The Use of Prime Generators to Implement Fast Twin Primes Sieve of Zakiya (SoZ), Applications to Number Theory, and Implications to the Riemann Hypotheses

Jabari Zakiya

Abstract

This paper describes the mathematical foundation of Prime Generators and their use in creating a fast Twin Primes Segmented Sieve of Zakiya (SSoZ), and also their applications to Number Theory, including Mersenne Primes, creating an exact Prime-counting Function, and implications for the Riemann Hypothesis.

The Use of Prime Generators to Implement Fast Twin Primes Sieve of Zakiya (SoZ), Applications to Number Theory, and Implications for the Riemann Hypotheses

Jabari Zakiya © Revised: January 10, 2019
jzakiya@gmail.com

Introduction

In my previous paper, **The Segmented Sieve of Zakiya (SSoZ)** (2014), I describe a general method to find prime numbers using an efficient prime sieve based on Prime Generators (PG). I expand upon that here, and present a customized version to specifically find Twin Primes, likely the fastest and most efficient method to do so. I also show how PGs can be used to characterize and find Mersenne Primes, create an exact Prime-counting Function, and allude to its implications to the Riemann Hypotheses.

Preamble

In 2008 I become aware of, and started delving into and understanding, the properties of what I term to be integer **cyclic generators**, and specifically **Prime Generators (PG)**. I subsequently developed the **Sieve of Zakiya (SoZ)**, based on their properties, that solely works in their integer space domain. After additional study and understanding of their properties, and maturation of coding techniques, I had a pretty good **SoZ** implementations, inherently faster than the classical **Sieve of Eratosthenes (SoE)** [9].

All my original code development was done using **Ruby** [15]. Around 2013 I started seriously looking at **segmented sieve** implementations, particularly to perform on multicore|threaded hardware systems. I subsequently looked at using compiled languages to perform parallel processing with, and released The Segmented Sieve of Zakiya (SSoZ) [1] in 2014. In that paper the coding examples were done in C++.

Since 2014, the capability to perform true parallel processing efficiently has matured, and exists in many more languages now, enabling wider exploitation of increasing hardware cores|threads. I thus began looking at different languages, and found **Nim** [16] easy enough to learn, and produce a fast SSoZ in. However, the reference Nim code provided herein can be easily translated into any language.

The SoZ and SSoZ algorithms perform very simple and regular operations and are best understood by visualizing (see pictures of) their structure and operations. They can be performed with any PG, which makes them universal generic algorithms, which can be adaptively configured to select a best PG to use based on input values and/or system resources.

Having worked with Prime Generators for over 10 years now I continue to learn new things, and ask new questions, about their structure, meaning, and use. They posses many interesting (even *magical*) mathematical properties. I briefly describe some of these properties that pertain more to pure math, showing their use to characterize and search for Mersenne Primes, to create an exact Prime-counting Function $\pi(x)$ [20], and with that, some similarities and implications to the Riemann Hypotheses [21].

However, the primary focus is to present new and better techniques to perform a Segmented Sieve of Zakiya (SSoZ). While the code components and techniques can be configured to perform various sieve types, I focus here on an implementation to find **Twin Primes** [8], to highlight the ease and simplicity of employing prime generators to such a task, and the subsequent performance benefits.

Mathematical Foundations of SoZ

The purpose of this section is to provide enough math to understand how|why mathematically the SoZ works, in order to conceptually understand its design components and their coding implementations.

Prime Generators

If we write out all the integers in a linear form like this: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14… there is no discernible way to distinguish primes from the list, *based on its linear structure*. However, instead of listing all the integers linearly we can write them out in a **cyclic representation**, as below. From mere observation, *we can begin to see* certain structures and patterns emerge as regards primes.

0, 2, 4, 6, 8, 10, 12…	0, 3, 6, 9, 12, 15, 18…	0, 4, 8, 12, 16, 20, 24, 28…
1, 3, 5, 7, 9, 11, 13…	1, 4, 7, 10, 13, 16, 19…	1, 5, 9, 13, 17, 21, 25, 29…
	2, 5, 8, 11, 14, 17, 20…	2, 6, 10, 14, 18, 22, 26, 30…
		3, 7, 11, 15, 19, 23, 27, 31…

In the first example of cycle 2, it basically divides the integers into the set of evens (top row) and odds, with all the primes on the odds row (except the even prime 2). For cycle 3, primes are on all three rows, and it's *not structurally evident* how to identify them. However for cycle 4, a clearer pattern emerges, as all the primes (past the first column) exist only on two rows, which have just odd integers. Progress!

