Improved Primality Testing and Factorization in Ruby revised
by
Jabari Zakiya © June 2013

Introduction

The Ruby standard library file **prime.rb** contains the class Integer methods **prime?** (which tests if an integer is prime) and **prime_division** (returns prime factorization of an integer). I present here simpler and significantly faster methods as their suggested replacements.

Prime Generators

All primes can be produced with a prime generator (PG) of the form: P_n = mod*k+r_i, r_i ε res[1..mod-1]. Modulus **mod** is an even integer, **k** = 0,1,2,3.., and $\mathbf{r_i}$ is a **residue** from a list of residues **res[1..mod-1]**. A prime *p* for *n* in P_n denotes a ***Strictly Prime (SP)*** PG. SP PGs moduli have form: P(*p*)! = 2*3*5...*p*, i.e. P(*p*)! = $\prod p_i$ is the factorial of the primes up to *p*. Their total residues are: rescnt = P(p_i-1)!=$\prod(p_i$-1). The residues r_i start at 1, end at mod-1, and include all the integers co-prime to mod from 3..mod.

The first SP PG P_3 = 6k+(1,5) generates all primes > 3. Here **mod**=2*3=6, rescnt = (2-1)(3-1) = 2 and the residues r_i are (1,5). Only when n%6 = 1 or 5 can n be prime, and there are N*(rescnt/mod) = N/3 prime candidates (pc) up to some N. The second SP PG P_5 = 30k+(1,7,11,13,17,19,23,29) generates all primes > 5 and the possible pc N*(8/30) = N*(4/15) are fewer than for P3. And so it goes for SP PG.

Table 1: P_3 = 6k+(1,5)

k	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**
r1	1	7	13	19	25	31	37	43	49	55	61	67	73	79	85	91	97	103	109
r2	5	11	17	23	29	35	41	47	53	59	65	71	77	83	89	95	101	107	113

Table 2: P_5 = 30k+(1,7,11,13,17,19,23,29) colored entries are non-primes

k	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**
r1	7	37	67	97	127	157	187	217	247	277	307	337	367	397	427	457	487	517
r2	11	41	71	101	131	161	191	221	251	281	311	341	371	401	431	461	491	521
r3	13	43	73	103	133	163	193	223	253	283	313	343	373	403	433	463	493	523
r4	17	47	77	107	137	167	197	227	257	287	317	347	377	407	437	467	497	527
r5	19	49	79	109	139	169	199	229	259	289	319	349	379	409	439	469	499	529
r6	23	53	83	113	143	173	203	233	263	293	323	353	383	413	443	473	503	533
r7	29	59	89	119	149	179	209	239	269	299	329	359	389	419	449	479	509	539
r8	31	61	91	121	151	181	211	241	271	301	331	361	391	421	451	481	511	541

Each value of k is a column in the tables which contain the **kth residues group** prime candidates (pc). For SP generators as their order increases the number of non-primes they generate decrease, so they become progressively more *efficient* at generating only primes. (For a more thorough explanation of prime generators see my papers [1] and [2])

Simplest Primality Test

If an integer N is divisible by any prime p_j <= sqrt(N), then its not prime, otherwise it is.

Algorithmically speaking: if (N % p_j)=0 for any prime p_j <= sqrt(N), N is not prime, otherwise it is.

So where do we get a list of all the primes p_j <= sqrt(N), especially when N becomes very large?

This is a job for our trusty SP prime generators. We don't need no *stinkin* list, we generate primes as needed. So let's look at a simple, but surprisingly fast, primality test algorithm and code.

The following two Ruby methods use P5 to generate the primes to check the primality of integer n.

Listing 1.

```
class Integer
   def primzp5?
      residues = [1,7,11,13,17,19,23,29,31]
      mod=30; rescnt=8

      n = self.abs
      return true  if [2,3,5].include? n
      return false if not residues.include?(n % mod) || n == 1

      sqrtN = Math.sqrt(n).to_i
      modk,r=0,1;  p=7        # first test prime pj
      while p <= sqrtN
        return false if n%p == 0
        r +=1; if r > rescnt; r=1; modk +=mod end
        p = modk+residues[r]  # next prime candidate
      end
      return true
   end

   def primzp5a?
      residues = [1,7,11,13,17,19,23,29,31]
      mod=30; rescnt=8

      n = self.abs
      return true  if [2,3,5].include? n
      return false if not residues.include?(n % mod) || n == 1

      sqrtN = Math.sqrt(n).to_i
      p=7         # first test prime pj
      while p <= sqrtN
        return false if
          n%(p)   == 0 or n%(p+4) ==0 or n%(p+6) == 0 or n%(p+10)==0 or
          n%(p+12)== 0 or n%(p+16)==0 or n%(p+22)== 0 or n%(p+24)==0
        p += mod  # first prime candidate for next kth residues group
      end
      return true
   end
end
```

