Ultimate Prime Sieve -- Sieve of Zakiya (SoZ)

by
Jabari Zakiya © 2008
jzakiya@yahoo.com

Introduction

In the first week of May 2008, to help me solve various Project Euler (http://projecteuler.net) problems I began searching for code to generate primes. Thus began a process that culminated in my developing a new class of Number Theory Sieves (NTS) to generate prime numbers, and test primality of numbers, that use minimum memory, are simple to code, and are much faster than all previously known methods.

Based on number theory, the new method employs a very simple and efficient process to generate a set of candidate primes up to a number N, and then uses a similarly simple number theory sieve to remove the non-primes. Both cases can inherently be done in parallel with multiple processors or software. The method will also accommodate a host of prime generator functions of the class described herein. I also used this number theory to then create the simplest, fastest, and deterministic primality tester.

I used Ruby 1.9.0-1 (www.ruby-lang.org) as my software development language, which you will see allows for very simple and expressive idioms for problem solving. Ruby is a pure Object Oriented interpreted language, which allows for easy and dynamic program development. My development environment was an Intel P4 2.8GHz, 1 GB, PCLinuxOS (www.pclinuxos.com) based laptop. All the code presented, however, should run with no problem on any system which has Ruby.

The Beginning

The well known classical method to find primes up to a number N is the Sieve of Eratosthenes (SoE). (See an animation of the process here: www.answers.com/topic/sieve-of-eratosthenes?cat=technology) Using this as my starting point, I did internet searches for Ruby code (because I like programming in it) to see what was out there. Lo and behold, I found a surfeit of code, but I liked the code below the best. Here's the original Ruby code to implement some prime sieves. I've just cosmetically reformatted it. You can find the original here: http://snippets.dzone.com/posts/show/3734

```
class Integer

   def primes1
      sieve = [nil, nil].concat((2..self).to_a)

      (2 .. Math.sqrt(self).floor).each do |i|
         next unless sieve[i]
         (i*i).step(self, i) { |j| sieve[j] = nil }
      end
      sieve.compact!
   end

   def primes2
      sieve = []
      3.step(self, 2) { |i| sieve[i] = i }
      sieve[2] = 2

      3.step(Math.sqrt(self).floor, 2).each do |i|
         next unless sieve[i]
         (i*i).step(self, i) { |j| sieve[j] = nil }
      end
      sieve.compact!
   end
```

```
   def primes3
      # cf. http://betterexplained.com/articles/another-look-at-prime-numbers/ and
      # http://everything2.com/index.pl?node_id=1176369
      n2, i = 0, 0;  sieve = []

      while n2 < self
         n1 = 6*i+1;  n2 = n1+4;  i += 1
         sieve[n1] = n1;  sieve[n2] = n2
      end

      sieve[1] = nil;  sieve[2] = 2;  sieve[3] = 3;  sieve[-1] = nil

      5.step(Math.sqrt(self).floor, 2) do |i|
         next unless sieve[i]
         (i*i).step(self, i) { |j| sieve[j] = nil }
      end
      sieve.compact!
   end
end
```

For non-Rubyist, these routines are implemented as methods inside the class Integer so you can use them like `integer.primes` and they produce as output an array of prime integers. So `100.primes` produces an array of 25 integer elements from [2,3,5,7....93,97]. You can check the number of primes found by doing `integer.primes.size`. Thus, `100.primes.size` should be 25. You can also implement them as functions by defining them as `def primes(num)` and just replace `self` with `num`, and then you can use them like `primes(num)` or more Rubyish, `primes num`.

All three routines are different implementations of the Sieve of Eratosthenes (SoE). There are two components to the algorithm: first an array of candidate primes is created, then the multiples of the first instance of each integer not previously eliminated are identified and eliminated through a sieve process.

Now `primes1` is just the literal brute-force SoE.

```
   def primes1
      sieve = [nil, nil].concat((2..self).to_a)
      (2 .. Math.sqrt(self).floor).each do |i|
         next unless sieve[i]
         (i*i).step(self, i) { |j| sieve[j] = nil }
      end
      sieve.compact!
   end
```

In Ruby `concat` appends to `[ nil, nil]` array the array of integers 2-to-N from `(2..self).to_a`. You can also do this in Ruby as `[ nil, nil] + (2..self).to_a`. This is done so `sieve[ 0] & [ 1} = nil` and then `sieve[ i] =i`. A primitive brute-force sieve (BFS) then begins, which uses the integers from 2-to-sqrt(N), which skips numbers already eliminated. Finally, `sieve.compact!` squeezes all the `nil` values from the array (in place) and leaves only the primes, and returns this result as the output.

Since the only even number prime is 2, we don't have to include the rest as candidate primes. `primes2` uses this fact to reduce the workload by initializing the sieve array with only the odd numbers (the even index elements will be `nil` values in Ruby). It's still a BFS, but only over the odd numbers now.

Note, I've speeded up the original code: `(i*i).step(self, i) { |j| sieve[ j] = nil }`
by replacing it with the following code: `(i*i).step(self, i<<1) { |j| sieve[ j] = nil }`

Since an odd+odd number is even and odd+even is odd, and since we aren't interested in even numbers as candidates, then we only need to step through the sieve process not by i but by 2*i (i << 1). Also note the prime number 2 now has to be explicitly included into the sieve array. Here's my version.

```
def primes2a
   # initialize sieve array with odd integers for odd indices, then 2
   sieve = [];  3.step(self, 2) {|i| sieve[i] = i};  sieve[2] = 2

   3.step(Math.sqrt(self).to_i, 2) do |i|
      next unless sieve[i]
      (i*i).step(self, i<<1) {|j| sieve[j] = nil }
   end
   sieve.compact!
end
```

OK, this is better but still a standard classical SoE. Things got interesting for me with `primes3`.

```
def primes3
   # cf. http://betterexplained.com/articles/another-look-at-prime-numbers/ and
   # http://everything2.com/index.pl?node_id=1176369

   n2, i = 0, 0;  sieve = []
   while n2 < self
      n1 = 6*i+1;  n2 = n1+4;  i += 1
      sieve[n1] = n1;  sieve[n2] = n2
   end

   sieve[1] = nil;  sieve[2] = 2;  sieve[3] = 3;  sieve[-1] = nil

   5.step(Math.sqrt(self).to_i, 2) do |i|
      next unless sieve[i]
      (i*i).step(self, i) { |j| sieve[j] = nil }
   end
   sieve.compact!
 end
```

Well, the code does work, but I didn't know why. I hate that! So now I'm into it. On to the books!

Some number theory

To produce the candidate primes `primes3` did this, which was new to me.

```
   n2, i = 0, 0;  sieve = []
   while n2 < self
      n1 = 6*i+1;  n2 = n1+4;  i += 1
      sieve[n1] = n1;  sieve[n2] = n2
   end
```

The explanation for this below, is from here: http://everything2.com/index.pl?node_id=1176369

All primes greater than 3 are of the form 6k-1 or 6k+1
All integers can be represented as 6k+m, where m ε {0, 1, 2, 3, 4, 5}, and k is some integer. Thus:
m=0: 6k is divisible by 6. Not prime
m=1: 6k+1 has no immediate factors. May be prime.
m=2: 6k+2 = 2 x (3k+1). Not prime
m=3: 6k+3 = 3 x (2k+1). Not prime
m=4: 6k+4 = 2 x (3k+2). Not prime
m=5: 6k+5 has no immediate factors. May be prime.
Therefore, the only candidates for primacy are 6k+1 and 6k+5. 6k+5 = 6m-1 for m=k+1. Therefore, P_3 = 6k+(1,5) all primes are of the form 6n±1 for some integer n.

For nomenclature purposes I'll call generators for primes > n, P_n with.

OK, so how does this math produce a more efficient sieve? For me, I need to see the numbers to see patterns, so I created a table of the generated prime candidates, where n1 = 6k+1, and n2 = 6k+5.

Table 1.

k	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
n1	1	7	13	19	25	31	37	43	49	55	61	67	73	79	85	91	97	103
n2	5	11	17	23	29	35	41	47	53	59	65	71	77	83	89	95	101	107

Look how well this produces prime candidates. In number theory the function (not the number) Pi(N) represents the number of primes <= N, with Pi(100) = 25. From the table we see 6k+(1,5) produces 32 prime candidates for n=100, thus, only 6 prime candidates (25, 49, 55, 77, 85, 95) turn out to be non-prime for n > 3. This is a very high primes/candidates ratio (p/c), much better than the brute force prime candidates generation process. It's even higher for Pi(107), which is the next k value past 100.