These three examples will generate all the integers, with cycles of 2, 3, and 4 (the linear list is cycle 1). They represent integer **cyclic generators** of form $\mathbf{I_n = modg * k + r}$, with r ε {0, 1, 2...modg-1}, k ≥ 0, where **modg** is the generator **modulus** (here 2, 3, or 4), and **r** a **residue** from the set of integers in the first column of each cycle. *Our goal is to construct generators good at generating primes*.

For **modg** even we can construct a *Prime Generator* (**PG**) with properties useful for creating fast prime sieves. To create a prime generator $\mathbf{P_n}$ from a cyclic $\mathbf{I_n}$ we eliminate its $\mathbf{r_i}$ which don't generate primes.

To visually illustrate this, using the cyclic generator I6 = 6 * k + r, r ε {0, 1, 2, 3, 4, 5}, k ≥ 0, we seek to eliminate the residues **r** (rows) which only have non-primes and keep those that contain primes.

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 * (3 * k + 1). No primes.
r = 3: 3, 9, 15, 21, 27, 33, 39, 45, 51...	r = 3: 6 * k + 3 = 3 * (2 * k + 1). No primes.
r = 4: 4, 10, 16, 22, 28, 34, 40, 46, 52...	r = 4: 6 * k + 4 = 2 * (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.

For I6 only the residues 1 and 5 can generate prime numbers, i.e. r ε {1, 5} for primes, which is only 1/3 of the integer space that could be generated for modg = 6. Thus, P6 = 6 * k + {1, 5} (or 6 * k ± 1) is a prime generator which generates all primes > 3, where I designate the prime generator modulus as **modpg.** However, because modpg = 6 = 2 * 3 is the product of the first two primes, P6 is also what I call a **Strictly Prime** (**SP**) prime generator, and I refer to it as P3, to distinguish it as such.

Mathematically then, a PG's residues $\mathbf{r_i}$ are the set of integers **r** ε {1 .. modpg-1} coprime to its **modpg** (have no common factors) i.e. their **greatest common divisor** is 1: gcd(**r**, **modpg**) = 1.

Generator Efficiency

While we can create a prime generator from any even **modg** (including 2) we want to use values to create *good* ones. By *good* I mean they have certain advantageous structural forms, which are:
1) For a given number of residues – **rescnt**, the fraction|ratio **rescnt / modpg** is the smallest possible.
2) The generated primes are *in order*; i.e. except for a small set of consecutive *base (excluded) primes* that form its modulus, all the other primes are generated **in order**, with no missing ones

Example of Form 1 PGs.
P6 = 6 * k + {1, 5}. We see for the I4 generator shown above we can form the PG P4 = 4 * k + {1, 3}. Both have two residues (rescnt = 2), and both generate all the primes *in order* with P4 all primes > 2 and P6 all primes > 3. However P6 is *better,* as its **rescnt / modpg** ratio is 1/3 vs 1/2 for P4. Thus the **number space** it has to generate to contain all the primes is smaller, 1/3 of all integers vs 1/2 for P4.

Example of Form 2 PGs.
P8 = 8 * k + {1, 3, 5, 7} is an in order generator, as its residues are all the primes between 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 consecutive.

Thus, when I refer to a generator's efficiency I'm alluding to the percentage of the total integer number space it has to generate to produce the primes up to some N. The smaller the ratio the fewer non-primes the generator will produce, which ultimately have to be sieved out to identify just the primes. Strictly Prime (SP) generators are structurally the most efficient. Fig 1 shows the efficiency of the first few SP generators, along with a few selected in-order non-SP modpg value PGs for comparison.

Fig 1

P_n	P3	P5	P60	P120	P150	P180	P7	P11	P13	P17
modulus (modpg)	6	30	60	120	150	180	210	2310	30030	510510
residues count (rescnt)	2	8	16	32	40	48	48	480	5760	92160
% of number space	33.33	26.67	26.67	26.67	26.67	26.67	22.86	20.78	19.18	18.05

Distribution of Primes

For a given prime generator, the primes are uniformly distributed along each residue. Fig 2 is the tabulation of primes along the residues for P5 for the first 50 million primes it identifies.

Fig 2

Distribution of the 50 Million primes[4 – 50000003] from P5[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 means the number of non-primes to be sieved are uniformly distributed along each residue too.

Canonical vs Functional PG Forms

In the paper I use the P5 SP generator as the reference generator for examples. For its mathematical description I use its canonical form, with first residue $\mathbf{r_0 = 1}$, followed by the first prime residue $\mathbf{r_1}$. P5 = 30 * k + {1, 7, 11, 13, 17, 19, 23, 29}. However, I use its functional form when using it to create a sieve, with $\mathbf{r_1}$ the first residue, and **last residue = (modpg + 1)**. So for P5 (et al) its functional form is: P5 = 30 * k + {7, 11, 13, 17, 19, 23, 29, 31}, which is the standard PG form I use going forward.