First create P5 residues list from 1 to mod+1, and check if n is 2, 3, or 5. Then check if the value n%30 is in the residues group; if not (or n is 1) then n is non-prime. Then set p=7 and test if n % p = 0 for every pc <= sqrt(n). If ever true, then n is non-prime; if not true, then n is prime. Method **primzp5?** generates and tests each individual pc successively, while **primzp5a?** generates and tests a residues group (columns in Table 2), manually unrolled, all at once. Now let's see the speed differences.

```
2.0.0p195 :100 > 6000000000000001.primzp5?      => true
2.0.0p195 :101 > tm{6000000000000001.primzp5?}  => 10.487644646
2.0.0p195 :102 > tm{6000000000000001.primzp5a?} => 8.964337622
2.0.0p195 :103 > tm{6000000000000001.prime?}    => 33.559378141
```

The timings for P5 show performing the primality test on unrolled residues groups is faster than serially testing individual pc. If we use bigger SP PGs we should expect (in theory) even better performance. In fact, that is what is observed on my system up to P17 (P19 and greater give slower results).

Listing 2. shows the code using P7, where **primzp7?** tests individual successive pc, while **primzp7a?** and **primzp7b?** test manually unrolled residues groups, each slightly differently. On my system, the 'a' and 'b' versions perform the same on MRI and Rubinius, but 'a' is somewhat faster than 'b' on JRuby, so in the benchmarks I just show the results using the 'a' version.

We could continue manually unrolling bigger SP prime generators, P11 (480 residues), P13 (5760), etc, but the code becomes unwieldy (you may want to try it for P11). Listing 3 shows generic methods **primzp?** and **primzpa?** which take a prime number input and creates the mod, rescnt, and residues group values for that SP PG, where the first version tests individual pc values serially and the 'a' version tests residues group values together (as the '7a' versions above). Algorithmically **primzp5/7?** are equivalent to **primzp? 5/7** while **primzp5/7a?** are (conceptually) equivalent to **primzpa? 5/7**.

The benchmarks show the manually unrolled residues in **primzp7a?** performs better than higher order SP PG in almost all the tests for both the 32/64-bit systems. However, if the generic versions can be coded to operate as the manually unrolled versions then larger SP primes should (in theory) be faster (barring memory and other physical system limitations).

In **primzpa?** the code: **return false if res.map {|r| n%(r+p)}.include? 0**
mimics manually unrolling and testing the residues values. There are multiple ways to equivalently code this, and different codings perform better with different VMs. Additionally, this code is ripe for true parallel implementation (each residues group is independently testable) with JRuby and Rubinius.

With a small change to **primzp?** we can create **factorzp** (Listing 4) to perform prime factorization. It is much simpler and faster than **prime_division** and produces a sorted list of prime factors, which can be easily formatted to match the latter's output.

The timings shown in Tables 3 and 4 display the VMs performance variations.

