Python's integer square root algorithm

Mark Dickinson

October 12, 2019

Abstract

We present an adaptive-precision variant of the Babylonian 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. 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 algorithms require the ability to detect whether a given integer is a square, and if so, to extract its root. (XXX Ref: Cohen.)

For small integers n, it's feasible to use floating-point arithmetic to compute 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 $n < 2^{52} + 2^{27}$. (XXX give more details in a later section.)

However, Python guarantees neither IEEE 754 format floating-point nor correct rounding of floating-point arithmetic operations. As such, any floating-point-based method for computing an integer square root would need a pure-integer fallback method for the case where the floating-point sqrt 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 algorithm that works entirely with integer arithmetic, avoiding floating-point.

In this paper we present a simple integer-only algorithm to compute the integer square root of an arbitrary nonnegative number. This algorithm has been implemented for CPython's math module and is available in Python 3.8 as math.isqrt.

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. XXX give reference). This is the approach used for example by Java's BigInteger class. (XXX needs reference.)

Heron's method is based on the observation that if x is an approximation to the (real) square root of an input 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 number of correct decimal places roughly doubles with each iteration. (XXX needs reference.)

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 analogue, 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 in practice, but we have to be a little careful with the details: in particular, correct termination of the algorithm 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

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

Rearranging gives

$$2\lfloor \sqrt{n} \rfloor - a \leq n/a,$$

which implies

$$2\lfloor \sqrt{n} \rfloor - a \leq \lfloor n/a \rfloor.$$

Rearranging again gives

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

and so

$$\lfloor \sqrt{n} \rfloor \leq \left\lfloor \frac{a + \lfloor n/a \rfloor}{2} \right\rfloor$$

which is the desired result. □

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

Proof. We have the following chain of equivalences:

$$\begin{aligned}
\lfloor \sqrt{n} \rfloor < a &\iff \sqrt{n} < a && (1)\\
&\iff n < a^2 && (2)\\
&\iff n/a < a && (3)\\
&\iff \lfloor n/a \rfloor < a && (4)\\
&\iff a + \lfloor n/a \rfloor < 2a && (5)\\
&\iff \frac{a + \lfloor n/a \rfloor}{2} < a && (6)\\
&\iff \left\lfloor \frac{a + \lfloor n/a \rfloor}{2} \right\rfloor < a && (7)\\
&\iff f(a) < a. && (8)
\end{aligned}$$

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

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

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

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

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 bounded-below strictly descreasing sequence of integers, which is impossible.

Moreover, the second lemma also provides a way to detect when we've reached the integer square root: if $\lfloor \sqrt{n} \rfloor \leq a$ 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.

While the sequence

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

must eventually *reach* the square root, it would be a mistake to say that it *converges* to the square root: if a is the integer square root of n, then it's possible to have $f(a) > a$. For example, if $n = 15$, the sequence above will

eventually alternate between 3 and 4, and more generally this happens for any n of the form $m^2 - 1$ for some $m \geq 2$.

From a computational perspective, an expression like $\lfloor n/a \rfloor$ is misleading: it resembles a composition of two operations, with the intermediate result living in the domain of rational numbers. But in reality most programming languages or arbitrary-precision integer arithmetic packages will provide integer division (or floor division) as a single integer-to-integer primitive. In Python, for example, we'd write the expression defining $f(a)$ as (a + n//a) // 2, where the // operator is Python's "floor division" operator. Here's the full algorithm translated directly to Python.

```
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
```

Here's a more natural and streamlined version of the algorithm. The termination condition $f(a) \geq a$ is replaced with the equivalent condition $\lfloor n/a \rfloor \geq a$. For our starting guess, we take the smallest power of two that exceeds $\sqrt{n}$. (In Python, n.bit_length() gives the smallest nonnegative integer k for which $n < 2^k$.)

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

The algorithm outlined above has been fairly widely adopted. It has the virtue of simplicity, and is reasonably efficient for inputs that aren't too large. Nevertheless, it's not perfect. The choice of initial guess is delicate, and a poor choice will have a significant effect on running time. 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. And there's potential inefficiency towards the end of the algorithm, too. To demonstrate this, consider the following example.