Generator Properties

Prime generators are specialized mathematical forms to generate primes, that allow for the creation of simple and fast prime sieves. Practically all their fundamental properties|characteristics are discernible solely from visual observation of their structure. If you look at them long enough certain things just pop out. Because SP generators are the most efficient I will limit discussion here to their properties, as they are exclusively used in the construction of the S/SoZ, specifically P5, P7, P11, P13, and P17.

Residue Value Properties

- there are an even number, with half symmetric (canonical form) to **modpg div 2**
- they are odd integers, with 1/4 of each ending with digits 1, 3, 7, or 9
- multiplication of numbers in a column map to a product value in another row|column
- each has a modular inverse $\mathbf{r^{-1}}$ (another residue value or itself) e.g. $\mathbf{r \cdot r^{-1} \bmod modpg = 1}$
- they always occur (canonical form) as **modular compliment pairs** e.g.: $\mathbf{modpg = r_i + r_j}$
- all the residues table values $\mathbf{r_1 < (r_1)^2}$ are prime

SP Modpg Properties

- they have the form: $\mathbf{modpgn = \prod(p_i) = 2 \cdot 3 \cdot 5 \cdot 7 \ldots. p_n}$
- the number of residues has form: $\mathbf{rescnt = \prod(p_i\text{-}1) = (2 -1) \cdot (3 -1) \cdot (5 -1) \ldots.(p_n\text{-}1)}$

For P5 its modulus is: modp5 = 30 = 2 * 3 * 5, and has 8 residues: rescnt = 8 = (2-1) * (3-1) * (5-1)
Each residue has a modular compliment: 30 = (1 + 29) = (7 + 23) = (11 + 19) + (13 + 17)
Each residue has a modular inverse: P5 residues [7, 11, 13, 17, 19, 23, 29, 31]
P5 inverses [13, 11, 7, 23, 19, 17, 29, 31]
We see here residues 11, 19, 29, and 31 are their own (self) inverses, i.e. (11 * 11) mod 30 = 1, etc.

Residues (PCs) Tables

The depiction of a cyclic generator creates a 2D array|matrix of integers. The rows correspond to the residue values and the columns are referenced by their column index **k** value. Each prime generator has a tabular functional form representation where each integer in the table is a **prime candidate (pc)**. They are the minimum set (and number) of integers that can possibly be prime ≤ N for a given PG.

In a **residues table** each column is a **residue group** (**resgroup**) of **pc** values, referenced by its index **k**. Each resgroup has a **base value** of **modk = modpg * k**, and each resgroup value can be **numerated** by adding its **residue** value to its base value, e.g. **pc_i = modk + ri** for each pc in a resgroup.

Performing the S/SoZ identifies the prime multiples (non-primes) in a PG's residues table for some N.

Computing Residues Products

To find the resgroup (col) for a pc value in the table we integer divide it by the PG modulus. To find its residue value, we find its integer remainder when dividing by the PG modulus. Thus a pc's regroup value is: **k = pc div modpg**, and its residue value is: **ri = pc mod modpg**, where herein / performs the **div** operation and % performs the **mod** operation.

Each prime is parametized by its residue value **r**, its resgroup value **k**, along with the PG's modulus value **modpg**, with **modk = modpg * k**. Thus, each prime has general form: **prime = modk + r**. Similarly, every **pc_i** member in the resgroup has a residue **ri** and has the value: **pc_i = modk + ri**. Thus, **product = prime * pc_i** translates into a table value with some new parameters **kp** and **ri**.

Using simple math, we can transform table multiplications into a computationally efficient form.

```
                  prime * pc_i
              (modk + r) * (modk + ri)
modk * modk + modk * (r + ri) + (r * ri)
       modk * (modk + r + ri) + (r * ri)
(modpg * k) * (prime + ri)  + (r * ri)
 modpg * [k * (prime + ri)] + (r * ri)
         |_________________|   |______|
                kk               rr
```

The original multiplication has now been transformed to the form: **product = modpg * kk + rr**

where **kk = k * (prime + ri)** and **rr = r * ri**, which has the general form: **pc = modpg * k + r**.

The (r * ri) term represents the base residues (k = 0) cross products (which can be pre-computed). From the (r * ri) term we extract two pieces of information. The first is its resgroup|column value, **kn = (r * ri) / modpg**. Then we determine its residue value, **rn = (r * ri) % modpg**, and translate it to a restrack index value, conceptually as **rt_i = residues.index(rn)**. Thus for P5, r = 7 is at residues[0], so that its rt_i = residues.index(7) = 0. (See section on Implementation Details.)

Thus, the product of any two members in a resgroup k translates to resgroup **kp = kk + kn** on **rt_i**.

