Ruby trunk - Feature #14383

Making prime_division in prime.rb Ruby 3 ready.

01/23/2018 02:18 AM - jzakiya (Jabari Zakiya)

Status:	Open
Priority:	Normal
Assignee:	yugui (Yuki Sonoda)
Target version:	Next Major

Description

I have been running old code in Ruby 2.5.0 (released 2017.12.25) to check for speed and compatibility. I still see the codebase in prime.rb hardly has changed at all (except for replacing Math.sqrt with Integer.sqrt).

To achieve the Ruby 3 goal to make it at least three times faster than Ruby 2 there are three general areas where Ruby improvements can occur.

- increase the speed of its implementation at the machine level
- rewrite its existing codebase in a more efficient|faster manner
- use faster algorithms to implement routines and functions

I want to suggest how to address the later two ways to improve performance of specifically the prime_division method in the prime.rb library.

I've raised and made suggestions to some of these issues here ruby-issues forum and now hope to invigorate additional discussion.

Hopefully with the release of 2.5.0, and Ruby 3 conceptually closer to reality, more consideration will be given to coding and algorithmic improvements to increase its performance too.

Mathematical correctness

First I'd like to raise what I consider *math bugs* in prime_division, in how it handles 0 and -1 inputs.

```
> -1.prime_division
 => [[-1,1]]

> 0.prime_division
Traceback (most recent call last):
        4: from /home/jzakiya/.rvm/rubies/ruby-2.5.0/bin/irb:11:in `<main>'
        3: from (irb):85
        2: from /home/jzakiya/.rvm/rubies/ruby-2.5.0/lib/ruby/2.5.0/prime.rb:30:in `prime_division
'
        1: from /home/jzakiya/.rvm/rubies/ruby-2.5.0/lib/ruby/2.5.0/prime.rb:203:in `prime_divisio
n'
ZeroDivisionError (ZeroDivisionError)
```

First, 0 is a perfectly respectable integer, and is non-prime, so its output should be [], an empty array to denote it has no prime factors. The existing behavior is solely a matter of prime_division's' implementation, and does not take this mathematical reality into account.

The output for -1 is also mathematically wrong because 1 is also non-prime (and correctly returns []), well then mathematically so should -1. Thus, prime_division treats -1 as a new prime number, and factorization, that has no mathematical basis. Thus, for mathematical correctness and consistency -1 and 0 should both return [], as none have prime factors.

```
> -1.prime_division
 => []

> 0.prime_division
 => []
```

```
> 1.prime_division
 => []
```

There's a very simple one-line fix to prime_division to do this:

```
# prime.rb

class Prime

  def prime_division(value, generator = Prime::Generator23.new)
    -- raise ZeroDivisionError if value == 0
    ++ return [] if (value.abs | 1) == 1
```

Simple Code and Algorithmic Improvements

As stated above, besides the machine implementation improvements, the other areas of performance improvements will come from coding rewrites and better algorithms. Below is the coding of prime_division. This coding has existed at least since Ruby 2.0 (the farthest I've gone back).

```
# prime.rb

class Integer

  # Returns the factorization of +self+.
  #
  # See Prime#prime_division for more details.
  def prime_division(generator = Prime::Generator23.new)
    Prime.prime_division(self, generator)
  end

end

class Prime

  def prime_division(value, generator = Prime::Generator23.new)
    raise ZeroDivisionError if value == 0
    if value < 0
      value = -value
      pv = [[-1, 1]]
    else
      pv = []
    end
    generator.each do |prime|
      count = 0
      while (value1, mod = value.divmod(prime)
             mod) == 0
        value = value1
        count += 1
      end
      if count != 0
        pv.push [prime, count]
      end
      break if value1 <= prime
    end
    if value > 1
      pv.push [value, 1]
    end
    pv
  end

end
```

This can be rewritten in more modern and idiomatic Ruby, to become much shorter and easier to understand.

```
require 'prime.rb'
```

```
class Integer
  def prime_division1(generator = Prime::Generator23.new)
    Prime.prime_division1(self, generator)
  end
end

class Prime

  def prime_division1(value, generator = Prime::Generator23.new)
    # raise ZeroDivisionError if value == 0
    return [] if (value.abs | 1) == 1
    pv = value < 0 ? [[-1, 1]] : []
    value = value.abs
    generator.each do |prime|
      count = 0
      while (value1, mod = value.divmod(prime); mod) == 0
        value = value1
        count += 1
      end
      pv.push [prime, count] unless count == 0
      break if prime > value1
    end
    pv.push [value, 1] if value > 1
    pv
  end

end
```

By merely rewriting it we get smaller|concise code, that's easier to understand, which is slightly faster. A *triple win!* Just paste the above code into a 2.5.0 terminal session, and run the benchmarks below.