After studying the table, and understanding what the numbers were doing, and applying the previous speed-up to the sieving process, I created a slightly more efficient and faster implementation.

```
def primes3a
   # all prime candidates > 3 are of form 6*k+1 and 6*k+5
   # initialize sieve array with only these candidate values
   n2, k = 6, 1;  sieve = []
   while n2 < self
      n1 = 6*k+1;  n2 = n1+4;  k += 1
      sieve[n1] = n1;  sieve[n2] = n2
   end
   # initialize sieve array with the primes < 6, resize array
   sieve[2]=2;  sieve[3]=3;  sieve[5]=5;  sieve=sieve[0..self]

   5.step(Math.sqrt(self).to_i, 2) do |i|
      next unless sieve[i]
      (i*i).step(self, i<<1) { |j| sieve[j] = nil }
   end
   sieve.compact!
 end
```

Up to this point I was just optimizing the original routines coding. Then I started to probe the math to understand what it was actually doing. This is a key point. ***Maximum software optimization comes from finding optimum algorithms, not from optimizing coding***. After studying Table 1 I finally saw how to optimize the prime candidates generation process and then the sieve process. Here's how.

To generate the prime candidates you just need to produce Table 1 up to the number N. So just do it! The code below on the left is the literal algorithm while the code on the right is the table translation.

```
n2, k = 6, 1;  sieve = []
while n2 < self
   n1 = 6*k+1;  n2 = n1+4;  k += 1
   sieve[n1] = n1;  sieve[n2] = n2
end
```

```
n1, n2 = 1, 5;  sieve = []
while n2 < self
   n1 += 6;  n2 += 6
   sieve[n1] = n1;  sieve[n2] = n2
end
```

The table translation only needs 2 variable and 2 simple adds instead of 3 variables and a multiplication and 3 additions needed to perform the literal algorithm. This is not only simpler but also much faster.

Now looking again at Table 1, to perform the sieve we need to eliminate all the non-primes, but first we need a way to identify them. Since $P_3 = 6k+(1,5)$ generates all the primes > 3 lets start looking at all the multiples of 5. From the table we get the pairs (n1=25, k=4), (n2=35, k=5) and the next multiples at (n1=55, k=9), (n2=65, k=10). So for prime p=5 the first instance of multiples of 5 are 5*5 and 7*5 and the distance between subsequent multiples of 5 is 6*5=30. Does this pattern hold for other primes? Well, for p=7 the first multiple is n2=5*7=35, and n1=7*7=49, and the next multiples are a distance of 6*7=42 greater at n2=77 and n1=91. So for p=7 the pairs for n2 are (n2=35, k=5), (n2=77, k=12), etc and for n1 they are (n1=49, k=8), (n=91, k=15), etc.

So do you see the pattern? The first multiples for each prime occur at 5*p or 7*p and the subsequent multiples are just additions of 6*p, which follows the number theory that the primes are of form 6n±1. Also notice from the table, multiples of p have a distance of k=p units, i.e. for p=5 for n1 k=4, 9, 14, etc; for n2 k=5, 10, 15, etc; for p=7 n1 k=8, 15, etc and for n2 k=5, 12, 19, etc. So now we know how to identify all prime multiples, which we then eliminate as prime candidates. The code below does this.

```
5.step(Math.sqrt(self).to_i, 2) do |i|
   next unless sieve[i]
   # p1 = 5*i,  k =6*i;  p2 = 7*i
   p1 = 5*i;  k = p1+i;  p2 = k+i
   while p1 <= self
      sieve[p1] = nil;  sieve[p2] = nil;  p1 += k;  p2 += k
   end
end
```

Now I can combine both the simpler/faster prime candidates generator with the similarly simpler and faster sieve process to produce a totally new NTS (SoZ) prime P_3 generator.

```
def primesP3
   # all prime candidates > 3 are of form 6*k+1 and 6*k+5
   # initialize sieve array with only these candidate values
   n1, n2 = 1, 5;  sieve = []
   while n2 < self
      n1 += 6;  n2 += 6;  sieve[n1] = n1;  sieve[n2] = n2
   end
   # initialize sieve array with the primes < 6, resize array
   sieve[2]=2;  sieve[3]=3;  sieve[5]=5;  sieve=sieve[0..self]

   5.step(Math.sqrt(self).to_i, 2) do |i|
      next unless sieve[i]
      # p1 = 5*i,  k =6*i,  p2 = 7*i
      p1 = 5*i;  k = p1+i;  p2 = k+i
      while p1 <= self
         sieve[p1] = nil;  sieve[p2] = nil;  p1 += k;  p2 += k
      end
   end
   sieve.compact!
end
```

The reason I need to do `sieve=sieve[0..self]` is because at least one prime candidate will extend beyond the index value N, so I resize the array to get rid of any of these. Since arrays in Ruby are dynamic I can resize them anytime I want, but for languages which require you to pre-size arrays just initialize the array to size N+6 if it squawks about storing past the end of the array. Now the sieve process will store past index value N too, but it's just add `nils`. That's ok, because `sieve.compact!` squeezes all the `nils` from the array, leaving just the primes.

Benchmarking verified this new SoZ implementation is significantly faster than all previous methods. Also, the algorithm, and its Ruby coding, is just so elegant in its simplicity and ease of understanding. You will see later how much you will come to appreciate this.

However, `primesP3` uses an array length equal to N. That's easy to reduce. `primesP3a` below uses only N/2 array elements and is only slightly slower only because the block `.map {|i| (i<<1)+3}` has to be performed to reconvert the numbers to their natural values. If you want though, you can forego this step, store the result, and do the reconversion later, if it makes more sense for a given application.

```
def primesP3a
   # all prime candidates > 3 are of form 6*k+1 and 6*k+5
   # initialize sieve array with only these candidate values
   # where sieve contains the odd integers representations
   # convert integers to array indices/vals by i = (n-3)>>1 = (n>>1)-1
   n1, n2 = -1, 1;  lndx = (self-1)>>1;  sieve = []
   while n2 < lndx
      n1 += 3;  n2 += 3;  sieve[n1] = n1;  sieve[n2] = n2
   end
   # now initialize sieve array with (odd) primes < 6, resize array
   sieve[0]=0;  sieve[1]=1;  sieve=sieve[0..lndx-1]

   5.step(Math.sqrt(self).to_i, 2) do |i|
      next unless sieve[(i>>)-1]
      # p1 = 5*i,  k = 6*i;  p2 = 7*i
      i6 = 6*i;  p1 = (i6-i-3)>>1;  p2 = (i6+i-3)>>1;  k = i6>>1
      while p1 < lndx
         sieve[p1] = nil;  sieve[p2] = nil;  p1 += k;  p2 += k;
      end
   end
   return [2] if self < 3
   [2]+([nil]+sieve).compact!.map {|i| (i<<1)+3 }
end
```

Some more number theory

Previously it was shown that P_3 = 6k+(1,5) produces all possible prime candidate integers > 3. But let's understand this more thoroughly. The first question to ask is where does the 6 come from? Well 6 is the product of the first two primes 2 and 3. Thus, any number divisible by 6 (i.e. n mod 6 = 0) is divisible by 2 and 3. So 6 is the modulus for P_3, modP_3 = 6. Now you can potentially produce a prime number by multiplying all the prime numbers from 2 up to some prime p, and then add 1 to it. So, (2*3)+1 = 7 is prime. This is where the 6k+1 comes from, which produces prime candidates for all k. Thus, 6*2+1=13, 6*3+1=19, 6*4+1=24, 6*5+1=31, etc, where 13, 19 and 31 are, in fact, prime.

But what about the 5? Here's an easy way to think about it which will help us later. The 5 is the single number (in this case) between 3 and 6 (modP_3) which isn't divisible by 2 or 3. Again, this should be obvious. So in this case, 5 is the single residue for P_3. Now knowing this general structure, we can extend the above process to create even more efficient prime candidates generators.

So if P_3 = 6k+(1,5) generates all the prime candidates > 3 can we find generators for p > 5? Yes!

Let's do the next one, P_5. P!(5)= modP_5 = 2*3*5 = 30. Now we need to find all the residue values, i.e. all the primes 5 < p < 30, which are 7,11,13,17,19,23,29. Thus, P_5 = 30k+ (1, 7, 11, 13, 17, 19, 23, 29).

This means to be prime, a number n > 5 must have n mod 30 ε {1, 7, 11, 13, 17, 19, 23, 29}, if not, it isn't prime. Alternatively, to be prime n + ε {1, 7, 11, 13, 17, 19, 23, 29} mod 30 must = 0. That's it!

Again, let's create a table to see how this works.

