Ruby master - Feature #12676

Significant performance increase, and code conciseness, for prime_division method in prime.rb

08/15/2016 01:48 AM - jzakiya (Jabari Zakiya)

Status:	Assigned	
Priority:	Normal	
Assignee:	marcandre (Marc-Andre Lafortune)	
Target version:		

Description

I earlier posted code to simplify the prime_division method in prime.rb.
This made the code much more concise and readable/understandable, while
also providing a small speed increase.

The code presented here for prime_division2 maintains the conciseness and
readability, but uses a different/simpler algorithm to provide a significant
speed increase over the current implementation of prime_division in prime.rb.

Timings for selected large primes are provided, run on CRuby 2.3.1.
System: System76 3.5GHz I7 cpu laptop, Linux 64-bit OS in Virtual Box.

```
n1 =   100_000_000_000_000_003
n2 =   200_000_000_000_000_003
n3 = 1_000_000_000_000_000_003
```

```
                        n1         n2         n3
prime_division         23.7       33.5       74.6
prime_division1        22.9       32.2       72.8
prime_division2        14.8       20.5       45.8
```

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

irb(main):015:0> n =   100_000_000_000_000_003; tm{ n.prime_division }
=> 23.730239721
irb(main):016:0> n =   100_000_000_000_000_003; tm{ n.prime_division1 }
=> 22.877657172
irb(main):017:0> n =   100_000_000_000_000_003; tm{ n.prime_division2 }
=> 14.758561827

irb(main):018:0> n =   200_000_000_000_000_003; tm{ n.prime_division }
=> 33.502851342
irb(main):019:0> n =   200_000_000_000_000_003; tm{ n.prime_division1 }
=> 32.23911595
irb(main):020:0> n =   200_000_000_000_000_003; tm{ n.prime_division2 }
=> 20.476660683

irb(main):021:0> n = 1_000_000_000_000_000_003; tm{ n.prime_division }
=> 74.630244055
irb(main):022:0> n = 1_000_000_000_000_000_003; tm{ n.prime_division1 }
=> 72.778948947
irb(main):023:0> n = 1_000_000_000_000_000_003; tm{ n.prime_division2 }
=> 45.802756121
```

1. Current code for prime_division in prime.rb.

```
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
```

2. Code simplification for current algorithm, increases conciseness/readability, with slight speedup.

```
def prime_division1(value, generator = Prime::Generator23.new)
  raise ZeroDivisionError if value == 0
  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
```

3. Change of algorithm, maintains conciseness/readability with significant speed increase.

```
def prime_division2(value, generator = Prime::Generator23.new)
  raise ZeroDivisionError if value == 0
  pv = value < 0 ? [-1] : []
  value  = value.abs
  sqrt_value = Math.sqrt(value).to_i
  generator.each do |prime|
    break if prime > sqrt_value
    while value % prime == 0
      pv << prime
      value /= prime
      sqrt_value = Math.sqrt(value).to_i
    end
  end
  pv << value if value > 1
  pv.group_by {|prm| prm }.map{|prm, exp| [prm, exp.size] }
end
```

History

#1 - 08/15/2016 03:53 AM - jzakiya (Jabari Zakiya)

- Tracker changed from Bug to Feature

#2 - 08/17/2016 06:19 PM - jzakiya (Jabari Zakiya)

By using a faster prime generator than the current one in prime.rb the performance for
the prime_division function can reach the desired 3x performance sought for Ruby 3 (3x3).

```
Timing comparisons for optimal method prime_division2 using different prime generators.
Prime::Generator23  is the current generator in prime.rb.
Prime::GeneratorP17 is fastest generator determined by timing all the base primes up to 23.

n1 =   100_000_000_000_000_003
n2 =   200_000_000_000_000_003
n3 = 1_000_000_000_000_000_003

                  n1         n2         n3
Generator23      14.2       19.7       44.6
GeneratorP17      8.8       12.2       27.8

irb(main):013:0> n =   100_000_000_000_000_003; tm{ n.prime_division2 Prime::Generator23.new}
=> 14.170281965
irb(main):014:0> n =   100_000_000_000_000_003; tm{ n.prime_division2 Prime::GeneratorP17.new}
=> 8.838717755

irb(main):015:0> n =   200_000_000_000_000_003; tm{ n.prime_division2 Prime::Generator23.new}
=> 19.664830177
irb(main):016:0> n =   200_000_000_000_000_003; tm{ n.prime_division2 Prime::GeneratorP17.new}
=> 12.209209681

irb(main):017:0> n = 1_000_000_000_000_000_003; tm{ n.prime_division2 Prime::Generator23.new}
=> 44.635148933
irb(main):018:0> n = 1_000_000_000_000_000_003; tm{ n.prime_division2 Prime::GeneratorP17.new}
=> 27.824091074

class Integer

  def prime_division2(generator = Prime::GeneratorP17.new)
    Prime.prime_division2(self, generator)
  end

end

class Prime

  def prime_division2(value, generator = Prime::GeneratorP17.new)
    raise ZeroDivisionError if value == 0
    pv = value < 0 ? [-1] : []
    value  = value.abs
    sqrt_value = Math.sqrt(value).to_i
    generator.each do |prime|
      break if prime > sqrt_value
      while value % prime == 0
        pv << prime
        value /= prime
        sqrt_value = Math.sqrt(value).to_i
      end
    end
    pv << value if value > 1
    pv.group_by {|prm| prm }.map{|prm, exp| [prm, exp.size] }
  end