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

 n = 500_000_000_000_000_000_008_244_213; tm{ pp n.prime_division }
[[3623, 1], [61283, 1], [352117631, 1], [6395490847, 1]]
 => 27.02951016

 n = 500_000_000_000_000_000_008_244_213; tm{ pp n.prime_division1 }
[[3623, 1], [61283, 1], [352117631, 1], [6395490847, 1]]
 => 25.959149721
```

Again, we get a *triple win* to this old codebase by merely rewriting it. It can be made 3x faster by leveraging the prime? method from the OpenSSL library to perform a more efficient|faster factoring algorithm, and implementation.

```
require 'prime.rb'
require 'openssl'

class Integer

  def prime_division2(generator = Prime::Generator23.new)
    return [] if (self.abs | 1) == 1
    pv = self < 0 ? [-1] : []
    value = self.abs
    prime = generator.next
    until value.to_bn.prime? or value == 1
      while prime
        (pv << prime; value /= prime; break) if value % prime == 0
        prime = generator.next
      end
    end
    pv << value if value > 1
    pv.group_by {|prm| prm }.map{|prm, exp| [prm, exp.size] }
  end
```

```
end
```

Here we're making much better use of Ruby idioms and libraries (enumerable and openssl), leading to a much greater performance increase. A bigger *triple win*. Pasting this code into a 2.5.0 terminal session gives the following results.

```
# Hardware: System76 laptop; I7 cpu @ 3.5GHz, 64-bit Linux

def tm; s=Time.now; yield; Time.now-s end

 n = 500_000_000_000_000_000_008_244_213; tm{ pp n.prime_division }
[[3623, 1], [61283, 1], [352117631, 1], [6395490847, 1]]
 => 27.02951016

 n = 500_000_000_000_000_000_008_244_213; tm{ pp n.prime_division1 }
[[3623, 1], [61283, 1], [352117631, 1], [6395490847, 1]]
 => 25.959149721

 n = 500_000_000_000_000_000_008_244_213; tm{ pp n.prime_division2 }
[[3623, 1], [61283, 1], [352117631, 1], [6395490847, 1]]
 => 9.39650374
```

prime_division2 is much more usable for significantly larger numbers and use cases than prime_division. I can even do multiple times better than this, if you review the above cited forum thread.

My emphasis here is to show there are a lot of possible *low hanging fruit* performance gains ripe for the picking to achieve Ruby 3 performance goals, if we look (at minimum) for simpler|better code rewrites, and then algorithmic upgrades.

So the question is, are the devs willing to upgrade the codebase to provide the demonstrated performance increases shown here for prime_division?

History

#1 - 01/23/2018 06:47 AM - shevegen (Robert A. Heiler)

I won't go into all points since your issue makes even my issue requests seem small. :-)

It may be easier to split your suggestions into separate issues though.

For example, the 0 and -1 situation, if it is a bug (probably is but I have not checked the official math definition for prime divisions myself yet), it may be better to split it into another issue and detach it from your other statements made, e. g. the code quality or the speedup gain from optimizing the ruby code.

I don't think that the ruby core devs are against smaller improvements at all, irrespective of the 3.x goal (which I think will be achieved via the JIT/mjit anyway) but it may be simpler to have smaller issues. Some of the bigger issues take longer to resolve. At any rate, that is just my opinion - feel free to ignore it. :)

You could perhaps also add the overall discussion to:

https://bugs.ruby-lang.org/projects/ruby/wiki/DevelopersMeeting20180124Japan

If an attendee notices the issue here (and if it is worth discussing; I think you indirectly also pointed out that some parts of the ruby core/stdlib do not receive equal attention possibly due to a lack of maintainers; I think that one problem may also be that many ruby hackers would not even know which area of core/stdlib may need improvements or attention. Not just the prime division situation you described, but also e. g. improvements to the cgi-part of ruby, even if these are minor - it's not easy to know which areas of standard distributed ruby need improvements.)

#2 - 01/23/2018 07:16 AM - mrkn (Kenta Murata)

Currently, prime_division can factorize any negative integers that are less than -1 like:

```
[2] pry(main)> -12.prime_division
=> [[-1, 1], [2, 2], [3, 1]]
```

Do you think how to treat these cases?

I think raising Math::DomainError is better for 1, 0, and any negative integers cases.

#3 - 01/23/2018 08:46 PM - jzakiya (Jabari Zakiya)

The major problem with prime_division trying to accommodate negative numbers is that, mathematically, prime factorization is really only considered over positive integers > 1.
I understand the creators intent, to be able to reconstruct negative integers from their prime factorization, but that's not what's done mathematically. 1|0 are not primes or composites so they can't be factored (they have no factors). You can see the Numberphile video explanation of this, or if you prefer, the wikipedia one.

From a serious mathematical perspective, prime_division (and the whole prime.rb lib) is inadequate for doing real high-level math. This is why I created the primes-utils gem, so I could do fast math involving primes. I also wrote the Primes-Utils Handbook, to provide and explain all the gem's source code.

The Coureutils library, a part of all [Li|U]nix systems, provides a world class factorization function factor. You're not going to create a Ruby version that will come close to it. I use factor for my default factoring function. Here's an implementation that mimics prime_division.

```
class Integer
   def factors(p=0)   # p is unused variable for method consistency
     return [] if self | 1 == 1
     factors = self < 0 ? [-1] : []
     factors += `factor #{self.abs}`.split(' ')[1..-1].map(&:to_i)
     factors.group_by {|prm| prm }.map {|prm, exp| [prm, exp.size] }
   end
end
```

And here's the performance difference against prime_division2, the fastest version of the previous functions.

```
n = 500_000_000_000_000_000_008_244_213; tm{ pp n.prime_division2 }
[[3623, 1], [61283, 1], [352117631, 1], [6395490847, 1]]
=> 9.39650374

n = 500_000_000_000_000_000_008_244_213; tm{ pp n.factors }
[[3623, 1], [61283, 1], [352117631, 1], [6395490847, 1]]
=> 0.007200317
```

If it were up to me, I would deprecate the whole prime.rb library and use the capabilities in primes-utils, to give Ruby a much more useful|fast prime math library. In fact, I'm doing a serious rewrite of it for version 3.0, using faster math, with faster implementations. When Ruby ever gets true parallel capabilities it'll be capable of making use of it.

But again in general, there are articles|videos showing how to speed up Ruby code. There are projects like Fast Ruby , specifically devoted to identifying and categorizing specific code constructs for speed. These resources can be used for evaluating the core codebase to help rewrite it using the fastest coding constructs.

Ruby is 20+ years old now, and I would imagine some (a lot of?) code has probably never been reviewed|considered for rewriting for performance gains. A lot of similarly aged languages have gone through this process (Python, Perl, PHP) and now it's Ruby's turn.

So while it's natural to talk and focus on the future machine implementation of Ruby 3, I think you can be getting current language speedups by making the ongoing codebase as efficiently and performantly written now, which will translate to an even faster Ruby 3.

This is also a way to get more than just the C guru devs involved in creating Ruby 3. I wouldn't mind evaluating existing libraries for simplicity|speed rewrites if I knew my work would be truly considered. This would provide casual users an opportunity to learn more of the internal workings of Ruby, while contributing to its development, which can only be to Ruby's benefit. How about a Library of the Week|Month or GoSC rewrite project, or a Make Ruby Faster Now

project! Lots of ways to make this effort fun and interesting.

#4 - 01/23/2018 11:45 PM - graywolf (Gray Wolf)

Unless I copy-pasted wrong your prime_division2 is /significantly/ slower for small numbers:

```
$ ruby bm.rb
prime_division   - current in prime.rb
prime_division_2 - proposed version in #14383 by jzakiya
`factor`         - spawning coreutils' factor command

`factor` is left out from the first benchmark, spawning 1_000_000
shells proves nothing.