Table 2.

k	n1	n2	n3	n4	n5	n6	n7	n8
0	1	7	11	13	17	19	23	29
1	31	37	41	43	47	49	53	59
2	61	67	71	73	77	79	83	89
3	91	97	101	103	107	109	113	119
4	121	127	131	133	137	139	143	149
5	151	157	161	163	167	169	173	179
6	181	187	191	193	197	199	203	209
7	211	217	221	223	227	229	233	239
8	241	247	251	253	257	259	263	269
9	271	277	281	283	287	289	293	299
10	301	307	311	313	317	319	323	329
11	331	337	341	343	347	349	353	359
12	361	367	371	373	377	379	383	389
13	391	397	401	403	407	409	413	419
14	421	427	431	433	437	439	443	449

A review of this will shows for k=3 all the prime candidates < 120 will be produced with the p/c ratio being 28/32. Compare this to P_3 where k=16 to generate the primes < 102 and the p/c ratio is 25/32. So we're selecting more primes, with fewer non-primes candidates, more efficiently, which would make you think this should be a faster prime generator. Here's the Ruby code.

```
def primesP5
   # all prime candidates > 5 are of form  30*k+(1,7,11,13,17,19,23,29)
   # initialize sieve array with only these candidate values
   n1, n2, n3, n4, n5, n6, n7, n8 = 1, 7, 11, 13, 17, 19, 23, 29;  sieve = []
   while n8 < self
      n1 +=30;   n2 += 30;   n3 += 30;   n4 += 30
      n5 +=30;   n6 += 30;   n7 += 30;   n8 += 30
      sieve[n1] = n1;  sieve[n2] = n2;  sieve[n3] = n3;  sieve[n4] = n4
      sieve[n5] = n5;  sieve[n6] = n6;  sieve[n7] = n7;  sieve[n8] = n8
   end
   # now initialize sieve with the primes < 30, resize array
   sieve[2]=2; sieve[3]=3; sieve[5]=5; sieve[7]=7; sieve[11]=11; sieve[13]=13
   sieve[17]=17; sieve[19]=19; sieve[23]=23; sieve[29]=29; sieve=sieve[0..self]

   7.step(Math.sqrt(self).to_i, 2) do |i|
     next unless sieve[i]
     # p1=7*i, p2=11*i, p3=13*i --- p6=23*i, p7=29*i, p8=31*i,  k=30*i
     n6 = 6*i;    p1 = i+n6;   p2 = n6+n6-i; p3 = p1+n6;  p4 = p2+n6
     p5 = p3+n6;  p6 = p4+n6;  p7 = p6+n6;   k = p7+i;    p8 = k+i
     while p1 <= self
        sieve[p1] = nil;  sieve[p2] = nil;  sieve[p3] = nil;  sieve[p4] = nil
        sieve[p5] = nil;  sieve[p6] = nil;  sieve[p7] = nil;  sieve[p8] = nil
        p1 += k; p2 += k; p3 += k; p4 += k; p5 += k; p6 += k; p7 += k ; p8 += k
```

```
      end
    end
    sieve.compact!
  end
```

And here's the reduced memory version.

```
  def primesP5a
    # all prime candidates > 5 are of form  30*k+(1,7,11,13,17,19,23,29)
    # initialize sieve array with only these candidate values
    # where sieve contains the odd integers representations
    # convert integers to array indices/vals by  i = (n-3)>>1 = (n>>1)-1
    n1, n2, n3, n4, n5, n6, n7, n8 = -1, 2, 4, 5, 7, 8, 10, 13
    lndx= (self-1) >>1;  sieve = []
    while n8 < lndx
      n1 +=15;   n2 += 15;   n3 += 15;   n4 += 15
      n5 +=15;   n6 += 15;   n7 += 15;   n8 += 15
      sieve[n1] = n1;  sieve[n2] = n2;  sieve[n3] = n3;  sieve[n4] = n4
      sieve[n5] = n5;  sieve[n6] = n6;  sieve[n7] = n7;  sieve[n8] = n8
    end
    # now initialize sieve with the (odd) primes < 30, resize array
    sieve[0]=0;  sieve[1]=1;  sieve[2]=2;  sieve[4]=4;  sieve[5]=5
    sieve[7]=7;  sieve[8]=8;  sieve[10]=10;  sieve[13]=13
    sieve = sieve[0..lndx-1]

    7.step(Math.sqrt(self).to_i, 2) do |i|
      next unless sieve[(i>>1)-1]
      # p1=7*i, p2=11*i, p3=13*i ---- p6=23*i, p7=29*i, p8=31*i,  k=30*i
      # maps to: p1= (7*i-3)>>1, --- p8=(31*i-3)>>1, k=30*i>>1
      n2=i<<1;  n4=i<<2;  n8=i<<3;  n16=i<<4;  n32=i<<5;  n12=n8+n4
      p1 = (n8-i-3)>>1;  p2 = (n12-i-3)>>1;  p3 = (n12+i-3)>>1;  p4 = (n16+i-3)>>1
      p5 = (n16+n4-i-3)>>1;  p6 = (n16+n8-i-3)>>1;  p7 = (n32-n4+i-3)>>1
      p8 = (n32-i-3)>>1;  k = (n32-n2)>>1
      while p1 < lndx
         sieve[p1] = nil;  sieve[p2] = nil;  sieve[p3] = nil;  sieve[p4] = nil
         sieve[p5] = nil;  sieve[p6] = nil;  sieve[p7] = nil;  sieve[p8] = nil
         p1 += k; p2 += k; p3 += k; p4 += k; p5 += k; p6 += k; p7 += k; p8 += k
      end
    end
    return [2] if self < 3
    [2]+([nil]+sieve).compact!.map {|i| (i<<1) +3 }
  end
```

These work and are faster than the P_3 versions! So can I extend this method to create a general class of prime generators up to (theoretically) any *Pn* value? Well, it's easy enough to write the Ruby code, so I did, and it didn't work. Well, it worked only for P_3 and P_5 but not for higher order generators. I tried and I tried, but once I was satisfied it wasn't coding errors, I accepted there was something wrong with my theory for extending my methodology. Ok, time to *REALLY* study the math now.

A time to digress

Early during my internet searching I found this blog site http://krenzel.info/?p=83 which introduced me to the Sieve of Atkin, which it said was supposed to be the fastest known method to generate primes. Here's the authors Python code to do it: http://krenzel.info/static/atkin.py Here's the link to the paper "Prime Sieves Using Binary Quadratic Forms" by A.O.L. Atkin and Daniel J. Bernstein (of qmail, et al, fame) where you can read and download the pdf of the paper.

http://www.ams.org/mcom/2004-73-246/S0025-5718-03-01501-1/S0025-5718-03-01501-1.pdf

So I downloaded and printed out the paper and the Python code. But after briefly looking at both, I quickly determined I wasn't inspired enough to devote the time (or number of brain cells) that I would need to really understand what was going on.

