Ruby trunk - Feature #13263

Add companion integer nth-root method to recent Integer#isqrt

02/28/2017 09:01 PM - jzakiya (Jabari Zakiya)

Status:	Open
Priority:	Normal
Assignee:	
Target version:	

Description

Following the heels of adding the method Integer#isqrt, to create exact integer squareroot values for arbitrary sized integers, based on the following threads:

https://bugs.ruby-lang.org/issues/13219
https://bugs.ruby-lang.org/issues/13250

I also request adding its companion method to compute any integer nth-root too.

Below are sample methods of high level Ruby code that compute exact results.

https://en.wikipedia.org/wiki/Nth_root_algorithm

The Newton's code is a Python version I tweaked to make it look like Integer#isqrt's form.

Benchmarks show the **bbm** method is generally faster, especially as the roots become larger, than using Newton's method, with an added benefits its simpler to code/understand, and has a lower sensitivity to the initial root value, and handling of small numbers.

```
class Integer
  def irootn(n)    # binary bit method (bbm) for nth root
    return nil if self < 0 && n.even?
    raise "root n is < 2 or not an Integer" unless n.is_a?(Integer) && n > 1
    num  = self.abs
    bits_shift = (num.bit_length - 1)/n + 1   # add 1 for initial loop >>= 1
    root, bitn_mask = 0, (1 << bits_shift)
    until (bitn_mask >>= 1) == 0
      root |= bitn_mask
      root ^= bitn_mask if root**n > num
    end
    root *= self < 0 ? -1 : 1
  end

  def irootn1(n)    # Newton's method for nth root
    return nil if self < 0 && n.even?
    raise "root n is < 2 or not an Integer" unless n.is_a?(Integer) && n > 1
    return self if self == 0 || (self == -1 && n.odd?)
    num = self.abs
    b = num.bit_length
    e, u, x = n-1, (x = 1 << (b-1)/(n-1)), x+1
    while u < x
      x = u
      t = e * x + num / x ** e
      u = t / n
    end
    x *= self < 0 ? -1 : 1
  end

  def irootn2(n)    # Newton's restructured coded method for nth root
    return nil if self < 0 && n.even?
    raise "root n is < 2 or not an Integer" unless n.is_a?(Integer) && n > 1
    return self if self == 0 || (self == -1 && n.odd?)
    num = self.abs
    b = num.bit_length
    e, x = n-1, 1 << (b-1)/(n-1) + 1
```

```
    while t = (e * x + num / x ** e)/n < x
      x = (e * x + num / x ** e)/n
    end
    x *= self < 0 ? -1 : 1
  end
end

require "benchmark/ips"

[50, 500, 1000, 2000, 4000, 5000].each do |exp|
  [3, 4, 7, 13, 25, 33]. each do |k|
    Benchmark.ips do |x|
      n = 10**exp
      puts "integer root tests for root #{k} of n = 10**#{exp}"
      x.report("bbm"     ) { n.irootn(k)  }
      x.report("newton1" ) { n.irootn1(k) }
      x.report("newton2" ) { n.irootn2(k) }
      x.compare!
    end
  end
end
```

Here are results.

```
def tm; t=Time.now; yield; Time.now-t end

2.4.0 :022 > exp = 111; n = 10**exp; r = 10; puts n, "#{ tm{ puts n.irootn(r)} }", "#{ tm{ puts n.
irootn1(r)} }", "#{ tm{ puts n.irootn2(r)} }"
125892541179
125892541179
125892541179
10000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000
4.6673e-05
6.5506e-05
0.000121357
 => nil
2.4.0 :023 > exp = 150; n = 10**exp; r = 50; puts n, "#{tm{ puts n.irootn(r)}}", "#{ tm{ puts n.ir
ootn1(r)}}", "#{ tm{ puts n.irootn2(r)} }"
1000
1000
1000
10000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
2.28e-05
1.8762e-05
0.000128852
 => nil
2.4.0 :024 >
```

The benchmarks show that irootn2 is the slowest but it has the same
form as Integer#isqt in the numeric.c and bignum.c files in trunk.
It probably can be tweaked to make it faster.

bignum.c, starting at line 6772
https://bugs.ruby-lang.org/projects/ruby-trunk/repository/revisions/57705/entry/bignum.c
numeric.c, starting at line 5131
https://bugs.ruby-lang.org/projects/ruby-trunk/repository/revisions/57705/entry/numeric.c