Listing 2.

```
class Integer
   def primzp7?
      residues = [1,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73,79,83,
         89,97,101,103,107,109,113,121,127,131,137,139,143,149,151,157,163,
         167,169,173,179,181,187,191,193,197,199,209,211]
      mod=210; rescnt=48

      n = self.abs
      return true  if [2, 3, 5, 7].include? n
      return false if not residues.include?(n%mod) || n == 1

      sqrtN = Math.sqrt(n).to_i
      modk,r=0,1;  p=11       # first test prime pj
      while p <= sqrtN
        return false if n%p == 0
        r +=1; if r > rescnt; r=1; modk +=mod end
        p = modk+residues[r]  # next prime candidate
      end
      return true
   end

   def primzp7a?
      residues = [1,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73,79,83,
         89,97,101,103,107,109,113,121,127,131,137,139,143,149,151,157,163,
         167,169,173,179,181,187,191,193,197,199,209,211]
      mod=210; rescnt=48

      n = self.abs
      return true  if [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43,
                       47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103,
                       107, 109, 113, 127, 131, 137, 139, 149, 151, 157,163,
                       167, 173, 179, 181, 191, 193, 197, 199, 211].include? n
      return false if not residues.include?(n%mod) || n == 1

      sqrtN = Math.sqrt(n).to_i
      p=11            # first test prime pj
      while p <= sqrtN
        return false if
          n%(p)    == 0 or n%(p+2)  ==0 or n%(p+6)  == 0 or n%(p+8)  ==0 or
          n%(p+12) == 0 or n%(p+18) ==0 or n%(p+20) == 0 or n%(p+26) ==0 or
          n%(p+30) == 0 or n%(p+32) ==0 or n%(p+36) == 0 or n%(p+42) ==0 or
          n%(p+48) == 0 or n%(p+50) ==0 or n%(p+56) == 0 or n%(p+60) ==0 or
          n%(p+62) == 0 or n%(p+68) ==0 or n%(p+72) == 0 or n%(p+78) ==0 or
          n%(p+86) == 0 or n%(p+90) ==0 or n%(p+92) == 0 or n%(p+96) ==0 or
          n%(p+98) == 0 or n%(p+102)==0 or n%(p+110)== 0 or n%(p+116)==0 or
          n%(p+120)== 0 or n%(p+126)==0 or n%(p+128)== 0 or n%(p+132)==0 or
          n%(p+138)== 0 or n%(p+140)==0 or n%(p+146)== 0 or n%(p+152)==0 or
          n%(p+156)== 0 or n%(p+158)==0 or n%(p+162)== 0 or n%(p+168)==0 or
          n%(p+170)== 0 or n%(p+176)==0 or n%(p+180)== 0 or n%(p+182)==0 or
          n%(p+186)== 0 or n%(p+188)==0 or n%(p+198)== 0 or n%(p+200)==0
        p += mod      # first prime candidate for next kth residues group
      end
      return true
   end
```

```
   def primzp7b?
      residues = [1,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73,79,83,
         89,97,101,103,107,109,113,121,127,131,137,139,143,149,151,157,163,
         167,169,173,179,181,187,191,193,197,199,209,211]
      mod=210; rescnt=48

      n = self.abs
      return true  if [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43,
                       47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103,
                       107, 109, 113, 127, 131, 137, 139, 149, 151, 157,163,
                       167, 173, 179, 181, 191, 193, 197, 199, 211].include? n
      return false if not residues.include?(n%mod) || n == 1

      sqrtN = Math.sqrt(n).to_i
      modk=0
      while (11+modk) <= sqrtN
        return false if
          n%(11+modk) == 0 or n%(13+modk) ==0 or n%(17+modk) == 0 or n%(19+modk) ==0 or
          n%(23+modk) == 0 or n%(29+modk) ==0 or n%(31+modk) == 0 or n%(37+modk) ==0 or
          n%(41+modk) == 0 or n%(43+modk) ==0 or n%(47+modk) == 0 or n%(53+modk) ==0 or
          n%(59+modk) == 0 or n%(61+modk) ==0 or n%(67+modk) == 0 or n%(71+modk) ==0 or
          n%(73+modk) == 0 or n%(79+modk) ==0 or n%(83+modk) == 0 or n%(89+modk) ==0 or
          n%(97+modk) == 0 or n%(101+modk)==0 or n%(103+modk)== 0 or n%(107+modk)==0 or
          n%(109+modk)== 0 or n%(113+modk)==0 or n%(121+modk)== 0 or n%(127+modk)==0 or
          n%(131+modk)== 0 or n%(137+modk)==0 or n%(139+modk)== 0 or n%(143+modk)==0 or
          n%(149+modk)== 0 or n%(151+modk)==0 or n%(157+modk)== 0 or n%(163+modk)==0 or
          n%(167+modk)== 0 or n%(169+modk)==0 or n%(173+modk)== 0 or n%(179+modk)==0 or
          n%(181+modk)== 0 or n%(187+modk)==0 or n%(191+modk)== 0 or n%(193+modk)==0 or
          n%(197+modk)== 0 or n%(199+modk)==0 or n%(209+modk)== 0 or n%(211+modk)==0
        modk += mod  # modulus for next kth residues group prime candidates
      end
      return true
   end
end
```