However, after finishing benchmarking my sieve algorithms, and seeing how much faster they were to the Ruby sieves code I downloaded, I wanted to compare them to the Sieve of Atkin (SoA). The easiest way to do this with the least amount of work was to code mine in Python and run the benchmarks included with that code. Here's the Python code for the SoA, SoE, and my P_3 and P_5 Python versions.

```
from random import randint
from math import sqrt, ceil, floor, log
import time
import sys

def sieveOfAtkin(end):
    end += 1
    lng = ((end/2)-1+end%2)
    sieve = [False]*(lng + 1)

    x_max, x2, xd = int(sqrt((end-1)/4.0)), 0, 4
    for xd in range(4, 8*x_max + 2, 8):
        x2 += xd
        y_max = int(sqrt(end-x2))
        n, n_diff = x2 + y_max**2, (y_max << 1) - 1
        if n%2 == 0:
            n -= n_diff
            n_diff -= 2
        for d in range((n_diff - 1) << 1, -1, -8):
            m = n%12
            if (m == 1 or m == 5):
                    m = n >> 1
                    sieve[m] = not sieve[m]
            n -= d

    x_max, x2, xd = int(sqrt((end-1)/3.0)), 0, 3
    for xd in range(3, 6*x_max + 2, 6):
        x2 += xd
        y_max = int(sqrt(end-x2))
        n, n_diff = x2 + y_max**2, (y_max << 1) - 1
        if n%2 == 0:
            n -= n_diff
            n_diff -= 2
        for d in range((n_diff - 1) << 1, -1, -8):
            if (n%12 == 7):
                    m = n >> 1
                    sieve[m] = not sieve[m]
            n -= d

    x_max, y_min, x2, xd = int((2 + sqrt(4-8*(1-end)))/4), -1, 0, 3
    for x in range(1, x_max + 1):
        x2 += xd
        xd += 6
        if x2 >= end: y_min = (((int(ceil(sqrt(x2 - end))) - 1) << 1) - 2) << 1
        n, n_diff = ((x**2 + x) << 1) - 1, (((x-1) << 1) - 2) << 1
        for d in range(n_diff, y_min, -8):
            if (n%12 == 11):
                    m = n >> 1
                    sieve[m] = not sieve[m]
            n += d

    primes = [2,3]
```

```
    if end <= 3 : return primes[:max(0,end-2)]

    for n in range(5 >> 1, (int(sqrt(end))+1) >> 1):
        if sieve[n]:
            primes.append((n << 1) + 1)
            for k in range(((n << 1) + 1)**2, end, 2*((n << 1) + 1)**2):
                sieve[k >> 1] = False

    s  = int(sqrt(end)) + 1
    if s%2 == 0: s += 1
    primes.extend([ i for i in range(s, end, 2) if sieve[i >> 1]])

    return primes


def sieveOfErat(end):
    if end < 2: return []

    #The array doesn't need to include even numbers
    lng = ((end/2)-1+end%2)

    # Create array and assume all numbers in array are prime
    sieve = [True]*(lng+1)

    # In the following code, you're going to see some funky
    # bit shifting and stuff, this is just transforming i and j
    # so that they represent the proper elements in the array

    # Only go up to square root of the end
    for i in range(int(sqrt(end)) >> 1):

        # Skip numbers that aren't marked as prime
        if not sieve[i]: continue

        # Unmark all multiples of i, starting at i**2
        for j in range( (i*(i + 3) << 1) + 3, lng, (i << 1) + 3):
            sieve[j] = False

    # Don't forget 2!
    primes = [2]

    # Gather all the primes into a list, leaving out the composite numbers
    primes.extend([(i << 1) + 3 for i in range(lng) if sieve[i]])

    return primes

def primesP3(val):
    # all prime candidates > 3 are of form  6*k+(1,5)
    # initialize sieve array with only these candidate values
    n1, n2 = 1, 5
    sieve = [False]*(val+6)
    while  n2 < val:
        n1 += 6;   n2 += 6;  sieve[n1] = n1;   sieve[n2] = n2
    # now load sieve with seed primes 3 < pi < 6, in this case just 5
    sieve[5] = 5

    for i in range( 5, int(ceil(sqrt(val))), 2) :
       if not sieve[i]:  continue
       #  p1= 5*i,  k = 6*i, p2 = 7*i,
       p1 = 5*i;  k = p1+i;  p2 = k+i
       while p2 <= val:
          sieve[p1] = False;  sieve[p2] = False;  p1 += k;  p2 += k
```

```
       if p1 <= val:  sieve[p1] = False

    primes = [2,3]
    if val < 3 : return [2]
    primes.extend( i for i in range(5, val+(val&1), 2)  if sieve[i] )

    return primes

def primesP5(val):
    # all prime candidates > 5 are of form  30*k+(1,7,11,13,17,19,23,29)
    # initialize sieve array with only these candidate values
    n1, n2, n3, n4, n5, n6, n7, n8 = 1, 7, 11, 13, 17, 19, 23, 29
    sieve = [False]*(val+30)
    while  n8 < val:
        n1 +=30;   n2 += 30;   n3 += 30;   n4 += 30
        n5 +=30;   n6 += 30;   n7 += 30;   n8 += 30
        sieve[n1] = n1;  sieve[n2] = n2;  sieve[n3] = n3;  sieve[n4] = n4
        sieve[n5] = n5;  sieve[n6] = n6;  sieve[n7] = n7;  sieve[n8] = n8
    # now load sieve with seed primes 5 < pi < 30
    sieve[7]  = 7;   sieve[11] = 11;  sieve[13] = 13;  sieve[17] = 17
    sieve[19] = 19;  sieve[23] = 23;  sieve[29] = 29

    for i in range( 5, int(ceil(sqrt(val))), 2) :
       if not sieve[i]:  continue
       # p1=7*i, p2=11*i, p3=13*i ---- p6=23*i, p7=29*i, p8=31*i,  k=30*i
       n6 = 6*i;    p1 = i+n6;   p2 = n6+n6-i;  p3 = p1+n6;  p4 = p2+n6
       p5 = p3+n6;  p6 = p4+n6;  p7 = p6+n6;    k  = p7+i;   p8 = k+i
       while p8 <= val:
          sieve[p1] = False;  sieve[p2] = False;  sieve[p3] = False;  sieve[p4] = False
          sieve[p5] = False;  sieve[p6] = False;  sieve[p7] = False;  sieve[p8] = False
          p1 += k; p2 += k; p3 += k; p4 += k; p5 += k; p6 += k; p7 += k; p8 += k

       if p7 <= val:  sieve[p7] = False
       if p6 <= val:  sieve[p6] = False
       if p5 <= val:  sieve[p5] = False
       if p4 <= val:  sieve[p4] = False
       if p3 <= val:  sieve[p3] = False
       if p2 <= val:  sieve[p2] = False
       if p1 <= val:  sieve[p1] = False

    primes = [2,3,5]
    if val < 3 : return [2]
    if val < 5 : return [2,3]
    if val < 7 : return [2,3,5]
    primes.extend( i for i in range(7, val+(val&1), 2)  if sieve[i] )

    return primes

print  sieveOfErat(1000)
print  sieveOfAtkin(1000)
print  primesP3(1000)
print  primesP5(1000)

for j in range(5,100000006, 1000000):

        print j,

        #a = time.time()
        #soe = sieveOfErat(j)
        #print time.time() -a,

        a = time.time()
```

```
    soa = sieveOfAtkin(j)
    print time.time() -a,


    a = time.time()
    soz1 = primesP3(j)
    print time.time() -a,


    a = time.time()
    soz2 = primesP3(j)
    print time.time() -a

  #if soa == soe:  print 'soa = soe for ', j
  if soa == soz1:  print 'soa = soz1 for', j
  if soa == soz2:  print 'soa = soz2 for', j
```

I ran this both under Python 2.4.3 and 2.5.2, and both my routines are faster than the SoA, and get progressively faster as the number N gets bigger. [A surprising programming note: my benchmarks run with Ruby 1.9.0-1 were significantly faster than with either Python version. The standard mantra is Python is faster than Ruby. Well maybe for Ruby 1.8. and older, but certainly not for Ruby 1.9.]

I eventually translated the Python SoA into Ruby, but my SoZ versions were faster than it too. Now, I felt very confident I really had found the fastest way to generate primes. But I still felt I should be able to generalize my method to work (theoretically) for any P_n.

Toward the Eureka insite

In the process of writing this paper, which took almost 4 weeks, I had to review all my prior work, and really tried to grok its meaning. And in order to explain it to others so they could understand it, I first had to. In the process I learned a lot more than I had originally, but now it made much more sense. It really came together when I went back to see why my general methodology didn't completely work.

My original general method, following the process I used to construct P_3 and P_5, started missing primes in this order: 331, 353, 379, 419, 541, 563, 607, 751, 773, 839... Well, subtracting 210 from each I got this sequence: 121, 143, 169, 209, 331, 353, 397, 441, 363, 629... Well I took these numbers and found the factors of each, but I couldn't see a general pattern to get the general method code to work.

At this point, I finally started to read the Atkin-Bernstein paper, and lo and behold, I saw the following.

The paper starts by identifying the prime P_5 generator, which I had independently hypothesized and written the code to implement. As a reminder, here it is again: P_5 = 30k+ (1, 7, 11, 13, 17, 19, 23, 29). But then they identify a class of three separate prime generators which form the basis of the SoA. These generators are, using their nomenclature: 1) 60k + n, n ε {1, 13, 17, 29, 37, ,41, 49, 53 } and 2) 60k + n, n ε {1, 7, 13, 19, 31, 37, 43, 49 } and 3) 60k + n, n ε {11, 23, 47, 59 }

Here is where thinking like an engineer benefited me. If you look at the SoA code, it treats and implements each function separately as a prime generator. But if you look at the 3 functions together what do you see? You see a common modulus of 60 with the common combined residues of (1,7,11,13,17,19,23,29,31,37,41,43,47,49,53,59). These residues are of the form 5 < p < 60, except that 49 isn't a prime. But hey, if it works for the SoA it had to work using my method. Right?

So I implemented the SoA generator using my methodology, and of course it works, and it's faster than the SoA. However, it's actually a tad slower than my P_5 code. This makes perfect sense because even though the SoA modulus is twice is big as the P_5 modulus it uses twice as many residues per pass (16 vs 8), so it still performs essentially the same number of operations. So the P_5 code is "better" by being slightly faster, and simpler. Why do more work to get essentially the same performance?

But even though I could implement the SoA generators using my NTS methodology the one hanging question that I didn't understand about the SoA generator was where did the 49 residue come from? At first I didn't care. I wrote the code and it worked. But then as I was writing this paper it finally all came together, and I was able to finally figure out how to construct a general method to perform P_n.

I wrote this paper in chunks, i.e. not in chronological order. I wrote most of the end first, and then went back in filled in the story of how I developed all the code. But one of the first sections I wrote was "Some more number theory." I wrote this section before I actually seriously read the SoA paper and before I wrote my SoZ implementation of it.

I went back to this section to try to figure out where that 49 had to come from. Somehow I knew my general method had to work, because it was only missing primes, not adding non-primes. But it wasn't intuitive where the 49 came from. So as I was reading this section over again, one thing stuck out in my head that lead me to that eureka insite that made everything fit together.

In explaining how to construct my generators I said the modulus is the product of consecutive primes from 2-to-n and the residues were the primes $n < p < \mod P_n$.. But 60 didn't fit my modulus model and 49 wasn't a prime, yet that generator worked, so obviously my theory was only identifying a subset of possible generators.

But then I looked at the requirement that for a number N to be prime: 1) N mod (modP_n) had to create one of the residues, ***OR*** 2) (N + residue) mod (modP_n) = 0. So I looked at P_3 = 6k+ (1, 5), and what do you see? Well the residues 1+5 = 6, which satisfies condition 2) for both residues. Then I looked at P_5 = 30k+ (1, 7, 11, 13, 17, 19, 23, 29). What do you see? (1+29) = (7+23) = (11+19) = (13+17) = 30. And for the SoA: 60k + (1, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 49, 53, 59) you get (1+59) = (7+53) = (11+49) = (13+47) = (17+43) = (19+31) = (23+37) = (29+31) = 60 = modP_{SoA}

So going back to my P_7, remember I said that my original constructor was missing primes, starting in this order: 331, 353, 379, 419, 541, 563, 607, 751, 773, 839... Well, if I now subtract multiples of 210 I get this: 121, 143, 169, 209, 121, 143, 187, 121, 143, 209... So there they are, my missing residues for P_7 -- (121, 143, 169, 187, 209). It all started to harmonize. Now I can see, said the blind man.

