Python's integer square root algorithm

Mark Dickinson

November 17, 2019

Abstract

We present an adaptive-precision variant of Heron's method for computing integer square roots of arbitrary-precision integers. The method 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

We start with a simple definition.

Definition 1. 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, the integer square root of n is simply the integer part of the exact 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 the ability to detect whether a given integer is a square, and if so, to extract its root.

For small n, it's feasible to use floating-point arithmetic to compute integer square roots. For example, assuming IEEE 754 binary64 format floating-point and a correctly-rounded square root operation, one can show that computing $\lfloor \sqrt{n} \rfloor$ directly gives the integer square root of n, provided that $n < 2^{52} + 2^{27}$. However, neither the Python language nor 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 computing 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. To avoid this complication, and to allow integer square roots to be computed for arbitrarily large integers, it's desirable to have an integer square root algorithm that works entirely with integer arithmetic.

In this article we present a simple and fast pure-integer algorithm to compute the integer square root of an arbitrary positive integer. This algorithm has been implemented for CPython's math module and is available from Python 3.8 onwards as math.isqrt.

Section 2 of this article reviews a well-known method for computing integer square roots based on Heron's method. This 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 integer-arithmetic version of Heron's method (also known as the Babylonian method, and expressible as a special case of the Newton–Raphson root-finding method). This is the approach used for example by Java's BigInteger class.

Heron's method is based on the observation that if x is a real approximation to the square root of a positive real number n, then

$$\frac{x + n/x}{2}$$

is an improved approximation. Iterating then allows the square root to be computed to any desired accuracy. It can be shown that the method converges towards the square root from any positive starting approximation x, and that once x gets close to the true square root the convergence is quadratic, so that the number of correct decimal places roughly doubles with each iteration.

Example 2. Given an approximation $x = 7/5 = 1.4$ to the square root $1.414213562\ldots$ of 2, a single iteration of Heron's method produces a new approximation $(7/5 + 2/(7/5))/2 = 99/70 = 1.414285714\ldots$. Applying a second iteration with $99/70$ as input gives $19601/13860 = 1.414213564\ldots$, 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 a fixed positive integer n, define a function f on the positive integers by

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

The idea is that—just like its real counterpart—the function f should transform a poor approximation to the integer square root of n into a better one, and so by applying f repeatedly we should eventually reach the integer square root. This works, but we have to be a little careful with the details. In particular, deciding when to stop iterating is delicate. The lemmas below make this precise.

Lemma 3. *For any positive integer a, $f(a)$ is greater than or equal to the integer square root of n.*

Proof. The AM-GM inequality applied to a and n/a gives

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

Hence

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

□

Lemma 4. *Given a positive integer a greater than the integer square root of n, $f(a)$ is smaller than a.*

Proof. We have the following chain of equivalences:

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

This completes the proof. □

Now suppose that we're given a starting guess a that exceeds the integer square root of n: $\lfloor\sqrt{n}\rfloor < a$. Combining the above lemmas, we have

$$\lfloor\sqrt{n}\rfloor \le f(a) < a.$$

If $f(a)$ is again greater than the integer square root of n, we have:

$$\lfloor\sqrt{n}\rfloor \le f(f(a)) < f(a) < a$$

and so on. So the sequence

$$a, f(a), f(f(a)), f(f(f(a))), \ldots$$

must eventually reach the integer square root. If not, we'd have an infinite strictly decreasing sequence of positive integers, violating the well-ordering principle.

Lemma 4 also provides a way to detect *when* we've reached the integer square root: if $\lfloor\sqrt{n}\rfloor \le a$ but the inequality $f(a) < a$ fails, then a cannot be greater than the integer square root, so it must *be* the integer square root. However, we need to be careful. While it's true that the sequence $a, f(a), f(f(a)), \ldots$ must eventually reach the square root, it would be a mistake to claim that it *converges* to the square root: we don't necessarily always reach a point where $f(a) = a$. For example, if $n = 15$, the sequence of iterates of f will eventually alternate between 3 and 4, and more generally this sort of alternation will occur for any n of the form $a^2 - 1$ for some $a \ge 2$.

