lfloor d/2 \rfloor$, so it follows that $0 \le d - e - 1 \le e$, and

$$\begin{aligned} 4(2^{d-e-1})^4 &= 4(4^{d-e-1})^2 \\ &= 4 \cdot 4^{d-e-1} \cdot 4^{d-e-1} \\ &\le 4 \cdot 4^{d-e-1} \cdot 4^e \\ &= 4^d \end{aligned}$$

Furthermore, since $c = \lfloor \log_4 n \rfloor$, $4^c \le n$, so $4^d \le n/4^{c-d}$, hence $4^d \le m$. Combining this with the inequality above, $4(2^{d-e-1})^4 \le m$, as required. □

The quantities c, d, e, and s are all small (machine-size) integers; only a and n are big integers. So the algorithm consists of two big-integer shifts, one big-integer addition and one big-integer division per iteration, along with a handful of operations with small integers.

The total number of iterations m of the while loop is remarkably simple: it's exactly $\lfloor \log_2 \lfloor \log_2 n \rfloor \rfloor$, assuming $n \ge 2$ (and zero iterations are required for $n = 1$). So for example, input values n satisfying $2^{32} \le n < 2^{64}$ require exactly five iterations, while values in the range $2^{64} \le n < 2^{128}$ require exactly six.

4 Near square roots and square detection

Occasionally all that's required is the ability to detect whether a given integer n is a perfect square. Typical efficient square-detection algorithms perform a series of cheap tests on the value n to eliminate classes of non-square integers - for example, by looking at the residue of n modulo 256 or 255. If n passes those tests, the final step is to compute the integer square root a of n and check whether $a^2 = n$. For that final step it's sufficient to compute a near square root rather than an integer square root: given an algorithm nsqrt that computes near square roots of positive integers, a positive integer n is a square if and only if n is equal to the square of nsqrt(n). This means that for the purposes of square detection, we can discard the final correction step from the algorithm shown in Listing 3. This is shown in Listing 5.

Similarly, if we somehow already know that an integer n is a perfect square and we want to compute its square root, we can omit the correction step in Listing 3: the nsqrt function will compute the square root directly.

5 Complexity analysis

It is straightforward to express the bit complexity of the algorithm presented in section 3 in terms of the bit complexities of multiplication and division, under some reasonable assumptions on those operations and on the representation of arbitrary-precision integers in use. In short, the complexity is dominated by a single operation on arbitrary-precision integers, namely whichever is the slower of (a) the division in the final iteration of the while loop, or (b) the multiplication in the correction step.

In this section, we'll analyse the complexity of the iterative algorithm presented in Listing 4. We write b for the bit length of n, and express the bit

Listing 5: Perfect square detection using nsqrt

```
def nsqrt(n):
    if n < 4:
        return 1
    else:
        e = (n.bit_length()-3) // 4
        a = nsqrt(n >> 2*e+2)
        return (a << e) + (n >> e+2) // a


def is_square(n):
    if n < 0:
        return False
    a = nsqrt(n)
    return a * a == n
```

complexity in terms of b. We start by identifying and eliminating operations that don't contribute significantly to the overall complexity.

The bit_length operation used in the first line of the function body has a complexity of $O(b)$, depending on the exact implementation and the in-memory representation of arbitrary-precision integers being used. In practice, assuming an integer representation based on some power of 2 with a directly encoded size field of some form, it's reasonable to expect it to be $O(1)$. (This assumption applies to both CPython and GMP.)

The integer quantities c, d, e in the listing are bounded by b, while s is smaller still, bounded by $1+\log_2(b)$. In the various computations involving these quantities, for example the computation of $2c - d - e + 1$ in the penultimate line of code, we use only linear-complexity operations: addition, subtraction, doubling and bit shifting. So each arithmetic operation on quantities of this size has bit complexity $O(\log(b))$. Recall from section 3 that the body of the while loop is executed exactly $\max(\lfloor \log_2(\log_2(n)) \rfloor, 0)$ times. As a result, the total bit complexity of the operations involving only c, d, e and s is $O(\log(b)^2)$, which will give negligible contribution to the bit complexity of the complete algorithm.

It remains to analyse the contribution to bit complexity from the arithmetic operations on integers of magnitude comparable to n. Each execution of the body of the **while**-loop involves four such operations:

- the left shift of a by $d - e - 1$;
- the right shift of n by $2c - d - e + 1$;
- the division of the right shift result by a;
- the addition of the division result to the left shift result.