Here's what's going on. My original constructor was P_7 = 210k + (1, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103, 107, 109, 113, 117, 131, 137, 139, 149, 151,157, 163, 167, 173, 179, 181, 191, 193, 197, 199) where the residues were the 42 primes between 7 and 210. But including the 1 in the residue list creates and odd number of residues, but we need an even number and they must add up in pairs to be the 210. Well 121, 143, 169, 187, and 209 when included gives me an even number of residues, which all create pairwise sums of 210. Thus, the residues must come in what I'll call *Modular Compliment Pairs* (MCP) which satisfies (res[i]+res[j]) mod 210 =0.

But there is one other condition that must be satisfied, the (modulus+1) must be prime. This is always true from my P! mod constructor, but 60=2*2*3*5 works because 60+1 = 61 is prime and all the other primes > 5 share no common factors with 60. [PS: I find later these conjectures aren't strictly true.]

Feeling very confident that I now knew what was really going on, I added the missing residues to my original P_7 code, and it worked, and it was faster than P_5 too. Here's the minimum memory version.

```
def primesP7a
   # all prime candidates > 7 are of form  210*k+(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
   # 131,137,139,143,149,151,157,163,167,173,179,181,187,191,193,197,199,209)
   # initialize sieve array with only these candidate values
   # where sieve contains the odd integers representatives
   # convert integers to array indices/vals by  i = (n )>>1 = (n>>1)-1
   n1, n2, n3, n4, n5, n6, n7, n8, n9, n10 = -1, 4, 5, 7, 8, 10, 13, 14, 17, 19
   n11, n12, n13, n14, n15, n16, n17, n18 = 20, 22, 25, 28, 29, 32, 34, 35
```

```
n19, n20, n21, n22, n23, n24, n25, n26 = 38, 40, 43, 47, 49, 50, 52, 53
n27, n28, n29, n30, n31, n32, n33, n34 = 55, 62, 64, 67, 68, 73, 74, 77
n35, n36, n37, n38, n39, n40, n41, n42 = 80, 82, 85, 88, 89, 94, 95, 97
n43, n44, n45, n46, n47, n48 = 98, 59, 70, 83, 92, 103
lndx= (num-1)>>1;  sieve = []
while n48 < lndx
  n1  += 105;  n2  += 105;  n3  += 105;  n4  += 105;  n5  += 105
  n6  += 105;  n7  += 105;  n8  += 105;  n9  += 105;  n10 += 105
  n11 += 105;  n12 += 105;  n13 += 105;  n14 += 105;  n15 += 105
  n16 += 105;  n17 += 105;  n18 += 105;  n19 += 105;  n20 += 105
  n21 += 105;  n22 += 105;  n23 += 105;  n24 += 105;  n25 += 105
  n26 += 105;  n27 += 105;  n28 += 105;  n29 += 105;  n30 += 105
  n31 += 105;  n32 += 105;  n33 += 105;  n34 += 105;  n35 += 105
  n36 += 105;  n37 += 105;  n38 += 105;  n39 += 105;  n40 += 105
  n41 += 105;  n42 += 105;  n43 += 105;  n44 += 105;  n45 += 105
  n46 += 105;  n47 += 105;  n48 += 105
  sieve[n1]  = n1;   sieve[n2]  = n2;   sieve[n3]  = n3;   sieve[n4]  = n4
  sieve[n5]  = n5;   sieve[n6]  = n6;   sieve[n7]  = n7;   sieve[n8]  = n8
  sieve[n9]  = n9;   sieve[n10] = n10;  sieve[n11] = n11;  sieve[n12] = n12
  sieve[n13] = n13;  sieve[n14] = n14;  sieve[n15] = n15;  sieve[n16] = n16
  sieve[n17] = n17;  sieve[n18] = n18;  sieve[n19] = n19;  sieve[n20] = n20
  sieve[n21] = n21;  sieve[n22] = n22;  sieve[n23] = n23;  sieve[n24] = n24
  sieve[n25] = n25;  sieve[n26] = n26;  sieve[n27] = n27;  sieve[n28] = n28
  sieve[n29] = n29;  sieve[n30] = n30;  sieve[n31] = n31;  sieve[n32] = n32
  sieve[n33] = n33;  sieve[n34] = n34;  sieve[n35] = n35;  sieve[n36] = n36
  sieve[n37] = n37;  sieve[n38] = n38;  sieve[n39] = n39;  sieve[n40] = n40
  sieve[n41] = n41;  sieve[n42] = n42;  sieve[n43] = n43;  sieve[n44] = n44
  sieve[n45] = n45;  sieve[n46] = n46;  sieve[n47] = n47;  sieve[n48] = n48
end
# now initialize sieve with the (odd) primes < 210, resize array
sieve[0]=0;   sieve[1]=1;   sieve[2]=2;   sieve[4]=4;   sieve[5]=5
sieve[7]=7;   sieve[8]=8;   sieve[10]=10; sieve[13]=13; sieve[14]=14
sieve[17]=17; sieve[19]=19; sieve[20]=20; sieve[22]=22; sieve[25]=25
sieve[28]=28; sieve[29]=29; sieve[32]=32; sieve[34]=34; sieve[35]=35
sieve[38]=38; sieve[40]=40; sieve[43]=43; sieve[47]=47; sieve[49]=49
sieve[50]=50; sieve[52]=52; sieve[53]=53; sieve[55]=55; sieve[62]=62
sieve[64]=64; sieve[67]=67; sieve[68]=68; sieve[73]=73; sieve[74]=74
sieve[77]=77; sieve[80]=80; sieve[82]=82; sieve[85]=85; sieve[88]=88
sieve[89]=89; sieve[94]=94; sieve[95]=95; sieve[97]=97; sieve[98]=98;
sieve = sieve[0..lndx-1]

11.step(Math.sqrt(self).to_i, 2) do |i|
  next unless sieve[(i>>1)-1]
  # p1=11*i, p2=13*i, p3=17*i --- p46=197*i, p47=199*i, p48=209*i,  k=210*i
  # maps to: p1= (11*i-3)>>1, --- p48=(209*i-3)>>1, k=210*i>>1
  p1  = (11*i-3)>>1;  p2  = (13*i-3)>>1;  p3  = (17*i-3)>>1;  p4  = (19*i-3)>>1
  p5  = (23*i-3)>>1;  p6  = (29*i-3)>>1;  p7  = (31*i-3)>>1;  p8  = (37*i-3)>>1
  p9  = (41*i-3)>>1;  p10 = (43*i-3)>>1;  p11 = (47*i-3)>>1;  p12 = (53*i-3)>>1
  p13 = (59*i-3)>>1;  p14 = (61*i-3)>>1;  p15 = (67*i-3)>>1;  p16 = (71*i-3)>>1
  p17 = (73*i-3)>>1;  p18 = (79*i-3)>>1;  p19 = (83*i-3)>>1;  p20 = (89*i-3)>>1
  p21 = (97*i-3)>>1;  p22 = (101*i-3)>>1; p23 = (103*i-3)>>1; p24 = (107*i-3)>>1
  p25 = (109*i-3)>>1; p26 = (113*i-3)>>1; p27 = (127*i-3)>>1; p28 = (131*i-3)>>1
  p29 = (137*i-3)>>1; p30 = (139*i-3)>>1; p31 = (149*i-3)>>1; p32 = (151*i-3)>>1
  p33 = (157*i-3)>>1; p34 = (163*i-3)>>1; p35 = (167*i-3)>>1; p36 = (173*i-3)>>1
  p37 = (179*i-3)>>1; p38 = (181*i-3)>>1; p39 = (191*i-3)>>1; p40 = (193*i-3)>>1
  p41 = (197*i-3)>>1; p42 = (199*i-3)>>1; p43 = (211*i-3)>>1; p44 = (121*i-3)>>1
  p45 = (143*i-3)>>1; p46 = (169*i-3)>>1; p47 = (187*i-3)>>1; p48 = (209*i-3)>>1
  k = (210*i)>>1
  while p1 < lndx
      sieve[p1]  = nil;  sieve[p2]  = nil;  sieve[p3]  = nil;  sieve[p4]  = nil
      sieve[p5]  = nil;  sieve[p6]  = nil;  sieve[p7]  = nil;  sieve[p8]  = nil
      sieve[p9]  = nil;  sieve[p10] = nil;  sieve[p11] = nil;  sieve[p12] = nil
```

```
          sieve[p13] = nil;  sieve[p14] = nil;  sieve[p15] = nil;  sieve[p16] = nil
          sieve[p17] = nil;  sieve[p18] = nil;  sieve[p19] = nil;  sieve[p20] = nil
          sieve[p21] = nil;  sieve[p22] = nil;  sieve[p23] = nil;  sieve[p24] = nil
          sieve[p25] = nil;  sieve[p26] = nil;  sieve[p27] = nil;  sieve[p28] = nil
          sieve[p29] = nil;  sieve[p30] = nil;  sieve[p31] = nil;  sieve[p32] = nil
          sieve[p33] = nil;  sieve[p34] = nil;  sieve[p35] = nil;  sieve[p36] = nil
          sieve[p37] = nil;  sieve[p38] = nil;  sieve[p39] = nil;  sieve[p40] = nil
          sieve[p41] = nil;  sieve[p42] = nil;  sieve[p43] = nil;  sieve[p44] = nil
          sieve[p45] = nil;  sieve[p46] = nil;  sieve[p47] = nil;  sieve[p48] = nil
          p1 += k; p2 += k; p3 += k; p4  += k; p5 += k; p6 += k; p7 += k
          p8 += k; p9 += k; p10 +=k; p11 += k; p12 +=k; p13 +=k; p14 +=k
          p15 +=k; p16 +=k; p17 +=k; p18 += k; p19 +=k; p20 +=k; p21 +=k
          p22 +=k; p23 +=k; p24 +=k; p25 += k; p26 +=k; p27 +=k; p28 +=k
          p29 +=k; p30 +=k; p31 +=k; p32 += k; p33 +=k; p34 +=k; p35 +=k
          p36 +=k; p37 +=k; p38 +=k; p39 += k; p40 +=k; p41 +=k; p42 +=k
          p43 +=k; p44 +=k; p45 +=k; p46 += k; p47 +=k; p48 +=k
      end
    end
    return [2] if self < 3
    [2]+([nil]+sieve).compact!.map {|i| (i<<1) +3 }
  end
```