Listing 1 expresses the algorithm described above in Python. A linguistic note: Python's // operator does "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 code in Listing 1 uses n as the initial estimate. That's highly inefficient for large n: the initial iterations will roughly halve the estimate each time,

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

```
def isqrt(n):
    def f(a):
        return (a + n//a) // 2

    # Starting guess can be anything not
    # smaller than the integer square root.
    a = n
    while True:
        fa = f(a)
        if fa >= a:
            return a
        a = fa
```

and many iterations will be needed to get into the general neighborhood of the square root. We can do better: *any* positive integer $a \geq \lfloor\sqrt{n}\rfloor$ can be used as a starting value. Alternatively, if we have a starting guess a that we know is close to the square root, but we don't know for sure that $a \geq \lfloor\sqrt{n}\rfloor$, we can apply a single iteration to replace a with $f(a)$ before running the main algorithm; by Lemma 3, $f(a)$ is guaranteed to be no smaller than the integer square root.

One simple, easy-to-compute possibility that's not too inefficient is to take a to be the smallest power of two exceeding $\lfloor\sqrt{n}\rfloor$. Before giving the code, we make a definition.

Definition 5. For a nonnegative integer n, the *bit length* of n, written $\text{length}(n)$, 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 0 is 0. For nonnegative integers n and e, we have $2^e \leq n$ if and only if $e < \text{length}(n)$, and $\text{length}(n) \leq e$ if and only if $n < 2^e$. 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 equivalences:

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

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 $f(a) \geq a$ has been replaced with the equivalent condition $\lfloor n/a\rfloor \geq a$.

Note: since we actually only need $\lfloor\sqrt{n}\rfloor \leq 2^e$ rather than $\lfloor\sqrt{n}\rfloor < 2^e$, we could tighten our initial bound slightly by using $\text{length}(n-1)$ instead of

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 d >= a:
            return a
        a = (a + d) // 2
```

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 worth it.

The algorithm above is well-known and well-used. It has the virtue of simplicity, and is reasonably efficient for inputs that aren't too large. Nevertheless, there's room for improvement. With the current initial guess, 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. The initial guess could perhaps be improved at the expense of some additional complexity. There's potential inefficiency towards the end of the algorithm, too. To demonstrate this last point, consider the following example.

Example 6. 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 returned. (Note that this example is representative of the worst-case behaviour rather than the typical behaviour of the algorithm.)

The algorithm introduced in the next section addresses these deficiencies, while retaining much of the simplicity of the algorithm in this section.

3 Variable-precision Heron's method

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

There are two key ideas. First, we vary the 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 at each iteration (which would require a per-iteration check-and-correct step), but instead allow the error to propagate, and we prove that with careful control of 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 the notion of a "near square root".

Definition 7. 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.

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 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. If j is not too large with respect 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 8. *Suppose that $n \ge 4$ and k is a positive integer satisfying $4k^4 \le n$. 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

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

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 $k/b \le 1$. From our (not yet used) assumption that $4k^4 \le n$ we have $k^2 \le n/4k^2$, 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 $k \le b$ and $k/b \le 1$. 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. □

Example 9. 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 based on binary, so multiplications and integer divisions by powers of two can be performed efficiently by bit-shifting. So when applying Theorem 8, we'll choose our k to be the largest power of two satisfying $4k^4 \le n$. 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 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 the interested reader to pursue them further.

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.

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
```

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
```

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, it's simpler to 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 just before we reach the **while** loop: we have $c = \lfloor \log_4 n \rfloor$, so $4^c \leq n < 4^{c+1}$ and $1 \leq \lfloor n/4^c \rfloor < 4$, so $a = 1$ 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 10. *Suppose that* $0 \leq s < \text{length}(c)$*, 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

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

Now we can apply the main theorem: if we can show that 2^{d-e-1} is an integer and that $4(2^{d-e-1})^4 \leq m$, it follows from Theorem 8 that a is a near square root of m. But from the definitions of d and e, $1 \leq d$ and $e = \lfloor d/2 \rfloor$, so it follows that $0 \leq d - e - 1 \leq e$, and

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

Furthermore, since $c = \lfloor \log_4 n \rfloor$, $4^c \leq n$, so $4^d \leq n/4^{c-d}$, hence $4^d \leq m$. Combining this with the inequality above, $4(2^{d-e-1})^4 \leq m$, 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 turns out to be remarkably simple: it's exactly $\lfloor \log_2 \lfloor \log_2 n \rfloor \rfloor$, assuming $n \geq 2$ (and zero iterations are required for $n = 1$). So for example, input values n satisfying $2^{32} \leq n < 2^{64}$ require exactly five iterations, while values in the range $2^{64} \leq n < 2^{128}$ require exactly six.

Listing 5: Fixed-precision variant, valid for $0 \le n < 2^{32}$