Thus, a hybrid method could be created that swtiches between the two.

```
def isqrt(num=self)

  b = num.bit_length
  x = 1 << (b-1)/2 | num >> (b/2 + 1)      # optimum first root extimate
  while (t = num / x) < x
    x = ((x + t) >> 1)
```

```
  end
  x
end

def irootn2(n)

  b = num.bit_length
  e, x = n-1, 1 << (b-1)/(n-1) + 1        # optimum first root estimate(?)
  while t = (e * x + num / x ** e)/n < x
    x = (e * x + num / x ** e)/n
  end
  x
end

def irtn(n)  # possible hybrid combination for all nth-roots

  b = num.bit_length
  if 2 < n  # for squareroot
    x = 1 << (b-1)/2 | num >> (b/2 + 1)
    while (t = num / x) < x
      x = ((x + t) >> 1)
    end
  else      # for roots > 2
    e, x = n-1, 1 << (b-1)/(n-1) + 1
    while t = (e * x + num / x ** e)/n < x
      x = (e * x + num / x ** e)/n
    end
  end
  x *= if self < 0 ? -1 : 1
end
```

So with just a little more work, a highly performant nth-root method can be added
to the std lib, as with Integer#isqrt, to take care of all the exact integer roots
for arbitrary sized integers, by whatever name that is preferable.

This will enhance Ruby's use even more in fields like number theory, advanced math, cryptography,
etc, to have fast primitive standard methods to compute these use case values.

History

#1 - 02/28/2017 10:38 PM - Student (Nathan Zook)

Newton's method has quadratic convergence. This means that a properly implemented Newton's method will blow away any BBM if more than just a few bits are needed.

"Properly implemented" is a big deal, because, as I have found with some research (see in particular http://www.agner.org/optimize/instruction_tables.pdf), recent Intel cpus can require >30x a long to do an integer divide as a multiply. I've not dug into our multi-precision library to see how Ruby implements things, but it can matter a very great deal. From the standpoint of Ruby implementations, the overhead of Ruby calls is a huge burden on these methods, and likely dominates much of your benchmarks. In the case of square roots, this was relatively easy to overcome, but for higher-order roots, this becomes more and more of a problem.

A general n-th root extractor makes sense, but I believe that it is worthwhile to do a bit of digging into these techniques before deciding on an approach.

#2 - 03/01/2017 02:46 AM - shyouhei (Shyouhei Urabe)

Jabari Zakiya wrote:

> This will enhance Ruby's use even more in fields like number theory, advanced math, cryptography,
> etc, to have fast primitive standard methods to compute these use case values.

I'm not immediately against this but can I ask you when is this method useful? Because being amateur, I don't know any real application of integer n-th root in cryptography etc.

#3 - 03/01/2017 09:08 PM - jzakiya (Jabari Zakiya)

Further testing shows Newton's method is sensitive to its implementation as you take larger roots.

Shown below are test results that show the **irootn1** Newton's implementation starts to give incorrect (smaller)
results past a certain size root value, but the **irootn2** Newton's implementation gives correct results (**bbm**

will always produce correct results). In the benchmarks, **irootn1** may sometimes be shown to be faster because of it producing, faster, smaller wrong results.

Because of this, **bbm** seems to be generally faster versus Newton's method, especially as the roots get bigger, because the operations to perform Newton's are cpu costly, requiring a multiplication, exponentiation, addition, and two divisions, at least once per round, for arbitrary sized integers.

On the other hand, **bbm** only requires 2|3 cheap cpu bit operations and one exponentiation per round.

So while on paper Newton's may seem it should be faster, its cpu implementation cost is much greater, and empirical evidence establishes **bbm** is generally faster than these implementations of Newton's.

But the most important point for me is, **as a user I always want correct results first and foremost.**

The only way to empirically (versus theoretically) establish which is faster is to do an optimized C version of **bbm** too, and do an apples-to-apples comparison against a Newton's version. But it is already clear which is simpler to code, with no worries it will produce incorrect answers. Also **bbm** takes much less electrical power to perform, because of its relatively smaller cpu operational costs.

I would that encourage that empirical results from accuracy testing, and benchmarking, should establish what is --better--, giving consideration to all relevant factors (not just speed).