Toward the Ultimate Prime Sieve

Now I believe I have the necessary theoretical framework to code the generalized ultimate prime sieve. Here's the code, where the n in Pn now stands for a prime (odd) number or modulus value (even).

```
  def primenePn(input=nil)
    # generalized Sieve of Zakiya, takes an input prime or modulus
    # all prime candidates > Pn are of form modPn*k+[1, res(Pn)]
    # initialize sieve array with only these candidate values
    # where sieve contains the odd integers representations
    # convert integers to array indices/vals by  i = (n-3)>>1 = (n>>1)-1

    # if no input value then just perform SoZ P7a as reference sieve
    return  primesP7a  if  input == nil
    seeds = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61]

    # if input is a prime (odd), determine modulus = P!(n) and seed primes
    if input&1 == 1
       # find seed primes <= Pn, compute modPn
       return 'INVALID OR TOO LARGE PRIME' if !seeds.include? input
       seeds = seeds[0 .. seeds.index(input)];  mod = seeds.inject {|a,b| a*b}
    else  # if input a modulus (even number), determine seed primes for it
       md, pp, mod = 2, 3, input
       seeds[1..-1].each {|i| md *=i;  break if mod < md;  pp = i}
       seeds = seeds[0 .. seeds.index(pp)]
    end

    # create array of prime residues for primes Pn < pk < modPn
    primes = mod.primesP7a;    residues  = primes - seeds

    # find modular compliments of primes [1, Pn < p < modPn] if necessary
    mcp = [];  tmp = [1]+residues
    while !tmp.empty?; b = mod - tmp.shift; mcp << b unless tmp.delete(b) end

    residues.concat(mcp).sort!

    # now initialize modPn, lndx and sieve then find prime candidates
    # set initial prime candidates to (converted) residue values
    lndx= (self-1)>>1;   m=mod>>1;   sieve = []
    pcn = [-1]+residues.map {|t| (t>>1)-1}
```

```
    while pcn.last < lndx
       pcn.each_index {|j| n = pcn[j]+m;  sieve[n] = n;  pcn[j] = n }
    end
    # now initialize sieve with the (odd) primes < modPn, resize array
    primes[1..-1].each {|x| p =(x-3)>>1;  sieve[p] = p }; sieve = sieve[0..lndx-1]

    # perform sieve with residue elements resPn+(modPn+1)
    residues << (mod + 1);   res0 = residues[0]
    res0.step(Math.sqrt(self).to_i, 2) do |i|
       next unless sieve[(i>>1)-1]
       # p1=res1*i, p2=res2*i, p3=res3*i -- pN-1=res(N-1)*i, pN=resN*i, k=mod*i
       # maps to: p1= (res1*i-3)>>1, --- pN=(((mod+1)-3)*i)>>1, k=mod*i>>1
       res = residues.map{|prm| (prm*i-3)>>1};   k = (mod*i)>>1
       while res[0] < lndx
          res.each_index {|j| sieve[res[j]] = nil;  res[j] += k }
    end
    return [2] if self < 3
    [2]+([nil]+sieve).compact!.map {|i| (i<<1) +3 }
end
```

Let me explain the general NTS method, and highlight some of the Rubyisms I used to code it.

First I'll allow for three types of inputs: a prime number, a modulus value, or nothing. If no input is given `primesPn()` will default to just doing `primesP7a` as the reference prime generator.

Now to do my NTS I need the seed primes form 2-to-n and the modulus these primes create. If the input is a prime number I determine the seeds primes from the reference primes seeds list, then I compute the modulus for those primes. If the input is a modulus value, I start with prime 2 and successively perform the P! process until the given modulus < the generated one. The code saves the prime value (pp) used before this condition is met then uses pp to resize the seeds array like before.

Then I find all the primes up to the modulus value using the reference prime generator (I could speed this up by allowing for passing these primes in as input array, but I wrote this to primarily test my theory and not for speed). The residue primes are then just the primes < mod minus the seed primes.

Then I determine if I need to add any non-prime modular compliments to the residue list. So I create a temporary array which includes the value "1" and the prime residues. Now to determine if a residue is missing a modular compliment I take my tmp array and shift out the first element and subtract it from the modulus. Then I test to see if this value is a remaining residue. If it is then I delete it from the list, if it isn't I add it to the modular compliments list. I keep doing this until the tmp array is empty. Below on top is the original way I coded this, underneath is the more rubyish way to express this.

```
while !tmp.empty?; b = mod - tmp.shift; tmp.include?(b) ? tmp.delete(b) : mcp << b end

while !tmp.empty?; b = mod - tmp.shift; mcp << b unless tmp.delete(b) end
```

Then I just add these modular compliments (if any) to the original residues array and sort it. Now I'm ready to generate the prime candidates in the sieve array. Instead of creating a list of variables n1, n2, n3, etc I create an array which holds those values (starting with the value -1), and generate the prime candidates as before by updating all these array elements after each pass. When I finish generating the prime candidates I then put all the (converted) primes < modulus into the sieve, as before.

Now I to do the sieve. First I add the (modulus + 1) to the residue list and use the first value in the list (it will always be a prime) to start stepping through the possible primes up to the sqrt(N). I multiply each residue by i, convert each value, then remove them from the sieve, then update them til finished.

Well, now that I have created a tool to test my theory to my heart's content let's see what it produces.

For the Mathematicians

First I did some sanity tests. Remember, if I do `integer.primesPn` without any input it will just do `integer.primesP7a` as the default, which it does. Then all of the prior versions should work when I use the appropriate input. So `primesP3a` is mimicked with inputs (3 or 6), `primesP5a` with (5 or 30) and `primesP7a` with (7 or 210). Now what about the SoA modulus, 60? Yep, that works too.

Now let's see if my general theory holds. I put in 11 and it....starts missing primes again. Urgh. The same with 13, 17, and 19. I stopped there. I'll get back to these later.

But then I looked at other moduli and found this. Remember, I conjectured the (modulus + 1) must be prime, and the modulus must be divisible by all the seed primes up to Pn. This was my test code:

```
(20_000_000.primesPn() - 20_000_000.primesPn(X)).size
```

When this returns 0 then the outputs were equal, otherwise it returns the number of the differences.
Replacing X with these moduli produced working generators: 110, 150, 180.
So these are all the working moduli I know work so far: 6, 30, 60, 110, 150, 180, and 210

Now we know why 6, 30 and 210 work, but why 110, 150, and 180? Well 150+1 and 180+1 are prime and their prime factorizations are: 150 = 2*3*(5*5), 180 = (2*2)*(3*3)*5 (remember: 60 = (2*2)*3*5) so these are composed of multiples of the first three consecutive primes. But how can 110 work? 110+1 isn't prime, though 110-1 is, and 110 = 2*5*11. Apparently some (modulus - 1) being prime will also work. I can see this because earlier for P_3 it was shown the primes are of the form 6k+1 and 6k-1.

The primes 13, 17, 19 won't work, their moduli+1 aren't prime, and have these prime factorizations: for P!(13) = 30031=59*509; for P!(17) = 510511=19*97*207; for P!(19) = 9699691 = 347*27953. So my strictly prime (SP) conjecture about the moduli created as P! was too restrictive.