  # Generates all integers which are greater than 2 and
  # are not divisible by either 2 or 3.
  #
  # This is a pseudo-prime generator, suitable on
  # checking primality of an integer by brute force
  # method.
  class Generator23 < PseudoPrimeGenerator
    def initialize
      @prime = 1
      @step = nil
      super
    end

    def succ
      if (@step)
        @prime += @step
        @step = 6 - @step
      else
        case @prime
        when 1; @prime = 2
        when 2; @prime = 3
        when 3; @prime = 5; @step = 2
        end
      end
      @prime
    end

    alias next succ

    def rewind
      initialize
    end
  end

  # P17 Strictly-Prime Prime Generator (SP PG).
  # Generates all the integers greater than 17 that
  # are not divisible by any prime <= 17.
  #
  # This is a pseudo-prime generator, suitable for
  # checking primality of an integer by brute force method.
  class GeneratorP17 < PseudoPrimeGenerator
    def initialize
      init_generator unless @current_prime
      @current_prime = 1
      super
    end

    # Initialize parameters for P17 Strictly-Prime Prime Generator (SP PG).
    # Increase/decrease base primes to pick a different SP PG.
    def init_generator
      base_primes = [2, 3, 5, 7, 11, 13, 17]
      @pg_mod   = base_primes.reduce(:*)
      @residues = base_primes.dup
      base_primes.last.step(@pg_mod, 2) {|r| @residues << r if @pg_mod.gcd(r) == 1}
      @residues << @pg_mod + 1
      @first_residues_index = base_primes.size
      @last_residues_index  = @residues.size - 1
      @modk = @r = 0
    end

    # Finds/returns next pseudo-prime, and updates pointer to the next one.
    def succ
      (@modk = 0;  @r = -1) if @current_prime < 2
      @r += 1
      (@r = @first_residues_index; @modk += @pg_mod) if @r > @last_residues_index
      @current_prime = @modk + @residues[@r]
    end

    alias next succ

    def rewind
      initialize
    end
  end
end
```

#3 - 08/19/2016 07:59 PM - jzakiya (Jabari Zakiya)

An additional change to the implementation of prime_division2 provides an
another 3x increase in speed over using the generators implemented as classes.

Using class based generators, and enumerable methods, incur a metaprogramming overhead
performance hit. Creating the generators as straight methods provides more flexibility
in their use, greatly reduces the code base, and significantly increases the speed of
the prime_division method.

Below is code for prime_division3 which uses the approach of prime_division2, but creates
the prime generator parameters as a simple method.

Timings show that the code/methodology for prime_division3 is now roughly 6x the speed of
the current prime_division in prime.rb, which leaps past the goal of Ruby 3x3 for Ruby 3
to have a 3x performance increase. These performance gains will become even greater with
any language speedups developed for Ruby 3.

Testing on both 32|64 bit Linux OS systems show P17 gives the fastest results overall.
This is a physical limitation of Ruby, not a mathematical one, as the residue array sizes

past P17 exceed a million elements. If this could be improved, using P19 (and greater)
would provide even higher performance. Even still, P19 is faster for (very) large numbers
past a certain size, as it reduces the number of prime candidates (pseudo primes) to perform
brute force testing with from P17's 18.05% of all integers to P19's 17.1%.

For the mathematical basis of this see the Primes Utils Handbook.
https://www.scribd.com/document/266461408/Primes-Utils-Handbook

```
n1 =   100_000_000_000_000_003
n2 =   200_000_000_000_000_003
n3 = 1_000_000_000_000_000_003

G23  = Prime::Generator23.new
GP17 = Prime::GeneratorP17.new

                       n1         n2         n3
prime_division        23.7       33.5       74.6
prime_division1       22.9       32.2       72.8
prime_division2 G23   14.2       19.7       44.6
primv_division2 GP17   8.8       12.2       27.8
prime_division3 2     12.5       17.5       39.3
prime_division3 3      7.7       10.9       24.2
prime_division3 5      6.0        8.6       18.9
prime_division3 7      4.9        6.9       15.6
prime_division3 11     4.6        6.3       14.2
prime_division3 13     4.2        5.7       12.9
prime_division3 17     3.9        5.5       12.1     fastest for this range of numbers
prime_division3 19     5.2        6.9       12.8     will become faster past some larger n

class Integer
  def prime_division3(pg_selector = 17)
    raise ZeroDivisionError if self == 0
    residues, base_primes, mod = init_generator(pg_selector)
    residues = base_primes + residues
    r1 = base_primes.size      # first_residue_index
    rn = residues.size - 1     # last_residue_index
    modk = r = 0

    pv = self < 0 ? [-1] : []
    value = self.abs
    sqrt_value = Math.sqrt(value).to_i
    while (prime = modk + residues[r]) <= sqrt_value
      while value % prime == 0
        pv << prime
        value /= prime
        sqrt_value = Math.sqrt(value).to_i
      end
      r += 1; (r = r1; modk += mod) if r > rn
    end
    pv << value if value > 1
    pv.group_by {|prm| prm }.map {|prm, exp| [prm, exp.size] }
  end

  private
  def init_generator(pg_selector)
    base_primes = [2, 3, 5, 7, 11, 13, 17, 19]
    pg_selector = 17 unless base_primes.include? pg_selector
    base_primes.select! {|prm| prm <= pg_selector }
    mod = base_primes.reduce(:*)
    residues = []; 3.step(mod, 2) {|r| residues << r if mod.gcd(r) == 1 }
    [residues << mod + 1, base_primes, mod]
  end
end
```

#4 - 08/22/2016 06:24 PM - jzakiya (Jabari Zakiya)

We can make it faster across ranges, with just a little bit more code,
by adaptively selecting the Prime Generator to use based on the number size.
The method select_pg has a profile of the best PG to use for given number ranges.
It's added to init_generator (along with the number), and provides an adaptively
selected pg, instead of a hard default previously used.

In prime_division4, if no pg_selector is given then its chosen by select_pg;
otherwise, if a valid pg_select value is given then it's used.