Test results below.

```
2.4.0 :567 > exp = 800; n = 10**exp; r = 74; puts ' ', "#{tm{ puts n.irootn(r)}}", "#{ tm{ puts n.irootn1(r)}}
", "#{ tm{ puts n.irootn2(r)} }" #,
64686076615
64686076615
64686076615

0.000136971
0.000171888
0.000681239
 => nil
2.4.0 :568 > exp = 800; n = 10**exp; r = 75; puts ' ', "#{tm{ puts n.irootn(r)}}", "#{ tm{ puts n.irootn1(r)}}
", "#{ tm{ puts n.irootn2(r)} }"
46415888336
34359738368
46415888336

0.000177774
3.7298e-05
0.000527569
 => nil
2.4.0 :569 > exp = 800; n = 10**exp; r = 100; puts ' ', "#{tm{ puts n.irootn(r)}}", "#{ tm{ puts n.irootn1(r)}
}", "#{ tm{ puts n.irootn2(r)} }"
100000000
67108864
100000000

0.000102902
1.6642e-05
0.000430577
 => nil
2.4.0 :570 > exp = 800; n = 10**exp; r = 200; puts ' ', "#{tm{ puts n.irootn(r)}}", "#{ tm{ puts n.irootn1(r)}
}", "#{ tm{ puts n.irootn2(r)} }"
10000
8192
10000

5.4696e-05
2.6753e-05
0.000941954
 => nil
2.4.0 :571 > exp = 800; n = 10**exp; r = 300; puts ' ', "#{tm{ puts n.irootn(r)}}", "#{ tm{ puts n.irootn1(r)}
}", "#{ tm{ puts n.irootn2(r)} }"
464
256
464

6.1939e-05
1.6915e-05
0.000279832
 => nil
2.4.0 :572 > exp = 800; n = 10**exp; r = 400; puts ' ', "#{tm{ puts n.irootn(r)}}", "#{ tm{ puts n.irootn1(r)}
```

```
}", "#{ tm{ puts n.irootn2(r)} }"
100
64
100

4.6034e-05
3.8532e-05
0.000322497
 => nil
2.4.0 :573 > exp = 800; n = 10**exp; r = 500; puts ' ', "#{tm{ puts n.irootn(r)}}", "#{ tm{ puts n.irootn1(r)}
}", "#{ tm{ puts n.irootn2(r)} }" #
39
32
39

4.1114e-05
6.3514e-05
0.000486907
 => nil
2.4.0 :574 >
```

#4 - 03/01/2017 10:54 PM - jzakiya (Jabari Zakiya)

Actually, this **bbm** version is generally a smidgen faster than the original, especially for perfect roots.

```
class Integer
  def irootn3(n)   # binary bit method (bbm) for nth root
    return nil if self < 0 && n.even?
    raise "root n is < 2 or not an Integer" unless n.is_a?(Integer) && n > 1
    num  = self.abs
    root = b_mask = 1 << (num.bit_length-1)/n
    numb = root**n
    until ((b_mask >>= 1) == 0) || numb == num
      root |= b_mask
      root ^= b_mask if (numb = root**n) > num
    end
    root *= self < 0 ? -1 : 1
  end
end

2.4.0 :781 > exp = 20000; n = 10**exp; r = 100; puts '', "#{tm{ puts n.irootn(r)}}", "#{ tm{ puts n.irootn3(r)
}}"
10000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
10000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000

0.077781852
0.056256279
 => nil
2.4.0 :782 > exp = 20000; n = 10**exp; r = 105; puts '', "#{tm{ puts n.irootn(r)}}", "#{ tm{ puts n.irootn3(r)
}}"
29935772947204898173058573713545176883262845807813845402640946129364486101284568000214134023311247816174990535
520060919558577709024260554789568837446173266139382305916243214137145048773203904
29935772947204898173058573713545176883262845807813845402640946129364486101284568000214134023311247816174990535
520060919558577709024260554789568837446173266139382305916243214137145048773203904

0.086875851
0.085639275
 => nil
2.4.0 :783 >
```

#5 - 03/03/2017 10:09 PM - Student (Nathan Zook)

If you get the wrong answer from Newton's, then you are doing it wrong. It may fail to converge, (which seems MOST unlikely in this case) but that is a different matter. But ultimately, there is 0 credibility to benchmarks that show BBM faster than Newton's past some small values, unless f(x) has a multiple root--which it never will here.