Finally, my initial theory also breaks down for `primesPn(11)` but in a very interesting way. The SP modulus for this is P!(11) = 2*3*5*7*11 = 2310, with 2310+1 prime, so it satisfies all the modulus conditions I hypothesized, but it still misses primes. I ran this to find the missing primes and residues.

```
(10000.primesPn()-10000.primesPn(11)).map {|i| i%2310}.uniq.sort
```

I found this test starts missing the first prime at the (prime) value 2791.

This produces an array of the unique modular compliment pairs that are needed to produce the missing primes. Here are these 32 numbers with their prime factorizations:

289 = 17*17, 391 = 17*23, 481 = 13*37, 493 = 17*29, 529 = 23*23, 559 = 13*43, 629 = 17*37
667 = 23*29, 793 = 13*61, 841 = 29*29, 893 = 19*47, 899 = 29*31, 923 = 13*71, 961 = 31*31
1037 = 17*61,1121= 19*59, 1189 = 29*41, 1273 = 19*67, 1349 = 19*71, 1387 = 19*73, 1411 = 17*83
1417 =13*109,1469=13*113,1517=37*41, 1643 = 31*53, 1681 = 41*41, 1751 =17*103,1781=13*137
1817 = 23*79, 1829=31*59, 1919=19*101, 2021= 43*47

Sure enough, what do you see? They all form modular compliments pairs of 2310, i.e. 289+2021=2310, 391+1919=2310, etc. But unlike from my conjectures, none are compliments to a prime, and they are composites of prime pairs between 13 and 137. I (currently) have no idea how you would know how to compute them if they are not compliment to a prime.[WAS (wild ass guess), I smell complex numbers.]

However, now that I know what the are, I know what to do with them to make `primesPn(11)` work.

I create an array with these mcps, and add them to the residue array when the input is 11 or 2310.

```
seeds = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37,41, 43, 47, 53, 59, 61]
mcps = [289, 391,  481,  493,  529,  559,  629,  667,  793,  841,  893,  899,
        923, 961, 1037, 1121, 1189, 1273, 1349, 1387, 1411, 1417, 1469, 1517,
       1643,1681, 1751, 1781, 1817, 1829, 1919, 2021]
```

```
residues.concat(mcp).sort!; residues.concat(mcps).sort! if [11, 2310].include? input
```

Now `primesPn(11:2310)` will work for these input.

But I was curious to know how many total residues were used, so here's the breakdown. There are 338 primes 11 < p < 2310, and 109 modular compliments to these primes, plus the 32 non-primes modular compliments, for a total of 479 numbers in the residues array. This means these 479 numbers, plus 1, will generate all the possible prime candidates by adding the modulus 2310 successively to them, and then these same 479 numbers, plus 2311, will be used to eliminate all the non-primes. Timings show `primesPn(11:2310)` is faster than for all the other generalized inputs forms, making it the ultimate NTS prime sieve generator I've found so far.

Now getting back to the modulus value 110, for what it's worth, here are its stats from the code:

seeds = [2, 3, 5]; mcp = [3, 9, 21, 27, 39, 49, 51, 57, 63, 69, 81, 87, 91, 93, 99], 15 values
residues = [7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73,79,83,89,97,101,103,107,109], 26

So the modular compliments are all multiples of 3 or 7. But remember 110 = 2*5*11. Are these mcps filling in for the missing primes 3 and 7 not being factors of the modulus? Thus, these 41 numbers, plus 1, generate all the prime candidates, then with 111 (a multiple of 3) they eliminate all the non-primes. This is deep! How many more moduli are like this?

Now look at what I call the efficiency of the generators, which I compute as ln(modulus/ # of residues). The modulus represents the range of numbers checked per pass, and the residues tell you the number of possible primes that can exist within each range. Higher efficiencies means more primes can be found faster and with less work. This should correlate with performance, and timings mostly show it does.

As you can see, the moduli for 30, 60, 150, and 180 all have the same efficiency ln(15/4), which means as I stated before, the larger moduli perform no better than P_5, so it's the "best" generator of this group because it's the simplest. Also note, they are all multiples of 30, as is 2310 (77*30). It should be an interesting area of research to see how many more multiple of 30 (2*3*5) moduli can be made to work.

Another thing that's shown is the strictly prime (SP) generators P_3, P_5, P_7 and P_{11} are the relatively most efficient, compared to the composite prime generators. This makes me think I was on track with my original conjecture about these generators, though they are still only a subset of all possible generators. Finally, the table shows these NTS generators try to compute the primes in a logarithmic manner, as they exist in numerical density. Theoretically, there should be more efficient generators than these.

Table 3.

modulus	6	30	60	110	150	180	210	2310
residues	2	8	16	42	40	48	48	480
efficiency = ln(mod/res)	1.0986	1.3217	1.3217	0.9628	1.3217	1.3217	1.4759	1.5712

While writing this another relationship popped out. For the SP generators P_3, P_5, P_7 and P_{11} the number of residues for each are (N-1)*(residues for preceding SP generator), where N is the SP number. So for P_5, it has (5-1)*(residues for P_3) = 4*2=8; for P_7, (7-1)*(8) = 48; and P_{11}, (11-1)*48 = 480. Is this just coincidence? Does it hold for all SP generators? Can we predict/find the next SP generator from this? Well P!(13) = 30030, and while 30031 isn't prime, 30029 is, and Pi(30030) = 3248. So does P_{13} work, and have (13-1)*480 = 5760 total residues?

There is some very interesting math underlying all of this, which awaits to be fully discovered.

Further optimizations

A major advantage of these NTS are their inherent capacity to generate in parallel the prime candidates (`ni +=k, sieve[ ni] =ni`) and then do the sieve process (`sieve[ pi] = nil, pi += k`). Readily available dual core processors should be able to easily do P_3 in parallel, while an 8 core system should be able to completely do P_5. These NTS are also tailor made for super computer implementation. I eagerly await the results of P_{11} done with a super computer. It may also be worthwhile to see if using threads (which are native to Ruby) would also yield a speedup in software for single processor systems.

If memory use optimization is a driving force there are many ways to represents the primes list with a compressed memory footprint. One simple method is to represent the odd numbers as bits in a binary string, where "1" is a prime, and "0" non-prime. The 25 odd primes for Pi(101) need 50 bits, and are:

10_10001001_01001101_00110010_01011010_01100101_10110111 = 2894d325a65b7

To convert to natural numbers do (bn<<1)+3, where bn is the bit position, starting with b0=0, or lsb. There are more ways to do this, but this shows one simple way to reduce memory usage.

Primality tester

Well, since I've now created the ultimate prime generators can I also create the ultimate primacy tester? After all, the theory should work in both directions. First, let's look at the Miller-Rabin primacy tester. Here's one Ruby version I got from this code here: http://snippets.dzone.com/posts/show/4636

```
# Miller-Rabin prime test in Ruby
# From: http://en.wikipedia.org/wiki/Miller-Rabin_primality_test
class Integer
   def prime?
     n = self.abs()
     return true  if n == 2
     return false if n == 1 || n & 1 == 0

     # cf. http://betterexplained.com/articles/another-look-at-prime-numbers/ and
     # http://everything2.com/index.pl?node_id=1176369

     return false if n > 3 && n % 6 != 1 && n % 6 != 5     # added

     d = n-1
     d >>= 1 while d & 1 == 0
     20.times do                                  # 20 = k from above
       a = rand(n-2) + 1;   t = d
       y = ModMath.pow(a,t,n)                     # implemented below
       while t != n-1 && y != 1 && y != n-1
         y = (y * y) % n;   t <<= 1
       end
       return false if y != n-1 && t & 1 == 0
     end
     return true
   end
end

module ModMath
   def ModMath.pow(base, power, mod)
     result = 1
     while power > 0
       result = (result * base) % mod if power & 1 == 1
       base = (base * base) % mod;  power >>= 1;
     end
     result
   end
end
```

Miller-Rabin is called a non-deterministic probabilistic primacy tester, meaning it makes good 'guesses' that a number is prime, but for some (small) percentage of numbers it will guess wrong. With any of the prime generators you can do a poor-mans primality test by doing this: `N.primes.last == N`, but this is slow and inefficient. With my now extensive knowledge of prime generation theory :-) I can develop a simpler/faster deterministic primality tester. All I have to do is perform the P_3 = 6k+ (1, 5) generator's sieve backwards. I'll call it `primz?` to distinguish it from the M-R code.

```
class Integer
   def primz?
      n = self.abs
      return true  if [2, 3, 5].include? n
      return false if n == 1 || n & 1 == 0
      return false if n > 5 && ( ! [1, 5].include?(n%6) || n%5 == 0)

      7.step(Math.sqrt(n).to_i,2) do |i|
        # p1= 5*i, k = 6*i,   p2 = 7*i
        p1 = 5*i;  k = p1+i;  p2 = k+i
        return false if  [(n-p1)%k , (n-p2)%k].include? 0
      end
      return true
   end
end
```

You can not get much simpler than this! Here's how/why it works.

First I check for 2, 3, and 5, then I check for 1 and even numbers (as done in the M-R code). Then for numbers > 5, if the number fails the P_3 = 6k+(1,5) test (N mod 6 != 1 or 5) OR it's divisible by 5, the number isn't prime. If the number passes these test then I merely need to do the following simple test.

