PRIMES-UTILS HANDBOOK

The Math and Code behind making the Rubygem

Jabari Zakiya

Edition 1 - 2016/01/18

Table of Contents i

Preface

I have confession, I didn't write this for you. Well, not initially. I had created the Rubygem prime-utils and was at the point where I needed to start documenting, for myself, all the changes I had made to the different versions. So going through them, I realized there was a lot of stuff I had started to forget why I did that. And to make it clear (to me), I needed to write out the history of these changes (not just their results) because that's where the real understanding of the code resided, in the changes.

So I started writing about the code, and its creation process, at a deeper and more descriptive level. After a while, I realized it would be interesting to turn this exercise into a book others could read as well. But I didn't want to just write another book about coding, I wanted to write a book about the math used to create the gem, and make it interesting to people who like math and|or who like to code. Because after all, if you're a person interested in prime number stuff (and math in general), you might be more likely to read this in the first place, and in the process learn some Ruby programming too.

That's when the real fun began. I mean, I couldn't put out sloppy math or code, so both had to pass (as lawyers would say) the strict scrutiny test. You never learn more about a topic than when you're tasked to explain or teach it to someone else. In this process, I learned much more about the math, and coding in Ruby, than I knew before this exercise, which has allowed me to explain them more thoroughly and clearer, hopefully making them both now easier for you to understand.

And as it turns out, Ruby is an excellent language to do math in. But if you're more into coding than math, you can still gain a lot from this book. Ruby allows you to write *beautiful* and self expressive code. And using its idioms well allows you to write highly functional, yet concise, code. Some may be amazed with the amount of tangible work I could do with so little code. So what I present here is the epitome of the marriage of the development of the math and code to perform the primes-utils methods.

Acknowledgments

I have been an ardent user and supporter of Free & Open Source Software (FOSS) since the turn of this century (2000), and would like to acknowledge a few of the thousands of people, and projects, which has made my work possible, and my life so much better for them.

Richard Stallman for creating the GNU (https://www.gnu.org/) project, Linus Torvalds for creating the Linux Kernel (https://www.kernel.org/), and Yukihiro 'Matz' Matsumoto for creating the Ruby language (https://www.ruby-lang.org/en/). LibreOffice (https://www.libreoffice.org/) used for writing this book, the KDE Kate text editor for writing code (https://www.kde.org/applications/utilities/kate/), Geogebra for some needed algebra (https://geogebra.org/), Wolfram Alpha (http://www.wolframalpha.com/) for providing nth prime values, and PCLinuxOS (http://www.pclinuxos.com/) to run it all from.

Jabari Zakiya
December 1, 2015
jzakiya@gmail.com

Introduction

Ruby comes with the prime.rb library which has the methods primes? and prime_division, which check if numbers are prime or determine their factors. However they're *slow*, which effectively limits the size of numbers you will want to operate on. And if you had other questions about primes you wanted fast answers to, you were out of luck, like what is some nth prime, or what are the primes (or how many are there) up to some number, or within a range? With primes-utils you can now easily get fast answers to these questions, and for very large numbers, with a suite of fast user friendly utility methods.

Installation

Being a rubygem primes-utils resides on Rubygems (www.rubygems.org) and can be found there at https://rubygems.org/gems/primes-utils. You can download it from its github homepage, located at https://github.com/jzakiya/primes-utils or install it in the usual manner using the cli gem command. Acquire it as normal with $ gem install primes-utils then in ruby require it as > require 'primes/utils'.

Test System

My reference system was a Lenovo V570 laptop with I5-2410M cpu @ 2.3 GHz, 6 GB ram, 640 GB HD. PCLinuxOS KDE-32 bits (http://www.pclinuxos.com/) was the base distribution, with rubies installed via RVM (https://rvm.io/). Windows 7 Home Premium 64-bit was available via dualboot, on which I used RubyInstaller (http://rubyinstaller.org/) based rubies for testing. Virtual Box (VB) was used to install various Linux and BSD distros to also test with.

At the time of writing, primes-utils has been tested to work on up to the following latest Ruby VMs: MRI (2.0 – 2.2, 1.9, 1.8.7), JRuby (1.7, 9.0), and Rubinius (2.5.), though the 2.5 series produces erroneous results for the primes method.

Changelog

2.7.0 – more tweaking adaptive pg selection ranges in select_pg; coded using between? instead of cover?
2.6.0 – much, much better adaptive pg selection algorithm used in select_pg
2.5.1 – corrected minor error in select_pg
2.5.0 – 9 more index primes under the 110-millionth in nths; fixed Ruby 1.8 incompatibility in primes; better|simpler technique for select_pg, significant speed increases for large ranges; used now in all sozcore2 client methods primes, primescnt, and primenth|nthprime; more code cleanups
2.4.0 – fixed error in algorithm when ks resgroup ≤ sqrt(end_num) resgroup; algorithm now split arrays when start_num > sqrt(end_num) in sozcore2, whose code also signficantly optimized, with API change adding pcs2start value to output parameters to use in primenth, which changed to use it; ruby idiom code opt for set_start_value; consolidated pcs_to_num | pcs_to_start_num functions into one new pcs_to_num, with associated changes in sozcore1|2; primes|cnt also significantly faster resulting from sozcore2 changes; massive code cleanups all-around; added private methods select_pg (to adaptively select the pg used in primes), and array_check (used in sozcore2 to catch array creation out-of-memory errors)
2.3.0 – primescnt now finds primes upto some integer much faster, for much larger integers; increased index nth primes to over 2 billionth; used in primenth|nthprime and primescnt
2.2.0 – for sozcore2: refactored to include more common code; changed output api; added memory error messages when prms and prms_range arrays creation fails; for primenth: used new function to compute parameter b and removed ceiling for it; increased number of index primes in nths; primes, primescnt, and primenth|nthprime also refactored, will use all available mem
2.1.0 – changed default PG in primes and primescnt from P13 to P5, significantly faster
2.0.0 – new methods primesf, primesmr, primescnt, primescntf, primescntmr, primes_utils also improved mem efficiency/speed and extended range for primes and primenth|nthprime changed default PG in primenth|nthprime from P11 to P7, major refactoring of all methods
1.1.1 – more efficient/faster code to count up to nth prime in primenth|nthprime
1.1.0 – new nth prime approximation method in primenth|nthprime
1.0.6 – fixed n=1 check error for prime?
1.0.5 – minor bug fix
1.0.4 – fixed n=0 case for primenth|nthprime; fixed subtle bug in primes, refactored to generalize code
1.0.3 – minor bug fix
1.0.2 – directly test for cli command factor on installed platform at start
1.0.1 – check if using Ruby 1.8 at start, if so, require 'rational' library for gcd method
1.0.0 – initial release (April 1, 2015) with methods prime?, primemr?, primes, prime_division|factors, primenth|nthprime

Using primes-utils

primes-utils provide easy to use fast methods for generating, counting, and testing for primes, as well as factoring numbers. However, since there are an infinite number of primes, and real computers have a finite amount of memory and time to work with (as well as us), they can't do everything. However, within a range of *reasonable* applications, they do provide a great deal of utility, performance, and user friendliness, not provided in standard Ruby. So with these provisos in mind, let's explore how to get the best out of these methods in normal everyday use.

List of Methods

The following is a list of the primes-utils methods and their purpose.

Primaltiy Testing – prime? and primemr?
Factoring – factors|prime_division
Primes List – primes, primesf, and primesmr
Primes Count – primescnt, primescntf, and primescntmr
Nth Prime – primenth|nthprime
Show Available Methods – primes_utils

Gem Output

When primes-utils is loaded it will create output that looks something like this:

```
> require 'primes/utils'
Using cli 'factor' for prime? primesf  primescntf  factors|prime_division
Available methods are: prime? primemr? primes primesf primesmr primescnt primescntf
primescntmr primenth|nthprime factors|prime_division
```

This first line tells you your system has factor, and which methods use it, and the next lines lists the available methods for your system. If your system doesn't have factor the output will be as follows:

```
> require 'primes/utils'
Using pure ruby versions for all methods
Available methods are: prime? primemr? primes primesmr primescnt primescntmr primenth|
nthprime factors|prime_division
```

Here you're told all the methods use their pure ruby versions, and shows the list of available methods for your system, except for primesf and primescntf, which exclusively use factor.

You can see the available methods at any time by using the primes_utils method on any integer value.

```
> 9.primes_utils
=> "prime? primemr? primes primesf primesmr primescnt primescntf primescntmr primenth|
nthprime factors|prime_division"
```

All primes-utils methods are mixins to class Integer, and will operate on their progeny.
Below shows the class ancestry of primes-utils for MRI Ruby.

```
> 1.class.ancestors
=> [Fixnum, Integer, Primes::Utils, Numeric, Comparable, Object, Kernel, BasicObject]
> 1234567890.class.ancestors
=> [Bignum, Integer, Primes::Utils, Numeric, Comparable, Object, Kernel, BasicObject]
```

Primality Testing

Ruby provides the method prime? in the primes.rb library for integer primality testing. In primes-utils there are two flavors of prime?, and additionally the method primemr?.

If your operating system provides the cli command factor, upon loading primes-utils will detect and use it to create prime?, otherwise it will create a pure ruby version for it. Why do this? Because factor [10] is a very fast C implementation of a very fast factoring algorithm which can process very big numbers. Most Unix derivative OSes, like the Linux and BSD based distros, natively come with factor, so prime? will use it with them. It's a simple process (and code) to also use it for a boolean primality test.

The pure ruby prime? performs fast brute force trial division with primes generated using the P7 PG (see Mathematical Foundations). It performed *best* among the tested Ruby VMs, particularly JRuby [4]. Though not nearly as fast as the factor version, it's significantly faster (and useful for large numbers) than the ruby standard version, especially for very large numbers up to the range of at least 10^{17}.

primemr? uses the Miller-Rabin primality test [8]. It's strictly a primality test (doesn't find factors) and is probabilistic, whereas the other algorithms are deterministic. However, it's *extremely fast*, and is the only practical game in town for testing huge numbers with hundreds, or even thousands, of digits.

prime?

Determine if the absolute value of an integer is prime. Return 'true' or 'false'. This replaces the prime? method in the prime.rb standard library.

```
101.prime? => true
100.prime? => false
-71.prime? => true
0.prime? => false
1.prime? => false
```

primemr?(k=20)

Using Miller-Rabin probabilistic primality test determine if the absolute value of an integer is prime. The reliability (probability) can be increased by increasing the default input parameter of k=20.

```
111111111111111111.primemr? => true
111111111111111111.primemr? 50  => true
111111111111111111.primemr? => false
-3333333333333333.primemr? => false
n=10**1700; (n+469).primemr? => true
0.primer? => false
1.primer? => false
```

For the factor prime? version, primes and nonprimes take the same time to process, but the pure ruby algorithm aborts faster for nonprimes, and works well on even large numbers with *small* prime factors.

Use prime? whenever possible because it's deterministic. Don't use primemr? for *small* numbers. It's much more reliable for numbers > 10^{10}. Your hardware will determine the practical limits for using prime? vs primemr? On a Raspberry PI, vs a 2.5 Ghz laptop, vs a 4.2 Ghz worskstation, these points will vary.

Factoring

factors (and alias prime_division from prime.rb) performs prime factorization, and also come in two flavors like prime?. If your system has factor it will be used it to create factors|prime_division. This provides significantly faster performance than the pure ruby version, and also extends the size of usable numbers to the 10^{70} range (YMMV – your mileage may vary).

The pure ruby version performs a fast prime division algorithm using P13 as the default PG, but allows changing the PG at run time from P3 – P19. Testing on my system showed P13 or P17 gave optimal performance. In theory, because larger prime generators create fewer prime candidates, they should be more efficient, and thus faster. However, the modulus, and number of residues, increase significantly, such that at some point they become less efficient for the hardware, OS, and Ruby, to represent and use.

factors(p=13) or prime_division(p=13)

Determine the prime factorization of the absolute value of an integer. This replaces the prime_division method in the `prime.rb` standard library. Output is an array of arrays of factors and exponents: [[p1,e1], [p2,e2]..[pn,en]]. The pure Ruby versions uses P13 as the default Strictly Prime (SP) Prime Generator (PG), which can be changed on input. Acceptable prime parameter inputs are: [3, 5, 7, 11, 13, 17 19]. The factor version also takes an unused input parameter to maintain user-interface consistency.

```
111111111111111111.prime_division => [[1111111111111111111, 1]]
111111111111111113.prime_division => [[3, 1], [47, 2], [61, 1], [89, 1], [30883085111, 1]]
123456789.factors => [[3, 2], [3607, 1], [3803, 1]]
123456789.factors 17 => [[3, 2], [3607, 1], [3803, 1]]
123456789.factors(17) => [[3, 2], [3607, 1], [3803, 1]]
-12345678.factors => [[2, 1], [3, 2], [47, 1], [14593, 1]]
0.factors => []
1.factors => []
```

Be aware, *factoring big primes will take much longer than composites* (for either version) so you might want to check the primality of large numbers first with primemr?. Only numbers (> 5) ending with the digits 1, 3, 7 or 9 can be prime, thus numbers ending with the digits 0, 2, 4, 5, 6 and 8 are composites.

Just for fun, see how long your system takes for doing the following:

```
> n = 10**60; (n+1309267).factors
> n = 10**70; (n+1309543).factors
> n = 10**80; (n+1309503).factors
```

The factor versions of prime? and factors|prime_division take about the same amount of time for the same inputs because they both wait for factor to finish before processing its output. The pure ruby versions can be significantly faster for nonprimes, versus comparably sized prime values.

Of course, factors|prime_division also serve to indirectly determine the primality of an integer, except unlike prime|mr?, you have to interpret the results instead of being given a 'true|false' answer. The first example above shows the number is prime, as it returns a single value with an exponent of 1.

Primes List

primes-utils provides the methods primes, primesf, and primesmr, which return an array of the primes that exist in a number range designated by a *start* and *end* value (inclusive).

primes is a pure ruby implementation using the SoZ. It's best to use for *large* ranges and numbers up to some N. Depending on the number size, its faster than the other two methods. It uses system memory to perform the SoZ, so when the `end_num`, or range size, or number of primes to output, becomes too large it will fail gracefully with error messages telling you which of these conditions occurred.

primesf uses the factor version of prime? to check the primality of prime candidates identified within a range by the default P13 PG. It's much faster than primes for numbers between 10^{17} – 10^{40} and for *small* ranges, and is not memory dependent. Of course, if your OS doesn't have factor it won't be available to use.

primesmr is the same code as primesf except is uses primemr? to check the prime candidates. It's super fast, and can be used with numbers in the 100s, to even 1000s, of digits. Since it uses primemr? it is *not* deterministic, and will (infrequently) give incorrect answers (usually miss a prime), especially for *small* numbers, but you can run it multiple times for confidence.

primes(start=0), primesf(start=0), primesmr(start=0)

Return an array of primes within the absolute value range: |start| - |end|. The order of the range doesn't matter if both given: start.primes end <=> end.primes start. If only one number provided, all the primes up to it will be returned. Memory error messages are provided for primes.

```
50.primes => [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
50.primesf 125 => [53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103, 107, 109, 113]
300.primes 250 => [251, 257, 263, 269, 271, 277, 281, 283, 293]
n=10**100; (n-250).primesmr(n+250) => []
1000.primes(5000).size => 501
(prms = 1000000.primes(1000100)).size => 6
prms.size => 6
prms => [1000003, 1000033, 1000037, 1000039, 1000081, 1000099]
-10.primes -50  => [11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
n=10**20; n.primes n+n -> ERROR1: range size too big for available memory. => nil
n=10**20; n.primes 100 -> ERROR2: end_num too big for available memory. => nil
n=10**8;(25*n).primes -> ERROR3: not enough memory to store all primes in output array.=>nil
0.primesf  => []
1.primesmr => []
```

With 2.0.0 primes became more memory efficient, making it faster and usable with larger numbers (ranges), and primesf and primesmr were introduced. Because the latter are not memory dependent they will attempt to process whatever your system can represent. Use sensibly. For fun try these:

```
> n = 10**8; (x=(n-500000).primes(n+500000)).size
> x.first 100
> x.last  100
> n = 10**9; (x=(n-500000).primes(n+500000)).size
> x.first 100
> x.last  100
> x[23800..24000]
> n = 10**15; n.primesf(n+40)
> n = 10**400; (n-600).primesmr(n+600)
```

Primes Counting

With version 2.0.0 primescnt, primescntmr, and primescntf were introduced to provide faster and more memory efficiently methods to count primes within a range. Prior to them you had to do something like `100000.primes(100200).size` to get a primes count. Each method has it optimal use niche.

primescnt is a client to sozcore2 and adaptively selects the PG to use with it. If you want to count primes within very large ranges, on the order > 10,000 integers, then you should use it. My reference system can do ranges greater than 2,000,000,000 (2 billion), starting at very larger numbers ($> 10^{12}$).

primescntf uses the factor version of prime? to check the primality of prime candidates identified within a range by the default P13 prime generator. It's much, much faster than primescnt for *small* ranges around *large* numbers, and more memory efficient, because it doesn't use arrays to sieve primes from to then count. Of course, if your OS doesn't have factor it won't be available to use.

primescntmr is the same code as primescntf except is uses primemr? to check the primality of the pcs identified by the prime generator within a range. It's biggest benefit is it's super fast, and can be used functionally with numbers in the 100s, and even 1000s, of digits. However, since it uses Miller-Rabin, it is *not* deterministic, and will (infrequently) give incorrect answers, especially for *small* numbers.

primescnt(start=0), primescntf(start=0), primescntmr(start=0)

Provide count of primes within the absolute value range: |start| - |end|. The order of the range doesn't matter if both given: start.primes end <=> end.primes start. If only one number provided, the count of all the primes up to it will be returned. Memory error messages are provided for primescnt.

```
100001.primescnt => 9592
100002.primescnt => 9592
100003.primescnt => 9593
100000.primescntf 100500 => 40
n=10**400; (n-500).primescntmr(n+500) => 1
-10.primescnt -50  => 11
n=10**20; n.primescnt n+n -> ERROR1: range size too big for available memory. => nil
n=10**20; n.primescnt 100 -> ERROR2: end_num too big for available memory. => nil
n=10**8; (25*n).primescnt => 121443371
0.primescntf  => 0
1.primescntmr => 0
```

So here's the best use practice for these:
1) for *smallish* numbers and/or very large ranges use primescnt.
2) for *smallish* ranges, especially around large numbers, use primescntf if available.
3) for very large numbers, use *smallish* ranges and primescntmr, and run multiple times for confidence.
4) use the deterministic methods primescnt and primescntf whenever possible.

Nth Prime

With 2.0.0 primenth|nthprime was significantly improved, to find up to the 1.1+ billionth prime, with fairly uniform processing time, using a hash of indexed nth prime values. But I mathematically set the max nth prime up to the 1,122,951,705th prime. In 2.2.0 I removed this ceiling, and added more index nth primes up to the 1.6+ billionth prime, which was raised to 2+ billionth in 2.3.0. The only limitations now are the amount of available memory it can use, and computation time. You also get a nice error message for values that exceed the available memory capacity.

The method can take a runtime parameter to select the Prime Generator it uses. Originally in 1.0.0, P11 was the default PG. Then after redesign, refactoring, and testing for 2.0.0, P7 became the default PG. As of 2.5.0, if no valid PG parameter given it will default to adaptively selecting the best SP PG (either P5, P7, P11). Currently, the largest SP PG that can be selected is P13. Under normal use there's really no need to manually select the PG, especially since now it adaptively selects it. However, if you're curious how performance varies with different PGs (the reason I used this feature for) you're still free to knock yourself out.

primenth(p=0) or nthprime(p=0)

Return for the absolute value of the nth prime designator, its value. Default Strictly Prime (SP) Prime Generator (PG) adaptively selected at runtime. Can change SP PG used on input. Acceptable primes range: [3 - 13]. Indexed nth primes now up to 2.1 billionth. Memory error message is provided.

```
1000000.primenth => 15485863
1500000.nthprime => 23879519
2000000.nthprime 11 => 32452843
-500000.nthprime => 7368787
1122951705.nthprime => 25741879847
n = 10**11; n.primenth -> ERROR1: range size too big for available memory. => nil
0.nthprime => 0
1.primenth => 2
```

Mathematical Foundations

To use the methods you don't *need* to know what makes them work, but to understand the code you do. But in order to understand why the code is doing what it does you first need to understand the math and reasoning being used that directs the code. However, these are pretty simple and straightforward, so pull up a chair, open up your mind, and let's go on a journey.