In addition, the correction step in the last line of code involves a squaring operation, a comparison, and possibly a decrement operation.

For the analysis of the **while** loop, we examine the state just before executing the final line in the body of the loop. At that point, the bit lengths of the various quantities involved are as follows:

- the bit length of n is at most $2c + 2$;
- the bit length of a is at most $e + 2$;
- the bit length of the left-shift result is also at most $d + 1$;
- the bit length of the right-shift result is at most $d + e + 1$;
- the bit length of the quotient is at most $d + 1$;
- the bit length of the sum (the new value for a) is at most $d + 2$.

For the two shift operations and the addition, the complexity is linear in the number of bits involved. Note in particular that, at least with CPython's implementation of arbitrary precision integers, the complexity of the right shift operation grows with the number of bits in the *result* of the shift, not the number of bits of n. So the complexity of all three operations together is $O(d + e)$, which is $O(b/2^s)$. Summing over all s, we get a total contribution of $O(b)$ for the execution of the while loop excluding the division.

The division is a division of a $d + e + 1$-bit quantity by an $e + 2$-bit quantity with a $d + 1$-bit result, or roughly, a division of an approximately $3b/2^{s+2}$-bit quantity by an approximately $b/2^{s+2}$ quantity. So the sizes involved roughly double on each iteration. We can reasonably assume that the complexity of the division operation grows faster than linearly, so that the total complexity from the divisions is dominated by the final division.

For the correction step, only the squaring operation has non-linear complexity, and the other operations are negligible in comparison. So overall, the complexity is dominated by either the final division operation, or the squaring operation in the correction step.

For CPython specifically, for Python versions prior to Python 3.12, multiplication uses the Karatsuba algorithm for large integer sizes, so has complexity $O(b^{1.585})$ (where the constant 1.585 is an upper bound for $\log(3)/\log(2) = 1.5849625...$), while division utilises a straightforward approach with quadratic complexity. So the math.isqrt function has bit complexity $O(b^2)$.

For CPython 3.12 and later, division uses a well-known divide-and-conquer algorithm due to Burnikel and Ziegler, with complexity $O(b^{1.585})$, and as a result the math.isqrt function also has bit complexity $O(b^{1.585})$.

6 Fixed-precision algorithms

The iterative algorithm shown in Listing 4 specialises easily to particular fixed-precision cases, giving an almost branch-free algorithm. For example, if $2^{30} \leq n < 2^{32}$, then $c = 15$, $s = 4$, and we can compute in advance the set of d and e values for each of the four iterations. For smaller positive n, we can find an f such that $2^{30} \leq 4^f n < 2^{32}$, apply the same algorithm to $4^f n$, then shift the resulting near square root right by f bits. This gives the algorithm shown in Listing 6. The corresponding 64-bit version is shown in Listing 7. While the

Listing 6: Fixed-precision variant, valid for $0 \leq n < 2^{32}$

```
def isqrt(n):
    e = (32 - n.bit_length()) // 2
    m = n << 2*e
    a = 1 + (m >> 30)
    a = (a << 1) + (m >> 27) // a
    a = (a << 3) + (m >> 21) // a
    a = (a << 7) + (m >> 9) // a
    a = a >> e
    return a if a*a <= n else a - 1
```

Listing 7: Fixed-precision variant, valid for $0 \leq n < 2^{64}$

```
def isqrt(n):
    e = (64 - n.bit_length()) // 2
    m = n << 2*e
    a = 1 + (m >> 62)
    a = (a << 1) + (m >> 59) // a
    a = (a << 3) + (m >> 53) // a
    a = (a << 7) + (m >> 41) // a
    a = (a << 15) + (m >> 17) // a
    a = a >> e
    return a if a*a <= n else a - 1
```

original iterative algorithm was developed for positive n, these fixed-precision variants also turn out to give the correct answer in the case $n = 0$.

For practical implementation, we can replace the initial steps of the algorithm with a lookup table. Listing 8 gives an implementation of a 32-bit integer square root in standard C. It makes use of a lookup table, and requires just a single division. As written, that division is of a 32-bit dividend by a 32-bit divisor, but both the divisor and the quotient fit into a 16-bit unsigned integer. The code also assumes the existence of a 32-bit "count leading zeros" function, clz32.