```
class Integer
  def prime_division4(pg_selector = 0)
    raise ZeroDivisionError if self == 0
    residues, base_primes, mod = init_generator1(self, pg_selector)
    residues = base_primes + residues
    r1 = base_primes.size      # first_residue_index
    rn = residues.size - 1     # last_residue_index
    modk = r = 0

    factors = self < 0 ? [-1] : []
    n = self.abs
    sqrt_n = Math.sqrt(n).to_i
    while (p = modk + residues[r]) <= sqrt_n
      while n % p == 0; factors << p; n /= p; sqrt_n = Math.sqrt(n).to_i end
      r += 1; (r = r1; modk += mod) if r > rn
    end
    factors << n if n > 1
    factors.group_by {|prm| prm }.map {|prm, exp| [prm, exp.size] }
  end

  private
  def init_generator1(num, pg_selector)
    base_primes = [2, 3, 5, 7, 11, 13, 17, 19, 23]
    pg_selector = select_pg(num.abs) unless base_primes.include? pg_selector
    base_primes.select! {|prm| prm <= pg_selector }
    mod = base_primes.reduce(:*)
    residues = []; 3.step(mod, 2) {|r| residues << r if mod.gcd(r) == 1 }
    [residues << mod + 1, base_primes, mod]
  end

  def select_pg(num)   # adaptively select fastest SP Prime Generator
    return 5  if num < 1 * 10**7  + 1000
    return 7  if num < 1 * 10**10 + 1000
    return 11 if num < 1 * 10**13 + 1000
    return 13 if num < 7 * 10**15 + 1000
    return 17 if num < 4 * 10**18 + 1000
    19
  end
end
```

#5 - 08/24/2016 03:57 PM - jzakiya (Jabari Zakiya)

One last simple tweek to increase overall peformance, in prime_division5.
Instead of selecting the optimum pg based on the number's size, first
suck out any factors of some base primes, then determine the optimum
pg based on the sqrt of the reduced factored number.

This significantly speedups large factorable numbers (while maintaining
the same performance for large primes) by choosing the optimun pg for
smaller numbers resulting from the factoring by the base primes.

```
class Integer
  def prime_division5(pg_selector = 0)
    raise ZeroDivisionError if self == 0
    base_primes = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
    pv = self < 0 ? [-1] : []
    value = self.abs
    base_primes.each {|prm| (pv << prm; value /= prm) while value % prm == 0 }
    sqrt_value = Math.sqrt(value).to_i
    num = self.abs == value ? value : sqrt_value
    residues, *, mod = init_generator1(num, pg_selector)
    rn = residues.size - 1;       # last_residue_index
    modk = r = 0

    while (prime = modk + residues[r]) <= sqrt_value
      while value % prime == 0;
        pv << prime
        value /= prime
        sqrt_value = Math.sqrt(value).to_i
      end
      r +=1; (r = 0; modk += mod) if r > rn
    end
    pv << value if value > 1
    pv.group_by {|prm| prm }.map{|prm, exp| [prm, exp.size] }
  end
  private
  def init_generator1(num, pg_selector)
    base_primes = [2, 3, 5, 7, 11, 13, 17, 19, 23]
    pg_selector = select_pg(num.abs) unless base_primes.include? pg_selector
    # puts "Using P#{pg_selector}"
    base_primes.select! {|prm| prm <= pg_selector }
    mod = base_primes.reduce(:*)
    residues = []; 3.step(mod, 2) {|r| residues << r if mod.gcd(r) == 1 }
    [residues << mod + 1, base_primes, mod]
  end

  def select_pg(num)    # adaptively select fastest SP Prime Generator
    return 5  if num < 1 * 10**7  + 1000
    return 7  if num < 1 * 10**10 + 1000
    return 11 if num < 1 * 10**13 + 1000
    return 13 if num < 7 * 10**15 + 1000
    return 17 if num < 4 * 10**18 + 1000
    19
  end
end
```

#6 - 08/25/2016 01:57 AM - hsbt (Hiroshi SHIBATA)

- Assignee set to yugui (Yuki Sonoda)

- Status changed from Open to Assigned

#7 - 08/25/2016 01:04 PM - jzakiya (Jabari Zakiya)

Question:
Why do you raise an error for the value '0'?

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

Why not also:

```
0.prime_division => []
```

This is more mathematically consistent because neither
are prime so neither have prime factors, and you can't
reconstruct either with Integer.from_prime_division.

'0' is a perfectly valid integer that shouldn't raise an error.

```
return [] if value >= 0 && value < 2
```

#8 - 08/25/2016 05:27 PM - jzakiya (Jabari Zakiya)

This is neater.

replace

```
raise ZeroDivisionError if self == 0
```

with

```
return [] if self | 1 == 1
```

#9 - 08/28/2016 08:00 PM - jzakiya (Jabari Zakiya)

OK, here's how to achieve what I believe is the ultimate brute force factoring technique,
by using more resources from the Ruby Standard Library.

If a number is prime we know we don't need to try and factor it.
Using the very fast --prime?-- method from the OPENSSL library, we first check
if the number is prime, and return [[self, 1]] if it is.

If the number isn't prime then we do the following, as shown in prime_division6`.

Whenever we find a factor of the number, we reduce the number by that factor and
then check if the new number is prime or '1'. If it's either of these we can stop
factoring. So in prime_division6, inside the while loop, when a factor is found,
we save it, reduce the number by it, take its square root, and jump out of the while
loop into the until loop, which checks if the reduced number is prime or 1. If not,
we use the same value of prime to start the while loop again (to check for multiple

factors of that prime). When that prime has no (more) factors of the number, we form
the next prime candidate (pseudo prime) and continue until finished.

This is now much, much faster, and allows for factoring (and prime determination)
of extremely large numbers, in reasonable times compared to the prior methods.

Below are some timing comparisons of factorings of some 21 digit numbers.