Remember, we are trying to find *prime* numbers. This is an important point, as you will see, so we must not lose focus on our purpose. A prime has only one divisor, or *prime factor*, itself ('1' is not considered a *prime*). This is important for finding primes and *factoring* numbers.

Now wouldn't it be helpful if we had a nice simple and complete formula, or expression, to generate all the (infinity of) primes we could ever want? Something that is actually useable and easy to code? Well it turns out I "discovered" a simple expression that *can* generate all the primes, which I have cleverly called *Prime Generators*. All you need to know to understand them is a little arithmetic and logic.

If we wanted to write out all the integers we could do it in a linear fashion like this:

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 ….

or we could do it like this:

0	6	12	18	24	30	36	42	48	54	60	66	72	78	84	90	96	102	…..
1	7	13	19	25	31	37	43	49	55	61	67	73	79	85	91	97	103	…..
2	8	14	20	26	32	38	44	50	56	62	68	74	80	86	92	98	104	…..
3	9	15	21	27	33	39	45	51	57	63	69	75	81	87	93	99	105	…..
4	10	16	22	28	34	40	46	52	58	64	70	76	82	88	94	100	106	…..
5	11	17	23	29	35	41	47	53	59	65	71	77	83	89	95	101	107	…..

Instead of writing the integers in one long row, we write them like a matrix of rows and columns, in a *cyclic* manner. Here the cycle is 6 rows, while the number of columns go for as long as you want to represent integers. I call this a cyclic ***Integer Generator (IG)***, $\mathbf{I_n = mod*k+r}$, where **n|mod** is the size of the cycle (here it's 6), **k** represents a column number (k >= 0), and here **r** ε {0,1,2,3,4,5} is a **residue**. So the above IG is designated I6, which can represent any integer, e.g. 103 = 6*17 + 1, where mod = 6, k = 17 and r = 1. Try picking any integer you want and determining its I6 parameters.

Now, let's compare how integers are represented using I5 versus I6.

0	**5**	10	15	20	25	30	35	40	45	50	…..
1	6	**11**	16	21	26	**31**	36	**41**	46	51	…..
2	**7**	12	**17**	22	27	32	**37**	42	**47**	52	…..
3	8	**13**	18	**23**	28	33	38	**43**	48	**53**	…..
4	9	14	**19**	24	**29**	34	39	44	49	54	…..

0	6	12	18	24	30	36	42	48	…..
1	**7**	**13**	**19**	25	**31**	**37**	**43**	49	…..
2	8	14	20	26	32	38	44	50	…..
3	9	15	21	27	33	39	45	51	…..
4	10	16	22	28	34	40	46	52	…..
5	**11**	**17**	**23**	**29**	35	**41**	**47**	**53**	…..

What do you notice about the primes past the first column? Well, for I5 there are primes on each row but for I6 only two rows have primes. This primes distribution difference is similar for all odd|even cycles. What does this tell us? Even cycles are better for *exclusively identifying* primes, as the primes cluster on just a few rows. From this little insight, we can create simple, and fast, *Prime Generators*.

Creating Prime Generators

Since we know only even numbered cyclic IGs can exclusively identify primes, I'll show how to use I6 to create a Prime Generator of the form $\mathbf{P_n = mod*k+r}$, with r ε {1...mod-1}, k ≥ 0.

I6 = 6*k+(0,1,2,3,4,5)	P6 = 6*k+(1,5)
r=0: 0, 6, 12, 18, 24, 30, 36, 42, 48...	r=0: 6*k+0 is divisible by 6. No primes.
r=1: 1, 7, 13, 19, 25, 31, 37, 43, 49...	r=1: 6*k+1 has no immediate factors. Has primes.
r=2: 2, 8, 14, 20, 26, 32, 38, 44, 50...	r=2: 6*k+2 = 2 x (3*k+1). No primes.
r=3: 3, 9, 15, 21, 27, 33, 39, 45, 51...	r=3: 6*k+3 = 3 x (2*k+1). No primes.
r=4: 4,10, 16, 22, 28, 34, 40, 46, 52...	r=4: 6*k+4 = 2 x (3*k+2). No primes.
r=5: 5,11, 17, 23, 29, 35, 41, 47, 53...	r=5: 6*k+5 has no immediate factors. Has primes.

We visually see for I6 all the primes reside on rows 1 and 5, but let's characterize this mathematically. On the right I've factored the expressions for each row. Now we can easily see that all the integers on rows 0, 2, 3, and 4 are composites (and thus have no primes), while row 1 and 5 expressions cannot be factored because residues 1 and 5 are *coprime* to 6. We can do this to for any IG to see which residues exclusively generate primes. This is mathematically characterized by this simple rule: for a number to be prime the *greatest common divisor* of **r ε {1,2,3,..mod-1}** and **mod** is 1, i.e. **gcd(r,mod) = 1**.

For I6 only the residues 1 and 5 are coprime to 6 and can generate prime numbers, i.e. r ε {1,5} for primes, and thus P6 = 6*k+(1,5) or (6*k±1) is a ***Prime Generator (PG)*** which generates all primes > 3, where here I designate the generator by its *modulus* (**mod**) value. And because mod = 6 = 2*3 is the product of the first two primes, P6 is also what I call a ***Strictly Prime (SP)*** generator designated as P3. A key characteristic of PGs is they significantly reduce the number space to consider|search to find primes. For P6, residues {1,5} generate only 1/3 (2 of 6) of the whole integer space, while eliminating 2/3 (4 of 6) of all integers from consideration of being prime (which is a simple non-primality test).

While any even **mod** will create a prime generator, we want to select values to create "*good*" ones. By good I mean 1) for a given number of residues – **rescnt**, the ratio **rescnt/mod** is the smallest possible, and 2) the generated primes are in order, i.e. except for a small number of consecutive *excluded primes* the generator skips over, all the other primes are generated with no missing ones.

Example of property 1.
P4 = 4*k+(1,3) and P6 = 6*k+(1,5) are Prime Generators with two residues (rescnt = 2). They both generate all the primes "in order" with P4 all primes > 2 and P6 all primes > 3. However P6 is "*better*" because its **rescnt/mod** ratio is 1/3 vs 1/2 for P4. This is the percentage of the number space needed to consider to find the primes up to some N, one-third versus half of all integers.

Example of property 2.
P8 = 8*k+(1,3,5,7) is an in order generator as its residues have all the primes > 2 and < 8, but
P10 = 10*k+(1,3,7,9) is missing 5 as a residue, thus its excluded primes (2,5) are not in order.

For me, discovering PGs was a true revelation, hopefully now you see how and why they work. They generate all the primes, along with a (relatively) small number of nonprimes, and we can theoretically make the ratio of nonprimes–to–primes as small as desired. With this in mind, next I'll introduce the concept of generator *efficiency*, which defines a metric of how well generators generate primes.

Prime Generator Efficiency

A Prime Generator's efficiency is a metric of the number space it uses to generate primes up to some N, which I define as the ratio **rescnt/mod**. The lower the percentage then the smaller the number space a PG generates over some range, thus the higher its efficiency.

Strictly Prime (SP) generators are the most efficient. Their moduli all have the form **mod = $\prod(p_i)$**, and their **rescnt** = **$\prod(p_i\text{-}1)$** for primes 2,3,5,...p_i. Table 1 shows the efficiency of the first few SP generators and a few selected in-order non-SP mod values.

Table 1.

P_n	P3	P5	P60	P120	P150	P180	P7	P11	P13	P17
modulus	6	30	60	120	150	180	210	2310	30030	510510
residues count (rescnt)	2	8	16	32	40	48	48	480	5760	92160
% of rescnt/mod * 100	33.33	26.67	26.67	26.67	26.67	26.67	22.86	20.78	19.18	18.05
ef(P_n) = ln(mod/res)	1.099	1.322	1.322	1.322	1.322	1.322	1.476	1.571	1.651	1.712

Another efficiency metric is the ratio of nonprime–to–prime pcs for a given range. Table 2 shows 1) as the number range grows, the number of nonprimes per prime ratio grows (because prime density decreases as numbers get bigger) and 2) within a range, the number of nonprimes per prime decreases as the PGs (mod) gets bigger. So for N=10**7, P3 generates about 4 nonprimes for each prime while for P5 it's 3, and while this ratio increases as N gets bigger, it increases at at smaller rate for P5 than P3.

Table 2.

	N = 10**7			N = 10**8			N = 10**9		
	no. of pcs	no. of primes	nonprms per prime	no. of pcs	no. of primes	nonprms per prime	no. of pcs	no. of primes	nonprms per prime
P3	3333332	664577	4.02	33333332	5761453	4.76	333333332	50847532	5.56
P5	2666665	664576	3.01	26666665	5761452	3.63	266666665	50847531	4.24
P7	2285712	664575	2.44	22857141	5761451	2.97	228571427	50847530	3.50
P11	2077920	664574	2.13	20779220	5761450	2.61	207792206	50847529	3.09
P13	1918080	664573	1.89	19180819	5761449	2.33	191808189	50847528	2.77
P17	1805250	664572	1.72	18052534	5761448	2.13	180525354	50847527	2.55

So as SP generators moduli increase they use a decreasing percentage of the number space to generate primes from, which is the same as saying their nonprimes–to–primes ratios decrease.

I use a variety of SP PGs in various methods, based on how they tested and benchmarked for specific uses and implementations. Currently P5|7|11 are adaptively selected to use in primes, primescnt, primenth, and P13 for factors and the 4 methods that use sozcore1. This shows that the exigencies of language design and hardware *always* ultimately determine what works best, and not just mathematical metrics. Done in a different language, and/or on different hardware, the optimum choices may vary.

Distribution of Primes

For a given Prime Generator, the primes are uniformly distributed along each residue. Below is the tabulation of primes along the residues for P5 for the first 50 million primes it identifies.

Table 3.

Distribution of the 50 Million primes[4 – 50000003] = [7 – 982,451,809] for Prime Generator P5: Expected Mean (avg) = 50,000,000 / 8 = 6,250,000								
# primes	r=1	r=7	r=11	r=13	r=17	r=19	r=23	r=29
50M	6,249,269	6,250,543	6,250,224	6,249,930	6,250,078	6,249,248	6,250,326	6,250,382

This also means the number of nonprimes to be sieved are uniformly distributed along each residue too.

Residues Properties

Residues come in ***modular compliment (mc) pairs (mcp)***, e.g. for any modulus: **mod = r_i + r_j**
This means each residue **r** has a sibling whose value is **mod-r**. So if you find the residues < **mod/2** the rest are just their **mc**s.

The ***** modular operation within a residue group performs a distinct mapping, e.g. **r_i * r_j → r_k**
This means if you multiply, within a resgroup, a pc with residue **r_i** by another pc with residue **r_j** their product is a pc value in a higher resgroup with residue **r_k**. This mapping is unique for each resgroup residues pair and is a key property used for creating a fast generic prime sieve that works with any PG.

The first residue **r_0** is always 1, the second residue **r_1** is always a prime, and the last is always **mod-1**. For SP generators all the generated pc values < $(r_1)^2$ are prime. This property can actually be used to generate the primes up some N, but becomes practically infeasible beyond a certain number size.

Numerical Bandwidth

Analogous to the electromagnetic spectrum, a PGs modulus is akin to the *numerical bandwidth* it spans within the integer number space (numerical spectrum). The larger the modulus value the larger its numerical bandwidth. The residues are the fundamental prime frequencies within the bandwidth. If you want to find primes you only need to modulate along those frequencies. The k values represent the residues groups discrete modulation bandwidth chunks (quanta) that primes can only reside in.

The concept of numerical bandwidth plays a key role in determining how to adaptively select the best PG to use to process numerical ranges of varying sizes. As range sizes become larger, SP PGs with larger moduli can produce faster results, as they reduce the number space they have to process over the range. This process is akin to ***tuning*** the appropriate PG to optimally match the range size. Large PGs for small ranges can be overkill, while small PGs for large ranges can be overwhelmed. The beauty is we can dynamically tune the process (algorithm) to pick the best PG, instead of always using just one.

Putting Prime Generators to Work

Now that you've seen how to create Prime Generators, and learned some of their characteristics, we'll now see how to put them to work.

Currently, the most well known algorithm to generate primes is the *Sieve of Eratosthenes* (SoE) [6]. It's fairly simple and straightforward. To find the primes up to some N, write down all the integers up to N, then starting with 2, mark off all its multiples (4, 6, 8, etc). Then using the next unmarked off integer, which is 3, mark off all its multiples up to N, and continue using unmarked off integers until you get up to N. What will be left are the primes up to N.

Now there are many ways to optimize the algorithm, the simplest being to skip all even integers > 2, since by definition they are multiples of 2. This reduces the number space to search thru by half. Then skip all integers past 5 which end with 5. Just these two modifications reduce the search space by 60%. There are many others, but they mostly amount to reducing the necessary searchable number space, to reduce the algorithmic work, in order to increase speed.

The algorithm is called a sieve because we put all the integers up to N into a big bowl with holes for nonprimes, shake it up real hard, and what doesn't fall out are the primes. So we don't directly identify the primes, we directly identify the nonprimes and indirectly are left with the primes. So the key to all fast sieves is to reduce the number of nonprimes the sieving algorithm needs to identify and eliminate.

With Prime Generators though, it's theoretically possible to construct one which could directly generate all the primes up to any finite N. However in the real world, their moduli become too big for practical use. But we know SP PGs inherently generate fewer nonprimes as their moduli grow, making them perfect candidates for fast and efficient sieving algorithms.

While the SoE is ancient, I created the *Sieve of Zakiya* (SoZ) in 2008, and have improved it to be a very fast, efficient, and simple generic sieving algorithm which can use any Prime Generator. It forms the basis for the methods primes, primescnt, and primenth|nthprime, and allowed me to select the best PG for them based on their use case. For primenth|nthprime users can even change the PG at runtime.

The pure ruby factors|prime_division also allows changing the PG at runtime. Its PG generates the primes needed to perform trial-division with to see if a number can be factored. Similarly for the pure ruby version of prime?, the PG generates the primes to perform trial-division.

The methods primesf, primesmr, primescntf, and primescntmr identify|count the primes over a number range. They are algorithmically very fast and efficient, as their PGs generate the fewest number of pcs possible for any input number range. Here, the primes are directly identified by testing each pc with a fast primality test method. They are also the most memory efficient because they don't perform a sieve, and thus can be used with extremely large numbers without inherent memory limitations.

So for primes-utils, I use PGs to 1) generate primes to use for trial-division for factoring, 2) to generate pcs within a number range to test for primality, and 3) for the SoZ, which next we'll explore in detail.

Sieve of Zakiya

As stated previously, the key to producing a fast sieve is to reduce the number of nonprimes you have to identify and eliminate. We see Prime Generators inherently already do this. By using PGs we already are working with a reduced number space and only have to identify the nonprimes for each residue. As it turns out, there's an extremely simple and fast way I created to do this, the *Sieve of Zakiya (SoZ)*.

The SoZ [1-4] first creates a table representing the prime candidates a PG generates up to some N. To identify the nonprimes, each prime within a residues group of the table multiplies every member of that resgroup. These rescnt products (first multiples) map to pcs on distinct restracks in higher resgroups. Starting from each first multiple, every primenth resgroup pc along a restrack is a prime multiple and marked as nonprime to the end of the restrack. Thus for every prime ≤ sqrt(N), all their multiples in the prime candidates table are marked as nonprime, leaving the primes ≤ N.

SoZ Algorithm

To find the primes ≤ N

1. choose a Prime Generator (PG), which has the form: Pn = mod*k + (1, r1..mod-1)
2. create a table of prime candidates ≤ N of form: pcs = [mod*k + (r1..mod+1)], where k = 0,1.. represents a residues group of rescnt (number of residues) prime candidates values
3. set prime = first prime candidate (pc)
4. set primesetp = prime * mod
5. perform sieve with prime value (can be done sequentially or in parallel):
    - multiply prime by each pc value in its residues group
    - mark each of the pc values of these products in the pcs table as nonprime
    - along the residue track of each product pc, successively add primestep and mark these pc values as nonprime for the entire residue track
6. set prime to next unmarked pc value in pcs table and repeat from 4 while prime ≤ sqrt(N)

One thing to note, though a PG's first residue value is always 1, I always start the pcstable with the second residue value (always a prime) and end each resgroup with the residue 1 pc values. This makes the algorithm simpler and faster to code because the first table pc is guaranteed to be prime this way.

Only the primes ≤ sqrt(N) are used to perform the sieve. For any of them, when one of their first multiples is > N the sieve for it can be terminated, as all other bigger first multiples will be > N.

A very nice property to note is that the prime multiples along each residues track can be eliminated simultaneously, as they are independent processes, thus enabling the sieve to be done in parallel. This makes the algorithm particularly conducive to fast implementations with multi[core|threaded] cpus and systems. The algorithm is also implementable as a segmented sieve (serial or parallel), which you can see in my paper *The Segmented Sieve of Zakiya (SSoZ)* [3].

Next we'll see a real example, and walk thru the algorithmic steps of the SoZ to show how simple it is.

SoZ Example

Find the primes up to N = 541 (100th prime) using SP Prime Generator P5.

Table 4. Prime Candidates Table Illustration of Sieve of Zakiya for P5(541)

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

We perform the SoZ algorithm as follows:

1. choose prime generator P5 = 30*k + (1,7,11,13,17,19,23,29)
2. create table of prime candidates of form: [30*k + (7,11,13,17,19,23,29,31)], k = 0,1,2..17
   the residues track pcs are: r1 = 30*k+7, r2 = 30*k+11, ... r7 = 30*k+29, r8 = 30*k+31
   where each column is a residues group of prime candidates for k = 0,1,2..17
3. set prime = 7 (first unmarked pc value)
4. set primestep = prime * mod = 7 * 30 = 210
5. perform sieve with prime = 7 (can be done sequentially or in parallel):
   multiply prime by each member of its residues group
   for prime = 7: 7 * [7,11,13,17,19,23,29,31] => [49,77,91,119,133,161,203,217]
   starting at these pc values, successively add primestep=210 and mark these pc values as nonprime along the residue track (row) they occur on, until the end of the table
6. repeat from 4 with successive unmarked pcs (11,13..23) for the primes ≤ (sqrt(541)= 23)

The colored pc values in Table 4 are prime multiples, with the same colors multiples of the same prime. Starting with 7, it multiplies its resgroup to produce its first multiples [49,77,91,119,133,161,203,217] which reside on distinct restracks (blue). Starting at 49 we add primestep (210) for its next multiple (259), continuing to the end of that restrack, then similarly with the other first multiples. We then use 11 (next unmarked pc), whose multiples are marked (red) similarly. The easier equivalent process is to mark each primenth column to the right of each product multiple as nonprime for each prime. Thus for 7 each 7th column is marked blue, for 11 each 11th column red, etc. Six colors are displayed because the sieve only uses the six primes up to 23, which is the largest pc (and prime here) ≤ sqrt(541).

The first color shown for each prime is its square value. When two primes are multiplied, like 7*11, their product (77) is marked first with the lower value's color (blue here) but is used again as the first multiple for the larger prime if needed. You can see in the table two jumps of 7 columns from 77 are blue (287, 497) and one 11 column jump is red (407).

The nonprime pcs in a resgroup play an interesting role in the sieving algorithm. Though skipped for generating first product multiples, we need them because their residue values are coprime to mod, and when multiplied by a prime they map that first product to its distinct restrack to begin the sieve from. So if prime 53 is used in the sieve, it still needs to multiple 49 to determine the distinct residue track and resgroup that residues pair (19 and 23) maps to so its multiples on that restrack can be eliminated.

Also, for the primes in the table, we only have to eliminate a reduced subset of their multiples, not all their multiples. Remember, a PG eliminates all multiples of its excluded primes from consideration, so for a prime in the table only its multiples with other primes in the table need to be eliminated. Thus for 7, 11, 13, etc, all their multiples with 2, 3, and 5 are already eliminated by the structure of P5.