Listing 3.

```
class Integer
   def primzp?(p=13)       # P13 is default prime generator here
     seeds  = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41]
     primes = []
     return 'PRIME OPTION NOT A SEEDS PRIME' if !seeds.include? P

     n = self.abs

     # find primes <= Pn, compute modPn then PG residues for p
     primes = seeds[0..seeds.index(p)]; mod = primes.inject {|a,b| a*b }
     residues=[1]; 3.step(mod,2){|i| residues << i if mod.gcd(i) == 1}
     residues << mod+1; rescnt = residues.size-1

     return true  if primes.include? n
     return false if not residues.include?(n % mod) || n == 1

     sqrtN = Math.sqrt(n).to_i
     modk,r = 0,1; p=residues[1] # first test prime pj for given Pp
     while p <= sqrtN
       return false if n%p == 0
       r += 1; if r > rescnt; r=1; modk += mod end
       p = modk + residues[r]    # next prime candidate
     end
     return true
   end
```

```
   def primzpa?(p=13)      # P13 is default prime generator here
     seeds  = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41]
     return 'PRIME OPTION NOT A SEEDS PRIME' if !seeds.include? p

     n = self.abs

     # find primes <= Pn, compute modPn then Prime Gen residues for Pn
     primes = seeds[0..seeds.index(p)]; mod = primes.inject {|a,b| a*b }
     mod = 30 if p > 5 and n < mod+2  # for Pp > P5 and n within Pp residues
     residues=[1]; 3.step(mod,2) {|i| residues << i if mod.gcd(i) == 1}*
     residues << mod+1; rescnt = residues.size-1

     return true  if primes.include? n
     return false if not residues.include?(n%mod) || n == 1

     sqrtN = Math.sqrt(n).to_i
     modk = 0;  p=residues[1]             # first test prime pj for given Pp
     res = residues[1..-1].map {|r| r-p } # residues distance from first prime
     while p <= sqrtN
       return false if res.map {|r| n%(r+p)}.include? 0
       p += mod   # first prime candidate for next kth residues group
     end
     return true
   end
end
```

Listing 4.

```
class Integer
   def factorzp(p=13)      # P13 is default prime generator here
     seeds  = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41]
     primes = []
     return 'PRIME OPTION NOT A SEEDS PRIME' if !seeds.include? p

     # find primes <= Pn, compute modPn then PG residues for p
     primes = seeds[0..seeds.index(p)]; mod = primes.inject {|a,b| a*b }
     residues=[1]; 3.step(mod,2){|i| residues << i if mod.gcd(i) == 1}
     residues << mod+1; rescnt = residues.size-1

     n = self.abs
     factors = []

     return factors << n if primes.include? n
     primes.each {|p| while n%p == 0; factors << p; n /= p end }
     return factors if n == 1 # for when n is product of only seed primes

     sqrtN= Math.sqrt(n).to_i
     modk,r = 0,1;  p=residues[1] # first test prime pj factor for given Pp
     while p <= sqrtN
       if n%p == 0
         factors << p; r -= 1; n /= p; sqrtN = Math.sqrt(n).to_i
       end
       r += 1; if r > rescnt; r = 1; modk += mod end
       p = modk + residues[r]     # next (or current) prime factor candidate
     end
     factors << n
     factors.sort   # return n if prime, or its prime factors
   end
end
```

Benchmarks
Tables 3 and 4 shows the timing results for 5 reference primes which span from 17 to 19 digits, and are nice multiples of each other (2x, 5x, 10x, etc). The timings compare the MRI, JRuby, and Rubinius virtual machines (VMs) performance for the different codings of the same algorithms.

Reference Primes
(2*10^16 + 3) = 20,000,000,000,000,003 (17 digits)
(1*10^17 + 3) = 100,000,000,000,000,003 (18 digits) 5x
(2*10^17 + 3) = 200,000,000,000,000,003 (18 digits) 2x, 10x
(1*10^18 + 3) = 1,000,000,000,000,000,003 (19 digits) 5x, 10x, 50x
(5*10^18 + 3) = 5,000,000,000,000,000,003 (19 digits) 5x, 25x, 50x, 250x

All primes are class Bignum integers for all 32-bit system VMs in Table 3 while for Table 4, for the 64-bit systems, they are Fixnum, except the largest prime is Bignum for MRI and Rubinius.