What IS special about this class of problems is that Newton's tends to blow up if your estimate is ever small.It is important to start above and work your way down. That is to say, your condition code is wrong. Doubling your initial estimate would be a substantial improvement in this regard. (It is helpful to actually print out the intermediate results to help understand what is happening.) Also, I do not understand why you do the exact same calculation twice in your Newton's work.

Seriously, though. I was only doing the ruby benchmarks before to get comparative information about O[] performance of NR verses Zimmerman. We cannot rely on them for anything else when we are going to C.

#6 - 03/04/2017 06:59 PM - jzakiya (Jabari Zakiya)

It would be really helpful if people produce empirical results of actual coded
examples of techniques, to establish their efficacies.

Math may suggest theoretical possibilities, but engineering determines their feasibility.

I've tried to be objective in assessing alternaitve techniques, creating objective tests, and
presentating empirical test results (accuracy and performance), that anyone can run themselves to verify.

Based on these empirical results, **bbm** is the "best" technique that satisfies a host of criteria I have listed.

The only thing Newton's method possibly has over **bbm** is speed, but only under certain conditions, and if
you select an optimized initial estimate. If the estimate is larger than necessary you get speed degradation.
If the estimate is too small you can get incorrect results. And Newton's speed advantage is only consistently
observed for an optimized version for squareroots, which cannot be applied generally to any nth-root, which
requires another specialized implementation.

Below I present a technique to create one optimized C impementation of **bbm** that will produce accurate results
for any nth-root of any integer. I think if something like this (or possibly better; I leave that to the C
devs gurus) is done, all the different relevant criteria I think should be considered (accuracy, speed, memory
use, power consumption, universal cpu efficiency) will be satisfied.

Here is my proposal on how to optimally implement (in C) the **bbm** technique, to use for all nth-roots.

```
class Integer
  def irootn(n)   # binary bit method (bbm) for nth root
    return nil if self < 0 && n.even?
    raise "root n is < 2 or not an Integer" unless n.is_a?(Integer) && n > 1
    num  = self.abs
#-------------------------------------------------------
    root = bit_mask = 1 << (num.bit_length - 1)/n
    numb = root**n
    until ((bit_mask >>= 1) == 0) || numb == num
      root |= bit_mask
      root ^= bit_mask if (numb = root**n) > num
    end
#-------------------------------------------------------
    root *= self < 0 ? -1 : 1
  end
end
```

Here is a possible C optimized implementation of the **bbm** that will work for all n-bit sized cpus.

This example assumes a 64-bit cpu for illustration purposes.

Let's use the example below to illustrate the technique.

Let: num = 10**50; num.size => 21 bytes to represent.

Therefore **num** is represented in memory as below, where each character is a 4-bit nibble.

```
              W2          W1               W0
num   =  xxxxxxxx yyyyyyyyyyyyyyyy zzzzzzzzzzzzzzzz
```

Now the first value the algorithm computes is the **bit_mask** size,

but the first computation is this: (num.bit_length - 1) => (167 - 1) = 166

We then (integer) divide this by the root value **n**.

Lets use the squareroot n = 2, but the algorithm works for any n >= 2.

Now: bit_mask = 1 << (166/2) => bit_mask = 1 << 83

In a naive implementation this value would take up 84 bits, or two (2) 64-bit words.

But in an optimized implementation we only need to use one cpu register, and
a word count value/register, to keep track of this value. So here the word
count is 2, and the initial value for the bit_mask is the portion of it that
fits in the upper word. As we shift **bit_mask** right, when it equals "0" we
decrement the word count to "1" and set the msb for **bit_mask** to "1", and continue.
When the **bit_mask** value hits "0" again, and we decrement its word count to "0"
we know we've finished the process (if we make it this far, if it wasn't a perfect root).

This bit_mask = 1 << 83 would computationally look like: bit_mask = 0x80000 0x0000000000000000

but be initially represented as: bit_mask = 0x80000; bit_mask_word_count = 2

So we only have to work with register sized values to work with **bit_mask**.