Thus we see from the table, out of 144 (18*8) pcs 47 nonprime pcs are sieved out leaving 97 primes. Those 97, plus 2, 3, and 5, the 3 excluded primes $p_j < (r1 = 7)$, constitute the 100 primes up to 541. Here the primes outnumber the nonprimes by over a 2 to 1. If we used a higher SP generator like P7, the ratio would be even higher over this range.

Finally, notice the SP PG property that the pcs $< (r1)^2$ are prime, where here all the pcs < 49 are prime.

Table 4 provides a good visual picture of how the SoZ works and how structurally more efficient it is compared to the SoE. A similar picture can be created for any PG to see how to find primes up to an N.

However, in many cases we don't want to find the primes up to some number N, but rather we're more interested in finding the primes, or their count, over some number range. This is particularly true in certain fields that need large primes to work with. It would be really inefficient to have to find all the primes up to a billion if you only really want to know the primes in some range in that number region.

The basic SoZ can be simply modified to enable it to efficiently find the primes over (theoretically) any number range, using any PG. It has the same essential structure of the basic algorithm, and uses the same operations, but just applied to the number range of interest, skipping over the nonrange pcs.

There are just two distinct cases the modified algorithm has to identify and contend with.

Case 1: the starting value of the range is > the square-root of the end value of the range.
Case 2: the starting value of the range is ≤ the square-root of the end value of the range.

For Case 1, two arrays are created, which the SoZ is used to sieve out the nonprimes from both.
For Case 2, the SoZ sieves one array of all pcs to `end_num`, then the range primes are selected from it.

Finding primes over a range is an essential task primes-utils performs, so let's see algorithmically how to approach this to get it done.

Finding Primes over a range

The basic SoZ algorithm to find primes up to some N needs to be only slightly modified to enable it to find the primes over a range, for which there are two distinct case scenarios to contend with.

Case 1: start_num > sqrt(end_num)

When the `start_num` for a range is greater than the square-root of the `end_num` (sqrt(`end_num`)) two pc tables|arrays are created, one for the pcs ≤ sqrt(`end_num`), the other for the pcs within the range.

Table 5. Split pcs tables for P5 for pcstable range [337 – 541]

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

Table 5 shows the pcs needed to find primes within the pcstable range [337 – 541]. This corresponds to an input range with start values ranging from 332-337 and end values from 541-546. That's because 337 is the first pc value in the table for any `start_num` > 331 (last pc in k10 resgroup) and 541 first for any `end_num` < 547 (first pc in k18 resgroup). Thus [335 – 545] is an input range the table represents. The task is to sieve the nonprimes within k11 – k17, which requires using the primes ≤ sqrt(541)=23 from the six pcs in k0 that are ≤ 23, which here all happen to be prime.

We start the sieve in the basic manner using 7, the first prime. It multiplies all the pcs for k0 as before, 7*[7,11,13,17,19,23,29,31] => [49,77,91,119,133,161,203,217]. Though using only the six primes up to 23, each prime still multiplies all its resgroup members to map these first multiplies to their unique res[group|track] pc locations. These values aren't in the table, but are used to jump from into the range.

To do these jumps I need to define some variables. Using zero-based indexing (table|array[0] is first), from Table 4 we see 49 is at location 12, on r5. From here, each 7th pc along r5 would normally be marked nonprime. Let `nps` indicate these prime multiple (nonprimes) locations. As before, a prime's jump size is `primestep = rescnt*prime` pcs. Now define `pcs2start` as the number of pcs < the first pc in the range. Finally, `maxpcs` is the number of pcs in the range [`start_num` – `end_num`].

Using these parameters, we determine for each sieve prime along each restrack whether it has a first prime multiple in the range in the following manner.

Starting from any `nps` on a restrack we compute the number of pcs from it to the first pc in the range, (`pcs2start – nps`) pcs. From this value we suck out any integral number of `primestep` pc jumps to find the remaining pcs left before the first pc in the range, which are `(pcs2start – nps)%primestep` pcs, i.e. (`0...primestep-1`) pcs. If 0, the number of pcs from `nps` to the first pc in the range is an exact multiple of `primestep` pcs. If not 0, we subtract it from `primestep` and check if this value is less than the number of pcs in the range. If it is, this value is the first prime multiple location in the range along that restrack, otherwise it's beyond the range, and the prime has no multiples in the range along that restrack.

Thus, to determine if a prime has a multiple along a restrack in some range, start from an `nps` and do:

nps = (pcs2start - nps) % primestep where % is the mod operator
nps = primestep - nps if nps != 0
if nps < maxpcs perform sieve along restrack from this location to end of range

Let's do an example for 7s first prime multiple 49.
For 49, `nps`=12, `primestep`=8*7=56, `pcs2start`= 8*11=88, and `maxpcs`=8*7=56, thus:

nps = (88 - 12) % 56 = 76 % 56 = 20
nps = 56 - 20 = 36 because 20 != 0
since 36 < 56, from address 36 perform sieve along its restrack to end of range

From Table 5, range[36]=469 is the first, and only, multiple of 7 in the range along r5, as a subsequent jump of 7 pcs along the restrack will exceed the range.

In Table 5 we see 7 has a multiple on each restrack. This is arithmetically predictable because `maxpcs` and `primestep` are equal (56), so `nps=(pcs2start - nps)%primestep` is a value from (0 – 55), which is less than `maxpcs`, and thus in the range. A resulting `nps` can only be 0 if it's on the same restrack as the first pc in the range, or equal `maxpcs-1` unless it's on the same restrack as the last pc in the range.

The pcstable range [427 – 539] shows these two conditions. The first multiple for 7 on r1 in Table 4 is 217, which is one primestep jump from the first location in the range of 427, thus its range `nps` is 0. Conversely for 119 on r7, it's exactly two `primestep` jumps from 539 so its `nps=maxpcs-1=30`. As an exercise, determine their necessary parameters and fill in the numbers and verify these results yourself.

Because the other sieve primes primesteps are > `maxpcs` some won't have multiples along each restrack in the range [337 – 541], but each sieve prime does have at least one multiple in the range, as we can see in Table 5 there are six colors in the range, representing a prime multiple for each sieve prime.

To be clear, we can use any prime multiple along a restrack, and not just the first, to project the prime into the desired range. For the input range [9,000 – 10,000] we need the primes < 100 to sieve with. From Table 4 we see 97 is the last pc that's < 100, so we would use all the primes up to it. However, we still have to sieve out any nonprimes up to 97. So starting with 7, it first sieves out it multiples <= 97 and then is projected to the range. So 7 first sieves out 49 on r5 then jumps to 259, and since 259 > 100 its the first multiple location (`nps`) used to compute 7s projection along r5 into the region. Then 7 sieves out 77 on r4 from the pcs < 100, and uses `nps` = 287 to project itself into the range along r4.

But what happens for ranges where first prime multiples are inside it? Does this technique still work? Yes it does (or I wouldn't be writing this), but with one arithmetic stipulation. The modular operator **%** used to preform (n % mod) must return positive values for negative numbers of n, e.g. -9 % 30 = 21. That's it, and everything works out as before. So let's look at an example and show that it works.

Table 6. Split pcs tables for P5 for pcstable range [41 – 173]

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												
r2	11	41	71	101	131	161												
r3	13	43	73	103	133	163												
r4	17	47	77	107	137	167												
r5	19	49	79	109	139	169												
r6	23	53	83	113	143	173												
r7	29	59	89	119	149													
r8	31	61	91	121	151													

Table 6 shows the pcs needed to find primes within the pcstable range [41 – 173], which corresponds to start values of 38-41 and end values of 173-178. Because `start_num`=41 > sqrt(`end_num`=173)=13 we create two arrays, and proceed as before. The sieve primes array contains just the 3 pcs [7, 11, 13], and the range array the pcs from [41 – 173], where range[0]=41. Here prime 7s first multiple, 49, is in the desired range but we'll see that everything still works out.

We still perform the following algorithm:

nps = (pcs2start - nps) % primestep
nps = primestep - nps if nps != 0
if nps < maxpcs perform sieve along restrack from this location to end of range

and for 49, `nps`=12, `primestep`=8*7=56, `pcs2start`= 9 and `maxpcs`=37, thus:

nps = (9 - 12) % 56 = -3 % 56 = 53
nps = 56 - 53 = 3 because 53 != 0
since 3 < 37, from address 3 perform sieve along its restrack to end of range

and we see range[3] = 49 will be marked as nonprime to start the sieve within this range on r5.
As an exercise, check that the other prime multiples range addresses are correctly identified too.

In both theses examples, the split tables are represented as linear arrays of pcs, and in each case we need to find the smallest gap size of pcs from the start of the range array to a prime's closest multiple. If the closest multiple is before the range we jump primestep pcs from there and see if it lands in|over the range. If the first multiple is past the start of the range, we project its location to before the range and then check if another forward primestep jump is in|over the range. Modular math makes it simple!

Case 2: start_num <= sqrt(end_num)

This is the much simpler case. If `start_num` is ≤ the sqrt(`end_num`) we create one pcstable for the range [0 – `end_num`], perform the basic SoZ on that, and then numerate|count the primes just within the range [`start_num` – `end_num`]. This is much simpler than creating two tables, where the range table consists of possible primes ≤ the sqrt(`end_num`) needed for sieving, and the sieving primes table consists of pcs in the range needed to be sieved, and figuring out the correct logic to do this efficiently. For example, for the input range [20 – 545], the pcstable range is [23 – 541], so we create Table 4 because the starting pc equals 23, the sqrt(541), and then just use the identified primes from [23 – 541].

Splitting the algorithm to process these two distinct cases of input ranges makes it much more efficient. In fact, it's the only effective way to find primes around big numbers without using up system memory. For example, for the input range [999,000 – 1,000,000] we create just two small arrays, one to find|hold the primes < 1000 and another for the pcstable for the range. This is much more memory efficient, and significantly faster, than sieving up to 1,000,000 just to identify the primes in the last 1000 integers.

We can now combine these two cases to create one algorithm to perform the SoZ over any range.

SoZ Algorithm for range [start_num – end_num]

1. choose a Prime Generator (PG), which has the form: Pn = mod*k + (1,r1..mod-1)
2. if start_num ≤ sqrt(end_num)
    a) create table of pcs ≤ end_num of form: pcs = [mod*k + (r1..mod+1)]
    b) set prime = first pc in table
    c) set primestep = prime*mod
    d) perform sieve with prime value (can be done sequentially or in parallel):
        - multiply prime by each pc value in its resgroup
        - mark these pc product values in pctable as nonprime
        - successively add primestep to these pcs and mark as noprime on each restrack
    e) set prime to next unmarked pc in table and repeat from c) while prime ≤ sqrt(end_num)
    f) select the primes in the range [start_num – end_num]
3. if start_num > sqrt(end_num)
    a) create sieve_primes table whose size is the number of pcs ≤ sqrt(end_num), and range_primes table whose size is the number of pcs (maxpcs) in the range
    b) set prime = first pc in sieve_primes table
    c) set primestep = prime * mod
    d) perform sieve with prime value, as above, on sieve_primes pcs
    e) for each restrack nps > sqrt(end_num), perform sieve on range_primes pcs by doing:
        - nps = (pcs2start - nps)%primestep
        - nps = (primestep - nps) if nps != 0
        - for nps = (0 – maxpcs-1) perform sieve in range starting from this address
    f) set prime to next unmarked sieve_primes table pc, repeat from c) while ≤ sqrt(end_num)
    g) select primes in range_primes table

Directly Finding Primes

Performing some kind of sieve is normally the way primes are identified from some set of integers, and using PGs can do that very efficiently. However, as I alluded to earlier, we can also use PGs to directly identify primes without having to perform a sieve. All we have to do is create an SP PG whose $(\mathbf{r_1})^2$ value is > `end_num`. But as I also said, this becomes impractical very quickly as numbers get large.

However, there's still a way to use any PG to psuedo-directly identify primes over a range without performing a sieve. All we need is a *fast number theoretic* method for primality testing, and apply it to the pcs a PG generates in a range. A really good benefit of this is we don't need to use any memory to represent integers to sieve over, making the coding much simpler. The task is to find a primality testing method fast (and accurate) enough to make this technique viable.

So from Table 5, we just generate the 56 pcs in the range, and directly test if each pc is prime. For large numbers and small ranges this works best. For ranges that contain large numbers of pcs, doing the SoZ may be faster than testing each pc individually. It just depends on the size of the parameters.

Starting with 2.0.0 I created the methods primesf, primescntf, primesmr, and primescntmr to perform the above technique to numerate|count primes over a range. They only differ by the primality testing method they use. This technique becomes the preferred (and really only practical) method to find primes within ranges of *very large* numbers. It's also much simpler to code than a sieve, much faster for its use cases, and easy to upgrade in the future to use the latest greatest primality testing method.

This ends my discussion here of how I put Prime Generators to work in primes-utils, though it doesn't end the discussion of all the ways they can be used. Hopefully you now have a good foundation for understanding what PGs are and how to use them. If you do, you should be able to follow the Ruby source code as it applies the algorithms and techniques presented to perform the primes-utils methods.

Ruby Fundamentals

As an aide to new programmers, and people coming from different languages, this is a short tutorial on Ruby. It uses specific source code snippets to better enable you to understand Ruby syntax and idioms. Hopefully you'll feel the urge to explore Ruby deeper to learn|appreciate its full power and flexibility.

Object Oriented

Ruby is a native Object Oriented (OO) language where mostly everything is an **object** of some **Class**. Classes are open, which allows you to add new methods to them, or change the behavior of existing ones. You typically create a **Module** of code of specific functionality to **Include** into a Class.

All the primes-utils methods are an **instance_method** for **class Integer**. This means they only operate on integers. So if you try something like **`(1.0).prime?`** or **`Math::PI.primemr?`** you'll get error messages because **1.0** and **Math::PI** are objects of **class Float**.

The general source code structure for the primes-utils rubygem has the following form:

```
module Primes
  module Utils
  ...
  ...
  end
end

class Integer; include Primes::Utils end
```

The Class ancestry tree for prime-utils for MRI 2.2.3 is shown below.

```
> 1.class.ancestors
=> [Fixnum, Integer, Primes::Utils, Numeric, Comparable, Object, Kernel, BasicObject]
> 1234567890.class.ancestors
=> [Bignum, Integer, Primes::Utils, Numeric, Comparable, Object, Kernel, BasicObject]
```

Public|Private Methods

Visibility to the user of methods is determined by whether they're created as `public` or `private`. Public methods are ones you can use (primes, primescnt, etc) while private methods are invisible to you and only accessible to other public and private methods. Thus, you can't use private method sozcore2 as `5.sozcore2`.

By default methods are created as public. The `private` keyword is used to designate methods following it as private. In the code I create a private, then a public, and then another private section of code.

Blocks

A code block is an amorphous group of expressions contained within curly brackets **{…}** or **do … end** statements that are passed to methods, where the later is normally used with code that spans multiple lines. *Blocks are powerful!* Both examples of passing a block to the `times` method are shown below.

```
rescnt.times {|i| pos[residues[i]] = i-1}
rescnt.times do |i|
  pos[residues[i]] = i-1
end
```

Booleans

Ruby has three boolean types, `true`, `false` and `nil`, where `nil` acts as boolean `false`. All objects not specifically `false` or `nil` are evaluated as boolean `true`. Thus **`if 3`** and **`if []`** are evaluated as **`if true`**. When a variable is assigned a value that doesn't exist it's assigned `nil`. Thus for the expression

`nth = nths.keys.sort.reverse.detect {|k| k < n}`, `nth = nil` if no key array value k is less than n.

In Ruby it's also typical that methods ending with **?** return boolean values. Thus `d.even?` (in primemr?) returns `true` if d is even or `false` if odd. Accordingly, prime? and primemr? behave similarly.

As in many languages, **!** denotes boolean negation, and can be used as **`!arg`** or **`! arg`** (also **`not arg`**).

Conditionals

`if <boolean>; <do something> end` can be written as **`<do something> if <boolean>`**. Ruby also has the ternary operator: **`<boolean> ? <do when true> : <do when false>`** which exists in C/C++ and other languages and is equivalent to: **`if <boolean> then <do true stuff> else <do false stuff> end`** In Ruby this returns a value which can be assigned to a variable, e.g. in primenth:

```
prmcnt = n > nth ? nth-1 : primes.size.
```

Also in primenth, the following code snippet is an example of *conditional compression*,

```
return n > 0 ? seeds[n-1] : 0 if n <= seeds.size
```

whose e quivalent long form is:

```
if n <= seeds.size
  if n > 0
    return seeds[n-1]
  else
    return 0
  end
end
```

`<do something> unless <boolean>` is equivalent to **`<do something> if !<boolean>`** so that `<do something>` will happen if `<boolean>` is `false` (do unless `true`).

Loops

Ruby has a rich set of loop forms, some unique to it. I use `times` and `while` and `each` loops in the code. A loop like: **`n.times do … end`** (n is an integer) will iterate n times. The form: **`n.times do |i| … end`** makes **i** available to use within the loop with the values (0,1,2,3,4,5..n-1), as Ruby uses zero indexing.

I use two forms of while loops. The most frequent is: **`while <boolean>;`** ... **`end`** where `<boolean>` is an expression, or value, which is either `true`, or `false|nil`. Like conditionals, you can also use the reverse form: **`<do something> while <boolean>`** where `<do something>` can be a single statement, or multiple statements enclosed in parentheses and semicolon separated, e.g. in primemr?.

Arrays

Ruby arrays are zero indexed and can contain just about any type of object (integers, floats, strings, other arrays, etc). `arry.size` gives the number of elements in `arry`, `arr[0]` the first element, `arr[-1]` the last, and `arry[1..-1]` gives a subarray of the second to last elements. Arrays have an extensive set of methods for its Class and are extensively used in the code.

Hashes

A hash is a data structure consisting of key|value pair objects of most any Class, though typically keys are strings, symbols, or numerical objects. The key|value pairs are mostly both integers in the code, e.g. for `nths`. Like arrays Ruby provides an extensive set of methods to operate on hash objects.

Assignments

Ruby provides multiple ways to assign variables values.

It provide math operators like **`r += 1`** and **`n /= p`** which perform the shown math operation on the old value and reassigns it back to the variable, equivalent to **`r = r+1`** and **`n = n/p`**.

A special assignment operator is `||=` which is used like **`v ||= 0`**. Here, if v is `nil` or `false` the value 0 is assigned to it, but if v already has some value it's unaffected. This is typically used to initialize a variable to some value if it was previously unassigned, or to do so after its conditional creation.

`primes = [2,3,5,7,11,13]` and **`factors = []`** are Array variables, where [] is a size zero empty array. **`modk,r,k = 0,0,0`** is equivalent to **`modk=0; r=0; k=0`** while **`v = val = 6`** assigns 6 to both variables. **`num, start_num = start_num, num`** switches the variables values.

Array elements can be assigned like: **`a, b, c = [1, 2, true]`** so that **`a = 1; b = 2; c = true`** while the snippet **`start_num, nth, nthflag = set_start_value(num,false)`** assigns the three left variables to the three element array output values from set_start_value.

Splatting

A special form of array element assignment is known as *splatting*, which provides a convenient way to assign only a subset of array elements to a few variables. The sozcore2 method returns an 8 element array of parameters, but depending on the methods that calls it, only 2, 6 or all 8 elements are used. By arranging the array elements appropriately, the use of splatting makes variable assignment easier, where the ***** splat operator eats the unused array elements so they don't have to be specifically assigned.

```
prms, m, * = sozcore2(num, start_num, mod)
prms, m, modk, residues, rescnt, pcs2start, *  = sozcore2(num, start_num, mod)
prms, m, modk, residues, rescnt, *, maxprms, r = sozcore2(num, start_num, mod) (Ruby 1.9+)
```

Method Chaining

A frequent practice in Ruby is to chain methods to do a sequence of operations with one-liners.
In set_start_value we see a good example of *method chaining*.

`nth = nths.keys.sort.reverse.detect {|k| k < n}` could be written as:

```
nth = nths.                 # starting with the nths Hash object
      keys.                 # create an array of nths' keys with the Hash#keys method
      sort.                 # sort the array elements with Array#sort method
      reverse.              # reverse array elements with Array#reverse method
      detect {|k| k < n}    # nth assigned first array value k < n, or nil if no k < n
```

The key thing to keep track of is what Class object you are operating on at each stage in the chain. In the example a Hash is the first stage object, which is turned into an Array of key values, whose output ultimately becomes a single Integer value, or `nil`, which is finally assigned to the `nth` variable.