Conclusions
All the methods shown here perform significantly faster than the standard Ruby methods **prime?** and **prime_division** for all VM versions tested, with some performing particularly better on a specific VM.

For the 32-bit system benchmarks JRuby is the fastest performing.
For the 64-bit system benchmarks Rubinius is the fastest performing.
primzp7a? is, generally, the fastest all-around primality testing algorithm implementation tested.
Testing unrolled (manually or programmed) residues is faster in JRuby than testing individual residues, but the opposite for MRI and Rubinius.

Improvement could entail using multiple threads/cores, memorization, machine coding, and selecting the PG based on the size of the number. In general smaller PG (e.g. P5 versus P13), which use fewer residues but more iterations, may 'fit' better in cache/memories or particular architectures resources. The 'sweet spot' on my system was P17 for testing large numbers (but not for really small numbers).

Using any of these implementations would make Ruby significantly more useful in math and science and applications involving prime testing and generation (see [1] and [2]).

Each Ruby VM should use the 'best' implementation particular to it.

Papers and code primeszp.rb download
http://www.4shared.com/dir/7467736/97bd7b71/sharing.html
https://gist.github.com/jzakiya/455f2357cdb08f4ee1c4

References
[1] Ultimate Prime Sieve – Sieve of Zakiya (SoZ), Jabari Zakiya, June 2008,
http://www.scribd.com/doc/73384039/Ultimate-Prime-Sieve-Sieve-Of-Zakiya
[2] The Sieve of Zakiya, Jabari Zakiya, December 2008
http://www.scribd.com/doc/73385696/The-Sieve-of-Zakiya

Table 3. All primes are class Bignum integers for all VMs..

Lenovo V570 laptop, Intel I5-2410M 64-bit, 2.3 GHz, 6 GB ram PCLOS KDE 32-bit, GCC 4.7.2, Ruby Versions via RVM															
Ruby vers	MRI = Ruby-2.0.0-p195 Rbx = Rubinius-2.0.0-rc1 JRb = JRuby-1.7.4														
Primes	(2*10^16 + 3)			(10^17 + 3)			(2*10^17 + 3)			(10^18 + 3)			(5*10^18 + 3)		
	MRI	Rbx	JRb	MRI	Rbx	JRb	MRI	Rbx	JRb	MRI	Rbx	JRb	MRI	Rbx	JRb
primzp7?	11.7	9.1	12.1	26.1	20.3	27.8	37.0	28.8	39.7	82.7	63.3	88.3	250.0	381.8	198.4
primzp7a?	9.5	7.8	3.5	21.6	17.5	7.9	30.3	24.6	11.5	67.2	53.8	26.5	176.4	351.5	59.1
primzpa? 7	13.7	8.7	6.5	30.8	19.4	14.3	43.7	27.2	20.7	97.1	60.1	46.9	244.6	362.4	104.8
primzp? 13	9.7	7.6	9.4	22.2	17.2	20.7	31.1	24.0	31.2	70.0	54.1	73.3	210.0	321.4	165.9
primzp? 17	9.7	7.3	9.3	22.1	16.2	20.3	31.0	22.7	30.1	69.5	51.2	71.5	207.9	303.1	157.9
primzpa? 13	11.4	7.2	5.6	25.6	20.1	11.9	37.0	23.5	17.2	81.9	52.3	39.3	201.8	306.9	88.9
primzpa? 17	11.8	7.1	5.5	26.3	20.0	11.5	38.0	23.6	17.1	83.9	53.0	38.1	230.2	492.1	85.9
factorzp 13	9.7	7.8	9.3	22.0	17.6	20.7	31.1	24.5	32.8	70.3	54.3	74.7	210.6	322.1	167.0
factorzp 17	9.7	7.5	9.2	22.1	16.5	19.9	31.1	23.1	31.3	69.6	51.4	73.9	207.0	306.2	160.2
prime?	49.3	65.1	96.1	112.1	137.2	209.5	164.3	195.2	238.6	392.9	448.8	731.1	859.7	1396	1202
prime_division	52.4	69.9	100.5	118.9	141.0	237.1	174.7	199.6	259.2	421.1	462.1	809.8	921.9	1422	1488

Table 4. All primes are class Fixnm for all VMs, except biggest prime is Bignum for MRI and Rbx.