Example 5. 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.

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 previous sections, 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: the algorithm produces at each iteration an approximation to the square root of $\lfloor n/4^s \rfloor$ for some integer s, with s decreasing 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 operation), 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 need a final 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 6. 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 the only near square root of n is a. Given a near square root a of n, the integer square root of n is clearly either a or $a-1$, depending on whether $a^2 \leq n$ or $a^2 > n$ (respectively). So an algorithm for computing near square roots provides us with a way to compute integer square roots.

Here's the idea of the algorithm: given a near square root d of $\lfloor n/k^2 \rfloor$ for some positive integer k, dk is then an approximation to $\sqrt{n}$, although possibly not a very good one. A single iteration of Heron's method applied to dk gives an improved approximation. Now comes the key point: if we're careful not to choose k too large with respect to n, we can prove that this improved approximation will again be a near square root of n.

The following theorem makes this precise. To keep calculations within the domain of the integers, it's convenient to restrict k to be even. So in the theorem below, k is replaced with $2m$.

Theorem 7. *Suppose that $n \geq 4$, and choose a positive integer m satisfying $4m^4 \leq n$. Given a near square root d of $\lfloor n/4m^2 \rfloor$, define a by*

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

Then a is a near square root of n.

In the notation of the previous section, a is defined by $a = f(2md)$.

Proof. By definition of near square root, we have

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

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

$$(d-1)^2 < \frac{n}{4m^2} < (d+1)^2.$$

Taking square roots throughout and multiplying by $2m$ gives

$$2m(d-1) < \sqrt{n} < 2m(d+1)$$

which can be rearranged as

$$|2md - \sqrt{n}| < 2m.$$

Squaring and dividing through by $4md$ gives

$$0 \le md + \frac{n}{4md} - \sqrt{n} < \frac{m}{d},$$

which implies that

$$-1 < md + \left\lfloor \frac{n}{4md} \right\rfloor - \sqrt{n} < \frac{m}{d},$$

Substituting the definition of a gives

$$-1 < a - \sqrt{n} < m/d.$$

To complete the proof, we need to know that $m \le d$, and so $m/d \le 1$. From the assumption that $4m^4 \le n$ we have $m^2 \le n/4m^2 < (d+1)^2$, so $m < d+1$. So now

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

from which $(a-1)^2 < n < (a+1)^2$, as required. □

Now we turn to implementation. Like many arbitrary-precision integer implementations, Python's integer implementation is based on binary, so multiplications and divisions by powers of two can be performed efficiently via bit shifts. For maximum efficiency when applying the lemma above, we choose our m to be the largest power of two satisfying $4m^4 \le n$. It's convenient to introduce another definition.

Definition 8. For a nonnegative integer n, the *bit length* of n, written $\text{length}(n)$, is the least nonnegative integer e for which $n < 2^e$.

Equivalently, 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 $2^e \le n$ if and only if $e < \text{length}(n)$.

When $m = 2^e$ is a power of two, the condition $4m^4 \le n$ of the lemma becomes $2^{2+4e} \le n$. That's equivalent to $2 + 4e < \text{length}(n)$, or $e \le (\text{length}(n) - 3)/4$. Taking $e = \lfloor (\text{length}(n) - 3)/4 \rfloor$ gives the following recursive near square root implementation, where the multiplication by m and the divisions by $4m$ and $4m^2$ are replaced by the corresponding bit-shift operations.

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

And that's it! Just six lines of simple Python code are needed to compute near square roots efficiently. Each step 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. Below we show an iterative version of the algorithm that makes both these improvements, and also adds the final check-and-correct step needed to return the integer square root rather than just a near square root. This version is closer to the actual CPython 3.8 implementation.

```
def isqrt(n):
    c = (n.bit_length() - 1) // 2
    m = c.bit_length()

    a = 1
    for s in range(1, m+1):  # s from 1 to m, inclusive
        d = c >> m-s
        e = d >> 1
        a = (a << d-e-1) + (n >> 2*c-d-e+1) // a
    return a if a*a <= n else a - 1
```