```
2.3.1 :006 > n = 500_000_000_000_000_000_009; n.prime_division6
 => [[500000000000000000009, 1]]

2.3.1 :007 > n = 500_000_000_000_000_000_010; n.prime_division6
Using P7
 => [[2, 1], [3, 1], [5, 1], [155977777, 1], [106852828571, 1]]

2.3.1 :008 > n = 500_000_000_000_000_000_011; n.prime_division6
Using P7
 => [[7, 1], [49009, 1], [1457458251108397, 1]]

2.3.1 :009 > n = 500_000_000_000_000_000_012; n.prime_division6
Using P11
 => [[2, 2], [482572373, 1], [259028504311, 1]]
```

Lenovo laptop, I5 2.3 GHz, 32-bit Linux OS system.

```
2.3.1 :002 > def tm; s = Time.now; yield; Time.now-s end
 => :tm

2.3.1 :003 > n = 500_000_000_000_000_000_009; tm{ n.prime_division6 }
 => 0.000715062

2.3.1 :004 > n = 500_000_000_000_000_000_010; tm{ n.prime_division6 }
Using P7
 => 15.500916273
2.3.1 :005 > n = 500_000_000_000_000_000_011; tm{ n.prime_division6 }
Using P7
 => 0.006549972
2.3.1 :006 > n = 500_000_000_000_000_000_012; tm{ n.prime_division6 }
Using P11
 => 44.197956157
```

System76 laptop, I7 3.5 GHz, Virtual Box 64-bit Linux OS system.

```
2.3.1 :052 > n = 500_000_000_000_000_000_009; tm{ n.prime_division6 }
 => 0.00027761

2.3.1 :053 > n = 500_000_000_000_000_000_010; tm{ n.prime_division6 }
Using P7
 => 6.524573098
2.3.1 :054 > n = 500_000_000_000_000_000_011; tm{ n.prime_division6 }
Using P7
 => 0.005872674
2.3.1 :055 > n = 500_000_000_000_000_000_012; tm{ n.prime_division6 }
Using P11
 => 19.085550067
```

Here's the complete code for prime_division6.

```
require 'openssl'

class Integer
  def prime_division6(pg_selector = 0)
    return [] if self | 1 == 1
    return [[self, 1]] if self.to_bn.prime?
    base_primes = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
    pv = self < 0 ? [-1] : []
    value = self.abs
    base_primes.each {|prm| (pv << prm; value /= prm) while value % prm == 0 }
    sqrt_value = Math.sqrt(value).to_i
    num = self.abs == value ? value : sqrt_value
    residues, *, mod = init_generator1(num, pg_selector)
    rn = residues.size - 1;       # last_residue_index
    modk = r = 0

    until value.to_bn.prime? or value == 1
```

```
      while (prime = modk + residues[r]) <= sqrt_value
        if value % prime == 0;
          pv << prime
          value /= prime
          sqrt_value = Math.sqrt(value).to_i
          break
        end
      r += 1; (r = 0; modk += mod) if r > rn
      end
    end
    pv << value if value > 1
    pv.group_by {|prm| prm }.map{|prm, exp| [prm, exp.size] }
  end

  private
  def init_generator1(num, pg_selector)
    base_primes = [2, 3, 5, 7, 11, 13, 17, 19, 23]
    pg_selector = select_pg(num.abs) unless base_primes.include? pg_selector
    # puts "Using P#{pg_selector}"
    base_primes.select! {|prm| prm <= pg_selector }
    mod = base_primes.reduce(:*)
    residues = []; 3.step(mod, 2) {|r| residues << r if mod.gcd(r) == 1 }
    [residues << mod + 1, base_primes, mod]
  end

  def select_pg(num)   # adaptively select fastest SP Prime Generator
    return 5  if num < 1 * 10**7  + 1000
    return 7  if num < 1 * 10**10 + 1000
    return 11 if num < 1 * 10**13 + 1000
    return 13 if num < 7 * 10**15 + 1000
    return 17 if num < 4 * 10**18 + 1000
    19
  end
end
```

#10 - 08/29/2016 04:56 AM - nobu (Nobuyoshi Nakada)

- Description updated

Jabari Zakiya wrote:

```
    return [] if self | 1 == 1
```

It seems an unnecessarily heavy operation when self is a huge integer.

#11 - 08/29/2016 06:53 PM - jzakiya (Jabari Zakiya)

Ah, but let's not forget the context that brings us here.

With prime_division6 we have blown past my intended purpose to make prime_division at
least 3x faster, to meet the Ruby 3x3 goal. We are now orders of magnitude faster than
the current implementation, which now creates the capability of processing previously
unthinkable large numbers in reasonable times.

Actually, the code:

```
self | 1 == 1
```

is quite easy to do in hardware. It just sets the lsb of the number to '1' and if
any other bits are '1' then it's 'false'. I don't understand the concern about using
it for large numbers because other calculations will use the initial large number as well.
It certainly won't affect anything (technically).

I changed the functionality to this implementation because it requires less cpu work,
and is shorter than the original functionally equivalent snippet, and also looks better (to me).

The main thing for me is the recognition that '0' and '1' should produce the same answer of [].

To support this, all *nix OS systems come with the cli command 'factor', part of the Unix
coreutils standard library. Doing the following gives the same (mathematically correct) answers.