The rough shape of the code is straightforward: we first deal with the special case where x is zero. Then we normalise x, shifting left by an even number of bits to ensure that $2^{30} \le x < 2^{32}$. Then we compute the integer square root of our new x, and finally shift back right to compensate for the original shift left. In the center of the function, for $2^{30} \le x < 2^{32}$, the line with the table lookup provides a near square root for $\lfloor x/2^{16} \rfloor$. The following line lifts that to a near square root y for x, and then the line after that corrects y downwards if necessary.

There are a couple of subtleties to note. Immediately before the correction step, from the bounds on x and the fact that y is a near square root of x, we know that $2^{15} \le y \le 2^{16}$. If $y = 2^{16}$ then it won't fit into a 16-bit unsigned integer, so we need to exclude this possibility. But if $y = 2^{16}$, then since y is a near square root of x, we must have $x > (2^{16} - 1)^2$. It then follows that in the previous line we end up using the very last entry in the lookup table to yield $y = 256$, and from there that after the lifting $y = 2^{16} - 1$, a contradiction. So $y = 2^{16}$ is impossible at this stage, and y will always fit into a 16-bit unsigned integer.

To make the lookup table work, the key fact is that given a 16-bit integer t satisfying $2^{14} \le t < 2^{16}$, we can compute a near square root of t using only the most significant 8 bits of t. More generally, we have the following lemma, which shows that we also have a small amount of freedom in choosing the lookup table entries to be used.

Lemma 13. *Suppose that n and k are positive integers satisfying $(k+1)^2 \le 4n$, and let $m = \lfloor n/k \rfloor$. Then $a := \lceil \sqrt{2k(m+1)} \rceil - 1$ and $b := \lceil \sqrt{2km} \rceil$ are both near square roots of n.*

Proof. We have:

$$\lceil \sqrt{2km} \rceil - 1 < \sqrt{2km} \le \sqrt{n} < \sqrt{2k(m+1)} \le \lceil \sqrt{2k(m+1)} \rceil.$$

Or in other words,

$$b - 1 < \sqrt{n} < a + 1.$$

So

$$(b-1)^2 < n < (a+1)^2.$$

All that remains is to show that $a \le b$, and then we have

$$(a-1)^2 \le (b-1)^2 < n < (a+1)^2 \le (b+1)^2,$$

showing that both a and b are near square roots of n.

But from our assumption that $k^2 \le n$ we have $k/2 < n/2k$

□

Listing 8: 32-bit integer square root in C

```c
#include <stdint.h>

// count leading zeros of nonzero 32-bit unsigned integer
int clz32(uint32_t x);

// isqrt32_tab[k] = isqrt(256*(k+64)-1) for 0 <= k < 192
static const uint8_t isqrt32_tab[192] = {
    127, 128, 129, 130, 131, 132, 133, 134, 135, 136,
    137, 138, 139, 140, 141, 142, 143, 143, 144, 145,
    146, 147, 148, 149, 150, 150, 151, 152, 153, 154,
    155, 155, 156, 157, 158, 159, 159, 160, 161, 162,
    163, 163, 164, 165, 166, 167, 167, 168, 169, 170,
    170, 171, 172, 173, 173, 174, 175, 175, 176, 177,
    178, 178, 179, 180, 181, 181, 182, 183, 183, 184,
    185, 185, 186, 187, 187, 188, 189, 189, 190, 191,
    191, 192, 193, 193, 194, 195, 195, 196, 197, 197,
    198, 199, 199, 200, 201, 201, 202, 203, 203, 204,
    204, 205, 206, 206, 207, 207, 208, 209, 209, 210,
    211, 211, 212, 212, 213, 214, 214, 215, 215, 216,
    217, 217, 218, 218, 219, 219, 220, 221, 221, 222,
    222, 223, 223, 224, 225, 225, 226, 226, 227, 227,
    228, 229, 229, 230, 230, 231, 231, 232, 232, 233,
    234, 234, 235, 235, 236, 236, 237, 237, 238, 238,
    239, 239, 240, 241, 241, 242, 242, 243, 243, 244,
    244, 245, 245, 246, 246, 247, 247, 248, 248, 249,
    249, 250, 250, 251, 251, 252, 252, 253, 253, 254,
    254, 255,
};

// integer square root of 32-bit unsigned integer
uint16_t isqrt32(uint32_t x)
{
    if (x == 0) return 0;

    int lz = clz32(x) & 30;
    x <<= lz;
    uint16_t y = 1 + isqrt32_tab[(x >> 24) - 64];
    y = (y << 7) + (x >> 9) / y;
    y -= x < (uint32_t)y * y;
    return y >> (lz >> 1);
}
```