1_000_000 times of 10

                       user     system      total        real
prime_division     1.496456   0.000059   1.496515 (  1.496532)
prime_division_2 104.094586   6.469827 110.564413 (110.565201)

1_000 times of 2**256

                       user     system      total        real
prime_division     0.069389   0.000000   0.069389 (  0.069391)
prime_division_2   0.325075   0.000000   0.325075 (  0.325088)
`factor`           0.163589   0.073234   0.992784 (  1.021447)

10 times of 500_000_000_000_000_000_008_244_213

                       user     system      total        real
prime_division   328.625069   0.033017 328.658086 (328.801649)
prime_division_2 118.690491   0.000119 118.690610 (118.691372)
`factor`           0.002487   0.000030   0.024145 (  0.025040)
```

script:

```
require 'prime'

require 'openssl'
require 'benchmark'

class Integer
  def prime_division_2(generator = Prime::Generator23.new)
    return [] if (self.abs | 1) == 1
    pv = self < 0 ? [-1] : []
    value = self.abs
    prime = generator.next
    until value.to_bn.prime? or value == 1
      while prime
        (pv << prime; value /= prime; break) if value % prime == 0
        prime = generator.next
      end
    end
    pv << value if value > 1
    pv.group_by {|prm| prm }.map{|prm, exp| [prm, exp.size] }
  end
end

def factor(num)
  return [] if num | 1 == 1
  factors = num < 0 ? [-1] : []
  factors += `factor #{num.abs}`.split(' ')[1..-1].map(&:to_i)
  factors.group_by {|prm| prm }.map {|prm, exp| [prm, exp.size] }
end

puts <<~EOF
prime_division   - current in prime.rb
prime_division_2 - proposed version in #14383 by jzakiya
`factor`         - spawning coreutils' factor command

`factor` is left out from the first benchmark, spawning 1_000_000
shells proves nothing.
EOF
```

```
puts
puts "1_000_000 times of 10"
puts
Benchmark.bm(16) do |x|
  x.report('prime_division')   { 1_000_000.times { 10.prime_division   } }
  x.report('prime_division_2') { 1_000_000.times { 10.prime_division_2 } }
end

puts
puts "1_000 times of 2**256"
puts
Benchmark.bm(16) do |x|
  num = 2**256
  x.report('prime_division')   { 1_000.times { num.prime_division   } }
  x.report('prime_division_2') { 1_000.times { num.prime_division_2 } }
  x.report('`factor`')         { 1_000.times { factor(num)          } }
end

puts
puts "10 times of 500_000_000_000_000_000_008_244_213"
puts
Benchmark.bm(16) do |x|
  num = 500_000_000_000_000_000_008_244_213
  x.report('prime_division')   { 10.times { num.prime_division   } }
  x.report('prime_division_2') { 10.times { num.prime_division_2 } }
  x.report('`factor`')         { 10.times { factor(num)          } }
end
```

#5 - 01/24/2018 03:33 AM - jzakiya (Jabari Zakiya)

Well, I did say "serious" math, didn't I.

```
2.5.0 :097 > 2**256
 => 115792089237316195423570985008687907853269984665640564039457584007913129639936

2.5.0 :099 > n = 2**256 + 1; tm{ pp n.factors }
[[1238926361552897, 1],
 [934616397153579777691635581996068965840512375416381885802803 21, 1]]
 => 11.187528889

2.5.0 :103 > n = 2**256 + 6; tm{ pp n.factors }
[[2, 1],
 [9663703905367, 1],
 [599108221708903554595341427309377510241603132709327340702349 0613, 1]]
 => 181.896643599

2.5.0 :105 > n = 2**256 + 7; tm{ pp n.factors }
[[92243, 1],
 [14633710594132193, 1],
 [39071613028785859, 1],
 [219548092480300808228971712976195385142 3, 1]]
 => 86.285821283

2.5.0 :107 > n = 2**256 + 8; tm{ pp n.factors }
[[2, 3],
 [3, 1],
 [683, 1],
 [4049, 1],
 [85009, 1],
 [2796203, 1],
 [31797547, 1],
 [81776791273, 1],
 [28225515294603308476042620861490152426 89, 1]]
 => 210.062944465