To understand the translation to the iterative version, it's helpful to look at how the quantity $c = \lfloor \log_4 n \rfloor$ changes over the course of the recursive algorithm. We can express c in terms of the bit-length of n as $c = \lfloor (\text{length}(n) - 1)/2 \rfloor$. Then $e = \lfloor (c-1)/2 \rfloor$, so $e + 1 = \lfloor (c+1)/2 \rfloor = \lceil c/2 \rceil$. So $\lfloor \log_4 \lfloor n/4^{e+1} \rfloor \rfloor = \lfloor \log_4 n \rfloor - e + 1$, which is equal to $c - \lceil c/2 \rceil = \lfloor c/2 \rfloor$.

So the sequence of values for $\lfloor \log_4 n \rfloor$ as the recursion progresses is

$$c, \lfloor c/2 \rfloor, \lfloor c/4 \rfloor, \lfloor c/8 \rfloor, \ldots, \lfloor c/2^m \rfloor = 0,$$

where m is the least nonnegative integer for which $c < 2^m$, or in other words, the bit-length of c.

In the iterative version of the algorithm, the variables d and e correspond to successive pairs in the above sequence, progressing from right to left. We start with $d = \lfloor c/2^{m-1} \rfloor$ and $e = \lfloor c/2^m \rfloor = 0$ on the first iteration, and end with $d = \lfloor c/2^0 \rfloor = c$ and $e = \lfloor c/2^1 \rfloor$ on the final iteration. Each iteration starts with a a near square root of $\lfloor n/4^{c-e} \rfloor$ and transforms that a into a near square root of $\lfloor n/4^{c-d} \rfloor$. In particular, the first iteration needs to start with a near square root of $\lfloor n/4^c \rfloor$, but we have $1 \le \lfloor n/4^c \rfloor < 4$, and so $a = 1$ is a suitable starting value. And after the final iteration, a is a near square root of $\lfloor n/4^{c-c} \rfloor = n$, as required.

Note that the quantities c, d, e, m 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 is remarkably simple: it's exactly $\lfloor \log_2 \lfloor \log_2 n \rfloor \rfloor$, assuming $n \geq 2$. So for example input values n satisfying $2^{32} \leq n < 2^{64}$ require exactly 5 iterations, while values in the range $2^{64} \leq n < 2^{128}$ require exactly 6.

4 Floating-point quick start

If we can assume IEEE 754 floating-point and semantics, we can use floating-point for low precision. In this section we assume IEEE 754 binary64 ("double precision") floating-point, and correctly-rounded arithmetic operations including square root.

Write $\mathrm{fsqrt}(x)$ for the correctly-rounded square root of an IEEE 754 binary64 floating-point value x, and $\mathrm{fsqr}(x)$ for the correctly-rounded square of x.

Lemma 9. *Suppose x is a positive finite binary64 floating-point number, satisfying $2^{-511} \leq x < 2^{512}$. Then $\mathrm{fsqrt}(\mathrm{fsqr}(x)) = x$.*

Proof. The bounds on x ensure that overflow and underflow are avoided, so that $\mathrm{fsqr}(x)$ is again positive and normal.

Provided that underflow and overflow are avoided, we have $\mathrm{fsqrt}(4^e x) = 2^e \,\mathrm{fsqrt}(x)$ and $\mathrm{fsqr}(2^e x) = 4^e \,\mathrm{fsqr}(x)$. Using this, we can assume that $\frac{1}{2} \leq x < 1$. □

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

Corollary 11. *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 $\mathrm{fsqrt}(\mathrm{rnd}\, n) = 2^{53}$.

5 To do

- Note that if a value is already known to be a square, no check-and-correct step is needed. - Similarly for square detection: take a near square root, square it, and check. - And square root ceiling is easy, too. - Running time analysis, and real-world timings to back it up. Asymptotically, isqrt of a $2n$-bit integer should be faster than division of a $2n$-bit integer by an n-bit dividend. - Fixed-precision versions. - Floating-point methods and floating-point accelerations. - Details on correctness of floating-point algorithm.