Now put $k = 128$ in the above to get:

Lemma 14. *Suppose that $2^{14} \le t < 2^{16}$, and let $s = \lfloor t/256 \rfloor$. Then $a \coloneqq \lceil \sqrt{256(s+1)} \rceil - 1$ and $b \coloneqq \lceil \sqrt{256s} \rceil$ are both near square roots of t.*

Proof. We have $\sqrt{256s} \le \sqrt{t} < \sqrt{256(s+1)}$. So

$$\lceil \sqrt{256s} \rceil - 1 < \sqrt{256s} \le \sqrt{t} < \sqrt{256(s+1)} \le \lceil \sqrt{256(s+1)} \rceil.$$

Or in other words,

$$b - 1 < \sqrt{t} < a + 1,$$

so

$$(b-1)^2 < t < (a+1)^2.$$

All that remains is to show that $a \le b$, and then we have

$$(a-1)^2 \le (b-1)^2 < t < (a+1)^2 \le (b+1)^2,$$

showing that both a and b are near square roots of t. But from our assumption that $2^{14} \le t$, we have $64 \le s$, and it follows that $128 \le \sqrt{256s} \le \sqrt{256(s+1)}$ and hence

$$256 \le \sqrt{256(s+1)} + \sqrt{256s}$$

Multiplying both sides by $(\sqrt{256(s+1)} - \sqrt{256s})/256$ gives:

$$\sqrt{256(s+1)} - \sqrt{256s} \le 1.$$

Now rearranging and taking ceilings gives

$$a = \lceil \sqrt{256(s+1)} \rceil - 1 \le \lceil \sqrt{256s} \rceil = b$$

as desired. □

In Listing 8, the lookup table value $T(k)$ at position k is given by $T(k) = \lfloor \sqrt{256(k+64) - 1} \rfloor$. So by the above lemma applied to $t = \lfloor n/2^{16} \rfloor$,

$$1 + T(\lfloor n/2^{24} \rfloor - 64) = \lceil \sqrt{256 \lfloor n/2^{24} \rfloor} \rceil$$

gives a near square root of $\lfloor n/2^{16} \rfloor$. It would have been more natural to absorb the addition of 1 into the table values, but that would have meant that the final table value would no longer fit in an 8-bit integer, which in turn would require the whole table to use 16-bit integers, wasting significant space.

Finally, a 64-bit integer square root algorithm is shown in Listing 9. The same subtleties apply as with the 32-bit integer square root: for values of x larger than $(2^{32} - 1)^2$, both 255 and 256 are near square roots of $\lfloor x/2^{56} \rfloor$. However, a result of 256 from the table lookup would lead to y overflowing a 32-bit unsigned integer, while with a value of 255, y remains representable throughout the algorithm. This is the opposite situation to the 32-bit case, where we needed the table lookup to produce 256 rather than 255 for x near the upper limits of the input range.

Listing 9: 64-bit integer square root in C

```c
#include <stdint.h>

// count leading zeros of nonzero 64-bit unsigned integer
int clz64(uint64_t x);

// isqrt64_tab[k] = isqrt(256*(k+65)-1) for 0 <= k < 192
static const uint8_t isqrt64_tab[192] = {
    128, 129, 130, 131, 132, 133, 134, 135, 136, 137,
    138, 139, 140, 141, 142, 143, 143, 144, 145, 146,
    147, 148, 149, 150, 150, 151, 152, 153, 154, 155,
    155, 156, 157, 158, 159, 159, 160, 161, 162, 163,
    163, 164, 165, 166, 167, 167, 168, 169, 170, 170,
    171, 172, 173, 173, 174, 175, 175, 176, 177, 178,
    178, 179, 180, 181, 181, 182, 183, 183, 184, 185,
    185, 186, 187, 187, 188, 189, 189, 190, 191, 191,
    192, 193, 193, 194, 195, 195, 196, 197, 197, 198,
    199, 199, 200, 201, 201, 202, 203, 203, 204, 204,
    205, 206, 206, 207, 207, 208, 209, 209, 210, 211,
    211, 212, 212, 213, 214, 214, 215, 215, 216, 217,
    217, 218, 218, 219, 219, 220, 221, 221, 222, 222,
    223, 223, 224, 225, 225, 226, 226, 227, 227, 228,
    229, 229, 230, 230, 231, 231, 232, 232, 233, 234,
    234, 235, 235, 236, 236, 237, 237, 238, 238, 239,
    239, 240, 241, 241, 242, 242, 243, 243, 244, 244,
    245, 245, 246, 246, 247, 247, 248, 248, 249, 249,
    250, 250, 251, 251, 252, 252, 253, 253, 254, 254,
    255, 255,
};

// integer square root of a 64-bit unsigned integer
uint32_t isqrt64(uint64_t x)
{
    if (x == 0) return 0;

    int lz = clz64(x) & 62;
    x <<= lz;
    uint32_t y = isqrt64_tab[(x >> 56) - 64];
    y = (y << 7) + (x >> 41) / y;
    y = (y << 15) + (x >> 17) / y;
    y -= x < (uint64_t)y * y;
    return y >> (lz >> 1);
}
```