The **root** value is then also set to the initial **bit_mask** value, but be represented as:

```
  root = 0x80000 0x0000000000000000; root_word_count = 2
```

The next computation is: numb = root**n

Now we know **numb** has the same bit size as **num** but we can do an optimized root**n computation because we know here only the msb is set, so we can just do the root**n computation on the MSword of root, and store that as **numb** with the bit length of **num**. This, again, eliminates doing an initial **n-eponentiation** on any arbitrary sized integer.

Now we start the loop: until (bit_mask >> 1) || numb == numb

Here again, we only shift the value representing the current word position for **bit_mask** stored in a cpu register, which is a fast/cheap inplace cpu operation for any sized cpu.

We can also do the numb == num comparison on a word-by-word basis, in a cpu register.

Now we do the operations: root |= bit_mask and root ^= bit_mask

Again, we only operate on the current **root** and **bit_mask** words Wi that are in a cpu register until we need to operate on the next lower word.

As we do the operation: if (numb = root**n) > num

the **root** exponentiation only needs to be performed on the non-zero Wi for **root**, as the lower "0" valued Wi will contribute nothing to the computation's value.

Thus, we've reduced the algorithm to a series of very fast, and computationally inexpensive, primitive cpu operations, that can easily be performed on any cpu of any size. We need no adds, multiplies, and divisions, which are much more cpu intensive/expensive, that are necessary to perform Newton's method.

So with one standard C optimized implemenation we will always get correct results for any/all integer roots, that will work optimally on any cpu, which uses the least amount of electrical power to perform (a key consideration for power sensitive platforms like phones, tablets, embedded systems, robots, and IoT things).

I am also 99.99% sure that this type of **bbm** optimized implementation will be faster than any other theoretically possible technique, as a general technique to always accurately produce the nth-root of an integer.

This can be empirically verified by creating this implemetation and doing an apples-to-apples comparison to other techniques, and assess them to the list of criteria I suggested, as well as other relevant criteria, for comparison.

#7 - 03/05/2017 08:17 PM - jzakiya (Jabari Zakiya)

An optimization for the initiall root**n can be as follows:

Given any number **num** with only one bit set, and thus: bits = num.bit_length

then it's exponentiation to any **n** is just: num**n => num << (num.bit_length - 1)*(n-1)

```
> num = 0x80000000  => 2147483648
> n = 1; (num**n).to_s(2)
 => "100000000000000000000000000000000"
> n = 1; (num  << (num.bit_length - 1)*(n-1)).to_s(2)
 => "100000000000000000000000000000000"
> n = 2; (num**n).to_s(2)
 => "10000000000000000000000000000000000000000000000000000000000000000"
> n = 2; (num  << (num.bit_length - 1)*(n-1)).to_s(2)
 => "10000000000000000000000000000000000000000000000000000000000000000"
> n = 3; (num**n).to_s(2)
 => "1000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000"
> n = 3; (num  << (num.bit_length - 1)*(n-1)).to_s(2)
 => "1000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000"
```

So in the C optimized **bbm** implementation the initial root**n exponentiation can be replaced with this simple machine level bit manipulation.

```
    root = bit_mask = 1 << (num.bit_length - 1)/n
    numb = (root << (root.bit_length - 1)*(n-1)    # fast cpu level root**n
```

```
    until ((bit_mask >>= 1) == 0) || numb == num
      root |= bit_mask
      root ^= bit_mask if (numb = root**n) > num
    end
```

It's interesting that if you just compare the benchmarks between using the ** operator and this method in hight level Ruby the ** operator is faster, but that's because in highlevel Ruby all the separate methods calls incur their individual overhead, while the ** operator incurs only one, and has a highly performant C implementation. (But if you compare the differnces of the **irootn** method using the different techniques, they perform the same using benchmark/ips, which is sort of expected since this initial operation occurs only once.)

Unless the C implementation of the ** operator already optimizes for this case, I have to think a

well done cpu level C implementation of: num << (num.bit_length - 1)*(n-1)

has to be faster, because all you're doing is setting one bit in some word Wi of a number.

```
require "benchmark/ips"

(2..10).each do |exp|
  [3, 4, 7, 13].each do |k|
    Benchmark.ips do |x|
      n = 2**exp; b = n.bit_length
      puts "integer exponentiation tests for power #{k} for n = 2**#{exp}"
      x.report("n**k"      ) { n**k  }
      x.report("n**k bits" ) { n << (b-1)*(n-1) }
      x.report("n**k bits1") { n << (n.bit_length-1)*(n-1) }
      x.compare!
    end
  end
end
```