```
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 6: Fixed-precision variant, valid for $0 \le n < 2^{64}$

```
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
```

4 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 $2^{30} \le n < 2^{32}$, 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 n, we can find an f such that $2^{30} \le 4^f n < 2^{32}$, 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 5. The corresponding 64-bit version is shown in Listing 6. While the original iterative algorithm was developed for positive n, these fixed-precision variants also turn out to give the correct answer in the case $n = 0$.

5 Using floating-point

If we have access to a floating-point type that provides IEEE 754-2008 floating-point format and semantics, we can use floating-point arithmetic to compute integer square roots directly for small n, and to accelerate the computation of integer square roots for larger n. Note that Python does not currently require IEEE 754 format or semantics, but that in practice use of IEEE 754 floating-point is almost ubiquitous on platforms that support Python, and Python's **float** type is highly likely to map to the IEEE 754-2008 binary64 "double precision" format.

In this section, 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 round-ties-to-even rounding mode.

First some definitions: given a real number x, write $\text{float}(x)$ for the nearest binary64 floating-point number to x, following the IEEE 754 rules with the default "roundTiesToEven" rounding rule. (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 the statement givenn in the introduction, that if n is not too large, $\lfloor \text{fsqrt}(n) \rfloor$ gives the integer square root of n.

Lemma 11. *Suppose that n is an integer satisfying $1 \le n < 2^{52}$. Then $\lfloor \text{fsqrt}(n) \rfloor$ is the integer square root of n.*

Proof. 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 easily small enough to be exactly representable as floats, it follows that

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

Note that we have a $\le$ in the right-hand inequality, because there's a possibility that float rounded $\sqrt{n}$ up to the next integer. To prove the lemma, we need to eliminate this possibility.

Choose an integer k such that $2^k \le a < 2^{k+1}$. Floats in the interval $[2^k, 2^{k+1}]$ are spaced by 2^{k-52}, so the next float down from $a+1$ is $a+1-2^{k-52}$, and it's enough for us to show 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 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 that's equivalent to $2k \le 51$, and from our assumptions, $a^2 \le n < 2^{52}$, so $2^k \le a < 2^{26}$, hence $k < 26$. □

Lemma 12. *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 $\text{float}(x) = 2^e \,\text{float}(y)$.*

Proof. The bounds on x and y ensure that $\text{float}(x) = 2^f \,\text{rint}(x/2^f)$, where $f = \lfloor \log_2 x \rfloor - 52$, and similarly for y. Here $\text{rint}(x)$ 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.

Lemma 13. *Suppose x is an IEEE 754 binary64 floating-point number satisfying $2^{-511} \leq x < 2^{512}$. Then* $\mathrm{fsqrt}(\mathrm{fsqr}(x)) = x$.

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

$$x = 2^e y.$$

Then

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

so applying Lemma 12,

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

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 12 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

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

Or in other words, it's enough to prove the original statement in the special case that $1/2 < x \leq 1$. So from this point on, we assume that $1/2 < x \leq 1$.

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

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

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 $1/2 \leq \sqrt{z}$, $\sqrt{z} + x > 1$. 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 the lemma, together with the fact that any nonnegative integer not exceeding 2^{53} can be exactly represented in binary64 floating-point.

Corollary 14. *Suppose n is a nonnegative integer satisfying $n \leq 2^{53}$. Then* $\mathrm{fsqrt}(\mathrm{fsqr}(n)) = n$.

Corollary 15. *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,

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

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 us a near square root for all $n \leq 2^{106} + 2^{54}$. For $n = 2^{106} + 2^{54} + 1 = (2^{53}+1)^2$, $\mathrm{rnd}\, n = 2^{106} + 2^{54}$ and $\mathrm{fsqrt}(\mathrm{rnd}\, n) = 2^{53}$.