2.5.0 :109 > n = 2**256 + 9; tm{ pp n.factors }
[[5, 1],
 [37181, 1],
 [210150995838577, 1],
 [29638510024302395306764118094101498566030625057480588178 97, 1]]
 => 8.70362184
```

#6 - 01/25/2018 04:26 PM - mame (Yusuke Endoh)

- File bm.rb added

Your prime_division2 uses OpenSSL::BN#prime?. You may know, it is a Miller-Rabin *probabilistic* primality test which may cause a false positive. In short, I suspect that your code may (very rarely) return a wrong result. Am I right? If so, it is unacceptable.

In my personal opinion, lib/prime.rb is not for practical use, but just for fun. The speed is not so important for lib/prime.rb. So I think it would be good to keep it idyllic.

BTW, I've released a gem namely faster_prime, a faster substitute for lib/prime.rb.

```
prime_division_current - current in prime.rb
prime_division_jzakiya - proposed version in #14383 by jzakiya
prime_division_mame    - https://github.com/mame/faster_prime

1_000_000 times of 10

                            user     system      total        real
prime_division_current   1.203975   0.000000   1.203975 (  1.204458)
prime_division_jzakiya  50.366586   8.994353  59.360939 ( 59.369606)
prime_division_mame      1.031790   0.000000   1.031790 (  1.031963)

1_000 times of 2**256

                            user     system      total        real
prime_division_current   0.057273   0.000219   0.057492 (  0.057495)
prime_division_jzakiya   0.230237   0.000000   0.230237 (  0.230288)
prime_division_mame      0.074282   0.000106   0.074388 (  0.074459)

10 times of 500_000_000_000_000_000_008_244_213

                            user     system      total        real
prime_division_current 222.418914   0.032561 222.451475 (222.491855)
prime_division_jzakiya  76.686295   0.000191  76.686486 ( 76.694430)
prime_division_mame      0.102406   0.000000   0.102406 (  0.102408)

$ time ruby -rfaster_prime -e 'p (2**256+1).prime_division'
[[1238926361552897, 1], [934616397153579777691635581996068965840512375416381885802803 21, 1]]

real    2m13.676s
user    2m13.645s
sys 0m0.008s
```

:-)

My gem is written in pure Ruby (not uses OpenSSL), but not so simple, so I have no intention to replace the standard lib/prime.rb, though.

#7 - 01/27/2018 08:30 PM - jzakiya (Jabari Zakiya)

Hi Yusuke.

Ah, we agree, prime.rb is not conducive for doing heavy-duty math. :-)

Please look at and play with my primes-utils gem.
It has a minimal universal useful set of methods for doing prime math.

Again, I'm in the process of rewriting it and adding more methods. Maybe you
want to collaborate with me and give Ruby a much better|serious prime math library.
Ruby is a serious language and deserves much better in areas of scientific and
numerical computation. This need has been recognized by such projects as
SciRuby, and I would like Ruby 3 to be better in these areas.

I installed your faster_prime gem and ran it on my laptop, and started
looking at the code. I haven't thoroughly studied it yet, but may I make
some suggestions to simplify and speed it up.

I don't know if you knew, but starting with Ruby 2.5, it now has Integer.sqrt.
We had a vigorous discussion on this issue here.
So you can replace your code below:

```
module Utils
  module_function

  FLOAT_BIGNUM = Float::RADIX ** (Float::MANT_DIG - 1)
  # Find the largest integer m such that m <= sqrt(n)
  def integer_square_root(n)
    if n < FLOAT_BIGNUM
```

```
      Math.sqrt(n).floor
    else
      # newton method
      a, b = n, 1
      a, b = a / 2, b * 2 until a <= b
      a = b + 1
      a, b = b, (b + n / b) / 2 until a <= b
      a
    end
  end
```

with Integer.sqrt. BTW, here's a fast|accurate pure Ruby implementation of it.