Error Handling

Ruby provides various means to catch and process error conditions in a structured manner. The only errors that will occur with normal use for a few methods are out-of-memory errors when processing numbers and ranges that require array lengths that exceed the available amount of system memory.

In version 2.2.0 I added error handling to sozcore2 to catch its out-of-memory errors, so if an array tries to be created with insufficient memory it will fail gracefully with an informative error message. In 2.4.0 I also added error handling in primes to handle when its output array didn't have enough memory too.

```
while m < maxprms             # find primes from sieved pcs in prms for range
  begin
    primes << modk + residues[r] if prms[m] == 1
  rescue Exception
    return puts "ERROR3: not enough memory to store all primes in output array."
  end
  r +=1; if r > rescnt; r=1; modk +=mod end
  m +=1
end
```

Writing code to catch known possible errors like these is good programming practice, and will save the programmer hours of troubleshooting, and the program's users loads of possible headaches.

Optimizations

Ruby has multiple ways to do the same thing, though some are faster than others. A good site that lists various optimizations is https://github.com/JuanitoFatas/fast-ruby [15], which presents benchmark results of comparative techniques. Some of these I was aware of before writing my code, but a few I've used after getting things to first work.

One subtle optimization I found was changing `prms` array values from `true|false` to 1|0. This allowed doing simple arithmetic on it to count the number of primes (1s). Originally I did `prms.reduce(:+)` to do that but `prms.count(1)` is faster.

The code below assigns the last value in an array that is less than some n. I originally did it the first way, but the second way (from the above website) benchmarked to be consistently somewhat faster.

```
nth = nths.keys.sort.select {|k| k < n}.last       # find largest indexed key < n
nth = nths.keys.sort.reverse.detect {|k| k < n}    # find largest indexed key < n
```

I also use `while` loops a lot because they are most times faster than other looping constructs.

Dependencies

primes-utils uses just a few methods that aren't automatically available and have to be '`required`' or explicitly invoked. It directly invokes from the **Math module** `Math.sqrt(n)` and `Math.log(n)`. I could have first done an **`include Math`** to load the module, and then just do `sqrt(n)|log(n)`, but I only used these two a few times so it was easy enough to invoke them. I also use the **openssl** standard library for primesmr? and thus do a **`require 'openssl'`** to use it. I also use the gcd method, which since ruby 1.9 has been a class Integer method. Before that it was in the **rational** library, so upon loading, I check for Ruby 1.8 and do a **`require 'rational'`** if it is. (Hopefully *you're* not still using Ruby 1.8. or 1.9)

Coding primes-utils

Coding is the art of story telling! It tells the story of how a problem or task is attempted to be solved or performed by a programer in a specific programming language. The same programmer with a different language may tell the story differently; another programmer using the same language may (probably) tell the story differently too. Thus, programming is a highly personal endeavor reflecting not just the cognitive understanding and conceptualization of a problem or task but also the personality of the programmer, and the choices s/he makes in using a language to tell the story. This is my story for coding primes-utils.

I tell the story of prime-utils here in the language Ruby. Ruby is a very nice language to tell stories in. It's a dynamic Object Oriented (OO) language. This means I didn't have to worry about defining types for variable, or the sizes of arrays *before* I could start writing the story, or be bothered with having to compile code before I could run short snippets. I could just write things and see how they behaved, immediately. Having this luxury allows you to experiment quickly, and extensively.

All my development of the math and algorithms underlying Prime Generators and the Sieve of Zakiya was done in Ruby. I have subsequently written versions in Forth, Python, and C/C++, but they were translations of the basic story told in Ruby, not different conceptualizations of the story. Maybe in a significantly different language like Haskell the story would be conceived differently. I don' know? But the Ruby code provided here can act as a Rosetta Stone reference for others storytellers.

There are 2 story lines being told as you read, one that illuminates the mathematical foundations of the code's functionality, the other illustrating the particular coding mechanisms and techniques used. I use the code's comments to explain what it's doing functionally, so even if you don't fully understand (care) exactly how the code works you *can* know what it's doing. These two stories are actually independent. If you're into the math, its story is compelling enough just to read alone, and if you're into coding, how I design and implement it makes an interesting and instructional read too.

I separate the code into two parts I'll explain; the **private methods** which do the heavy computational processing and provide common functionality, and the **public methods** that are available to the users. My explanations are focused on the operation and functionality of the code and not how a particular instruction or snippet works. There are scores of books, articles, and videos that explain these things. I'll be more focused on explaining what the code is doing on the conceptual and functional level.

However, hopefully you will see that by picking good descriptive names for variables and methods, and providing clear descriptive code comments, the code describes itself and becomes self-documenting. This is true for any language, but can be enhanced even more with Ruby by using its native idioms and instructions to write (and think) in instead of translating from another language paradigm to it.

For example, you will see no `for` loops in the code, but lots of `each` and `times` loops. Ruby has `for` loops, but it's idiomatic to use the later forms. Another thing that really frees your mind is being able to stuff anything into an array (including other arrays) besides numbers and strings. And because arrays are dynamic, you can grow, shrink, and reassign them without memory allocation concerns.

Ruby also allows you to write conceptually concise code that does a lot, if you use its idioms wisely. For example in primemr? I originally wrote these two line as:

```
return true  if n==2 or n==3
return false if n%6 != 1 && n%6 != 5 or n < 2
```

But in the course of writing this book I did a lot of code cleaning and *idiomizing*, so now they are:

```
return true  if [2,3].include? n
return false unless [1,5].include?(n%6) and n > 1
```

Actually here the original code is shorter, so why change it? Because for me the changed form is more idiomatic, more uniform, and more flexible. If I want to check for more small prime values of n in the first form I'd need to add a `'or n==x'` statement for each one, but only need to add them to the array for the second form. And for the second line, in the first form even I had to rethink what the logic was doing after seeing it again, while the second form is logically clearer – return false for n (it's not prime) unless the residue of n with 6 is 1 or 5 and n is greater than 1. To pick a different PG residue test I only need to change the array to its residues and use its mod value. Simple, clear, and straightforward.

Here using the arrays allows me to succinctly represent the math of checking against a list of small primes or a list of a PG's residues. It wasn't the way I originally wrote it because I used the thinking paradigm I learned growing up with procedural languages. So as with human languages, software languages determine the scope of how you perceive the world, and express its reality and functioning.

Ruby also has native error handling capabilities that almost coerces (shames) you into using it. Here is the code snippet in 2.1.0 sozcore2 which creates the arrays `prms` and `prms_range` for sieving primes.

```
ks = (start_num-2).abs/mod    # start_num's resgroup value
maxpcs = maxprms              # if ks = 0 use this for prms array
if ks > 0                     # if start_num > mod+1
  maxpcs = pcs2sqrtN          # find primes in pcs upto sqrtN
  pcs2ks = rescnt*ks          # number of pcs upto ks resgroup
  max_range  = maxprms-pcs2ks # number of pcs from ks resgroup to end_num
  prms_range = Array.new(max_range,1) # pc array to generate range primes
end
prms=Array.new(maxpcs,1)      # for primes upto sqrtN and end_num if ks=0
```

This is perfectly functional code *except* when the array sizes exceed the available memory needed to create them. So in 2.2.0 – 2.3.0 I added error handling for those errors, which looked like this:

```
ks = (start_num-2).abs/mod    # start_num's resgroup value
maxpcs = maxprms              # if ks = 0 use this for prms array
begin                         # start arrays mem error handling block
  if ks > 0                   # if start_num > mod+1
    maxpcs = pcs2sqrtN        # find primes in pcs upto sqrtN
    pcs2ks = rescnt*ks        # number of pcs upto ks resgroup
    max_range =maxprms-pcs2ks # number of pcs from ks resgroup to end_num
    begin
      prms_range = Array.new(max_range,1) # pc array to hold range primes
    rescue                    # for not enough memory error for prms_range
      return puts "ERROR1: range size too big for available memory."
    end
  end
  prms=Array.new(maxpcs,1)    # for primes upto sqrtN and end_num if ks=0
    rescue                    # for not enough memory error for prms
      return puts "ERROR2: end_num too big for available memory."
end
```

This too is perfectly functional code, which catches out-of-memory errors during creation of each array, outputs a descriptive error message for each case, and then gracefully exits the method.

And to tell the truth, if I hadn't written this book it may still be that way. However, in the course of writing this book I revisited the code and realized it was really convoluted. Just looking at it I saw I could simplify and make it clearer by putting the error handling code for `prms` directly around it like I did for `prms_range`. After doing that the code looked like this:

```
split_arrays = start_num > sqrtN # flag, true if two arrays used for sieve
maxpcs = maxprms              # init array size for all pcs to end_num
if split_arrays               # if start_num > sqrtN create two arrays
  maxpcs = pcs2sqrtN          # number of pcs|array size, for pcs <= sqrtN
  max_range = maxprms-pcs2start # number of pcs in range start_num to end_num
  prms_range = array_check(max_range,1) # array to represent pcs in range
  begin
    prms_range = Array.new(max_range,1) # pc array to hold range primes
  rescue                      # for not enough memory error for prms_range
    return puts "ERROR1: range size too big for available memory."
  end
end
begin
  prms = Array.new(maxpcs,1)  # for primes upto sqrtN and end_num if ks=0
rescue                        # for not enough memory error for prms
  return puts "ERROR2: end_num too big for available memory."
end
```

But less than a minute after writing this I said to myself, “man that's ugly” and rewrote it as this,

```
split_arrays = start_num > sqrtN # flag, true if two arrays used for sieve
maxpcs = maxprms              # init array size for all pcs to end_num
if split_arrays               # if start_num > sqrtN create two arrays
  maxpcs = pcs2sqrtN          # number of pcs|array size, for pcs <= sqrtN
  max_range = maxprms-pcs2start # number of pcs in range start_num to end_num
  prms_range = array_check(max_range,1) # array to represent pcs in range
  return puts "ERROR1: range size too big for available memory." unless prms_range
end
prms = array_check(maxpcs,1)  # array for pcs upto sqrtN, or end_num
return puts "ERROR2: end_num too big for available memory." unless prms
```

creating the helper method array_check to do the common error handling, now in the code for 2.4.0+.

```
def array_check(n,v)  # catch out-of-memory errors on array creation
  begin Array.new(n,v) rescue return end  # return an array or nil
end
```

This turned functional ugly code into functional *beautiful* code! All the error handling distractions have been sucked out, leaving a short and concise snippet of self-documenting code which is similar in look and size to the original snippet, with the addition of the context specific error message code.

This illustrates that writing functional code is mechanical, writing beautiful code is an art. Ruby, maybe more than any other language, allows you to write beautiful code, if you use its powers well.

But Ruby also allows you to write code in a very personalized style. I like writing code so that each line can contain a complete statement of functionality. So for array_check, in probably any book on Ruby you would see it written something like:

```
def array_check(n,v)  # catch out-of-memory errors on array creation
  begin               # begin error handling block
    Array.new(n,v)    # attempt to create an array
  rescue              # catch error on array creation
    return            # return implicit 'nil' as output for error
  end                 # end error handling block
end
```

I didn't see the need to take 5 loc to say what I could do in 1 for a simple private helper method.

I stated earlier Ruby allowed me to experiment extensively because I could do it quickly. You will see this clearly for the explanation of sozcore2. Here I'll use a shorter example to make the point.

While this code conceptually expresses succinctly the factoring process in the factors method,

```
while (p = modk+residues[r]) <= sqrtN
  while n%p == 0; factors << p; n /= p; sqrtN = Math.sqrt(n).to_i end
  r +=1; if r > rescnt; r=1; modk +=mod end
end
```

this is the code I ultimately used.

```
while (p = modk+residues[r]) <= 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
end
```

Why? I tried six variations on these themes and this was consistently a little bit faster. So here I used the faster code, even though the first was shorter and conceptually clearer, because speed ruled there.

Benchmarking was very important in determining code selection. There was micro benchmarking at the code snippet|instruction level and then macro benchmarking at the method level. I used the gem benchmark-ips to perform micro benchmarks for code snippet|instructions comparisons, and created my own timer word – **`def tm; s=Time.now; yield; Time.now-s end`** – used as **`tm{n.factors}`** to get macro timing results in seconds.

The code has greatly evolved since its April 1, 2015 1.0.0 initial release of 5 methods. There have been added methods, found and fixed bugs, algorithmic improvements, various coding optimizations, and significant refactoring. To the last point, it has now reached the stage where I think I have DRYed it out to make the code as compact as reasonably worthwhile. I've been able to do all this in less than 500 lines of verbosely commented code. Personally, I think it's some of the most functional and *beautiful* code I have ever done for a project of this size and complexity.

The source code shown here is for version 2.5.0, which marks a mature milestone in the history of this project. I don't see it changing much more unless Ruby changes significantly to allow me to do things faster|better, or I find faster algorithmic approaches, or I add more functionality.

I'll first explain the **private methods** in basically the order they appear in the source code. They do the number crunching and perform the common functions. They are the results of extensive refactoring to DRY out the code. They become the lego blocks that are reused to form the **public methods** which are the methods provided for you to use.

Creating the math and code was a very visual experience for me, so it'll probably be helpful for you to refer back to **Mathematical Foundations** and its tables to *see* what the code is doing as you go along. The firmer your understanding of the math is the easier the code will be to follow and understand.

So for all the brave and curious, let's dive into the code, and see how|what makes primes-utils works.

Private Methods

make_residues_rescnt(mod)

```
def make_residues_rescnt(mod)
  residues=[1]; 3.step(mod,2) {|r| residues << r if mod.gcd(r) == 1}
  residues << mod+1
  [residues, residues.size-1]   # return residues array and rescnt
end
```

This is the simplest method. It takes a PGs `mod` value and returns an array of its `residues` and `rescnt`. The `residues` array has form [1, r1, r2....mod-1, mod+1] and size of `rescnt+1`.

pcs_to_num(num, mod, rescnt, residues, lte)

```
# lte= true: first output element is number of pcs <= num
# lte=false: num pcs <, residue index, and resgroup value, for (start_)num pc
def pcs_to_num(num,mod,rescnt,residues,lte)
  num -=1; lte ? (num |=1; k=num.abs/mod) : k = (num-1).abs/mod
  modk = mod*k; r=1
  r +=1 while num >= modk+residues[r]
  [rescnt*k + r-1, r, modk]     # [num pcs, r index, num modulus]
end
```

This little method's function is simple, but the road to this current form is anguished. First its function.

It takes as inputs the `[start|end]_num` of a range, a PGs `mod`, `rescnt`, `residues` variables, and a `lte` flag and returns a 3 element array of: the number of pcs < | <= to `num`, `num's` residue index, and it base value. For `num=end_num`, `lte` is set to `true` and the first output element is the num of pcs to `end_num` (inclusive). For `num=start_num`, `lte` is set `false` and the first element is the number of pcs < `start_num` (exclusive).

I say it has an anguished history because I finally decided to create it by combining 2 methods into 1. In my original coding the two functions were done with separate methods, which made them super simple to use conceptually. However, the code to do them was so close, my urge to DRY it while writing this HANDBOOK kept pushing me to see if I could find a ***simple*** way to combine them. What you see is the final result of that long effort. I like it because it reduces the methods and code. I lament it because I had to introduce a small level of complexity by including a flag to select the functionality. But since it's only used in the private methods sozcore[1|2] its contextual use is clearly understood (and documented). Life's a tradeoff. Here are the original two methods for comparison.

```
def pcs_to_num(num,mod,rescnt,residues) # find number of pcs <= num
  num = (num-1 | 1).abs         # if N even number then subtract 1
  k=num/mod; modk = mod*k; r=1  # init parameters for kth resgroup
  r +=1 while num >= modk+residues[r] # find r index for 1st pc > num
  rescnt*k + r-1                # number of pcs to num
end

def pcs_to_start_num(num,mod,rescnt,residues) # find num pcs < start_num
  k = (num-2).abs/mod           # num's residue group value
  modk = mod*k; r = 1           # init parameters for kth resgroup
  r +=1 while modk + residues[r] < num # find r index for pc of num
  [rescnt*k + r-1, r, modk]     # parameters for 1st pc >= start_num
end
```

When finding the number of pcs <= `num` only the first output element is used and the other 2 ignored, which is seen in this example in sozcore1: `maxpcs,* = pcs_to_num(num,mod,rescnt,residues,true)`

sozcore1(num, start_num, method_flag)

```
# Use default SP Prime Generator to parametize the pcs within a range
# inputs:  end_num|start_num of range; method_flag to numerate|count primes
# outputs: maxpcs-m - number of pcs in the range
#          r        - residue index value for start_num pc of range
#          mod      - mod value for PG
#          modk     - base value for start_num's resgroup
#          rescnt   - number of residues for PG
#          residues - array of residues plus mod+1 for PG
#          primes array|primes.size - primes array or size based on method_flag
def sozcore1(num, start_num, method_flag)
  num = num.abs;   start_num = start_num.abs
  num, start_num = start_num, num  if start_num > num

  primes = [2,3,5,7,11,13]      # excluded primes for P13 default SP PG
  mod = primes.reduce(:*)       # P13 modulus: 2*3*5*7*11*13 = 30030
  residues, rescnt = make_residues_rescnt(mod)
  maxpcs,* = pcs_to_num(num,mod,rescnt,residues,true) # num pcs <= end_num

  # init 'primes' w/any excluded primes in the range, or [] if none
  primes.select! {|p| p >= start_num && p <= num}

  # compute parameters for start_num pc, then create output parameters array
  m, r, modk = pcs_to_num(start_num, mod, rescnt, residues, false)
  [maxpcs-m, r, mod, modk, rescnt, residues, method_flag ? primes : primes.size]
end
```

The job of sozcore1 is very simple; given values for a number range, and a `method_flag`, it returns an array of parameters which can be used to create an array of primes within the range, or their count. Its the result of refactoring and DRYing out the primes[f|mr] | primescnt[f|mr] methods, which now use it. They take its output to numerate the pcs in the range and test their primality (with prime? or primemr?) and either provide an array of the range primes, or their count. Let's see how it works.

After initializing the range variables, `primes` is set for the excluded primes of the default SP PG, then its `mod` and `residues|rescnt` are created. Using them, the number of all the pcs <= `end_num's` pc value is determined and assigned to `maxpcs`. If `end_num` is inside the excluded primes this value will be zero.

If the range is partly or completely within the excluded primes we have to account for those primes. If just `start_num` is in the excluded primes, `primes` holds that subset (e.g. [7,11,13]) and `maxpcs` > 0. If `end_num` is in the excluded primes `primes` holds that subset and `maxpcs` will be 0. If `start_num` is not in the excluded primes `primes` is set to an empty array, and will just contain the primes found in the range.

Then for `start_num's` pc address we find m, the number of pcs less than it, its residue index r, and resgroup value `modk`, which are passed to the output. The first output array element is the number of pcs in the range, `maxpcs-m` (which is > 0 for `end_num` not in the excluded primes), and the last is either `primes`, or its array size. When `method_flag` is `true` we want the prime values, otherwise their count.

I put the selection of the default PG, and parameters, here, which make the user methods super simple clients of its output. They only differ by the pcs primality testing method, which can be easily changed. I use P13 because it performed best on my system, and ideally you want to use the largest SP PG that works best on yours.