```
jzakiya@jzakiya-VirtualBox ~ $ factor 0
0:
jzakiya@jzakiya-VirtualBox ~ $ factor 1
```

```
1:
jzakiya@jzakiya-VirtualBox ~ $ factor 2
2: 2
```

Thus, when use ask the question what are the factors of '0' or '1' the answer is 'none'.

In fact, if you want to make the prime_division function as fast as possible we can use the 'factor' command to implement it with, as shown below (this is what my 'primes-utils' gem uses if it detects the OS is a *nix variant).

This will give instantaneous results until you exceed the allowable number size for 'factor'.

```
class Integer
  def factors
    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
```

So instead of just leveraging the resources of the Ruby Standard Library we can also leverage the resources of the Unix standard environment.

Obviously, we can't make a factoring function that can work on every arbitrary large number, on a real physical computer, within a realistic time frame for computation.

But prime_divsion6|factors more than exceed the goals for Ruby 3x3, which for me, and what I use Ruby for, makes it much more attractive|usable to applied math and science domains, and engineering.

Lastly, (on this topic) I would suggest changing (or providing an alias) to the name 'prime_division'.

What the method is functionally doing is providing a list of a number's prime factors,
but the current name is focusing on a process mechanism. I think it's clearer to use the following:

Instead of:

```
   n.prime_division
```

rename it to:

```
   n.prime_factors
```

or what I like the best (because it's shorter and more consistent with the Unix 'factor' command):

```
   n.factors
```

Then you can rename:

```
   Integer.from_prime_division
```

to a more descriptive

```
   Integer.from_prime_factors
```

And lastly, lastly, on a related issue, I also suggest the following.

The technique in prime_division6 is dependent on using a fast 'prime?' method, which Ruby comes with in its OPENSSL standard library.

I would suggest/urge using it in the prime.rb library as it's 'prime?' method.

I raised this issue of why create/use slow implementations for a primality tester when Ruby already has a fast one.

See: https://bugs.ruby-lang.org/issues/12673

I would suggest doing this in prime.rb for 'prime?', which keeps it and 'prime_division(6)' in one nice neat place. Then, if you find/create a faster implementation of 'prime?' only one method has to change without changing the semantics (or dependencies) on any other method that uses it.

```
require 'openssl'

class Integer

  def prime?
```

```
    self.to_bn.prime?
  end

  def prime_division7(pg_selector = 0)
    return [] if self | 1 == 1
    return [[self, 1]] if self.prime?
    base_primes = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
    pv = self < 0 ? [-1] : []
    value = self.abs
    base_primes.each {|prm| (pv << prm; value /= prm) while value % prm == 0 }
    sqrt_value = Math.sqrt(value).to_i
    num = self.abs == value ? value : sqrt_value
    residues, *, mod = init_generator1(num, pg_selector)
    rn = residues.size - 1        # last_residue_index
    modk = r = 0

    until value.prime? or value == 1
      while (prime = modk + residues[r]) <= sqrt_value
        if value % prime == 0
          pv << prime
          value /= prime
          sqrt_value = Math.sqrt(value).to_i
          break
        end
      r += 1; (r = 0; modk += mod) if r > rn
      end
    end
    pv << value if value > 1
    pv.group_by {|prm| prm }.map{|prm, exp| [prm, exp.size] }
  end

  private
  def init_generator1(num, pg_selector)
    base_primes = [2, 3, 5, 7, 11, 13, 17, 19, 23]
    pg_selector = select_pg(num.abs) unless base_primes.include? pg_selector
    # puts "Using P#{pg_selector}"
    base_primes.select! {|prm| prm <= pg_selector }
    mod = base_primes.reduce(:*)
    residues = []; 3.step(mod, 2) {|r| residues << r if mod.gcd(r) == 1 }
    [residues << mod + 1, base_primes, mod]
  end

  def select_pg(num)   # adaptively select fastest SP Prime Generator
    return 5  if num < 1 * 10**7  + 1000
    return 7  if num < 1 * 10**10 + 1000
    return 11 if num < 1 * 10**13 + 1000
    return 13 if num < 7 * 10**15 + 1000
    return 17 if num < 4 * 10**18 + 1000
    19
  end
end
```

#12 - 09/02/2016 08:20 PM - jzakiya (Jabari Zakiya)

A simplification.

Because the cli command factor handles the values '0' and '1' correctly we can
eliminate that first test for them in factors, so the code just becomes as shown below.

```
class Integer
  def factors
    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
```

#13 - 09/06/2016 05:21 AM - jzakiya (Jabari Zakiya)

Here is an additional coding efficiency speedup.

By testing whether the number is prime or '1' immediately after extracting any
base prime factors we can eliminate doing the prime generator selection process
for those cases. This provides an additional performance increase for those cases.

```
require 'openssl'

class Integer

  def prime?
    self.to_bn.prime?
  end

  def prime_division8(pg_selector = 0)
    return [] if self | 1 == 1
    return [[self, 1]] if self.prime?
    base_primes = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
    pv = self < 0 ? [-1] : []
    value = self.abs
    base_primes.each {|prm| (pv << prm; value /= prm) while value % prm == 0 }

    unless value.prime? or value == 1
      sqrt_value = Math.sqrt(value).to_i
      num = self.abs == value ? value : sqrt_value
      residues, *, mod = init_generator1(num, pg_selector)
      rn = residues.size - 1        # last_residue_index
      modk = r = 0

      until value.prime? or value == 1
        while (prime = modk + residues[r]) <= sqrt_value
          if value % prime == 0
            pv << prime
            value /= prime
            sqrt_value = Math.sqrt(value).to_i
            break
          end
          r += 1; (r = 0; modk += mod) if r > rn
        end
      end
    end
    pv << value if value > 1
    pv.group_by {|prm| prm }.map{|prm, exp| [prm, exp.size] }
  end