```
class Integer
  def sqrt       # Newton's method version used in Ruby for Integer#sqrt
    return nil if (n = self) < 0  # or however you want to handle this case
    return n if n < 2
    b = n.bit_length
    x = 1 << (b-1)/2 | n >> (b/2 + 1)    # optimum initial root estimate
    while (t = n / x) < x; x = ((x + t) >> 1) end
    x
  end
end
```

Also in same file, you can do this simplification:

```
def mod_sqrt(a, prime)
  return 0 if a == 0

  case
  when prime == 2
    a.odd? ? 1 : 0   =>   a & 1   =>  or better =>  a[0]  # lsb of number
  ....
```

Finally, in prime_factorization.rb we can simplify and speedup code too. Don't count the primes factors when you find then, just stick then in an array, then when finished with factorization process you can use Enumerable#group_by to format output of them in one step. Saves code size|complexity, and is faster.

I also restructured the code like mine to make it simpler. Now you don't have to do so many tests, and you get rid of all those individual yields, which complicates and slow things down. You will have to modify the part of your code that uses prime_factorization, but that code should be simpler|faster too.

```
require "faster_prime/utils"

module FasterPrime
  module PrimeFactorization
    module_function

    # Factorize an integer
    def prime_factorization(n)
      return enum_for(:prime_factorization, n) unless block_given?

      return if n == 0
      if n < 0
        yield [-1, 1]
        n = -n
      end

      SMALL_PRIMES.each do |prime|
        if n % prime == 0
          c = 0
          begin
            n /= prime
            c += 1
          end while n % prime == 0
          yield [prime, c]
        end
        if prime * prime > n
          yield [n, 1] if n > 1
          return
        end
        return if n == 1
      end
```

```
      if PrimalityTest.prime?(n)
        yield [n, 1]
      else
        d = nil
        until d
          [PollardRho, MPQS].each do |algo|
            begin
              d = algo.try_find_factor(n)
            rescue Failed
            else
              break
            end
          end
        end

        pe = Hash.new(0)
        prime_factorization(n / d) {|p, e| pe[p] += e }
        prime_factorization(d) {|p, e| pe[p] += e }
        pe.keys.sort.each do |p|
          yield [p, pe[p]]
        end
      end
    end
  end
```

Change to below, with appropriate changes for factoring algorithm inside loop.

```
require "faster_prime/utils"

module FasterPrime
  module PrimeFactorization
    module_function

    # Factorize an integer
    def prime_factorization(n)
      return enum_for(:prime_factorization, n) unless block_given?

      # 'factors' will hold the number of individual prime factors
      return [] if n.abs | 1 == 1    # for n = -1, 0, or 1
      factors = n < 0 ? [-1] | []    # if you feel compelled for negative nums
      n = n.abs

      SMALL_PRIMES.each do |prime|   # extract small prime factors, if any
        (factors << prime; n /= prime) while n % prime == 0
      end

      # at this point n is either a prime, 1, or a composite of non-small primes
      until PrimalityTest.prime?(n) or n == 1   # exit when n is 1 or prime
        # if you're in here then 'n' is a composite thats needs factoring
        # when you find a factor, stick it in 'factors' and reduce 'n' by it
        # ultimately 'n' will be reduced to a prime or 1

        # do whatever needs to be done to make this work right
        d = nil
        until d
          [PollardRho, MPQS].each do |algo|
            begin
              d = algo.try_find_factor(n)
            rescue Failed
            else
              break
            end
          end
        end
      end

      # at this point 'n' is either a prime or 1
      factors << n if n > 1         # stick 'n' in 'factors' if it's a prime
      # 'factors' now has all the number of individual prime factors
      # now use Enumerable#group_by to make life simple and easy :-)
      # the xx.sort is unnecessary if you find the prime factors sequentially
      factors.group_by(&:itself).sort.map { |prime, exponents| [prime, exponents.size] }
    end
  end
```

This is now a generic template for factorization. To upgrade just use better|faster
PrimalityTest and factorization functions. You also see it has separated each
distinct algorithmic function into single area of concern, in a conceptually more
functional programming style. This is now also able to be possibly implemented in
parallel because of the isolation of functional areas of concern.

#8 - 01/28/2018 09:21 PM - jzakiya (Jabari Zakiya)

Also, FYI, to go along with using Integer.sqrt, you can save some code (and increase performance)
using the [OpenSSL library](), which already has some of the methods in utils.rb.