Thus sozcore1, with its user methods, pseudo-directly list|count primes over a range without sieving.

sozcore2(num, start_num, mod)

```
    # Perform SoZ with given Prime Generator and return array of parameters
    # inputs:  end_num and start_num of range and mod value for PG
    # outputs: prms - binary (0,1) array of pcs within a range or to end_num
    #          m    - num of pcs in prms < start_num; so prms[m] = start_num
    #          modks    - mod value for start_num's resgroup
    #          residues - array of residues plus mod+1 for PG
    #          rescnt   - number of residues for PG
    #          pcs2start- number of pcs < start_num pc
    #          maxprms  - number of pcs to find primes from; prms array size
    #          rs   - residue index location for first pc >= start_num
    def sozcore2(num, start_num, mod)
      residues, rescnt = make_residues_rescnt(mod)  # parameters for the PG
      maxprms,* = pcs_to_num(num,mod,rescnt,residues,true) # num pcs <= end_num

      # for start_num pc, find num pcs <, residue index, and resgroup mod value
      pcs2start, rs, modks = pcs_to_num(start_num, mod, rescnt, residues, false)

      sqrtN = Math.sqrt(num).to_i   # sqrt of end_num (end of range)
      pcs2sqrtN,* = pcs_to_num(sqrtN,mod,rescnt,residues,true) # num pcs <= sqrtN

      split_arrays = start_num > sqrtN # flag, true if two arrays used for sieve
      maxpcs = maxprms              # init array size for all pcs to end_num
      if split_arrays               # if start_num > sqrtN create two arrays
        maxpcs = pcs2sqrtN          # number of pcs|array size, for pcs <= sqrtN
        max_range = maxprms-pcs2start # number of pcs in range start_num to end_num
        prms_range = array_check(max_range,1) # array to represent pcs in range
        return puts "ERROR1: range size too big for available memory." unless prms_range
      end
      prms = array_check(maxpcs,1)  # array for pcs upto sqrtN, or end_num
      return puts "ERROR2: end_num too big for available memory." unless prms

      # residues offsets to compute a pcs address in its resgroup in prms
      pos =[]; rescnt.times {|i| pos[residues[i]] = i-1}

      # Sieve of Zakiya (SoZ) to eliminate nonprimes from prms and prms_range
      modk,r,k=0,0,0
      pcs2sqrtN.times do |i|        # sieve primes from pcs upto sqrt(end_num)
        r +=1; if r > rescnt; r=1; modk +=mod; k +=1 end
        next unless prms[i] == 1    # when a prime location found
        prm_r = residues[r]         # its residue value is saved
        prime = modk + prm_r        # its value is numerated
        prmstep = prime * rescnt    # its primestep computed
        kcon = k * prmstep          # its inner loop constant computed
        residues[1..-1].each do |ri|# now perform sieve with it
          # convert (prime * (modk + ri)) pc value to its address in prms
          # computed as nonprm = (k*(prime + ri) + kn)*rescnt + pos[rr]
          kn,rr  = (prm_r * ri).divmod mod            # residues product res[group|track]
          nonprm = kcon + (k*ri + kn)*rescnt + pos[rr]# 1st prime multiple address with ri
          while nonprm < maxpcs; prms[nonprm]=0; nonprm +=prmstep end
          if split_arrays                             # when start_num > sqrtN
            nonprm = (pcs2start - nonprm)%prmstep     # (start_num - closest multiple) pcs
            nonprm = prmstep - nonprm if nonprm != 0  # location in range, or beyond
            while nonprm < max_range; prms_range[nonprm]=0; nonprm += prmstep end
          end
        end
      end

      # determine prms array parameters and starting location value m for start_num
      split_arrays ? (prms = prms_range; maxprms = max_range; m = 0) : m = pcs2start
      [prms, m, modks, residues, rescnt, pcs2start, maxprms, rs] # parameters output
    end
```

What I'm going to do here is use this version of sozcore2, used first in 2.4.0, as the basis to explain the math and ruby implementation of the SoZ algorithm. It will be extensive. Then I'll show another version I almost used in 2.5.0, which is actually (nominally) faster, and explain why I abandoned it.

The purpose of sozcore2 is to perform the Sieve of Zakiya for an input number range using a given PG, and return an array of parameters from which the primes can be numerated|counted by user methods primes, primescnt and primenth|nthprime. It's been improved from 2.3.0, because I caught an error in the algorithm, corrected it, and further optimized it. It is by far algorithmically and computationally the biggest and most complex method, but all it's doing is the SoZ over a range, as previously explained.

I think because of its complexity the best way to explain it is to use a real example, so I'll use the example in Table 5, finding the primes over the pcs range [337 – 541] using P5. You can look at the code and see what it's doing, but what's more important is *why* it's doing it. If you really *get it* at that level you can translate it to any other language, or even improve it. Let me know if you do. Let's start.

Unlike in sozcore1, the input range must be entered in order, where `num=end_num`, and for P5, `mod=30`. So for our example, we would invoke the method like `sozcore2(541,337,30)` or `sozcore2 541,337,30`. It starts by creating the PGs `residues|rescnt` parameters from `mod`, which for P5 produces the results `residues = [1,7,11,13,17,19,23,39,31]` and `rescnt = 8`.

Then pcs_to_num computes the number of pcs up to `end_num=541`, resulting in `maxprms=8*18=144`. I then determine where `start_num=337` is in the table, and how many pcs are below it using pcs_to_num, producing `pcs2start=8*11=88`, its `residues` index `rs=1`, and base resgroup val `modks=30*11=330` (`337 = 330+residues[1]`). I then calculate `sqrtN=23`, and the number of pcs to its table location, `pcs2sqrtN=6`.

I now determine if I sieve over a single array up to `end_num` or over split (2) arrays|ranges. I create the boolean flag `split_array = start_num > sqrtN`. In our example this is `true`, so we will create 2 arrays to represent the pcs in the split ranges in Table 5. I then set `maxpcs=maxprrms=144`. This would be the array size for a single array when `split_array` is `false`. We'll now create the arrays for each range.

Arrays are created from system memory. Originally, I didn't account for cases where the amount of memory needed to create an array was less than what was available, so the method would crash ingloriously. In my quest to become a better programmer, and provide a better user experience, in 2.2.0 I employed ruby's error handling features to catch these out-of-memory errors, to gracefully fail with clear messages in these cases. Starting with 2.4.0, I created the helper method array_check to create an array and catch out-of-memory errors, and pass the array back, or `nil` if it wasn't created. If their was an error, a distinct error message is displayed for each case telling you what was the cause of the error.

Proceeding, I first ask if I need to create 2 arrays. Since it's yes, I set `maxpcs = pcs2sprtN`, the number of pcs to the sqrtN, and size for the sieve primes array. I then set `max_range = maxprms-pcs2start = 144-88 = 56`, which is the number of pcs in the range, and size of the range primes array. I then attempt to create the `prms_range` array of size `max_range` with array_check, where each array element is set to '1' (default as prime). If insufficient memory it returns `nil`, and the error message for it will be displayed – `ERROR1: range size too big for available memory.` – and `nil` returned, else the array was created.

I now create the `prms` array. If we don't split arrays, it represents a single array of all the table pcs. Since we're splitting arrays, it attempts to create an array representing just the sieve primes pcs and displays the message – `ERROR2: end_num too big for available memory.` – if not enough memory.

Assuming we've created our split arrays, we're now ready to perform the SoZ algorithm using them.

So now we have the `prms` array of size 6, and the `prms_range` array of size 56, of all '1's (primes). (I originally used `true|false` as array elements but using '1' for primes allows faster arithmetic counting.) The 6 `prms` elements represent the pcs ≤ sqrt(`end_num=541`) in Table 5, and for `prms_range` the 56 pcs in the range. We find all the primes in `prms` (here they're all prime, but we still perform the sieve on it) and use them to sieve out all their multiples (locations) in `prms_range`. We'll end up with `prms_range` an array of 1|0 (nomprimes) values. The first location, `prms_range[0]`, is the `start_num` pc value whose residue index and resgroup values, along with `prms_range`, are returned to numerate|count the primes.

Taking up where we left off, we do: **`pos = []; rescnt.times {|i| pos[residues[i]] = i-1}`** Originally `pos` was a hash, here it's an (dynamic) array, which is a little faster. It computes the index offsets for each residue value, `pos[1]=-1..pos[29]=6`. I could do `residues.index(rr)-1` directly in the code, but it's way slower. Adding `pos[rr]` to a pc's resgroup value computes its address within `prm`. Note, for pcs with residue 1, `pos[1]=-1` makes sure to compute its `prm` address to be on the last restrack of a resgroup, not the first in the next one. We now have all the pieces to implement the SoZ code loop.

The sieve is performed inside a loop of `pcs2sqrtN` iterations, which first finds all the primes in `prms` and uses them to sieve the nonprimes in `prms` and then `prms_range`. The line **`pcs2sqrtN.times do |i|`** starts the process, and `|i|` (iterating as 0,1,2,3,4,5) is the index value for each element in `prms`. As we loop we increment|reset the resindex r to range from 1..8, and increment `modk` and k after doing `rescnt=8` loops. They're used to numerate the pc values for each array position as we go from one resgroup to the next.

Starting inside the loop with r=1, `modk`=0, k=0, and i=0 we check if the first element in `prms` is prime, (which it always will be as previously explained) and continue (if `prms[i] == 1` is `false`, it's nonprime, and we skip the loop body and start another iteration). We set `prm_r=7`, `prime=7`, `prmstep=56`, `kcon=0`. We're now ready to do the loop that performs the SoZ sieve for both `prms` and `prms_range`, but first I need to explain a *little bit of math* so you can|will understand why it's doing what it's dong.

Per the SoZ algorithm (feel free to review) each prime multiplies each of its resgroup pcs, and uses each first multiple pc value as the starting point along a restrack to eliminate its additional multiples. So we first need to find the location in `prms` for these first multiples. There are at least two ways to mathematically do this, but the way I do it now seems to be the more efficient to code, as I'll explain.

Each resgroup pc has the same modk value. Let r be a prime's residue value (`prm_r` in code) and ri for any pc. We identify the first prime multiples – **prime * pc** = **prime * (modk + ri)** – as follows:

```
          prime * (modk + ri)
      (modk + r) * (modk + ri)
      modk * modk + modk * (r + ri) + (r * ri)
      modk * (modk + r + ri) + (r * ri)
(mod * k) * (prime + ri) + (r * ri)
 mod * [k * (prime + ri)] + (r * ri)
       |_______________|   |_____|
               kk            rr
```

Thus, the first multiple values have form – **mod*kk + rr** – where **kk = k*(prime + ri)** and **rr = r*ri.** This expression creates values in the number domain (product values), which we need to translate to the residues domain (pcstable locations). Mathematically we do k = pc/mod, and r = pc% mod, to get a pc's resgroup|residue values (pc = mod*k + r) then do **k*rescnt + pos[r]** for its pcstable location. We'll now do an equivalent transform without having to multiply **prime*pc**, to avoid creating large products.

For the base residues (k=0) cross-products **rr**, their resgroups are **kn = rr/mod** and residue track offsets **pos[rr % mod]**. Thus, a product's resgroup when k > 0 is then **ki = kk + kn = k*(prime+ri) + kn** and its location – **nonprm** = ki*rescnt + pos[rr % mod] = **(k*(prime+ri) + kn)*rescnt + pos[rr % mod]**.

I can manipulate this further: nonprm = (k*prime + k*ri + kn)*rescnt + pos[rr % mod]
nonprm = k*prime*rescnt + (k*ri + kn)*rescnt + pos[rr % mod]
nonprm = k*primestep + (k*ri + kn)*rescnt + pos[rr % mod]
nonprm = kcon + (k*ri + kn)*rescnt + pos[rr % mod]

where kcon = k*prime*rescnt = k*primestep is a constant that can be calculated outside the sieve loop. Notice, now prime's value is not used in the inner loop, only its residue and the value of kcon.

Thus, the following code (first used in 2.4.0) performs the SoZ sieve over any range:

```
residues[1..-1].each do |ri|                  # perform sieve for prime
  # convert (prime * (modk + ri)) pc value to its index|address in prms
  # computed as nonprm = (k*(prime + ri) + kn)*rescnt + pos[rr]
  kn,rr  = (prm_r * ri).divmod mod            # residues product res[group|track]
  nonprm = kcon + (k*ri + kn)*rescnt + pos[rr]# 1st prime multiple address with ri
  while nonprm < maxpcs; prms[nonprm]=0; nonprm +=prmstep end
  if split_arrays                             # when start_num > sqrtN
    nonprm = (pcs2start - nonprm)%prmstep     # (start_num - closest multiple) pcs
    nonprm = prmstep - nonprm if nonprm != 0  # location in range, or beyond
    while nonprm < max_range; prms_range[nonprm]=0; nonprm += prmstep end
  end
end
```

For the first `prime` with residue `prms_r`, we iterate thru residues (7, 11...29, 31), with `ri=7` first, and compute `kn` and `rr`, then the first multiple location `nonprm`. We start sieving the `prms` array in the first `while` loop and stop when `nonprm's` value exceeds `maxpcs` (number of pcs in `prms`). Because we have split arrays, we then project `nonprm's` last value up to the range, as we've seen how to do earlier. With the new `nonprm` value we sieve the `prms_range` array (second `while` loop) until it exceeds `max_range`. We then perform the same process with the remaining `ri`. When we exhaust them we iterate the main loop, find the next prime, and sieve with it until we use all the primes in `prms`. As shown, the sieve algorithm loop takes just 10 loc (excluding comments) to perform for each prime, over both ranges.

Finally we perform the last two lines:

```
split_arrays ? (prms = prms_range; maxprms = max_range; m = 0) : m = pcs2start
[prms, m, modks, residues, rescnt, pcs2start, maxprms, rs] # parameters output
```

If we have split arrays we want to return `prm_range's` data, and reassign `prms` and `maxprms` to reflect that, and set `m=0`, `prms_range's` first pc address index. If just one array, `prms` and `maxprms` are correct and we just need to set `m=pcs2start`, for the address of the first pc in the range. We then create the output parameters array, which is returned, as it's the last statement in the method.

There is some beautiful math going on within the sieve loop which I'll just touch on a bit here.

The expression – **mod * [k * (prime + ri)] + (r * ri)** – is the product of *any* two resgroup pcs, not just primes.

From Table 4, if we multiply 67*91=6097, we can do it either way and still get the same answer (try your own).

67*91 = 30*[2*(67 + 31)] + (7*31) = 60*(98) + 217 = 5880 + 217 = 6097
91*67 = 30*[2*(91 + 7)] + (7*31) = 60*(98) + 217 = 5880 + 217 = 6097

Its pcstable parameters are: k = 6097 / 30 = 203, r = 6097 % 30 = 7, so it has a residue of 7 in resgroup 203, and its pcstable address is: k*rescnt + pos[7] = 203*8 + 0 = 1624. This is how I originally performed the inner loop. However, the products become relatively much larger than their array addresses as they increase, and depending on the language and hardware, could exceed the capacity of a system to represent them. Doing the math of the inner loop code from above we get the same address for this prime multiple using much smaller numbers.

kn, rr = (7 * 31).divmod 30 => kn = 7 and rr = 7, with kcon = k*rescnt*prime = 2*8*67 = 1072
nonprm = kcon + (k*ri + kn)*rescnt + pos[rr] = 1072 + (2*31 + 7)*8 + pos[7] = 1624+0 = 1624

Here all the values are much less than the product, making this method more universal to implement. Both – **(k*(prime + ri) + kn)*rescnt** – and it's variant – **kcon + (k*ri + kn)*rescnt** – are arithmetically similar, performing 2 multiplications and additions, but the later is faster in Ruby. With advanced compilers for languages like C/C++, etc it may make no difference which version you use with them.

Finally, here's part of what this expression is saying about the characteristics of prime multiples.

If we do this: (k*(prime + ri) + kn)*rescnt + pos[rr]
k*prime*rescnt + k*ri*rescnt + kn*rescnt + pos[rr]
(k*prime + k*ri + kn)*rescnt + pos[rr]

we get an expression where – (k*prime + k*ri + kn) – gives the accumulative resgroup for the product. It's composed of three parts: a part due to the prime value, a part due to the pc's residue ri, and a part due to their cross-product resgroup. We see kn is not dependent on the resgroup value k, so it acts as (using electrical engineering vernacular) the DC, or steady-state, component, which always exists. The value k*(prime + ri) is the harmonic part, the number of resgroups to jump starting from resgroup kn. If we stick in numbers we get: (2*67 + 2*31 +7) = (134 + 62 + 7) = 203, which equals k = 6097 / 30, and again, pos[rr] tells us the restrack offset for the product's residue.

Though mathematically simple (we're only doing plain arithmetic here) it's theoretically instructive. Every prime multiple pc is the linear composition of three parts, which can be computed separately and added together to get its resgroup location, using only each pc's residue, and not its value.

Knowing this, I knew I should be able to precompute some of these values to optimize the inner loop calculations. I was always being frustrated in this effort because my early attempts to optimize this process in ruby, first using hashes, then arrays, produced slower code than doing the straight arithmetic. Then one day (in fact late in writing this HANDBOOK) it was like the sun rose and cleared out the fog and I could see clearly how to do it. Then after doing all of this, and rewriting the code, I ultimately decided not to use it. The reason why pitted the costs of theoretical possibility against practical reality.

If we start with this expression – **nonprm = kcon + (k*ri + kn)*rescnt + pos[rr]**
I realized I could precompute for each prime residue r the values ri, kn, and pos[rr] for each resgroup pc it multiplies and store them in a [rescnt x rescnt] array. I finally figured out how to do it correctly, to create a faster inner loop. Then I realized I could do even better doing the following manipulation:

nonprm = kcon + (k*ri + kn)*rescnt + pos[rr]
nonprm = kcon + k*(ri*rescnt) + (kn*rescnt + pos[rr])
nonprm = kcon + k*rx + ry

where **rx = ri*rescnt** and **ry = kn*rescnt + pos[rr]** can be precomputed for each r|ri residues pair. To precompute these values I created the following private methods.

```
def soz_loop_params(mod)        # create array of a PG's SoZ inner loop params
  residues=[1]; 3.step(mod,2) {|r| residues << r if mod.gcd(r) == 1}
  residues << mod+1;  rescnt = residues.size-1
  resp = Array.new(rescnt) {Array.new(rescnt)}
  rescnt.times do |i|
    rescnt.times do |j|
      r = residues[i+1]; ri = residues[j+1]
      kn,rr = (r * ri).divmod mod
      rx = ri*rescnt;  ry = kn*rescnt + residues.index(rr)-1
      resp[i][j] = [rx,ry]
    end
  end
  resp
end

def resp5;  soz_loop_params(30)   end
def resp7;  soz_loop_params(210)  end
def resp11; soz_loop_params(2310) end
```

This, in fact, did provide an additional small speed increase, while creating a simpler array structure than my first attempt. The relevant code changes from the 2.4.0 version of sozcore2 are shown below:

```
  # select array for PG to do optimized SoZ inner loop calculations for P[5|7|11]
  sozresp = if mod==210; resp7 else mod > 210 ? resp11 : resp5 end

  # Sieve of Zakiya (SoZ) to eliminate nonprimes from prms and prms_range
  modk,r,k=0,0,0
  pcs2sqrtN.times do |i|        # sieve primes from pcs upto sqrt(end_num)
    r +=1; if r > rescnt; r=1; modk +=mod; k +=1 end
    next unless prms[i] == 1    # when a prime location found
    prm_r = residues[r]         # its residue value is saved
    prime = modk + prm_r        # its value is numerated
    prmstep = prime * rescnt    # its primestep computed
    kcon = k * prmstep          # its inner loop constant computed
    sozresp[r-1].each do |rx,ry|# now perform sieve with it
      # convert (prime * (modk + ri)) pc value to its address in prms
      # computed as nonprm = kcon + k*(ri*rescnt) + (kn*rescnt + pos[rr])
      nonprm = kcon + k*rx + ry                  # 1st prime multiple address with ri
      while nonprm < maxpcs; prms[nonprm]=0; nonprm +=prmstep end
      if split_arrays                            # when start_num > sqrtN
        nonprm = (pcs2start - nonprm)%prmstep    # (start_num - last multiple) pcs
        nonprm = prmstep - nonprm if nonprm != 0  # location in range, or beyond
        while nonprm < max_range; prms_range[nonprm]=0; nonprm += prmstep end
      end
    end
  end
```

Where before I computed the offset residues array `pos` I now set the variable `sozresp` (SoZ residue parameters) to contain the array for the SoZ inner loop parameters |rx,ry| for the input PG. They are computed by the private method soz_loop_params, and the arrays generated and stored in the resp#

methods. Ruby allows iterating over 2D array rows by merely doing – `sozresp[r-1].each` – instead of having to do – `sozresp[r-1][0.-1].each` – to explicitly iterate over the columns.

Adding this additional (relatively small amount of) code worked fine, and did produce measurable (but small) speed increases. And I was feeling real good about myself, and pleased I finally figured out how to reduce that inner code section to just one loc of two additions and one multiplication. And I was on the verge of using it in 2.5.0. Then I started benchmarking versions. I mean extensive rigorous testing.