#8 - 03/06/2017 02:58 AM - jzakiya (Jabari Zakiya)

More efficient.

```
    root = bit_mask = (b = 1 << (num.bit_length - 1)/n)
    numb = root << b*(n-1)                              # fast cpu level root**n
    until ((bit_mask >>= 1) == 0) || numb == num
      root |= bit_mask
      root ^= bit_mask if (numb = root**n) > num
    end
```

#9 - 03/06/2017 07:44 PM - jzakiya (Jabari Zakiya)

A further simplification can be done for numb = root**n

```
root = bit_mask = 1 << (b = (num.bit_length - 1)/n)
numb = root**n
numb = root << b*(n-1)
numb = (1 << b) << b*(n-1)
numb = 1 << (b + b*(n-1))
numb = 1 << b*(1 + (n-1))
numb = 1 << b*n
```

Which means **root** and **numb** can be done in parallel now by the compiler.

Also, because **root** and **bit_mask** are the same size, only one **word_count** variable is needed, to track which word of **root** is being worked on, not one for each.

You also only need on word_count variable for the size of **numb** and **num**.
Then the comparisons **numb == num** and **numb > num** can be done on a word-by-word basis, starting from each MSword. If the MSword of **numb** is > or < than that for **num** then it's "true" or "false"; if equal continue with next lower Mswords as necessary.

This reduces the implementations to just bit manipulations, with 1|2 bit twiddles and one shift operation per loop, and one real arithmetic operation, the ** exponentiation inside the loop.

```
    root = bit_mask = 1 << (b = (num.bit_length - 1)/n)
    numb = 1 << b*n                                   # fast cpu level root**n
    until ((bit_mask >>= 1) == 0) || numb == num
      root |= bit_mask
      root ^= bit_mask if (numb = root**n) > num
    end
```

#10 - 03/07/2017 04:31 PM - jzakiya (Jabari Zakiya)

These are the correct benchmarks to show the differences in performance doing root**n.
Even in highlevel Ruby, the 1 << b*n shows it is faster.

```
require "benchmark/ips"

(50..60).each do |exp|
  [3, 4, 7, 13].each do |n|
    Benchmark.ips do |x|
      num = 2**exp; root = 1 << (b = (num.bit_length-1)/n)
      puts "root**n tests for root #{n} of num = 2**#{exp}"
      x.report("root**n"        ) { root**n  }
      x.report("1 << b*n"       ) { 1 << b*n }
      x.report("root << b*(n-1)") { root << b*(n-1) }
      x.compare!
    end
  end
end
```

#11 - 03/13/2017 09:25 PM - jzakiya (Jabari Zakiya)

Just FYI.

I simplified Newton's general nth-root method from the original implementation I posted.
It's faster, and seems to produce the correct results all the time (from the tests I've run).
For some roots (mostly smallish) of certain numbers it's faster than **bbm** by some
percentage difference, but in general **bbm** is still faster, by whole number factors, across the board.

```
original implementation of Newton's general nth-root method

def irootn2(n)
  return nil if self < 0 && n.even?
  raise "root n is < 2 or not an Integer" unless n.is_a?(Integer) && n > 1
  return self if self == 0 || (self == -1 && n.odd?)
  num = self.abs
  b = num.bit_length
  e, x = n-1, 1 << (b-1)/(n-1) + 1       # optimum first root estimate(?)
  while t = (e * x + num / x ** e)/n < x
    x = (e * x + num / x ** e)/n
  end
  x
end

simpler/faster implementation of Newton's general nth-root method

def irootn2(n)   # Newton's method for nth root
  return nil if self < 0 && n.even?
  raise "root n is < 2 or not an Integer" unless n.is_a?(Integer) && n > 1
  return self if self == 0 || (self == -1 && n.odd?)
  num = self.abs
  b = num.bit_length
  e, x = n-1, 1 << (b-1)/(n-1) + 1       # optimum first root estimate(?)
  while (t = e * x + num / x ** e)/n < x
    x = t/n
  end
  x *= self < 0 ? -1 : 1
end
```

#12 - 03/14/2017 03:42 AM - jzakiya (Jabari Zakiya)