7 Using floating-point

Floating-point methods can be used either to compute integer square roots directly for small n, or to accelerate the computation of an integer square root for larger n by providing an accurate starting guess for an iterative algorithm. In this section we give some details. Note however that CPython's implementation of math.isqrt does not make any use of floating-point.

Neither the Python language nor (prior to Python 3.12) the CPython reference implementation of that language, provides a guarantee of either IEEE 754 format floating-point or correct rounding of floating-point arithmetic operations. As such, any floating-point-based method for direct computation of an integer square root would need a correctness check along with a pure-integer fallback method for the case where the floating-point square root operation fails to provide sufficient accuracy. For this reason, use of floating-point was avoided in the implementation of math.isqrt.

Despite the lack of language guarantees, use of IEEE 754 floating-point is almost ubiquitous on platforms that support Python, and on almost all systems currently in use Python's **float** type maps to the IEEE 754 "double precision" binary64 interchange format, defined in section 3.6 of IEEE 754-2019. Moreover, floating-point square root operations implemented in hardware are invariably correctly rounded. From this point onwards, we assume that Python's **float** type *does* use the binary64 format, and that basic arithmetic operations and the standard library math.sqrt function are all correctly rounded, using the default IEEE 754 "roundTiesToEven" rounding mode.

First some definitions: given a real number x, write $\text{float}(x)$ for the nearest (under "roundTiesToEven") binary64 floating-point number to x. If x has magnitude $2^{1024} - 2^{970}$ or greater, $\text{float}(x)$ is the appropriately-signed infinity. If x is already a finite floating-point number, we implicitly regard it as a real number when necessary. Given a finite nonnegative binary64 floating-point number x, write $\text{fsqrt}(x)$ for the correctly-rounded square root of x, and $\text{fsqr}(x)$ for the correctly-rounded square of x. In other words, $\text{fsqrt}(x) = \text{float}(\sqrt{x})$ and $\text{fsqr}(x) = \text{float}(x^2)$.

We first prove that if n is not too large then $\lfloor \text{fsqrt}(n) \rfloor$ gives the integer square root of n.

Lemma 15. *Suppose that n is a nonnegative integer satisfying $n < 2^{52}$. Then $\lfloor \text{fsqrt}(n) \rfloor$ is the integer square root of n.*

Proof. The lemma is clearly true for $n = 0$, so for the remainder of the proof we assume that n is positive.

Write a for the integer square root of n. Then $a^2 \le n < (a+1)^2$, so $a \le \sqrt{n} < a + 1$, and since both a and $a + 1$ are small enough to be exactly representable as floats, and the float operation is monotonic, it follows that

$$a = \text{float}(a) \le \text{float}(\sqrt{n}) \le \text{float}(a+1) = a + 1.$$

So

$$a \le \text{fsqrt}(n) \le a + 1,$$

and to prove the lemma, it suffices to eliminate the possibility that $\text{float}(\sqrt{n}) = a + 1$.