One thing about me, I don't become too emotionally involved in my code, past a certain point. So when I saw only small speed increases my spider senses started tingling, and I went into test mode. I thought I'd see a bigger difference than the 2.4.0 code, and I wanted to know why I didn't. So I started testing the 2.4.0 code against the newer code.

One thing I first realized was the 2.4.0 code was using the older adaptive technique in select_pg, so I then tested the 2.4.0 code using the new method. When I did that, all the significant differences that existed between the versions, which mostly occurred for large ranges, evaporated. That's because the new adaptive technique picked a bigger PG for large ranges. This actually confirmed what I suspected; the nominal micro implementation change gains were swamped by the macro algorithmic change gains.

Added to this finding were other niggling costs of changing the 2.4.0 code. More code and complexity, extra memory needed for each PG loop parameter array, and the loss of flexibility to easily use any PG, ultimately didn't warrant using a design that gave no real speed increases. However, this exercise was still useful. This approach may be useful to get optimum performance from specialty or customized applications (software or hardware).

This finally ends the journey through the math and coding to implement sozcore2. It has corrected a subtle SoZ algorithmic implementation error in prior versions to 2.4.0, and I believe represents (currently) the optimum algorithm with the optimum ruby implementation. I know it's a mouthful to digest, but for the hardy diner it's a meal worth digesting. Just take your time, and eat slowly, and you may be able to savor all the nutritional value (math and ruby) it has to offer.

approximate_nth(n)

```
def approximate_nth(n)          # approximate nthprime value >= real value
  b = 0.5722*n**0.0088          # derived equation to compute close nth values
  a = b*(Math.log(Math.log(n)))
  (n*(Math.log(n)+a)+3).to_i    # use nth approximation as end_num of range
end
```

This is used inside primenth to compute a *close* approximation to an nth prime that's just a *little bit* greater than it. This will be used as the `end_num` for a range containing the real nth prime value. For example, for n = 100 the real 100th prime is 541, and this method returns 554 as the `end_num` of a range which contains it. This ensures the range is always large enough to contain the real nth prime. This current version first appeared in 2.2.0 and was the culmination of a lot of blood, sweat, and tears. I first created a Calc [14] spreadsheet of handpicked values for **b** (like 0.64) that made **a** give good approximations for each nth prime in `nths`. I then used Geogebra [12] to curve fit each **[key,b]** pair to get the equation **b(n)**, then tweaked the coefficient (0.57..) to get a better fit over `nths` range. Simple!

set_start_value(n, hshflag)

```
def set_start_value(n, hshflag) # find largest index nthprime|val <= n
  if hshflag
    return [nths[n], 0, true] if nths.has_key? n       # if n is key in nths table
    nth = nths.keys.sort.reverse.detect {|k| k < n}    # find largest indexed key < n
    [nth ? nths[nth] : 0, nth ||= n+1, false]          # [start_num, nth, false]
  else
    return [0,nths.key(n),false] if nths.has_value? n  # if n is value in nths table
    v=val= nths.values.sort.reverse.detect {|v| v < n} # find largest indexed val < n
    [v ||= 0, val ? nths.key(val) : 0, true]           # [start_num, nth, true]
  end
end
```

This also has an interesting history. Before 2.3.0 it essentially did the code inside the **if** statement and was used in primenth to set `start_num` for a range containing an nth prime. The input was just **n**, the nth prime designator (e.g. n = 100 for 100th prime). It first checks if n is a key in the `nths` hash, and if so, it returns a 3 element array of the prime value for n, 0, and `true`. Thus since 1,000,000 is a key in `nths`, if n is 1000000 the array [15485863, 0, `true`] is returned. If n isn't in `nths`, then I want to find the largest (closest) key < n. This is done with the second line. So if I want the nine-millionth prime then n=9000000, and the largest key less than it is 7,500,000, which is assigned to `nth`. Since `nth` evaluates to `true`, the output would be [132276691, 7500000, `false`]. Now if n is < the smallest key, like n=1000 for the one-thousandth prime, there is no key < n, then `nth = nil`, and the output is [0, 1001, false].

While looking how to upgrade primescnt from 2.2.0 inspiration struck. I realized I could count primes up to some N more efficiently by starting from a known nth prime less than it. Since I had a hash of nth primes, why not start counting from them to reduce the range size to process? So that's what I did.

If `hshflag` is `false`, n is an integer value we want to find the primes up to, and we do the `else` branch. It first checks if n is an nth prime value in `nths`, and if so returns 0, the key for the value, and `false`. So if n=15485863, [0, 1000000, `false`] is returned. If n isn't in `nths`, I want to find the largest value < n, and return it, its key (nth count), and `true`. So if n=100000000, v=val=86028121, which evaluate to `true`, thus the ouptut is [86028121, 5000000, `true`]. If n is < the smallest value, e.g. n=12345678, there is no `nths` value < n, then `v=val=nil`, and the output is [0, 0, `true`]. Thus, the method normally returns a prime or 0 for `start_num` of the reduced range, the number of primes up to it, and a flag, as its output.

nths

```
    def nths                          # hash table index of reference nth primes
      nths={1000000 => 15485863,   5000000 =>   86028121,   7500000 =>  132276691,
         10000000 =>  179424673,  12500000 =>  227254201,  15000000 =>  275604541,
         18500000 =>  344032387,  25000000 =>  472882027,  31000000 =>  593441843,
         37500000 =>  725420401,  43500000 =>  848321917,  50000000 =>  982451653,
         56000000 => 1107029837,  62500000 => 1242809749,  68500000 => 1368724829,
         75000000 => 1505776939,  81500000 => 1643429659,  87500000 => 1770989609,
         93500000 => 1898979367, 100000000 => 2038074743, 106500000 => 2177624377,
        112500000 => 2306797469, 125000000 => 2576983867, 137500000 => 2848518523,
        150000000 => 3121238909, 162500000 => 3395057291, 175000000 => 3669829403,
        187500000 => 3945592087, 200000000 => 4222234741, 212500000 => 4499683009,
        225000000 => 4777890881, 237500000 => 5056862311, 250000000 => 5336500537,
        262500000 => 5616787769, 275000000 => 5897707297, 287500000 => 6179208157,
        300000000 => 6461335109, 312500000 => 6743943629, 325000000 => 7027107881,
        337500000 => 7310793337, 350000000 => 7594955549, 362500000 => 7879581839,
        375000000 => 8164628191, 387500000 => 8450100349, 400000000 => 8736028057,
        412500000 => 9022375487, 425000000 => 9309109471, 437500000 => 9596238593,
        450000000 => 9883692017, 462500000 =>10171564687, 475000000 =>10459805417,
        487500000 =>10748372137, 500000000 =>11037271757, 512500000 =>11326513039,
        525000000 =>11616020609, 537500000 =>11905863799, 550000000 =>12196034771,
        562500000 =>12486465863, 575000000 =>12777222833, 587500000 =>13068237251,
        600000000 =>13359555403, 612500000 =>13651119389, 625000000 =>13942985677,
        637500000 =>14235122851, 650000000 =>14527476781, 662500000 =>14820071503,
        675000000 =>15112928683, 687500000 =>15406031899, 700000000 =>15699342107,
        712500000 =>15992957251, 725000000 =>16286768243, 737500000 =>16580801137,
        750000000 =>16875026921, 762500000 =>17169527171, 775000000 =>17464243799,
        787500000 =>17759139259, 800000000 =>18054236957, 812500000 =>18349591409,
        825000000 =>18645104897, 837500000 =>18940846207, 850000000 =>19236701629,
        862500000 =>19532780327, 875000000 =>19829092147, 887500000 =>20125592731,
        900000000 =>20422213579, 912500000 =>20719050323, 925000000 =>21016060633,
        937500000 =>21313231963, 950000000 =>21610588367, 962500000 =>21908128993,
        975000000 =>22205818561, 987500000 =>22503733657, 1000000000=>22801763489,
       1012500000=>23099993743, 1025000000=>23398391231, 1037500000=>23696858797,
       1050000000=>23995554823, 1062500000=>24294392179, 1075000000=>24593421187,
       1087500000=>24892587403, 1100000000=>25191867719, 1112500000=>25491361037,
       1125000000=>25790970053, 1137500000=>26090709563, 1150000000=>26390560513,
       1162500000=>26690560601, 1175000000=>26990744987, 1187500000=>27291009337,
       1200000000=>27591444869, 1212500000=>27892051267, 1225000000=>28192760279,
       1237500000=>28493648629, 1250000000=>28794583627, 1262500000=>29095694269,
       1275000000=>29396966971, 1287500000=>29698366099, 1300000000=>29999858327,
       1312500000=>30301430881, 1325000000=>30603183581, 1337500000=>30905024497,
       1350000000=>31207047449, 1362500000=>31509131153, 1375000000=>31811397571,
       1387500000=>32113702069, 1400000000=>32416190071, 1412500000=>32718790873,
       1425000000=>33021414143, 1437500000=>33324275711, 1450000000=>33627220709,
       1462500000=>33930284893, 1475000000=>34233442279, 1487500000=>34536683891,
       1500000000=>34840062373, 1512500000=>35143545889, 1525000000=>35447088559,
       1537500000=>35750747297, 1550000000=>36054501641, 1562500000=>36358440731,
       1575000000=>36662430631, 1587500000=>36966563321, 1600000000=>37270791697,
       1612500000=>37575137933, 1625000000=>37879532671, 1637500000=>38184009763,
       1650000000=>38488677419, 1662500000=>38793413899, 1675000000=>39098225629,
       1687500000=>39403174463, 1700000000=>39708229123, 1712500000=>40013309359,
       1725000000=>40318523009, 1737500000=>40623800311, 1750000000=>40929166261,
       1762500000=>41234743751, 1775000000=>41540289619, 1787500000=>41845958971,
       1800000000=>42151671491, 1812500000=>42457500313, 1825000000=>42763499629,
       1837500000=>43069571603, 1850000000=>43375710643, 1862500000=>43681898699,
       1875000000=>43988172667, 1887500000=>44294549347, 1900000000=>44601021791,
       1912500000=>44907564593, 1925000000=>45214177441, 1937500000=>45520935011,
       1950000000=>45827700419, 1962500000=>46134655219, 1975000000=>46441643177,
       1987500000=>46748693981, 2000000000=>47055833459, 2012500000=>47363059687
      }
    end
```

`nths` returns a hash whose keys are integers that designate an nth prime, and whose values are the nth primes. Ex: the one-millionth prime is 15485863 so nths[1000000] = 15485863. This is the current list for 2.5.0, which included 9 more index primes under the 110-millionth than 2.4.0.

select_pg(num, start_num)

```
def select_pg(num, start_num)   # adaptively select PG
  range = num - start_num
  pg = 5
  if start_num <= Math.sqrt(num).to_i     # for one array of primes upto N
    pg =  7 if num >  50*10**4
    pg = 11 if num > 305*10**5
  else                                    # for split array cases
    pg =  7 if (range.between?(10**6, 10**7-1) && start_num < 10**8)     ||
               (range.between?(10**7, 10**8-1) && start_num < 46*10**8)  ||
               (range.between?(10**8, 10**9-1) && start_num < 16*10**10) ||
               (range >= 10**9 && start_num < 26*10**12)
    pg = 11 if (range.between?(10**8, 10**9-1) && start_num < 55*10**7)  ||
               (range >= 10**9 && start_num < 45*10**9)
  end
  primes = [2,3,5,7,11,13].select! {|p| p <= pg}
  [primes, primes.reduce(:*)]   # [excluded primes, mod] for PG
end
```

This method first appeared in 2.4.0, and was used in primes to adaptively selects the PG to optimize the speed of sozcore2. In 2.5.0 it was modified again, and added to primescnt and primenth, which are also clients of sozcore2. It used a simpler technique, and was faster for some use cases. However, it also produced some extremely suboptimal performance for large range sizes, based on its simplistic model. After performing detailed benchmarking over various range sizes and starting values, I determined an optimal PG profile for each case. This was nonlinear over the number space my system (cpu, available memory) could process. This profile ultimately became a table of the best PG to use for different start values vs range sizes. For 2.6.0, the table was translated into code to perform the selection process, and was implemented using the cover? method. For 2.7.0 I switched to using between? (which is faster and also reads better) and logical operators `&&` and `||`, while maintaining the same code structure.

Further work needs to be done on different systems (ARM cpus, etc) with greater available memory. As usable memory increases, larger start values and range sizes can be processed, which should make using larger SP PGs optimal. These are scenarios I currently haven't had the resources to explore.

array_check(n, v)

```
def array_check(n,v)  # catch out-of-memory errors on array creation
  Array.new(n,v) rescue return            # return an array or nil
end
```

In Ruby parlance this method is known as *syntactical sugar*, or helper method in Rails. I just DRYed up the code to perform handling of array creation out-of-memory errors into this method, so I could take it out of sozcore2, where it's used twice. It reduces the loc in sozcore2 and also makes it easier to read and understand. If an error occurs it returns `nil`, which causes a distinct error message to be displayed in sozcore2 for each array creation failure, otherwise it creates and returns the array. I can abbreviate writing it as as shown, instead of the long form as: `begin Array.new(n,v) rescue return end` because it's solely encapsulated within the method definition.

Public Methods

The primes-utils methods you can actually use are public methods. As explained earlier, they are all instance_methods of class Integer, so they use and send messages to just Integer objects. Having factored out the common useable pieces to the private method, they have become smaller|simpler to code over time. I'll present them by their functional categories, which will make it easier to follow.

Factoring and Primality Testing

A process used to factor a number also determines if it's prime, but not necessarily vice versa. Since this is conceptually the easiest to understand I'll start here to delve into the code.

Using factor

Unix, and derivative OSes like Linux and BSDs, come with the cli command factor [10], a very fast C program which performs a sophisticated factoring algorithm applicable to *large* numbers. It is so much faster than prime_division from the `prime.rb` standard library, and useable for so much bigger numbers, it would be almost criminal not to use it. (I'm a firm believer in harnessing the full power of FOSS.)

Here is typical output factor produces for two nonprimes:

```
$ factor 323321
323321: 29 11149

$ factor 308700
308700: 2 2 3 3 5 5 7 7 7
```

Here is typical output for a prime number:

```
$ factor 313933
313933: 313933
```

We can call factor (or other cli commands) from inside ruby in the following manner:

```
> `factor 1000`
=> "1000: 2 2 2 5 5 5\n"
```

This returns a string for the answer. It's a simple matter to process this string to get the same output format as with prime_division, an array of form [[p1,e1],[p2,e2]....[pn,en]], where each entry is a two element array of a prime factor and exponent (times of occurrence within the output). Here's the code.

factors | prime_division w/factor

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

alias  prime_division  factors
```

The first line makes an OS system call with factor on `self`, which returns a string output as above. This string is split into an array of string numbers, of which the factors are the 2nd to last elements, which are mapped from string values to integers. `factors` is now an array of integer values. The second line then creates a hash of zero initial values, and uses the prime values in `factors` as keys, and their count

in the array as their key values. The output is the sorted conversion to array of the hash key|value pairs. I then alias prime_division to factors to be used to replace the `prime.rb` version.

It's even simpler to use factor as a primality tester.

prime? w/factor

```
def prime?
  `factor #{self.abs}`.split(' ').size == 2
end
```

Here all we need to do is test if factor's string array size is 2. If `true` the number is prime, if `false` the array size is > 2, which means the number has at least 2 prime factors and thus is nonprime.

For the (unfortunate) people whose OS doesn't have factor I provide a pure ruby implementation of the basic brute force factoring algorithm that goes something like this:

1) For an integer N, create a list of primes up to the sqrtN
3) Divide N by each prime in the list
4) If no prime divides N, then N is prime, return [[N,1]]
5) For each prime that divides N, determine how many times it does, and save results
6) Return factors [[p1,e1],[p2,e2]....[pn,en]]

Pretty simple. The only real question is how do we get a list of primes up to the sqrtN for any N? We know the answer to this is to pick a Prime Generator and generate them.

factors | primes_division (pure ruby*)*

```
def factors(p=13)
  # Return prime factors of n in form [[p1,e1],[p2.e2]..[pn.en]]
  # Uses P13 SP PG as default Prime Generator
  seeds = [2, 3, 5, 7, 11, 13, 17, 19]
  p = 13 unless seeds.include? p

  primes = seeds[0..seeds.index(p)]
  mod = primes.reduce(:*)   # modulus: modPn = 2*3*5*7*..*Pn
  residues, rescnt = make_residues_rescnt(mod)

  n = self.abs              # number to factor
  factors = []              # init empty factors array

  return [] if n < 2
  return [[n,1]] if primes.include? n
  primes.each {|p| while n%p == 0; factors << p; n /= p end }

  sqrtN = Math.sqrt(n).to_i
  modk,r=0,1
  while (p = modk+residues[r]) <= 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
  end
  factors << n if n > 1
  h=Hash.new(0); factors.each {|f| h[f] +=1}; h.to_a.sort
end
```

Here I use P13 as the default PG to generate the primes, but you change it at run time to any PG in the `seeds` array, which has max value for P19. I'll explain why you might want to change this later. The next section of code creates the `primes` array of excluded primes for the selected PG, and then its `mod` and `rescnt` values and `residues` array. After a few preliminaries, we start factoring N with the primes in `primes` with the line: `primes.each {|p| while n%p == 0; factors << p; n /= p end }` This line sucks out the number of times each excluded prime is a factor of N, reducing N each time. The next section of code is the core of the brute force factoring algorithm.

We first compute `sqrtN`, then set `p=residues[1]`, which is the first residue, and prime, for the PG being used. For P13, that would be 17, for P17 it would be 19, etc. Remember, we've already sucked out the factors for the primes less than this. The `while` loop iterates over all the pcs <= `sqrtN`. The `if` branch then does the following for each pc that divides the current N (only the primes will): 1) it stores the prime p into `factors`; 2) it sets r to use p again to check for more factors of it; 3) it reduces N by p; and 4) computes a new `sqrtN` for the reduced N.

So as we find prime factors of N, we divide them out and create a smaller N to factor, for which we only have to check primes up to its smaller `sqrtN`. Thus, we continually reduce N's size, and the number of pcs we need to use, as we go. The worst case scenario (for time and computation) is for N prime, because every pc <= `sqrtN` will then be checked. (If N is *very large* you might want to check it first with primemr? so you don't try factoring a large prime.)

After the `while` loop ends, N is reduced to a remaining prime, or was prime initially, which is stored in `factors`, or it's '1' (which isn't stored) if N is factored completely. We then just perform the hash trick shown previously to create the output.

The run time option to choose a PG is to optimize speed. I picked P13 as the default because on my reference systems it performs well overall, though P17 was actually faster for *large* numbers. For a given system (cpu, memory size, ruby version, OS, etc) the fastest PG will likely be between P13–P19. The way to find out which is best for your system is to *test, test, and test!*

Primality Testing

There are two basic approaches we can use to test if an integer is prime – we can determine if it can be factored, or we can test it for various *number theoretic* characteristics. We've seen how we can factor a number, so what are these number theory properties we can test for?

Two simple visual tests to check if N is nonprime is to see if its even (i.e. if N > 2 and ends with digits 0, 2, 4, 6, 8), or whether it ends in 5 (which means its divisible by 5). The evenness test eliminates half of all the integers, and the 5 test 10% more. Thus, all primes > 5 must end with digits 1, 3, 7, or 9, which are 40% of all integers, and conversely, 60% can easily be visually checked to be nonprime.

A more sophisticated test is to check if **N % mod** is a residue value for some PG. For PGs > P5|30, this test subsumes the above two tests, and arithmetically eliminates an increasing percentage of integers from prime consideration for increasing SP PGs. Using just P3 eliminates 2/3 (67%) of the integers from being prime, with only **N % 6 => {1,5}** integers being viable prime candidates. (Unfortunately, I was never taught this simple test through all my years of formal schooling.)

The `prime.rb` library provides prime? for primality testing, and like with prime_division, I create two versions. I already showed the version that uses factor, which provides fast testing of *very large* numbers. In *Improved Primality Testing and Factorization in Ruby* [4] I presented various pure ruby methods for primality testing. In primes-utils I use the version below which benchmarked the best.

prime? (pure ruby)

```
def prime?       # Uses P7 Strictly Prime Generator
  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
  is_a_pc = residues.include?(n%mod)  # true if n is a prime candidate
  return false unless (n > 1 and is_a_pc) or [2,3,5,7].include? n
  return true  if n <= 211 and not [121,143,169,187,209].include? n

  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
  true          # n is prime (100%|deterministically)
end
```

I use P7 as the PG here, and list its residue values to make it easier to see (remember) what's going on. I first check if N is a prime candidate, or a P7 excluded prime, and return `false` if not (it's a nonprime). N is prime if it's <= 211 and not one of the 5 composite P7 residues, otherwise it's a prime candidate we have to further test. I then (by brute force) test if N is a multiple of any prime <= `sqrtN`. For each successive kth resgroup whose first pc is <= `sqrtN`, I test if any of their 48 pcs divides N. If any does, the `while` loop returns `false`, as N is nonprime. If none do, the `while` loop ends and `true` is returned.

This coding is *prime* (pun intended) for parallelization. Each residue test is independent and can be done in parallel. In ruby, as it goes thru the `or` chain, the first `true` test (which means a pc divides N) terminates further testing and returns `true` for the whole chain to `if`, and thus `false` is returned. If no pc divides N the `or` chains are `false`, and fall thru after the last resgroup with pcs <= sqrtN is tested.

It so happens, this version runs best overall on JRuby, and comparatively no worse than other codings using larger PGs. However, using this technique for larger PGs doesn't increase speed, which means it hits a [soft|hard]ware barrier, because (everything being equal) it should. This is an instance where the real tangibles of language and hardware design totally determine which version works best on a system. This version seems to hit a sweet spot on my system across various Ruby VMs and 32/64-bit OSes. If you want to see how the code in my paper performs on your system you can get it here [5] and run it.

In prime-utils I provide primemr? as a number theoretic primality test using the Miller-Rabin [8] test. Its main advantages are it's very fast, and can work with *extremely large* numbers, while its (minimal) disadvantage is it's a statistical test (in this form). However, this is not a real (versus theoretical) problem, because I code it so you can increase its reliability parameter at run time, and because it's so fast, you can run it multiple times on a given number to satisfy *your* confidence level.

In fact, I implement primemr? to first do the **N % 6 => {1,5}** test, to reduce the number of nonprimes subjected to the M-R algorithm to increase its reliability even more. It uses the `openssl` library to use its mod_exp(d,n) method to perform fast exponentiation and modular reduction on unlimited sized numbers. It's a coding improvement over the ruby code shown in [9].

primemr?

