Ruby - Bug #13492

Integer#prime? and Prime.each might produce false positives

04/21/2017 07:06 PM - stomar (Marcus Stollsteimer)

Status:	Closed		
Priority:	Normal		
Assignee:	marcandre (Marc-Andre Lafortune)		
Target version:	2.5		
ruby -v:	r58436	**Backport:**	2.2: UNKNOWN, 2.3: UNKNOWN, 2.4: UNKNOWN

Description

There is a bug in Integer#prime? that might result in the method returning true for (very big) numbers that are not prime. Similarly, Prime.each might yield numbers that are not prime.

Integer#prime? uses Math.sqrt(self).to_i to determine the upper limit up to which trial divisions are carried out. However, Math.sqrt uses floating point arithmetic, which might lead to considerable differences between its result and the correct integer sqrt.

In the case where Math.sqrt returns a number that is too low, the trial division aborts too early and might miss a higher prime factor. Therefore, a non-prime number might erroneously reported to be prime.

Note that this bug probably has very low impact for practical use cases. The affected numbers are very big, probably well beyond the range where calculating Integer#prime? in a reasonable amount of time is feasible. For double precision floats, Math.sqrt().to_i should produce wrong results for integers from around 10**30 or 10**31 upwards.

Example:

```
p = 150094635296999111   # a prime number

n = p * p  # not prime
n          # => 22528399544939171409947441934790321
```

n is not a prime number and has only one prime factor, namely p. n can only be identified correctly as non-prime when a trial division with p is carried out.

However, Integer#prime? stops testing before p is reached:

```
Math.sqrt(n).to_i  # => 150094635296999104 (!)
p                  # => 150094635296999111
```

It would therefore erroneously return true (after a very long time of waiting for the result).

(To be precise: the method tests in batches of 30, and the highest tested number in this case would actually be 150094635296999101; the remaining numbers up to the (wrong) limit are known to be multiples of 2 or 3, and need not be tested.)

Prime.each has the same problem. It uses Prime::EratosthenesGenerator by default, which calculates the upper limit with Math.sqrt().floor (via Prime::EratosthenesSieve).

For trunk, this bug can easily be fixed by using the (new) Integer.sqrt method, see attached patch.

I'm not sure whether a patch for Ruby 2.4 and 2.3 (where Integer.sqrt is not available) is necessary.

History

#1 - 04/21/2017 07:08 PM - stomar (Marcus Stollsteimer)

- File 0001-prime-upper-limit.patch added

#2 - 04/21/2017 07:12 PM - stomar (Marcus Stollsteimer)

By monkey patching Math.sqrt to be extremely imprecise, the general effect can be seen better, i.e. for low numbers.

Affected are Integer#prime? and Prime.each via Prime::EratosthenesSieve.

```
module Math
```

```
  class << self
    alias :sqrt_org :sqrt
  end

  # imprecise sqrt (last digit dropped)
  def self.sqrt(n)
    sqrt_org(n).floor(-1)
  end
end

Math.sqrt(200)  # => 10


require "prime"

ubound = 1000  # expected in 1..1000:  168 prime numbers


### failure
(1..ubound).to_a.count {|n| n.prime? }  # => 171
Prime.each(ubound).to_a.size            # => 188
Prime.each(ubound, Prime::EratosthenesGenerator.new).to_a.size
                                        # => 188

sieve = Prime::EratosthenesSieve.instance
sieve.get_nth_prime(167)  # expected: 997
                                # => 859
sieve.instance_variable_get(:@primes)[25..35]
         # => [101, 103, 107, 109, 113, 121, 127, 131, 137, 139, 143]
  # expected: [101, 103, 107, 109, 113, 127, 131, 137, 139, 149, 151]


### success (for Prime.prime? a pseudo prime generator is sufficient,
#            false positives do not hurt here)
(1..ubound).to_a.count {|n| Prime.prime?(n) }  # => 168
(1..ubound).to_a.count {|n| Prime.prime?(n, Prime::Generator23.new) }
                                               # => 168

# Prime::TrialDivisionGenerator works fine
Prime.each(ubound, Prime::TrialDivisionGenerator.new).to_a.size
                                               # => 168
```

#3 - 04/22/2017 04:26 PM - stomar (Marcus Stollsteimer)

The proposed change breaks an essentially unrelated test.

A test for Prime::EratosthenesSieve redefines Integer() to test the behavior for timeouts.

The test could be made to succeed by using

```
root = Integer(Integer.sqrt(segment_max))
```

instead of

```
root = Integer.sqrt(segment_max)
```

in Prime::EratosthenesSieve#compute_primes, which seems kind of stupid.

Any idea how the test could be fixed in a better way?

Here the test case (from test/test_prime.rb):

```
  def test_eratosthenes_works_fine_after_timeout
    sieve = Prime::EratosthenesSieve.instance
    sieve.send(:initialize)
    begin
      # simulates that Timeout.timeout interrupts Prime::EratosthenesSieve#compute_primes
      def sieve.Integer(n)
        n = super(n)
        sleep 10 if /compute_primes/ =~ caller.first
        return n
      end

      assert_raise(Timeout::Error) do
        Timeout.timeout(0.5) { Prime.each(7*37){} }
```

```
      end
    ensure
      class << sieve
        remove_method :Integer
      end
    end

    assert_not_include Prime.each(7*37).to_a, 7*37, "[ruby-dev:39465]"
  end
```

#4 - 05/19/2017 09:35 AM - akr (Akira Tanaka)

The patch seems good.

#5 - 05/20/2017 12:39 AM - marcandre (Marc-Andre Lafortune)

- Status changed from Open to Closed

- Assignee set to marcandre (Marc-Andre Lafortune)

Good catch.

I tweaked the timeout test by patching Integer.sqrt.

#6 - 05/20/2017 09:38 AM - stomar (Marcus Stollsteimer)

@marcandre (Marc-Andre Lafortune)

I actually was working on the essentially same fix for the test case and about to commit; only I did want to fix the test for Math.sqrt first (which could be backported) and then switch to Integer.sqrt.

#7 - 06/29/2017 04:42 PM - usa (Usaku NAKAMURA)

If a patch is provided, I'll merge it to ruby_2_3.

Files

0001-prime-upper-limit.patch	1.53 KB	04/21/2017	stomar (Marcus Stollsteimer)