Since n is positive, so is a, and we can find an integer k such that $2^k \le a < a+1 \le 2^{k+1}$. Any two consecutive floating-point numbers in the interval $[2^k, 2^{k+1}]$ are separated by exactly 2^{k-52}, so the largest floating-point number that's strictly smaller than $a+1$ is $a+1-2^{k-52}$. To avoid the possibility that $\sqrt{n}$ rounds up to $a+1$, it's enough to show that $\sqrt{n}$ is closer to $a+1-2^{k-52}$ than to $a+1$; that is, that

$$\sqrt{n} < a + 1 - 2^{k-53}$$

or equivalently that

$$a + 1 - \sqrt{n} > 2^{k-53}.$$

But we have

$$a + 1 - \sqrt{n} = \frac{(a+1)^2 - n}{a + 1 + \sqrt{n}}.$$

The numerator on the right-hand side above is at least 1, and since $\sqrt{n} < a+1$, the denominator is less than $2(a+1)$, which is in turn less than or equal to 2^{k+2}. So

$$a + 1 - \sqrt{n} > 2^{-k-2}.$$

So we only need to show that $2^{-k-2} \ge 2^{k-53}$. But from our assumptions, $a^2 \le n < 2^{52}$, so $a < 2^{26}$, so k is at most 25 and if follows that $2^{-k-2} \ge 2^{-27} > 2^{-28} \ge 2^{k-53}$, as required. □

The upper bound can be extended slightly:

Corollary 16. *Suppose that n is a nonnegative integer satisfying $n < 2^{52} + 2^{27}$. Then $\lfloor \mathrm{fsqrt}(n) \rfloor$ is the integer square root of n. However, for $n = 2^{52} + 2^{27} = (2^{26}+1)^2 - 1$, $\lfloor \mathrm{fsqrt}(n) \rfloor$ gives $2^{26}+1$, while the integer square root of n is 2^{26}.*

Proof. We already proved the statement for $n < 2^{52}$. For any n in the range $2^{52} \le n < 2^{52} + 2^{27}$ the correct integer square root is 2^{26} and it's enough to observe that the operation $\lfloor \mathrm{fsqrt}(n) \rfloor$ is monotonic in n and that it gives the correct answer for both 2^{52} and for $2^{52} + 2^{27}$. The last statement follows by direct computation. □

Next we show that if all we need is a near square root of n (as defined in section 3), then we can use a much larger upper bound on n. First, a lemma leading to a well-known theorem: that the floating-point square root operation recovers a float x from its (floating-point) square.

Lemma 17. *Suppose that x and y are positive real numbers such that $x = 2^e y$ for some integer e, and that both x and y lie in the half-open interval $[2^{-1022} - 2^{-1076}, 2^{1024} - 2^{970})$. Then $\mathrm{float}(x) = 2^e \,\mathrm{float}(y)$.*

Proof. The bounds on x and y ensure that $\mathrm{float}(x) = 2^f \,\mathrm{rint}(x/2^f)$, where $f = \lfloor \log_2 x \rfloor - 52$, and similarly for y. Here $\mathrm{rint}(x)$ is the operation that rounds a real number to the nearest integer, rounding halfway cases to the even integer. The result follows directly from this. □

The following fact is well known, but we state and prove it here for convenience.

Theorem 18. *Suppose x is an IEEE 754 binary64 floating-point number satisfying $2^{-511} \le x < 2^{512}$. Then* $\mathrm{fsqrt}(\mathrm{fsqr}(x)) = x$.

Proof. We first use the previous lemma to reduce to the case where $1/2 < x \le 1$. Choose an integer e and floating-point number y such that $1/2 < y \le 1$ and

$$x = 2^e y.$$

Then

$$x^2 = 2^{2e} y^2$$

so applying Lemma 17,

$$\mathrm{float}(x^2) = 2^{2e}\,\mathrm{float}(y^2).$$

By definition of fsqr, this can be rewritten as

$$\mathrm{fsqr}(x) = 2^{2e}\,\mathrm{fsqr}(y).$$

Now taking square roots of both sides,

$$\sqrt{\mathrm{fsqr}(x)} = 2^e \sqrt{\mathrm{fsqr}(y)}.$$

Applying Lemma 17 again,

$$\mathrm{float}(\sqrt{\mathrm{fsqr}(x)}) = 2^e\,\mathrm{float}(\sqrt{\mathrm{fsqr}(y)}),$$

or in other words

$$\mathrm{fsqrt}(\mathrm{fsqr}(x)) = 2^e\,\mathrm{fsqrt}(\mathrm{fsqr}(y)).$$

So to prove that $\mathrm{fsqrt}(\mathrm{fsqr}(x)) = x$, it's enough to prove that $\mathrm{fsqrt}(\mathrm{fsqr}(y)) = y$, since we then have

$$\mathrm{fsqrt}(\mathrm{fsqr}(x)) = 2^e\,\mathrm{fsqrt}(\mathrm{fsqr}(y)) = 2^e y = x.$$

So from this point on, we assume that $1/2 < x \le 1$.