```
# Miller-Rabin prime test in Ruby (modified from Rosetta Code)
# Ruby Rosetta Code: http://rosettacode.org/wiki/Miller-Rabin_primality_test

require 'openssl'
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
```

Listing and Counting Primes

We'll now examine methods that provide nonexistent functionality in the `prime.rb` standard library.

One of my primary reasons for creating primes-utils was to add functionality for *all things primes* not in `prime.rb`. Two of the most common of these functions is listing and counting primes over a range. From a coding perspective, 95% of the code is the same. I create a common process to identify the primes, then decide whether to return a numerated list of them, or their count, based on the method. The selection process of which to perform is hardwired in the public methods that implement these functions, while the common computational code resides in the private methods.

The methods primes, primesf, and primesmr provide a list of prime values over a range while the methods primescnt, primescntf, and primescntmr are optimized to provide just their count. These public methods act primarily as clients to the private methods sozcore1 and sozcore2, and merely have to minimally operate on their output data to return their results. As you'll see, they're very compact and much less computationally complex.

Because primesf, primesmr, primescntf, and primescntmr are so similar we'll look at them all together. They are clients to sozcore1, so they pseudo-directly generate the primes. Thus, they have no inherent memory limitations, and are only limited by the numerical constraints of factor, primemr?, and time.

primes[f|mr]

```
 def primes[f|mr](start_num=0)
  # List primes within a number range: end_num - start_num

  sozdata = sozcore1(self, start_num, true)  # true for primes list
  return sozdata[1] unless sozdata[0]        # if range within excluded primes

  pcs_in_range, r, mod, modk, rescnt, residues, primes = sozdata

  pcs_in_range.times do         # list primes from this num pcs in range
    prime = modk + residues[r]
    primes << prime if prime.[prime?|primemr?]
    r +=1; if r > rescnt; r=1; modk +=mod end
  end
  primes
end
```

primescnt[f|mr]

```
def primescnt[f|mr](start_num=0)
  # Count primes within a number range: end_num - start_num

  sozdata = sozcore1(self, start_num, false) # false for primes count
  return sozdata[1] unless sozdata[0]        # if range within excluded primes

  pcs_in_range, r, mod, modk, rescnt, residues, primescnt = sozdata

  pcs_in_range.times do         # count primes from this num pcs in range
    primescnt +=1 if (modk + residues[r]).[prime?|primemr?]
    r +=1; if r > rescnt; r=1; modk +=mod end
  end
  primescnt
end
```

`start_num` and `end_num` are fed into sozcore1, with a `method_flag`, which is `true` for returning prime values, and `false` for their count. The array `sozdata` captures its output, whose first element is tested. If `false`, the second element is returned as output, which is an array of `seeds` primes (or their count) if the range is within the `seeds` primes of sozcore1. If `end_num` is greater than the `seeds` values, `sozdata` captures sozcore1's full output array `[pcs_in_range, r, mod, modk, rescnt, residues, primes|count]`, whose first element evals to `true`, so we skip the `return` and assign the outputs to the shown variables.

`pcs_in_range` is the number of pcs in the range, r the `residues` index for its first pc, with `modk` value, and `mod`, `rescnt`, and `residues` the PG's parameters in sozcore1. The last element is an array of partial seed primes (or their count) if `start_num` is within the seed primes, or an empty array [] (or 0) if not.

Finally, we loop `pcs_in_range` times and numerate and test the primality of each pc within the range, and either output an array of the primes, or their count, depending on the method. The primality test is done with either prime? (which uses factor) or primemr?. You really can't get much simpler than this!

The code is written to use any PG. In this instance P13 (P17) in sozcore1 performs best on my system. Merely replacing its `primes` array can change it to any SP generator, to optimize it for any other system.

We'll now look at primes and primescnt, which are sozcore2 clients. They have to do a little bit more work than the sozcore1 clients, but are still very simple.

primes

```
def primes(start_num=0)
  # List primes between a number range: end_num - start_num
  # Adaptively selects best Strictly Prime (SP) Prime Generator
  num = self.abs;  start_num = start_num.abs
  num, start_num = start_num, num  if start_num > num

  primes, mod = select_pg(num, start_num) # adaptively select PG
  prms, m, modk, residues, rescnt, x, maxprms, r = sozcore2(num, start_num, mod)
  return unless prms            # exit gracefully if sozcore2 mem error

  # init 'primes' w/any excluded primes in range then extract primes from prms
  primes.select! {|p| p >= start_num && p <= num}
  while m < maxprms             # list primes from sieved pcs in prms for range
    begin
      primes << modk + residues[r] if prms[m] == 1
    rescue Exception
      return puts "ERROR3: not enough memory to store all primes in output array."
    end
    r +=1; if r > rescnt; r=1; modk +=mod end
    m +=1
  end
  primes
end
```

primescnt

```
def primescnt(start_num=0)
  # Count primes between a number range: end_num - start_num
  # Adaptively selects best Strictly Prime (SP) Prime Generator
  num = self.abs;  start_num = start_num.abs
  num, start_num = start_num, num  if start_num > num

  if start_num < 3              # for all primes upto num
    start_num, nth, nthflag = set_start_value(num,false)
    return nth unless nthflag   # output num's key|count if a ref nth value
  end

  primes,mod = select_pg(num, start_num)  # adaptively select PG
  prms, m, * = sozcore2(num, start_num, mod)
  return unless prms            # exit gracefully if sozcore2 mem error

  # init prmcnt for any excluded primes in range then count primes in prms
  prmcnt = primes.count {|p| p >= start_num && p <= num}
  prmcnt = nth-1 if nthflag && nth > 0    # start count for small range
  prmcnt + prms[m..-1].count(1)
end
```

We start off for both making sure the range values are non-negative and assigned to the right variables. Then in primes, the `primes` array and `mod` are created for a selected PG. Then sozcore2 is invoked and its output assigned to the shown variables ('x' isn't used here). If it incurred and out-of-memory error we exit primes because `prms` wasn't created, and will be evaluated as `nil`. We then check if the range is partly inside the excluded primes (e.g. `50.primes 3`), or fully (e.g. `5.primes 2`), and reassign `primes` to contain just them if so. If `start_num` is beyond the excluded primes, `primes` will be set to an empty array. Then starting at `prms` address m, we iterate over `prms` and check if each address is a prime, and numerate and store them in `primes` if they are (provided `ERROR3` doesn't occur), then return `primes` when done. If the range is fully inside the excluded primes, `maxpcs` and m are 0 and the loop is skipped, and `primes` is returned with those excluded primes.

To find the count of the primes over a range you could just do (and still can) `start.primes(end).size`. In fact, it was the only way to do it until 2.0.0, when I introduced dedicated methods to count primes. The version in 2.3.0 – 2.4.0 was significantly faster for a range up to some N, and over large ranges, or ranges around large numbers, than the previous version of primescnt. In 2.5.0 I added adaptive PG selection, which made it even faster. It's also faster than primescnt[f|mr] under certain scenarios. So let's see how it works.

After assigning the range values, if `start_num` is less than 3 the range contains all the primes up to N, so set_start_value is invoked with `end_num` and a `false` flag, and its output assigned to the shown variables. If `end_num` happens to be a value in the `nths` hash table, then it's some nth prime, and `nthflag` will be `false` and `nth`, the key for the value, is returned as the prime count up to N. If `end_num` is not in the hash table, `nthflag` is `true`, `start_num` is set to the largest value in the table less than `end_num` (or 0), and `nth` its key|count (or 0). This `start_num` value will be used to create a smaller range to find the remaining primes up to `end_num`.

If `start_num` is > 2 (set_start_value is skipped) it keeps its original value as the start of some range, which may include excluded primes. The variables `nth` and `nthflag` are not assigned then, and are thus `nil`. Then select_pg adaptively selects the best PG to use for the derived numerical range.

Then sozcore2 is invoked, and only its first two outputs used, `prms` and m, the array of the sieved range pcs and the address for `start_num's` pc in it. If sozcore2 incurred an out-of-memory error then `prms` wasn't created, and will be evaluated as `nil` when tested, and primescnt exited.

The last three lines create the output, where the first two determine the number of primes that must be accounted for that aren't in `prms` but are still in the range. If the range includes all the primes up to N, or some portion of the PG's excluded primes, we have to include those excluded primes for the range. Thus, the first line sets `prmcnt` to be the count of any excluded primes inside the range, or zero if the start of the range is beyond the excluded primes.

The next line accounts for when `start_num` is < 3 (want all the primes up to N) and it's potentially replaced by a larger value closer to `end_num`, used to create a smaller range to sieve the remaining primes from. If this occurred, we want `prmcnt` to be all the primes up to the larger `start_num` value. At this stage in the code, if `nthflag` is `true` then `nth` is either the key for the closest value in the `nths` table < N, or 0 when there is no table value < N. If the first two conditions are true, `prmcnt` is reset to `nth-1` (we have to subtract 1 because the range includes that prime value). The last line then merely has to count the range primes in `prms` and add them to `prmcnt` to get the total count for the full range.

Finding the Nth Prime

Originally in 1.0.0, I created a crude approximation of the nthprime and counted up from the first (k=0) resgroup to the nth prime. In 2.0.0 I created a hash of indexed nth primes, in steps of 12.5 million up to the 1,112,500,000th prime, to create a small range containing the nth prime, then count from the lower indexed prime up to the nth prime. I also set a ceiling for what the max nth prime could be to eliminate memory errors and put some reasonable bounds on computation time. With 2.2.0 I upped the indexed primes to the 1,600,000,000th, created a better function to approximate the nth prime, and removed the ceiling on the max nth, so now you can go until you run out of memory (whereupon you get a nice informative error messages) or patience. I upped the indexed primes again to 2,012,500,000th in 2.3.0.

primenth | nthprime

```
def primenth(p=0)
  # Return value of nth prime
  # Adaptively selects best SP PG, unless valid input PG given at runtime
  seeds = [2, 3, 5, 7, 11, 13]
  (primes=seeds[0..seeds.index(p)]; mod=primes.reduce(:*)) if seeds.include? p

  n = self.abs                  # the desired nth prime
  return n > 0 ? seeds[n-1] : 0  if n <= seeds.size

  start_num, nth, nthflag = set_start_value(n,true)
  return start_num if nthflag   # output nthprime value if n a ref prime

  num = approximate_nth(n)      # close approx to nth >= real nth
  primes, mod = select_pg(num, start_num) unless primes
  prms, m, modk, residues, rescnt, pcs2start, * = sozcore2(num, start_num, mod)
  return unless prms            # exit gracefully if sozcore2 mem error

  # starting at start_num's location, find nth prime within given range
  prmcnt = n > nth ? nth-1 : primes.size
  pcnt = prmcnt + prms[m..-1].count(1)  # number of primes upto nth approx
  return "#{pcnt} not enough primes, nth approx too small" if pcnt < n
  while prmcnt < n; prmcnt +=1 if prms[m] == 1; m +=1 end
  k, r = (m + pcs2start).divmod rescnt
  mod*k + residues[r]
end
```

As of 2.5.0, the default PG is adaptively selected by select_pg. If an explicit PG is chosen at runtime (up to P13) its parametized. If you pass an object that's not in `seeds` it defaults to adaptive selection.

We first check if n is a small nth prime in the `seeds` array and return its value if it is. If not, we pass n and `true` to set_start_value, and if n is a key (index prime) in `nths` `nthflag` will be `true`, and `start_num` is returned as the nth prime. If n not in `nths`, `nthflag` is `false` and `start_num` is either the value of the largest key < n, or 0. This is the lower bound for a range containing the nth prime, while the value computed by approximate_nth is the range's upper bound. We then create the excluded primes array and `mod` for the selected PG, invoke sozcore2, and exit if an out-of-memory error occurred.

Then `prmcnt` is set to `nth-1` if n > returned hash key, or the number of excluded primes, when n < the smallest hash key, which means the range is [2 – `num`]. We then set `pcnt` to all the primes up to `end_num`, and exit if there aren't enough to find the nth prime n (which should never happen by design). Then the `while` loop computes the address m in `prms` for the nth prime, i.e. the number of pcs in `prms` up to it. The term `(m + pcs2start)` represents all the pcs up to the nth prime, which is numerated in the standard way by computing its resgroup k, and residue r, and then computing its value – `mod*k + residues[r]`.

Future Development

The functionality that primes-utils contains provided the means to answer some basic questions about primes I needed to know in order to solve problems I was interested in. Obviously, there are a great many questions and problems that exist regarding various aspects of the primes. But it provides a much faster and useful set of tools for people using Ruby to apply to problems involving primes than those that come natively in the language.

There are some changes and improvements to Ruby that would make primes-utils better. Currently it does not provide fast byte or bit addressable arrays. There are rubygems which provide some of these capabilities, but they are slow compared to its native array implementation. The arrays `prms` and `prms_range`, used for sieving primes in sozcore2, waste lots of memory because of this, which reduces the range sizes they can process before running out of memory. Since the MRI VM is written in C, providing the ability it has of creating byte addressable arrays would expand their processing power. There has been some discussion that Ruby 3.0 (?) may enable users to create data 'types' which may allow for a native byte type. If a native byte addressable array type were available it would increase the number range by at least 4 to 8 times more for 32|64 bit OSes.

Also, as currently written the methods are single-threaded, though the SoZ, and some other functions, are inherently parallel algorithms. Theoretically, significant performance gains exist if Ruby allowed native mulit-thread|core parallel processing, requiring it to remove the Global Interpretor Lock (GIL). Matz has stated this is a goal of Ruby 3.0, along with signficant speed increases over Ruby 2. Though JRuby and Rubinius use native threads|cores, MRI is primes-utils reference VM. These language limitations also make it infeasible to implement a Segmented Sieve of Zakiya (SSoZ) [3], which could also potentially increase memory efficiency and speed. There are also possibly achievable performance gains using programming tricks like memoization, which I currently haven't explored much.

Another possibility to explore is using faster compiled languages to implement pieces of functionality with, and interface with their compiled executables in Ruby. I already do that by using factor if it's available, but I didn't have to compile, package, and provide it, the OS did. C is the obvious first choice to try for this, though newer compiled languages (like Rust, etc) may be better, especially for providing parallelism|concurrency. Of course, the code can also be translated into other dynamic languages (Python, Javascript, Elixir), and it would be interesting to ultimately see how it looks, and performs, in a host of other languages.

I think the most significant additional performance gains will result from enhancing the algorithms, particularly for adaptively selecting (tuning) the PGs. The one currently used is quite rudimentary, and only based on range size. This can undoubtedly be made more sophisticated and generalized. However, performance also seems to be partly tied to the characteristics of the hardware (memory, cache size, architecture) along with the language. One size doesn't seem to fit all.

Ultimately I would like to see at least some of these techniques|methods make it into the Ruby standard library, to make it more natively useful for numerical and mathematical applications.

Glossary

greatest common divisor|gcd – largest common divisor of two integers
mod – short for modulus; an even number which determines all of a PG's parameters
modk – value of the modulus times the kth resgroup number, equal to mod*k
rescnt – residues count; the number of residues for a prime generator
residues – the set of integers r ε {1, r1..mod-1} less than and coprime to mod, i.e, gcd(r,mod) = 1
residue track|restrack – a row in a PG's prime candidates table, and the values along it
residues group|resgroup – the prime candidates contained in the kth column of a pcs table
prime candidate(s)|pc(s) – the integer values a PG generates
pcs table|pcstable – matrix|table representation of the pcs a prime generator can generate
k – resgroup numerical index value k = 0, 1, 2...
ks – resgroup index for the start_num of a range
excluded primes – the set of small prime numbers a prime generator does not generate
prime generator|PG – an expression of form Pn = mod*k + r, which can generate the primes
sequential prime generator – a PG that generates all the primes in sequence past its excluded primes
Strictly Prime|SP generator – a PG with modulus equal to the factorial of consecutive primes
FOSS – Free & Open Source Software
loc – lines of code
DRY – Don't Repeat Yourself, a key mantra for Ruby programmers
SoE – Sieve of Eratosthenes
SoZ – Sieve of Zakiya
VM – Virtual Machine

References

[1] https://www.scribd.com/doc/73385696/The-Sieve-of-Zakiya
[2] https://www.scribd.com/doc/73384039/Ultimate-Prime-Sieve-Sieve-Of-Zakiya
[3] https://www.scribd.com/doc/228155369/The-Segmented-Sieve-of-Zakiya-SSoZ
[4] https://www.scribd.com/doc/150217723/Improved-Primality-Testing-and-Factorization-in-Ruby-revised
[5] https://gist.github.com/jzakiya/455f2357cdb08f4ee1c4
[6] https://en.wikipedia.org/wiki/Sieve_of_Eratosthenes
[7] https://en.wikipedia.org/wiki/Dirichlet's_theorem_on_arithmetic_progressions
[8] https://en.wikipedia.org/wiki/Miller-Rabin_primality_test
[9] http://rosettacode.org/wiki/Miller-Rabin_primality_test
[10] https://en.wikipedia.org/wiki/Factor_%28Unix%29
[11] https://en.wikipedia.org/wiki/GNU_Core_Utilities
[12] https://geogebra.org/
[13] https://www.wolframalpha.com
[14] https://www.libreoffice.org/
[15] https://github.com/JuanitoFatas/fast-ruby

Source Code

primes-utils/lib/primes/utils.rb