Look back at the code for `primesP3`. I don't need the part that generates candidate primes, because if it passes all the input tests it's a prime candidate, so I just need some of the sieve code. Finally, after 3 weeks of thinking about the how to do this, after I finished writing the prime generation part of this paper above, and took time to focus on how to do this, it all became clear.

The sieve for `primesP3` uses the 3 numbers p1 = 5*i, p2 = 7*i, and k = 6*i. For each i, p1 and p2 are set as base numbers and then successive multiples of k are added to both. These values for p1 and p2 constitute non-primes, so if N is equal to any of these values it isn't prime. So the test becomes: for each i subtract the base (first) value for p1, p2 from N, then if either of these results is divisible by k, i.e. [(N-p1) or (N-p2)] mod k = 0, the sieve would mark N as non-prime. Pick numbers from Table 1 and do it by hand. I couldn't stop smiling when I finally figured it out and saw how simple the code is.

My cursory benchmarks show my prime test is at least an order of magnitude faster than the M-R code shown herein, while having the advantage of being a deterministic test, so you can absolutely positively be certain if a number is/not prime. The GIMPS project folks (Great Internet Mersenne Prime Search: http://www.mersenne.org) should really love this.

And just as you can use a prime generator as a poor-mans primality tester you can use a prime tester as a poor-mans prime generator. Here's code that will generate primes between a range x..y:

```
def primesxy(x,y); (x..y).map {|p| p.primz? ? p:nil}.compact! end
```

Now you can verify primes up to some N like this: `10_0000.primesPn(11) == primesxy(2,10_0000)`

or can generate the primes within ranges like this: `primesxy(10_000, 20_000)`
or determine the number of primes within a range: `primesxy(50_000, 90_000).size  => 3580`

Remember, generating primes this way is much slower than the native generators, but for small ranges it's perfectly useful, and quick. Now we have a larger set of tools to find out just about anything we want to know about primes. I also wanted to show how easy it is to do this stuff in Ruby.

Now that I have the ultimate handy dandy primality tester, one of the first things I did with it was to find where the next SP = P!(n) modulus + 1 was prime past P!(11) = 2310. So here it is:

```
((2*3*5*7*11*13*17*19*23*29*31)+1).primz?  =>true
```

where P!(31) = 200,560,490,130. So now I want to know Pi(P!(31)), because it, minus the 11 seeds primes 2 -- 31, gives me the number of prime residues for this modulus. So I enter into my laptop `200560490130.primesPn(11).size` and go to bed, but in the morning I see an out of memory error in Ruby staring at me. My poor ole laptop just didn't have the bits to finish the job.

Now Pi(n) ~ n/ln(n), gives Pi(200,560,490,130) ~ 7,706,638,035 primes, and that's a low estimate.
If it works, it has an approximate efficiency of: ln(200560490130.0 / 7706638035) = 3.2590

These numbers are starting to make my head hurt, so I'll stop here.

Final thoughts

It's now the first week in June 2008 as I write, so what you see here has been a month long journey of exploration and discovery. I release this work now into the wild so others can study, use, and extend.

One area of immediate use (and financial gain) I can see these results applied to is the GIMPS project, which is trying to find the Mersenne primes 2^P-1, where P is a prime. The last one found was #44 in September 2006 for P = 32,582,657, which produces a 9,808,358 digit prime. There just so happen to be a $100,000 bounty for the first person/group to discover the first 10 million digit prime. With my primacy tester it should be easier and faster now to find with certainty the candidate exponents P, and then check the results. Hey, just send me one-tenth of a penny for every digit you find. :-)

Another area of immediate applied use is for work in cryptography, which relies on large primes as the basis for security in many cases. In fact, any application or area of research which needs to find or check primes, or factor numbers, should be able to apply these findings to make life easier and faster.

However, my greatest anticipation is to see the results of parallel implementation of these NTS for the different generators. I also would like to see some of these routines (especially `primz?`) become part of standard software libraries (Ruby, Python, Perl, Java, C, etc). Finding/testing for primes is such a standard thing that's done, especially in computational numerics, it would be a no-brainer to do.

I also want to thank Yukihiro "Matz" Matsumoto for inventing Ruby. I was able to do all this work with my laptop and the Ruby interpretor, enabling me to test and probe my math ideas on the fly, with this very expressive language. The insites of my thinking were directly aided by the simplicity, efficiency, and elegance Ruby provided in enabling their formulation and their subsequent expression.

The code from this paper can be downloaded from here: http://snippets.dzone.com/posts/show/5610
and the pdf of this paper is here: http://www.4shared.com/dir/7467736/97bd7b71/sharing.html

Timing Results

Ruby has a standard benchmarking library, which you can use to benchmark these routines in parallel. But so I could do quick and immediate timing tests I used the following code below, with examples.

```
def tm(code); s=Time.now; eval code; Time.now-s end

tm '100000.primesP7a'   or   tm '23949023321.primz?'
```

Table 4.

n+1	$1x10^5$	$3x10^5$	$5x10^5$	$1x10^6$	$3x10^6$	$5x10^6$	$1x10^7$	$2x10^7$	$3x10^7$
Pi(n+1)	9,592	25,997	41,538	78,498	216,816	348,513	664,579	1,270,607	1,857,860
SoA	0.114	0.365	0.446	0.800	2.139	3.164	6.305	12.482	18.631
primesP60	0.023	0.073	0.170	0.225	0.700	1.512	2.304	4.735	7.589
primesPn3	0.066	0.217	0.361	0.695	2.181	3.697	7.524	15.044	22.441
primesPn5	0.046	0.156	0.256	0.498	1.519	2.261	5.401	10.470	15.219
primesPn7	0.048	0.125	0.204	0.413	1.285	2.134	4.364	8.476	12.526
primesPn11	0.073	0.188	0.271	0.472	1.270	2.069	3.994	8.356	12.005
primesP3a	0.038	0.088	0.145	0.305	0.892	1.483	3.044	6.566	8.925
primesP5a	0.023	0.088	0.132	0.246	0.703	1.754	2.355	5.415	6.377
primesP7a	0.022	0.066	0.103	0.230	0.629	1.114	2.288	4.591	5.898

All times are in seconds, run with Ruby 1.9.0-1, with an Intel P4, 2.8 Ghz, 1 GB based laptop.

The SoA is the direct Ruby translation of the Python SieveOfAtkin code shown herein. `primesP60` is my hand-coded minimum memory implementation of the SoA generators. The SP generators P_3, P_5, P_7 and P_{11} were performed by my generalized implementation, to do an apples-to-apples comparison of the generators efficiency. The timings indicate that, for implementations being equal , P_{11} is the fastest generator of the four, especially as N gets larger.

Extra!, Extra!

Just before I was about to put this paper to rest I took about 15 minutes to test if P_{13} would work. I first found the composite prime mcps, using the previously shown code, which found 1086 of them. Then I generated the 3242 prime residues $13 < p < 30030$ for P_{13}, then I separately ran the code (all done in the Ruby interpretor) to generate the regular mcp (1431), and sure enough 1086 + 3242 + 1431 = 5759, so (1+#res) gives a total of 5760, which matches the predicted number of residues of (13-1)*480 = 5760. Then I modified `primesPn` so I could pass in an array of the composite prime mcps and select them, so I now can do `N.primesPn(13,mcps)` to test P_{13} (or others) with my general code. And...it worked!

n+1	$1x10^5$	$3x10^5$	$5x10^5$	$1x10^6$	$3x10^6$	$5x10^6$	$1x10^7$	$2x10^7$	$3x10^7$
primesPn13	1.201	1.577	2.158	2.517	3.851	5.488	8.148	13.003	17.536

Its efficieny is ln(30030/5760) = 1.651. If done in parallel it should be the ultimate prime sieve (so far). Again let me say, because Ruby is so easy to express ideas in it allowed me to think about, probe, and implement the math without thinking about the programming. It really made this work realizble for me.

Sieve of Zakiya/NTS Prime Generators

P_3 = 6k+(1,5)

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

P_{60} = 60k+(1, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 49, 53, 59)

P_{110} = 110k+(1, 3, 7, 9, 11, 13, 17, 19, 21, 23, 27, 29, 31, 37, 39, 41, 43, 47, 49, 51, 53, 57, 59, 61, 63, 67, 69, 71, 73, 79, 81, 83, 87, 89, 91, 93, 97, 99, 101, 103, 107, 109)

P_{150} = 150k+(1, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 49, 53, 59, 61, 67, 71, 73, 77, 79, 83, 89, 91, 97, 101, 103, 107, 109, 113, 119, 121, 127, 131, 133, 137, 139, 143, 149)

P_{180} = 180k+(1,7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 49, 53, 59, 61, 67, 71, 73, 77, 79, 83, 89, 91, 97, 101, 103, 107, 109, 113, 119, 121, 127, 131, 133, 137, 139, 143, 149, 151, 157, 161, 163, 167, 169, 173, 179)

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

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

P_{13} = 30030k+(1, [5759 residue values: 3242 primes, 1431 prime mc, 1086 composite mc])