Let $z = \mathrm{fsqr}(x)$ be the closest float to x^2. Then $1/4 \le z \le 1$ and since floats are spaced at most 2^{-53} apart within the interval $[1/4, 1]$,

$$|z - x^2| \le 2^{-54}.$$

Now we have

$$\sqrt{z} - x = \frac{(\sqrt{z} - x)(\sqrt{z} + x)}{\sqrt{z} + x} = \frac{z - x^2}{\sqrt{z} + x}$$

and since $1/2 < x$ and $1/2 \le \sqrt{z}$, $\sqrt{z} + x > 1$. It follows that

$$|\sqrt{z} - x| < 2^{-54}$$

and so since floats in the interval $[1/2, 1]$ are spaced exactly 2^{-53} apart, it follows that x is the unique closest float to $\sqrt{z}$, so $x = \mathrm{fsqrt}(z)$ as required. □

The next corollary follows immediately from Theorem 18, together with the fact that any nonnegative integer not exceeding 2^{53} can be exactly represented in binary64 floating-point.

Listing 10: Integer square root using floating-point quick start

```
import math


def isqrt(n):
    c = (n.bit_length() - 1) // 2
    s = (c//53).bit_length()
    d = c >> s
    a = int(math.sqrt(n >> 2*c-2*d))
    while s > 0:
        e = d
        s = s - 1
        d = c >> s
        a = (a << d-e-1) + (n >> 2*c-d-e+1) // a
    return a if a*a <= n else a - 1
```

Corollary 19. *Suppose n is a nonnegative integer satisfying $n \leq 2^{53}$. Then* $\mathrm{fsqrt}(\mathrm{fsqr}(n)) = n$.

Now we're in a position to prove that for sufficiently small n, the floating-point square root can be used to compute a near square root of n.

Corollary 20. *Suppose n is a positive integer satisfying $n \leq 2^{106}$. Then* $\lfloor \mathrm{fsqrt}(\mathrm{rnd}(n)) \rfloor$ *is a near square root of n.*

Proof. First suppose that n is a perfect square: $n = a^2$. Then $\mathrm{rnd}(n) = \mathrm{fsqr}(a)$, so from the previous corollary, $\mathrm{fsqrt}(\mathrm{rnd}(n)) = \mathrm{fsqrt}(\mathrm{fsqr}(a)) = a$.

Now suppose that n is not a perfect square, so that $a^2 < n < (a+1)^2$ for some nonnegative integer $a < 2^{53}$. Then rnd is monotonic, so

$$\mathrm{rnd}(a^2) \leq \mathrm{rnd}(n) \leq \mathrm{rnd}((a+1)^2)$$

or equivalently,

$$\mathrm{fsqr}(a) \leq \mathrm{rnd}(n) \leq \mathrm{fsqr}(a+1).$$

Similarly, fsqrt is monotonic, so applying fsqrt throughout and using the previous corollary,

$$a \leq \mathrm{fsqrt}(\mathrm{rnd}(n)) \leq a+1.$$

Hence $\lfloor \mathrm{fsqrt}(\mathrm{rnd}\, n) \rfloor$ is either a or $a+1$, so a is a near square root of n. □

In fact, $\lfloor \mathrm{fsqrt}(\mathrm{rnd}\, n) \rfloor$ gives a near square root of n for all $n \leq 2^{106} + 2^{54}$. The smallest n for which it fails to give a near square root is $n = 2^{106} + 2^{54} + 1 = (2^{53}+1)^2$: for this n, the only possible near square root is 2^{53+1}, which is not representable in the IEEE 754 binary64 floating-point format.

Listing 10 shows a version of the iterative algorithm in Listing 4, modified to use floating-point arithmetic to compute the initial value of a. If $n < 2^{106}$, a is already a near square root of n and the **while** loop is executed zero times. More generally, the number of iterations required is $\lfloor \log_2((\log_2 n)/53) \rfloor$ for $n \geq 2^{53}$, and 0 otherwise.