Lenovo V570 laptop, Intel I5-2410M 64-bit, 2.3 GHz, 6 GB ram [Virtual Box] Linux Mint 14 KDE 64-bit, GCC 4.7.2, Ruby Versions via RVM															
Ruby vers	MRI = Ruby-2.0.0-p195 Rbx = Rubinius-2.0.0-rc1 JRb = JRuby-1.7.4														
Primes	(2*10^16 + 3)			(10^17 + 3)			(2*10^17 + 3)			(10^18 + 3)			(5*10^18 + 3)		
	MRI	Rbx	JRb	MRI	Rbx	JRb	MRI	Rbx	JRb	MRI	Rbx	JRb	MRI	Rbx	JRb
primzp7?	3.00	1.93	14.5	6.75	4.34	30.5	9.75	6.12	44.9	21.4	13.7	98.2	279.5	313.7	225.3
primzp7a?	1.31	0.75	3.45	3.01	1.54	7.55	4.24	2.18	10.7	9.53	4.78	24.1	250.6	165.6	53.9
primzpa? 7	5.10	1.95	6.67	11.4	4.15	15.1	16.4	5.94	20.9	37.5	13.6	46.8	321.6	218.7	104.6
primzp? 13	2.52	1.62	10.6	5.67	3.61	23.5	8.08	5.07	32.3	17.9	11.9	73,7	239.3	155.4	164.6
primzp? 17	2.43	1.62	10.6	5.37	3.59	23.4	7.65	5.06	31.7	17.1	11.4	72.0	237.2	146.2	159.5
primzpa? 13	4.21	1.38	5.75	9.42	3.29	12.6	13.2	4.58	18.1	29.5	10.8	39.5	270.1	155.9	89.4
primzpa? 17	4.03	1.75	5.74	8.98	3.65	12.7	12.7	5.14	18.2	27.9	11.7	38.6	282.1	154.5	87.5
factorzp 13	2.52	1.63	10.8	5.73	3.65	23.1	8.10	5.20	32.7	17.9	11.7	73.1	238.2	154.7	163.2
factorzp 17	2.49	1.64	10.5	5.38	3.49	22.2	7.75	5.03	32.6	17.0	11.3	70.2	235.5	146.4	159.8
prime?	45.2	56.6	91.8	104.8	133.1	199.1	147.5	177.9	253.7	317.5	394.6	540.0	1160	1279	1180
prime_division	49.4	57.1	105.5	109.6	128,6	239.2	157.4	188.4	301.0	354.6	419.9	632.5	1200	1281	1433

Revised Methods – 2013-8-28

Here are much faster, and more standard, methods for doing primality testing and factorization in Ruby.

[Un/L]inux comes with the standard cli command "factor".

```
$ factor 30409113
30409113: 3 7 1448053                # here the number is composite

$ factor 6000000000000001
6000000000000001: 6000000000000001   # here the number is a prime
```

This can be used to create ***much faster*** and more portable code that will work exactly the same with all versions of Ruby run under *nix systems.

Here's the code:

```
class Integer
   def factors
     factors = `factor #{self.abs}`.split(' ')[1..-1].map {|i| i.to_i}
     h = Hash.new(0); factors.each {|f| h[f] +=1}; h.to_a.sort
   end

   #def primality?
   #  return true if `factor #{self.abs}`.split(' ')[1..-1].size == 1
   #  return false
   #end

   # This is better coded version: 2013-10-18
   def primality?
     `factor #{self.abs}`.split(' ').size == 2
   end
end

2.0.0p247 :054 > 30409113.factors
 => [[3, 1], [7, 1], [1448053, 1]]

2.0.0p247 :055 > 6000000000000001.primality?
 => true
```

These names are used to not conflict with the other methods in the code base.

Now Ruby can work with **REALLY BIG NUMBERS** with consistent fast performance.

This should even work for Windoze builds, as they use *nix emulators.

