Python's integer square root algorithm

Mark Dickinson

December 26, 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 $a^2 \le n < (a+1)^2$.

Equivalently, the integer square root of n is 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 adaptation of Heron's method to the domain of positive integers.

Heron's method, also known as the Babylonian method, is a procedure for approximating the square root of a given positive real number t. It's based on the idea that if x is a positive real approximation to $\sqrt{t}$, then

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

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 2. 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 idea is that, just like its real counterpart h, the function g should transform a poor approximation to the integer square root of n into an improved approximation, and so by applying g repeatedly we hope to 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 3. *For any positive integer a, $\lfloor \sqrt{n} \rfloor \leq g(a)$.*

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

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

Dividing through by $2a$ gives

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

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

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

□

Lemma 4. *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 $\lfloor \sqrt{n} \rfloor \le a$, 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 $i \ge 0$, $a_{i+1} = g(a_i)$. We'll show that this sequence must eventially reach $\lfloor \sqrt{n} \rfloor$.

Lemma 5. *For all $i \ge 0$, $\lfloor \sqrt{n} \rfloor \le a_i$.*

Proof. This follows directly from Lemma 3 for $i \ge 1$, and from our assumption on a for $i = 0$. □

Lemma 6. *For all $i \ge 0$, $\lfloor \sqrt{n} \rfloor = a_i$ if and only if $a_i \le a_{i+1}$.*

Proof. By Lemma 5, either $\lfloor \sqrt{n} \rfloor < a_i$ or $\lfloor \sqrt{n} \rfloor = a_i$. If $\lfloor \sqrt{n} \rfloor = a_i$ then by Lemma 5 again, $a_i \le a_{i+1}$. If $\lfloor \sqrt{n} \rfloor < a_i$ then $a_{i+1} < a_i$ by Lemma 4. □

Theorem 7. *There exists an $i \ge 0$ 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 $a_i \le a_{i+1}$, and so by Lemma 6, 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 3, $g(a)$ 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 8. For a nonnegative integer n, the *bit length* of n, which we'll write simply as $\mathrm{length}(n)$, 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:

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

So taking $e = \lfloor(\operatorname{length}(n)+1)/2\rfloor$, 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 $\operatorname{length}(n-1)$ in place of $\operatorname{length}(n)$. 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.

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 9. 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 we aim 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 prove 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 10. 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

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

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 11. *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. □

Example 12. 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 $a = 110 + \lfloor 46696/440 \rfloor = 216$.

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
```

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 when applying Theorem 11, we'll take for our k the largest power of two satisfying $4k^4 \leq n$. Writing $k = 2^e$, we have:

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

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.

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, at the end of every iteration of that

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
```

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 entry to 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 13. *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 11 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. □

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
```

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 $n \ge 2$ (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 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 positive 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 binary floating-point type that provides IEEE 754 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 "double precision" binary64 interchange format, defined in section 3.6 of IEEE 754-2019.

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 "roundTiesToEven" rounding mode.

First some definitions: given a real number x, write $\mathrm{float}(x)$ for the nearest (under "roundTiesToEven") binary64 floating-point number to x. If x has magnitude $2^{1024} - 2^{970}$ or greater, $\mathrm{float}(x)$ 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 $\mathrm{fsqrt}(x)$ for the correctly-rounded square root of x, and $\mathrm{fsqr}(x)$ for the correctly-rounded square of x. In other words, $\mathrm{fsqrt}(x) = \mathrm{float}(\sqrt{x})$ and $\mathrm{fsqr}(x) = \mathrm{float}(x^2)$.

We first prove the statement given in the introduction, that if n is not too large, $\lfloor \mathrm{fsqrt}(n) \rfloor$ gives the integer square root of n.

Lemma 14. *Suppose that n is a nonnegative integer satisfying $n < 2^{52}$. Then $\lfloor \mathrm{fsqrt}(n) \rfloor$ 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

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

So

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

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

Since n is positive, so is a, and we can find an integer k such that $2^k \le a < 2^{k+1}$. 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 $2^{-k-2} \geq 2^{k-53}$. But that's equivalent to $2k \leq 51$, and from our assumptions, $a^2 \leq n < 2^{52}$, so $2^k \leq a < 2^{26}$, hence $k < 26$. □

Lemma 15. *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 16. *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 15,

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

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 15 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 $1/2 < x \le 1$. So from this point on, we assume that $1/2 < x \le 1$.

Let $z = \text{fsqr}(x)$ 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 $x = \text{fsqrt}(z)$ as required. $\square$

The next corollary follows immediately from Theorem 16, together with the fact that any nonnegative integer not exceeding 2^{53} can be exactly represented in binary64 floating-point.

Corollary 17. *Suppose n is a nonnegative integer satisfying $n \le 2^{53}$. Then $\text{fsqrt}(\text{fsqr}(n)) = n$.*

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 18. *Suppose n is a positive integer satisfying $n \le 2^{106}$. Then $\lfloor \text{fsqrt}(\text{rnd}(n)) \rfloor$ is a near square root of n.*

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

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

$$\text{rnd}(a^2) \le \text{rnd}(n) \le \text{rnd}((a+1)^2)$$

or equivalently,

$$\text{fsqr}(a) \le \text{rnd}(n) \le \text{fsqr}(a+1).$$

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

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

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

In fact, $\lfloor \text{fsqrt}(\text{rnd}\, n) \rfloor$ gives a near square root of n for all $n \le 2^{106} + 2^{54}$. 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, $\text{rnd}\, n = 2^{106} + 2^{54}$ and $\text{fsqrt}(\text{rnd}\, n) = 2^{53}$.

Listing 7 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 $n \ge 2^{53}$, and 0 otherwise.

Listing 7: 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
```