```
                            require 'openssl'

mod_pow(a,b,n)       =>     a.to_bn.mod_exp(b,n)

mod_inv(a,n)         =>     a.to_bn.mod_inverse(n)
```

#9 - 01/29/2018 06:21 AM - yugui (Yuki Sonoda)

- Target version set to Next Major

- Assignee set to yugui (Yuki Sonoda)

Faster is better -- with certain conditions. Considering those conditions, I would like to take the following approach.

1. Remove the dependency from mathn.rb to prime.rb
2. Move prime.rb from lib/ to a bundled gem
3. Apply some of the patches.

This is because of the following conditions I would like the change to satisfy.

1. Simplicity of the dependency graph in the standard library
    - I don't want to let mathn have too much dependency.
2. Keeping the focus of the standard library
    - We are moving more libraries from lib/ to bundled gems. The solution must be consistent to this trend.
3. Certain degree of backward compatibility
    - the algorithm must be deterministic.

#10 - 01/29/2018 10:30 PM - jzakiya (Jabari Zakiya)

Thank you for taking this project into consideration.

FYI, there is C open source code for both the Coreutils [factor]() function and the [APR-CL]() deterministic
primality test. However, [Miller-Rabin]() is faster, and can be implemented as deterministic, at least up
to 25-digits, so maybe a good Integer#prime? method should be a hybrid combination, depending on number size.
The documentation for the APR-CL code says it's good up to 6021-digits on 64-bit systems
(not that anyone should expect to process those size numbers in any reasonable times).

To make sense of all these possibilities, a determination of the max number size|range has
to be made, since we are talking about an infinite set of numbers (primes). I wouldn't
think Ruby (or any general language) would have a primes library that will find the
largest [Mersenne Primes](), for example. However, with the right math and implementation,
thousands of digits numbers can be easily worked with within reasonable times and memory usage.

As an example, below is output of my Integer#primesmr method, for numerating primes
within a number range, here for 100, 1000, and 2000 digit numbers. It combines the M-R
test with my math techniques using prime generators to identify a minimum set of
prime candidates within the ranges, and then checking their primality.

This is part of my [primes-utils]() gem, which has a [Primes-Utils Handbook](),
which explains all the math, and provides the full gem source code.

It so far has the methods primes|mr|f, primescnt|mr|f, nthprime, prime|mr? and
factors|prime_division in current version 2.7. For the (next) 3.0 release I'm adding
next_prime and prev_prime too. I think these are a good (minimum) set of general
purpose methods that provide the majority of information most people would seek in
a primes library. I'll probably create single hybrid methods also where now I have multiple
versions. I'm playing with JRuby for using parallel algorithms I already have implemented
in C++/Nim until CRuby can provide capability.

Again, thanks for the Ruby team agreeing to take on this effort.

```
2.5.0 :017 > n= (10**100+267); tm{ p n.primesmr n+5000 }
[10000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000267, 100000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000949, 1000000000000
```

```
0000000000000000000000000000000000000000000000000000000000000000000000000000000000001243, 10000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000001293, 100000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000001983, 1000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000002773, 10000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000002809, 100000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000002911, 1000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000002967, 10000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000003469, 100000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000003501, 1000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000003799, 10000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000004317, 100000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000004447, 1000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
4491]
 => 0.06310091
2.5.0 :018 > n= (10**1000+267); tm{ p n.primesmr n+5000 }
[1000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000453, 100000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000001357, 10000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000002713, 1000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000004351]
 => 11.089284953
2.5.0 :019 > n= (10**2000+267); tm{ p n.primesmr n+5000 }
[1000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000004561]
 => 56.129938705
2.5.0 :020 >
```

#11 - 01/30/2018 06:01 PM - jzakiya (Jabari Zakiya)

FYI, I re-ran the examples above (and an additional one) using the Miller-Rabin implementation in my primes-utls 3.0 development branch. Not only is it deterministic up to about 25-digits, but it's way faster too. It can probably be made faster by using a WITNESS_RANGES list with fewer witnesses, and can now be easily extended as optimum witnesses are found for larger number ranges.

I provide this to raise the issue that absolute determinism for a primality test function maybe shouldn't be an absolute criteria for selection. Of course, it is the ideal, but sometimes striving for perfection can be the enemy of good enough.

Again, hybrid methods may be able to combine the best of all worlds.