  private
  def init_generator1(num, pg_selector)
    base_primes = [2, 3, 5, 7, 11, 13, 17, 19, 23]
    pg_selector = select_pg(num.abs) unless base_primes.include? pg_selector
    # puts "Using P#{pg_selector}"
    base_primes.select! {|prm| prm <= pg_selector }
    mod = base_primes.reduce(:*)
    residues = []; 3.step(mod, 2) {|r| residues << r if mod.gcd(r) == 1 }
    [residues << mod + 1, base_primes, mod]
  end

  def select_pg(num)   # adaptively select fastest SP Prime Generator
    return 5  if num < 1 * 10**7  + 1000
    return 7  if num < 1 * 10**10 + 1000
    return 11 if num < 1 * 10**13 + 1000
    return 13 if num < 7 * 10**15 + 1000
    return 17 if num < 4 * 10**18 + 1000
    19
  end
end
```

There is a potential way to DRY up the code, and combine the two places where the number is checked for being prime or '1'. This would entail creating a fast way to pick the best prime generator after each factor reduction of the number. Doing this shouldn't be hard to code, the question is will the implementation provide a general performance increase, or not. Below is psuedo code for doing this.

```
  def prime_division_x(pg_selector = 0)
    return [] if self | 1 == 1
    return [[self, 1]] if self.prime?
    base_primes = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
    pv = self < 0 ? [-1] : []
    value = self.abs
    base_primes.each {|prm| (pv << prm; value /= prm) while value % prm == 0 }

    until value.prime? or value == 1
```

```
      sqrt_value = Math.sqrt(value).to_i
      r, rn, mod, modk, residues = primegen_select(value, sqrt_value, pg_selector)
      while (prime = modk + residues[r]) <= sqrt_value
        if value % prime == 0
          pv << prime
          value /= prime
          break
        end
        r += 1; (r = 0; modk += mod) if r > rn
      end
    end

    pv << value if value > 1
    pv.group_by {|prm| prm }.map{|prm, exp| [prm, exp.size] }
  end
```

#14 - 10/02/2016 02:52 AM - jzakiya (Jabari Zakiya)

IMHO the current version has an inconsistency in outputs between 1 and -1.

```
>  1.prime_division8 => []
> -1.prime_division8 => [[-1,1]]
```

For logical consistency, and mathematical correctness, -1 output should be the same as 1.

```
>  1.prime_division8 => []
> -1.prime_division8 => []
```

This can be corrected with a minor modification to the first loc as follows.

```
 return [] if self | 1 == 1   =>   return [] if self.abs | 1 == 1
```

#15 - 10/04/2016 07:17 PM - marcandre (Marc-Andre Lafortune)

- Assignee changed from yugui (Yuki Sonoda) to marcandre (Marc-Andre Lafortune)

Thanks for your comments and propositions.

First, let me note that prime_division returns a factorization, such that n.prime_division.inject(1){|value, (prime, index)| value * prime**index} returns n (see int_from_prime_division)

This explains the results for negative numbers and 0. I feel this is logical. Because of that and for compatibility reasons, this will not change.

I am not against optimizing prime_division or similar, as long as:

- code remains of a reasonable size
- memory consumption isn't overly big
- compatibility is maintained (for all possible usage, i.e. with no arguments, one argument, two arguments, ...)
- does not require another library; checking for the presence of other methods could be acceptable though, as long as there's a backup in case they are absent.

I have not read all of your multiple comments. Would it be possible for you to summarize them and to make a single proposal that respects all the above conditions?

#16 - 10/04/2016 08:07 PM - marcandre (Marc-Andre Lafortune)

Nobuyoshi Nakada wrote:

> Jabari Zakiya wrote:
>
> > ```
> > return [] if self | 1 == 1
> > ```
>
> It seems an unnecessarily heavy operation when self is a huge integer.

Isn't that quite efficient though, even for Bignum? Bignum#|(Fixnum) is optimized (bigor_int)

In any case, using even? is dirt cheap even on huge Bignum, and easier to read, in case it is necessary.

#17 - 10/05/2016 08:13 AM - Eregon (Benoit Daloze)

Marc-Andre Lafortune wrote:

> Nobuyoshi Nakada wrote:

> > Jabari Zakiya wrote:
> >
> > ```
> > return [] if self | 1 == 1
> > ```
> >
> > It seems an unnecessarily heavy operation when self is a huge integer.
>
> Isn't that quite efficient though, even for Bignum? Bignum#|(Fixnum) is optimized (bigor_int)
>
> In any case, using even? is dirt cheap even on huge Bignum, and easier to read, in case it is necessary.

It has to allocate another Bignum, with just one bit changed (it's |, not &, (1<<100)|1 is (1<<100)+1).
I would recommend the much clearer and efficient return [] if self == 0 or self == 1.
Or am I missing something?

#18 - 10/05/2016 08:20 AM - Eregon (Benoit Daloze)

I meant return [] if self == 0 or self == 1 (corrected on the tracker).

#19 - 10/05/2016 02:45 PM - marcandre (Marc-Andre Lafortune)

lol, I confused it with &, even if that didn't make any sense, sorry.

#20 - 10/08/2016 04:08 AM - jzakiya (Jabari Zakiya)

I am confused some by these recent comments, and would appreciate clarification.

Since 1 is not prime and returns [] then for mathematical consistency and correctness so should -1.

I don't understand how the code I presented created a problem. It presents no problems in my benchmarks.
The alternative to it then would be:

```
    return [] if self == 0 or self.abs == 1
```

Actually, the code fixes handling -1 correctly and also correctly mathematically handles 0 (no error raised).
If you allow these mathematical errors to remain the code produces the same results as prime_division and is
thus a drop in replacement for it (but much faster). Have you tested it?

Besides these math errors, the current version of prime_divison is slow and uses much more code than necessary.
My purpose is to make prime factorization in prime.rb as fast as possible, particulary to exceed Matz's Ruby 3x3 goal.
The code I've presented is orders of magnitude faster than 3x, with the added benefit of greatly reducing the codebase
necessary to achieve this performance increase. On my 3.5GHz I7 Linux laptop it can factor 30+ digit numbers in seconds.
Have you benchmarked it against prime_division? Are there other more important metrics that the code is being judged against?

As a user of Ruby for math and engineering purposes I need accuracy, speed, and ease of use.
The present handling of -1 and 0 are just plain mathematical errors. In fact, from a mathematical perspective *all negative
integers are defined as non-prime*, so from that perspective you don't even need to try to prime factor negative numbers.
In my primes-utils gem I just do self.abs to require processing only positve integers.

Also, by using the OpenSSL Standard Library you can replace the slow implementation of prime? with it's counterpart
to increase performance, and save code too.

Below is the even more concise improved code. As you see, it's much shorter than the current codebase in prime.rb.
If Ruby ever gets true parallel programming capabilities it could possibly be upgraded to take advantage of that too.
If you have questions please let me know.

```
require 'openssl'

class Integer

  def prime?
    self.to_bn.prime?
  end

  def prime_division9(pg_selector = 0)
    return [] if self.abs | 1 == 1
    return [[self, 1]] if self.prime?
    pv = self < 0 ? [-1] : []
    value = self.abs
    base_primes = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
    base_primes.each {|prm| (pv << prm; value /= prm) while value % prm == 0 }

    unless value.prime? or value == 1
```

```
      residues, *, mod = init_generator1(Math.sqrt(value).to_i, pg_selector)
      rn = residues.size - 1
      modk = r = 0

      until value.prime? or value == 1
        while (prime = modk + residues[r])
          (pv << prime; value /= prime; break) if value % prime == 0
          r += 1; (r = 0; modk += mod) if r > rn
        end
      end
    end
    pv << value if value > 1
    pv.group_by {|prm| prm }.map{|prm, exp| [prm, exp.size] }
  end

  private
  def init_generator1(num, pg_selector)
    base_primes = [2, 3, 5, 7, 11, 13, 17, 19, 23]
    pg_selector = select_pg(num.abs) unless base_primes.include? pg_selector
    # puts "Using P#{pg_selector}"
    base_primes.select! {|prm| prm <= pg_selector }
    mod = base_primes.reduce(:*)
    residues = []; 3.step(mod, 2) {|r| residues << r if mod.gcd(r) == 1 }
    [residues << mod + 1, base_primes, mod]
  end

  def select_pg(num)   # adaptively select fastest SP Prime Generator
    return 5  if num < 21 * 10**6  + 1000
    return 7  if num < 20 * 10**9  + 1000
    return 11 if num < 10 * 10**12 + 1000
    return 13 if num < 70 * 10**14 - 1000
    return 17 if num < 43 * 10**17 - 1000
    19
  end
end
```

#21 - 10/08/2016 06:35 AM - marcandre (Marc-Andre Lafortune)

I thought I was quite clear, but I will try to be clearer.

n.prime_divison returns a factorization of n such that when you take their product of the powers you get back n. Trying to express this mathematically:

```
n =  ∏ f_i ^ p_i
```

Wheref_i is either -1 , 1 or a prime, p_i is positive integer. We could allow f_i to be 0 and p_i to be 1 in case n == 0 instead of raising (i.e. returning [[0, 1]], but certainly not [], which would give a product of 1 (https://en.wikipedia.org/wiki/Product_(mathematics)#Empty_product). This is why the current library provides a factorization that I consider correct for negative integers:

```
-8.prime_division # => [[-1, 1], [2, 3]]
(-1 ** 1) * (2 ** 3) # => -8, which is the desired result
```

So I don't see a compelling reason for results for negative numbers to change, so it would be nice if you adapted your code to conform to this, otherwise I'll do it.

Also, as stated, require 'openssl' is not acceptable in lib/prime.rb, but it would be fine to take advantage of it if it is loaded, and documentation should mention that.

Finally, I just listed all the conditions that must be met, I didn't mean that your proposal didn't meet any of them, just some of them. In particular I never meant to imply that your solution wouldn't be faster.

#22 - 10/10/2016 05:15 PM - jzakiya (Jabari Zakiya)

Thanks for your explanation. I understood (after looking at the source code) what was being done,
and how it was being done, but from a user perspective I was inquiring *why* it was being done that way.

Also purely from a user perspective, the method name prime_factors is much clearer and intuitive than
prime_division. It says what the method will produce and not what it's doing.