The revised paper and code has been added to here:

Papers and code primeszp.rb download
http://www.4shared.com/dir/7467736/97bd7b71/sharing.html
https://gist.github.com/jzakiya/455f2357cdb08f4ee1c4

First Four Sieve of Zakiya (SoZ) Strictly Prime Generators

P_3 = 6k+(1,5)

P_5 = 30k+ (1, 7, 11, 13, 17, 19, 23, 29)

P_7 = 210k+(1, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103, 107, 109, 113, 121, 127, 131, 137, 139, 143, 149, 151, 157, 163,167, 169, 173, 179, 181, 187, 191, 193, 197, 199, 209)

P_{11} = 2310k+(1, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103, 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 169, 173, 179, 181, 191, 193, 197, 199, 211, 221, 223, 227, 229, 233, 239, 241, 247, 251, 257, 263, 269, 271, 277, 281, 283, 289, 293, 299, 307, 311, 313, 317, 323, 331, 337, 347, 349, 353, 359, 361, 367, 373, 377, 379, 383, 389, 391, 397, 401, 403, 409, 419, 421, 431, 433, 437, 439, 443, 449, 457, 461, 463, 467, 479, 481, 487, 491, 493, 499, 503, 509, 521, 523, 527, 529, 533, 541, 547, 551, 557, 559, 563, 569, 571, 577, 587, 589, 593, 599, 601, 607, 611, 613, 617, 619, 629, 631, 641, 643, 647, 653, 659, 661, 667, 673, 677, 683, 689, 691, 697, 701, 703, 709, 713, 719, 727, 731, 733, 739, 743, 751, 757, 761, 767, 769, 773, 779, 787, 793, 797, 799, 809, 811, 817, 821, 823, 827, 829, 839, 841, 851, 853, 857, 859, 863, 871, 877, 881, 883, 887, 893, 899, 901, 907, 911, 919, 923, 929, 937, 941, 943, 947, 949, 953, 961, 967, 971, 977, 983, 989, 991, 997, 1003, 1007, 1009, 1013, 1019, 1021, 1027, 1031, 1033, 1037, 1039, 1049, 1051, 1061, 1063, 1069, 1073, 1079, 1081, 1087, 1091, 1093, 1097, 1103, 1109, 1117, 1121, 1123, 1129, 1139, 1147, 1151, 1153, 1157, 1159, 1163, 1171, 1181, 1187, 1189, 1193, 1201, 1207, 1213, 1217, 1219, 1223, 1229, 1231, 1237, 1241, 1247, 1249, 1259, 1261, 1271, 1273, 1277, 1279, 1283, 1289, 1291, 1297, 1301, 1303, 1307, 1313, 1319, 1321, 1327, 1333, 1339, 1343, 1349, 1357, 1361, 1363, 1367, 1369, 1373, 1381, 1387, 1391, 1399, 1403, 1409, 1411, 1417, 1423, 1427, 1429, 1433, 1439, 1447, 1451, 1453, 1457, 1459, 1469, 1471, 1481, 1483, 1487, 1489, 1493, 1499, 1501, 1511, 1513, 1517, 1523, 1531, 1537, 1541, 1543, 1549, 1553, 1559, 1567, 1571, 1577, 1579, 1583, 1591, 1597, 1601, 1607, 1609, 1613, 1619, 1621, 1627, 1633, 1637, 1643, 1649, 1651, 1657, 1663, 1667, 1669, 1679, 1681, 1691, 1693, 1697, 1699, 1703, 1709, 1711, 1717, 1721, 1723, 1733, 1739, 1741, 1747, 1751, 1753, 1759, 1763, 1769, 1777, 1781, 1783, 1787, 1789, 1801, 1807, 1811, 1817, 1819, 1823, 1829, 1831, 1843, 1847, 1849, 1853, 1861, 1867, 1871, 1873, 1877, 1879, 1889, 1891, 1901, 1907, 1909, 1913, 1919, 1921, 1927, 1931, 1933, 1937, 1943, 1949, 1951, 1957, 1961, 1963, 1973, 1979, 1987, 1993, 1997, 1999, 2003, 2011, 2017, 2021, 2027, 2029, 2033, 2039, 2041, 2047, 2053, 2059, 2063, 2069, 2071, 2077, 2081, 2083, 2087, 2089, 2099, 2111, 2113, 2117, 2119, 2129, 2131, 2137, 2141, 2143, 2147, 2153, 2159, 2161, 2171, 2173, 2179, 2183, 2197, 2201, 2203, 2207, 2209, 2213, 2221, 2227, 2231, 2237, 2239, 2243, 2249, 2251, 2257, 2263, 2267, 2269, 2273, 2279, 2281, 2287, 2291, 2293, 2297, 2309)