```
# need rubygems to load gems, and rational for 'gcd' method for 1.8
%w/rubygems rational/.each{|r| require r} if RUBY_VERSION =~ /^(1.8)/

require "primes/utils/version"

module Primes
  module Utils
    # Upon loading, determine if platform has cli command 'factor'

    private

    @@os_has_factor = false
    begin
      if `factor 10`.split(' ') == ["10:", "2", "5"]
        @@os_has_factor = true
      end
    rescue
      @@os_has_factor = false
    end

    public

    if @@os_has_factor   # for platforms with cli 'factor' command

      def prime?
        `factor #{self.abs}`.split(' ').size == 2
      end

      def factors(p=0)   # p is unused dummy variable for method consistency
        factors = `factor #{self.abs}`.split(' ')[1..-1].map(&:to_i)
        h = Hash.new(0); factors.each {|f| h[f] +=1}; h.to_a.sort
      end

      def primesf(start_num=0)
        # List primes within a number range: end_num - start_num
        # Uses 'prime?' to check primality of prime candidates in range
        sozdata = sozcore1(self, start_num, true)  # true for primes list
        pcs_in_range, r, mod, modk, rescnt, residues, primes = sozdata

        pcs_in_range.times do         # list primes from this num pcs in range
          prime = modk + residues[r]
          primes << prime if prime.prime?
          r +=1; if r > rescnt; r=1; modk +=mod end
        end
        primes
      end

      def primescntf(start_num=0)
        # Count primes within a number range: end_num - start_num
        # Uses 'prime?' to check primality of prime candidates in range
        sozdata = sozcore1(self, start_num, false) # false for primes count
        pcs_in_range, r, mod, modk, rescnt, residues, primescnt = sozdata

        pcs_in_range.times do         # count primes from this num pcs in range
          primescnt +=1 if (modk + residues[r]).prime?
          r +=1; if r > rescnt; r=1; modk +=mod end
        end
        primescnt
      end

      puts "Using cli 'factor' for prime? primesf  primescntf  factors|prime_division"

    else  # use pure ruby versions for platforms without cli command 'factor'
```

```
  def prime?       # Uses P7 Strictly Prime Generator
    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
    is_a_pc = residues.include?(n%mod)  # true if n is a prime candidate
    return false unless (n > 1 and is_a_pc) or [2,3,5,7].include? n
    return true  if n <= 211 and not [121,143,169,187,209].include? n

    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
    true          # n is prime (100%|deterministically)
  end

  def factors(p=13)
    # Return prime factors of n in form [[p1,e1],[p2,e2]..[pn,en]]
    # Uses P13 SP PG as default Prime Generator
    seeds = [2, 3, 5, 7, 11, 13, 17, 19]
    p = 13 unless seeds.include? p

    primes = seeds[0..seeds.index(p)]
    mod = primes.reduce(:*)   # modulus: modPn = 2*3*5*7*..*Pn
    residues, rescnt = make_residues_rescnt(mod)

    n = self.abs              # number to factor
    factors = []              # init empty factors array

    return [] if n < 2
    return [[n,1]] if primes.include? n
    primes.each {|p| while n%p == 0; factors << p; n /= p end }

    sqrtN = Math.sqrt(n).to_i
    modk,r=0,1
    while (p = modk+residues[r]) <= 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
    end
    factors << n if n > 1
    h=Hash.new(0); factors.each {|f| h[f] +=1}; h.to_a.sort
  end

  puts "Using pure ruby versions for all methods"
end

# Replace slow ruby library method prime_division with faster version
alias  prime_division  factors
```

```
def primenth(p=7)
  # Return value of nth prime
  # Adaptively selects best SP PG, unless valid input PG given at runtime
  seeds = [2, 3, 5, 7, 11, 13]
  (primes=seeds[0..seeds.index(p)]; mod=primes.reduce(:*)) if seeds.include? p

  n = self.abs                  # the desired nth prime
  return n > 0 ? seeds[n-1] : 0  if n <= seeds.size

  start_num, nth, nthflag = set_start_value(n,true)
  return start_num if nthflag   # output nthprime value if n a ref prime key

  num = approximate_nth(n)      # close approx to nth >= real nth
  primes, mod = select_pg(num, start_num) unless primes
  prms, m, modk, residues, rescnt, pcs2start, * = sozcore2(num, start_num, mod)
  return unless prms            # exit gracefully if sozcore2 mem error

  # starting at start_num's location, find nth prime within given range
  prmcnt = n > nth ? nth-1 : primes.size
  pcnt = prmcnt + prms[m..-1].count(1)  # number of primes upto nth approx
  return puts "#{pcnt} not enough primes, nth approx too small" if pcnt < n
  while prmcnt < n; prmcnt +=1 if prms[m] == 1; m +=1 end
  k, r = (m + pcs2start).divmod rescnt
  mod*k + residues[r]
end

alias  nthprime  primenth       # to make life easier

def primes(start_num=0)
  # List primes between a number range: end_num - start_num
  # Adaptively selects best Strictly Prime (SP) Prime Generator
  num = self.abs;  start_num = start_num.abs
  num, start_num = start_num, num  if start_num > num

  primes, mod = select_pg(num, start_num) # adaptively select PG
  prms, m, modk, residues, rescnt, x, maxprms, r = sozcore2(num, start_num, mod)
  return unless prms            # exit gracefully if sozcore2 mem error

  # init 'primes' w/any excluded primes in range then extract primes from prms
  primes.select! {|p| p >= start_num && p <= num}
  while m < maxprms             # list primes from sieved pcs in prms for range
    begin
      primes << modk + residues[r] if prms[m] == 1
    rescue Exception
      return puts "ERROR3: not enough memory to store all primes in output array."
    end
    r +=1; if r > rescnt; r=1; modk +=mod end
    m +=1
  end
  primes
end

def primescnt(start_num=0)
  # Count primes between a number range: end_num - start_num
  # Adaptively selects best Strictly Prime (SP) Prime Generator
  num = self.abs;  start_num = start_num.abs
  num, start_num = start_num, num  if start_num > num

  if start_num < 3              # for all primes upto num
    start_num, nth, nthflag = set_start_value(num,false)
    return nth unless nthflag   # output num's key|count if a ref nth value
  end

  primes,mod = select_pg(num, start_num)  # adaptively select PG
  prms, m, * = sozcore2(num, start_num, mod)
  return unless prms            # exit gracefully if sozcore2 mem error
```

```
  # init prmcnt for any excluded primes in range then count primes in prms
  prmcnt = primes.count {|p| p >= start_num && p <= num}
  prmcnt = nth-1 if nthflag && nth > 0    # start count for small range
  prmcnt + prms[m..-1].count(1)
end


# Miller-Rabin test (modified from: http://rosettacode.org/wiki/Miller-Rabin_primality_test)

require 'openssl'

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

def primesmr(start_num=0)
  # List primes within a number range: end_num - start_num
  # Uses 'primemr' to check primality of prime candidates in range
  sozdata = sozcore1(self, start_num, true)  # true for primes list
  pcs_in_range, r, mod, modk, rescnt, residues, primes = sozdata

  pcs_in_range.times do         # list primes from this num pcs in range
    prime = modk + residues[r]
    primes << prime if prime.primemr?
    r +=1; if r > rescnt; r=1; modk +=mod end
  end
  primes
end

def primescntmr(start_num=0)
  # Count primes within a number range: end_num - start_num
  # Uses 'primemr' to check primality of prime candidates in range
  sozdata = sozcore1(self, start_num, false) # false for primes count
  pcs_in_range, r, mod, modk, rescnt, residues, primescnt = sozdata

  pcs_in_range.times do         # count primes from this num pcs in range
    primescnt +=1 if (modk + residues[r]).primemr?
    r +=1; if r > rescnt; r=1; modk +=mod end
  end
  primescnt
end

def primes_utils                # display list of available methods
  methods = %w/prime? primemr? primes primesf primesmr primescnt primescntf
               primescntmr primenth|nthprime factors|prime_division primes_utils/
  (methods - (@@os_has_factor ? [] : %w/primesf primescntf/)).join(" ")
end
```

```
private

def make_residues_rescnt(mod)
  residues=[1]; 3.step(mod,2) {|r| residues << r if mod.gcd(r) == 1}
  residues << mod+1
  [residues, residues.size-1]   # return residues array and rescnt
end

# lte= true: first output element is number of pcs <= num
# lte=false: num pcs <, residue index, and resgroup value, for (start_)num pc
def pcs_to_num(num,mod,rescnt,residues,lte)
  num -=1; lte ? (num |=1; k=num.abs/mod) : k = (num-1).abs/mod
  modk = mod*k; r=1
  r +=1 while num >= modk+residues[r]
  [rescnt*k + r-1, r, modk]     # [num pcs, r index, num modulus]
end

# Use default SP Prime Generator to parametize the pcs within a range
# inputs:  end_num|start_num of range; method_flag to numerate|count primes
# outputs: maxpcs-m - number of pcs in the range
#          r        - residue index value for start_num pc of range
#          mod      - mod value for PG
#          modk     - base value for start_num's resgroup
#          rescnt   - number of residues for PG
#          residues - array of residues plus mod+1 for PG
#          primes array|primes.size - primes array or size based on method_flag
def sozcore1(num, start_num, method_flag)
  num = num.abs;   start_num = start_num.abs
  num, start_num = start_num, num  if start_num > num

  primes = [2,3,5,7,11,13]      # excluded primes for P13 default SP PG
  mod = primes.reduce(:*)       # P13 modulus: 2*3*5*7*11*13 = 30030
  residues, rescnt = make_residues_rescnt(mod)
  maxpcs,* = pcs_to_num(num,mod,rescnt,residues,true) # num pcs <= end_num

  # init 'primes' w/any excluded primes in the range, or [] if none
  primes.select! {|p| p >= start_num && p <= num}

  # compute parameters for start_num pc, then create output parameters array
  m, r, modk = pcs_to_num(start_num, mod, rescnt, residues, false)
  [maxpcs-m, r, mod, modk, rescnt, residues, method_flag ? primes : primes.size]
end

# Perform SoZ with given Prime Generator and return array of parameters
# inputs:  end_num and start_num of range and mod value for PG
# outputs: prms - binary (0,1) array of pcs within a range or to end_num
#          m    - num of pcs in prms < start_num; so prms[m] = start_num
#          modks    - mod value for start_num's resgroup
#          residues - array of residues plus mod+1 for PG
#          rescnt   - number of residues for PG
#          pcs2start- number of pcs < start_num pc
#          maxprms  - number of pcs to find primes from; prms array size
#          rs   - residue index location for first pc >= start_num
def sozcore2(num, start_num, mod)
  residues, rescnt = make_residues_rescnt(mod)  # parameters for the PG
  maxprms,* = pcs_to_num(num,mod,rescnt,residues,true) # num pcs <= end_num

  # for start_num pc, find num pcs <, residue index, and resgroup mod value
  pcs2start, rs, modks = pcs_to_num(start_num, mod, rescnt, residues, false)

  sqrtN = Math.sqrt(num).to_i   # sqrt of end_num (end of range)
  pcs2sqrtN,* = pcs_to_num(sqrtN,mod,rescnt,residues,true) # num pcs <= sqrtN

  split_arrays = start_num > sqrtN # flag, true if two arrays used for sieve
  maxpcs = maxprms              # init array size for all pcs to end_num
  if split_arrays               # if start_num > sqrtN create two arrays
    maxpcs = pcs2sqrtN          # number of pcs|array size, for pcs <= sqrtN
    max_range = maxprms-pcs2start # number of pcs in range start_num to end_num
```

```
    prms_range = array_check(max_range,1) # array to represent pcs in range
    return puts "ERROR1: range size too big for available memory." unless prms_range
  end
  prms = array_check(maxpcs,1)  # array for pcs upto sqrtN, or end_num
  return puts "ERROR2: end_num too big for available memory." unless prms

  # residues offsets to compute a pcs address in its resgroup in prms
  pos =[]; rescnt.times {|i| pos[residues[i]] = i-1}

  # Sieve of Zakiya (SoZ) to eliminate nonprimes from prms and prms_range
  modk,r,k=0,0,0
  pcs2sqrtN.times do |i|        # sieve primes from pcs upto sqrt(end_num)
    r +=1; if r > rescnt; r=1; modk +=mod; k +=1 end
    next unless prms[i] == 1    # when a prime location found
    prm_r = residues[r]         # its residue value is saved
    prime = modk + prm_r        # its value is numerated
    prmstep = prime * rescnt    # its primestep computed
    kcon = k * prmstep          # its inner loop constant computed
    residues[1..-1].each do |ri|# now perform sieve with it
      # convert (prime * (modk + ri)) pc value to its address in prms
      # computed as nonprm = (k*(prime + ri) + kn)*rescnt + pos[rr]
      kn,rr  = (prm_r * ri).divmod mod           # residues product res[group|track]
      nonprm = kcon + (k*ri + kn)*rescnt + pos[rr]# 1st prime multiple address with ri
      while nonprm < maxpcs; prms[nonprm]=0; nonprm +=prmstep end
      if split_arrays                            # when start_num > sqrtN
        nonprm = (pcs2start - nonprm)%prmstep     # (start_num - closest multiple) pcs
        nonprm = prmstep - nonprm if nonprm != 0  # location in range, or beyond
        while nonprm < max_range; prms_range[nonprm]=0; nonprm += prmstep end
      end
    end
  end
  # determine prms array parameters and starting location value m for start_num
  split_arrays ? (prms = prms_range; maxprms = max_range; m = 0) : m = pcs2start
  [prms, m, modks, residues, rescnt, pcs2start, maxprms, rs] # parameters output
end

def approximate_nth(n)          # approximate nthprime value >= real value
  b = 0.5722*n**0.0088          # derived equation to compute close nth values
  a = b*(Math.log(Math.log(n)))
  (n*(Math.log(n)+a)+3).to_i    # use nth approximation as end_num of range
end

def set_start_value(n, hshflag) # find largest index nthprime|val <= n
  if hshflag
    return [nths[n], 0, true] if nths.has_key? n       # if n is key in nths table
    nth = nths.keys.sort.reverse.detect {|k| k < n}    # find largest indexed key < n
    [nth ? nths[nth] : 0, nth ||= n+1, false]          # [start_num, nth, false]
  else
    return [0,nths.key(n),false] if nths.has_value? n  # if n is value in nths table
    v=val= nths.values.sort.reverse.detect {|v| v < n} # find largest indexed val < n
    [v ||= 0, val ? nths.key(val) : 0, true]           # [start_num, nth, true]
  end
end

def nths                        # hash table index of reference nth primes
  nths={1000000 => 15485863,   5000000 =>   86028121,   7500000 =>  132276691,
     10000000 =>  179424673,  12500000 =>  227254201,  15000000 =>  275604541,
     18500000 =>  344032387,  25000000 =>  472882027,  31000000 =>  593441843,
     37500000 =>  725420401,  43500000 =>  848321917,  50000000 =>  982451653,
     56000000 => 1107029837,  62500000 => 1242809749,  68500000 => 1368724829,
     75000000 => 1505776939,  81500000 => 1643429659,  87500000 => 1770989609,
     93500000 => 1898979367, 100000000 => 2038074743, 106500000 => 2177624377,
    112500000 => 2306797469, 125000000 => 2576983867, 137500000 => 2848518523,
    150000000 => 3121238909, 162500000 => 3395057291, 175000000 => 3669829403,
    187500000 => 3945592087, 200000000 => 4222234741, 212500000 => 4499683009,
    225000000 => 4777890881, 237500000 => 5056862311, 250000000 => 5336500537,
    262500000 => 5616787769, 275000000 => 5897707297, 287500000 => 6179208157,
    300000000 => 6461335109, 312500000 => 6743943629, 325000000 => 7027107881,
```

```
    337500000 => 7310793337, 350000000 => 7594955549, 362500000 => 7879581839,
    375000000 => 8164628191, 387500000 => 8450100349, 400000000 => 8736028057,
    412500000 => 9022375487, 425000000 => 9309109471, 437500000 => 9596238593,
    450000000 => 9883692017, 462500000 =>10171564687, 475000000 =>10459805417,
    487500000 =>10748372137, 500000000 =>11037271757, 512500000 =>11326513039,
    525000000 =>11616020609, 537500000 =>11905863799, 550000000 =>12196034771,
    562500000 =>12486465863, 575000000 =>12777222833, 587500000 =>13068237251,
    600000000 =>13359555403, 612500000 =>13651119389, 625000000 =>13942985677,
    637500000 =>14235122851, 650000000 =>14527476781, 662500000 =>14820071503,
    675000000 =>15112928683, 687500000 =>15406031899, 700000000 =>15699342107,
    712500000 =>15992957251, 725000000 =>16286768243, 737500000 =>16580801137,
    750000000 =>16875026921, 762500000 =>17169527171, 775000000 =>17464243799,
    787500000 =>17759139259, 800000000 =>18054236957, 812500000 =>18349591409,
    825000000 =>18645104897, 837500000 =>18940846207, 850000000 =>19236701629,
    862500000 =>19532780327, 875000000 =>19829092147, 887500000 =>20125592731,
    900000000 =>20422213579, 912500000 =>20719050323, 925000000 =>21016060633,
    937500000 =>21313231963, 950000000 =>21610588367, 962500000 =>21908128993,
    975000000 =>22205818561, 987500000 =>22503733657, 1000000000=>22801763489,
    1012500000=>23099993743, 1025000000=>23398391231, 1037500000=>23696858797,
    1050000000=>23995554823, 1062500000=>24294392179, 1075000000=>24593421187,
    1087500000=>24892587403, 1100000000=>25191867719, 1112500000=>25491361037,
    1125000000=>25790970053, 1137500000=>26090709563, 1150000000=>26390560513,
    1162500000=>26690560601, 1175000000=>26990744987, 1187500000=>27291009337,
    1200000000=>27591444869, 1212500000=>27892051267, 1225000000=>28192760279,
    1237500000=>28493648629, 1250000000=>28794583627, 1262500000=>29095694269,
    1275000000=>29396966971, 1287500000=>29698366099, 1300000000=>29999858327,
    1312500000=>30301430881, 1325000000=>30603183581, 1337500000=>30905024497,
    1350000000=>31207047449, 1362500000=>31509131153, 1375000000=>31811397571,
    1387500000=>32113702069, 1400000000=>32416190071, 1412500000=>32718790873,
    1425000000=>33021414143, 1437500000=>33324275711, 1450000000=>33627220709,
    1462500000=>33930284893, 1475000000=>34233442279, 1487500000=>34536683891,
    1500000000=>34840062373, 1512500000=>35143545889, 1525000000=>35447088559,
    1537500000=>35750747297, 1550000000=>36054501641, 1562500000=>36358440731,
    1575000000=>36662430631, 1587500000=>36966563321, 1600000000=>37270791697,
    1612500000=>37575137933, 1625000000=>37879532671, 1637500000=>38184009763,
    1650000000=>38488677419, 1662500000=>38793413899, 1675000000=>39098225629,
    1687500000=>39403174463, 1700000000=>39708229123, 1712500000=>40013309359,
    1725000000=>40318523009, 1737500000=>40623800311, 1750000000=>40929166261,
    1762500000=>41234743751, 1775000000=>41540289619, 1787500000=>41845958971,
    1800000000=>42151671491, 1812500000=>42457500313, 1825000000=>42763499629,
    1837500000=>43069571603, 1850000000=>43375710643, 1862500000=>43681898699,
    1875000000=>43988172667, 1887500000=>44294549347, 1900000000=>44601021791,
    1912500000=>44907564593, 1925000000=>45214177441, 1937500000=>45520935011,
    1950000000=>45827700419, 1962500000=>46134655219, 1975000000=>46441643177,
    1987500000=>46748693981, 2000000000=>47055833459, 2012500000=>47363059687
  }
end

def select_pg(num, start_num)   # adaptively select PG
  range = num - start_num
  pg = 5
  if start_num <= Math.sqrt(num).to_i     # for one array of primes upto N
    pg =  7 if num >  50*10**4
    pg = 11 if num > 305*10**5
  else                                    # for split array cases
    pg =  7 if (range.between?(10**6, 10**7-1) && start_num < 10**8)     ||
               (range.between?(10**7, 10**8-1) && start_num < 46*10**8)  ||
               (range.between?(10**8, 10**9-1) && start_num < 16*10**10) ||
               (range >= 10**9 && start_num < 26*10**12)
    pg = 11 if (range.between?(10**8, 10**9-1) && start_num < 55*10**7)  ||
               (range >= 10**9 && start_num < 45*10**9)
  end
  primes = [2,3,5,7,11,13].select! {|p| p <= pg}
  [primes, primes.reduce(:*)]   # [excluded primes, mod] for PG
end
```

```
    def array_check(n,v)  # catch out-of-memory errors on array creation
      Array.new(n,v) rescue return            # return an array or nil
    end

  end
end

class Integer; include Primes::Utils end

puts "Available methods are: #{0.primes_utils}"  # display methods upon loading
```