```
this:   47824821.prime_factors   is intuitively clearer than this:   47824821.prime_division
```

I suggest at least aliasing this for >= Ruby 2.4 because Ruby 3 will have backward incompatibiities anyway,
and since I would imagine prime.rb is not used by a whole lot of people, creating at least an alias
puts the name on a conceptually better foundation of what user would expect. Looking in code and seeing
the use of n.prime_factors is just so much clearer.

Finally, if you don't want to directly require openssl you can still use the technique that uses a fast prime? method.

You can paste the code below directly into an irb session (or load from a file).

I created two versions of prime_factors, one using prime? from openssl and the other using the upgraded prime? method from the current trunk code for prime.rb (Thanks, again, for incorporating my suggestion for it into trunk.)

https://github.com/ruby/ruby/blob/trunk/lib/prime.rb

For the randomly choseen 27 digit integer shown prime_division takes twice as much time as prime_factors and prime_factors1, because half its time (21 sec) is used to perform trial division on the last|largest prime.

Thus, using this technique enables the code to be greatly simplified (conceptually and in locs) while significantly increasing speed, in this case 2x faster. However, prime_factors is still 7x slower, and needs more total code, than prime_division9 (from previous post) which is still the better method, especially to deal with large primes.

I present this to note and demonstrate you can apply this technique within the existing codebase to make prime factorization faster, simpler to code, and easier to upgrade by using a faster generator or prime? method. (In my primes-utils gem I use a Miller-Rabin primality test method, but to uses the openssl method mod_exp(n,m). This just reinforces the fact that openssl math methods are very useful|fast for working with arbitrary sized numbers.)

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

class Integer

  def prime?
    return self >= 2 if self <= 3
    return true if self == 5
    return false unless 30.gcd(self) == 1
    (7..Math.sqrt(self).to_i).step(30) do |p|
      return false
        self%(p)    == 0 || self%(p+4)  == 0 || self%(p+6)  == 0 || self%(p+10) == 0 ||
        self%(p+12) == 0 || self%(p+16) == 0 || self%(p+22) == 0 || self%(p+24) == 0
    end
    true
  end

  def prime_factors(generator = Prime::Generator23.new)
    return [] if self | 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


  def prime_factors1(generator = Prime::Generator23.new)
    return [] if self | 1 == 1
    pv = self < 0 ? [-1] : []
    value  = self.abs
    prime = generator.next
    until value.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

System: System76 laptop; I7 3.5GHz cpu

> n = 500000000000000000008244213; n.to_s.size
=> 27
```

```
> n = 500000000000000000008244213; n.prime_factors
=> [[3623, 1], [61283, 1], [352117631, 1], [6395490847, 1]]

n.prime_division     42.61 secs
n.prime_factors1     23.43 secs
n.prime_factors      21.05 secs
n.prime_division9     3.35 secs
```

#23 - 10/11/2016 12:01 AM - jzakiya (Jabari Zakiya)

A typo in prime? method:

```
  return false  =>  return false if
```

#24 - 10/12/2016 04:43 PM - jzakiya (Jabari Zakiya)

I know this may be past your pay grade, but I present this to bring awareness to it.
In general for Ruby 2.x, while loops perform the best, but in JRuby they perform the worst.
I have seen various technical explanations for this, but I hope Ruby 3 does something to
make its other looping constructs as fast. See timing differences below.

```
require 'openssl'

class Integer
  def prime_factors(generator = Prime::Generator23.new)
    return [] if self | 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

  def prime_factors_1(generator = Prime::Generator23.new)
    return [] if self | 1 == 1
    pv = self < 0 ? [-1] : []
    value  = self.abs
    prime = generator.next
    until value.to_bn.prime? or value == 1

      loop do

        (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

System: System76 laptop; I7 cpu @ 3.5GHz; VirtualBox 64-bit Linux distro

> n = 500000000000000000008244213; n.prime_factors
=> [[3623, 1], [61283, 1], [352117631, 1], [6395490847, 1]]

                     Ruby 2.3.1   JRuby 9.1.5.0
n.prime_division     25.93 secs     27.53 secs
n.prime_factors      10.14 secs     32.12 secs
n.prime_factors_1    13.61 secs     15.97 secs
```

#25 - 11/18/2016 03:46 PM - jzakiya (Jabari Zakiya)

I refactored the last version, prime_division9, to make it simpler, which
also makes it a litlle faster. I put the base_primes factors testing in a
new int_generator2 version, which DRYs out that process into one place.

init_generator2 now takes the input number, sucks out any base_primes factors,
then selects a 'best' prime generator based on the reduced factored value, or
uses a user selected pg value. It now also returns the reduced factored value
and an array of its base_primes factors, along with the residues array and mod
value for the selected prime generator.

This refactoring now separates the main algorithmic functions into more
logically distinct methods, allowing their modification|improvement to be done
independent from each other, which allows prime_division10 to become more concise.

Also as shown, prime_division10 produces what I consider to be the mathematically
and logically correct output for -1 of [], to match that for 1, instead of [[-1,1]].
To produce the existing output for -1 change < -1 to < 0 in the first loc.

```
require 'openssl'

class Integer

  def prime?
    self.to_bn.prime?
  end

  def prime_division10(pg_selector = 0)
    pv = self < -1 ? [-1] : []   # change < -1 to < 0 for prime.rb behavior
    value = self.abs

    unless value.prime? or (value | 1) == 1
      residues, mod, value, factors = init_generator2(value, pg_selector)
      last_residues_index = residues.size - 1
      modk = r = 0

      until value.prime? or value == 1
        while (prime = modk + residues[r])
          (factors << prime; value /= prime; break) if value % prime == 0
          r += 1; (r = 0; modk += mod) if r > last_residues_index
        end
      end
      pv += factors
    end

    pv << value if value > 1
    pv.group_by {|prm| prm }.map{|prm, exp| [prm, exp.size] }
  end

  private
  def init_generator2(num, pg_selector)
    factors = []
    base_primes = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
    base_primes.each {|prm| (factors << prm; num /= prm) while num % prm == 0 }
    pg_selector = select_pg(Math.sqrt(num).to_i) unless base_primes.include? pg_selector
    #puts "Using P#{pg_selector}"
    mod = base_primes.select! {|prm| prm <= pg_selector }.reduce(:*)
    residues = []; 3.step(mod, 2) {|r| residues << r if mod.gcd(r) == 1 }
    [residues << mod + 1, mod, num, factors]
  end

  def select_pg(num)   # adaptively select fastest SP Prime Generator
    return 5  if num < 21 * 10**6  + 1000
    return 7  if num < 20 * 10**9  + 1000
    return 11 if num < 10 * 10**12 + 1000
    return 13 if num < 70 * 10**14 - 1000
    return 17 if num < 43 * 10**17 - 1000
    19
  end
end
```