```
> n= (10**100+267); tm{ p n.primesmr n+5000 }
 => 0.056422744 3.0 dev
 => 0.06310091  2.7

> n= (10**1000+267); tm{ p n.primesmr n+5000 }
 => 7.493292636  3.0 dev
 => 11.089284953 2.7

> n= (10**2000+267); tm{ p n.primesmr n+5000 }
 => 36.614877874 3.0 dev
 => 56.129938705 2.7

> n= (10**3000+267); tm{ p n.primesmr n+5000 }  => 10000000000........1027
 => 192.866720666 3.0 dev
 => 286.244139793 2.7

# Miller-Rabin version in Primes-Utils 2.7

def primemr?(k=20)  # increase k for more reliability
  n = self.abs
  return true  if [2,3].include? n
  return false unless [1,5].include?(n%6) and n > 1

  d = n - 1
  s = 0
  (d >>= 1; s += 1) while d.even?
  k.times do
    a = 2 + rand(n-4)
    x = a.to_bn.mod_exp(d,n)     # x = (a**d) mod n
    next if x == 1 or x == n-1
    (s-1).times do
      x = x.mod_exp(2,n)         # x = (x**2) mod n
      return false if x == 1
      break if x == n-1
    end
    return false if x != n-1
  end
  true  # n is prime (with high probability)
end

# Miller-Rabin version in Primes-Utils 3.0 dev

# Returns true if +self+ is a prime number, else returns false.
def primemr?
  primes = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43]
  return primes.include? self if self <= primes.last
  return false unless primes.reduce(:*).gcd(self) == 1
  wits = WITNESS_RANGES.find {|range, wits| range > self}  # [range, [wit_prms]] or nil
  witnesses = wits && wits[1] || primes
  witnesses.each {|p| return false unless miller_rabin_test(p) }
  true
end

private
# Returns true if +self+ passes Miller-Rabin Test on witness +b+
def miller_rabin_test(b)             # b is a witness to test with
  n = d = self - 1
  d >>= 1 while d.even?
  y = b.to_bn.mod_exp(d, self)       # x = (b**d) mod n
  until d == n || y == n || y == 1
    y = y.mod_exp(2, self)           # y = (y**2) mod self
```

```
    d <<= 1
  end
  y == n || d.odd?
end

WITNESS_RANGES = {
  2_047 => [2],
  1_373_653  => [2, 3],
  25_326_001 => [2, 3, 5],
  3_215_031_751 => [2, 3, 5, 7],
  2_152_302_898_747 => [2, 3, 5, 7, 11],
  3_474_749_660_383 => [2, 3, 5, 7, 11, 13],
  341_550_071_728_321 => [2, 3, 5, 7, 11, 13, 17],
  3_825_123_056_546_413_051 => [2, 3, 5, 7, 11, 13, 17, 19, 23],
  318_665_857_834_031_151_167_461 => [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37],
  3_317_044_064_679_887_385_961_981 => [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41]
}
```

#12 - 01/31/2018 03:56 PM - jzakiya (Jabari Zakiya)

This version is probably better, as it's a little faster, and takes a complete
list of witnesses for a given number, which can be determined separately.

```
    # Returns true if +self+ is a prime number, else returns false.
    def primemr?
      primes = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43]
      return primes.include? self if self <= primes.last
      return false unless primes.reduce(:*).gcd(self) == 1
      wits = WITNESS_RANGES.find {|range, wits| range > self} # [range, [wit_prms]] or nil
      witnesses = wits && wits[1] || primes
      miller_rabin_test(witnesses)
    end
```

```
    private
    # Returns true if +self+ passes Miller-Rabin Test on witness +b+
    def miller_rabin_test(witnesses)     # use witness list to test with
      neg_one_mod = n = d = self - 1
      d >>= 1  while d.even?
      witnesses.each do |b|
        s = d
        y = b.to_bn.mod_exp(d, self)     # y = (b**d) mod self
        until s == n || y == 1 || y == neg_one_mod
          y = y.mod_exp(2, self)         # y = (y**2) mod self
          s <<= 1
        end
        return false unless y == neg_one_mod || s.odd?
      end
      true
    end
```

Files

bm.rb	1.94 KB	01/25/2018	mame (Yusuke Endoh)