Using a 1-bit greater initial estimate than for **bbm** makes Newton's nth-root implementation
significantly faster across the board than before (with seemingly correct answers).

```
def irootn2(n)   # Newton's method for nth root
  return nil if self < 0 && n.even?
  raise "root n is < 2 or not an Integer" unless n.is_a?(Integer) && n > 1
  return self if self == 0 || (self == -1 && n.odd?)
  num = self.abs
  b = num.bit_length

  #e, x = n-1, 1 << (b-1)/(n-1) + 1       # optimum first root estimate(?)
  e, x = n-1, 1 << (b-1)/n + 1           # much better first root estimate

  while (t = e * x + num / x ** e)/n < x
    x = t/n
```

```
  end
  x *= self < 0 ? -1 : 1
end
```

#13 - 03/21/2017 04:39 PM - jzakiya (Jabari Zakiya)

FYI for general interest and curiosity.

In Ruby 2.4.0 the 3 implementations below of Newton's general nth-root method all produce
correct results, using an initial root value that's 1-bit larger than the actual value.

Using **benchmark-ips** they are all basically equivalent in speed, with **Newton3**
being a smidgen faster across a range of number/root sizes. It is interesting to see
how they differ in speed (minimally) based on the particular number and/or root value.

It is also interesting to see that when implemented and run with Crystal (current 0.21.1),
while Crystal is faster (as expected), it is not multiple orders faster, and the performance
profile is similar between the different implementations. (Replace 1 << ... with 1.to_big_i << ...)

Thus, Ruby's use of glibc, gmp, et al, libraries appears to be very, very good for doing this math,
(at least to the accuracies of these libraries). It would still be intersting to see how much faster
an optimized version of **bbm** would be (as I've proposed, or better), compared to Newton, especially
since the stock implementation is still faster than any of the Newton implementations for some number/root sizes.

```
def irootn(n)   # Newton's method for nth root
  return nil if self < 0 && n.even?
  raise "root n not an Integer >= 2" unless n.is_a?(Integer) && n > 1
  return self if (self | 1) == 1 || (self == -1 && n.odd?)
  num = self.abs
```

```
                    Newton1                                                 Newton2
         Newton3
  e, u, x = n-1, (x = 1 << (num.bit_length-1)/n + 1), x+1    e, x = n-1, 1 << (num.bit_length-1)/n + 1    e, x
 = n-1, 1 << (num.bit_length-1)/n + 1
  while u < x                                                while (t = (e * x + num / x ** e))/n < x     whil
e (t = (e * x + num / x ** e)/n) < x
    x = u                                                       x = t/n                                      x
 = t
    t = e * x + num / x ** e                                 end                                          end
    u = t / n
  end

  x *= self < 0 ? -1 : 1
end
```

#14 - 03/22/2017 08:17 PM - jzakiya (Jabari Zakiya)

Noticed after Ruby 2.4.1 upgrade today (Wed 2017.03.22) performance is generally a bit slower.

#15 - 04/04/2017 10:55 PM - jzakiya (Jabari Zakiya)

FYI

Looking at the GNU Multiple Precision Arithmetic Library I see it has functions for
arbitrary size integer squareroot and nth-roots.

Doesn't Ruby already use this library?
Have they been considered/tested in Ruby? Are they better than the suggested alternatives?

https://gmplib.org/
https://gmplib.org/gmp-man-6.1.2.pdf

```
p. 36 of gmblib 6.1.2 manual

5.8 Root Extraction Functions

int mpz_root (mpz t rop, const mpz t op, unsigned long int n) [Function]
Set rop to bp n opc; the truncated integer part of the nth root of op. Return non-zero if the
computation was exact, i.e., if op is rop to the nth power.

void mpz_sqrt (mpz t rop, const mpz t op) [Function]
Set rop to bpopc; the truncated integer part of the square root of op.
```

#16 - 04/05/2017 12:49 AM - shyouhei (Shyouhei Urabe)

jzakiya (Jabari Zakiya) wrote:

> FYI
>
> Looking at the GNU Multiple Precision Arithmetic Library I see it has functions for arbitrary size integer squareroot and nth-roots.
>
> Doesn't Ruby already use this library?

Yes. https://bugs.ruby-lang.org/issues/8796

> Have they been considered/tested in Ruby?

You can try compiling ruby by configure --with-gmp.

> Are they better than the suggested alternatives?

This proposed function is something new to us, so not tested yet.