Explicitly for primes then: **kp = k * (prime + ri) + (r * ri) / modpg**

To describe this verbally, to find the resgroup of the product of any two resgroup members, numerate one member (here a prime), call its residue **r**, add the other's residue **ri** to it, multiply that sum by the resgroup value, then add to that the resgroup of their residues cross product. Values from Table 1 give:

Ex: kp = (97 * 109) / 30 = 3 * (97 + 19) + (7 * 19) / 30 = 3 * (109 + 7) + (19 * 7) / 30 = 352

The advantage of this multiplication translation, versus the straight multiplication, is that all values stay relatively much smaller than N as it becomes bigger, which makes it more efficient and faster on older, and/or limited, hardware systems. There are also various ways to perform the translation using diverse techniques, such as lookup tables for its parts. (See discussion in **PRIMES-UTILS HANDBOOK** [5]) And as we'll see later, it allows for the easy computation of the **nextp** arrays used in the SSoZ.

Residue multiplications are the most numerous complex arithmetic operations in the SoZ and SSoZ, so doing them efficiently significantly determines their ultimate performance.

Number Theory and Riemann Hypotheses

Before presenting the use of Prime Generators in the creation of fast prime sieves I want to *briefly* show their use for advancing the understanding of some fundamental topics in Number Theory.

Prime Generator Sequences

Each prime generator has a characteristic **Prime Generator Sequence (PGS)**, which can be used to generate all the elements in its residues table up to some N.

```
P3 = 6 * k + {1, 5} = 1   5   7  11  13  17  19  23  25  29  31..
                      +   4   2 | 4   2 |  4   2  |  4   2  |  4   2
```

Here, to get the next|previous value in the table we add|subtract either 2 or 4 to the current value.

```
P5 = 30 * k + {1.. 29} =  1  7   11   13  17   19   23   29   31  37   41   43   47   49   53   59   61…
                          +   6  4   2    4   2    4    6    2 | 6   4    2    4    2   4    6    2
```

```
PGS P7:   10  2  4  2  4  6  2  6  4  2  4  6  6  2  6  4  2  6  4  6  8  4   2  4
           2  4  8  6  4  6  2  4  6  2  6  6  4  2  4  6  2  6   4  2  4  2 10  2 |
```

We can easily visually discern some of their general fundamental properties.

Prime Generator Sequence Properties

- the number of terms in the sequence is the same as the number of residues
- the terms are all even numbers (because the table values are all odd)
- the sum of the fundamental sequence terms equals the PG's modulus **modpg**
- they have a characteristic symmetry around their midpoints
- will identify every prime (past the modulus primes missing in the sequence) up to any N
- will produce every value in a PG's residues table for any N
- a sequence can be traversed forward or backward from any value

These sequences are very useful in optimizing prime searches over ranges of numbers without the need to actually parametize a PG. Again, the longer sequences for higher efficiency PG's will reduce the number space (pcs) to examine much more, though in practice shorter ones are easier to program.

For example, in the code for `genPGparameters` I use P3's PGS to identify (and reduce) the values to test to find the residues for P5 up to P17, versus simply testing every value (or odds) up to their moduli. However, for doing searches over very long ranges, the sequences for larger generators could pay for themselves in increased performance (reduction in time) in the long run.

Doing math with sequences is very similar to using **residues tables**. For any value in the sequence, to jump to another **resgroup** you merely add|subtract the **modk** value to it, and the sequence picks up from that point using the new value.

Essentially then, a PGS encodes information about a PG's residues table (and thus the primes) in a different form. They each have their utility and advantages to work in. I am certain there unique properties constitute a worthy field of dedicated academic study.

Mersenne Primes Search

By definition, prime generators find every prime, thus every special type or class of prime has some PG parametization. Here I provide a *brief* presentation of a process using prime generators to perform a very efficient search for Mersenne Primes. A fuller presentation is the topic of another paper.

Mersenne Primes have the form: $\mathbf{M_p = 2^p - 1}$ for p prime. P4 and P3|6 only have two residues, and it's easy to show Mersenne Primes exist for only one of them for each, for P3|6 = 6 * k + 1 and P4 = 4 * k + 3.

Proof: $2^p - 1 = 6k + 1$ for odd p > 2

$$2^p - 1 = 6k + 1$$
$$2^p - 2 = 6k$$
$$2(2^{p-1} - 1) = 6k$$
$$2^{p-1} - 1 = 3k$$

Proof: $2^p - 1 \neq 6k + 5$

$$2^p - 1 = 6k + 5$$
$$2^p - 6 = 6k$$
$$2^{p-1} - 3 \neq 3k$$

Since $2^{p-1} - 1$ and $2^{p-1} - 3$ are odd 3k must be odd, so k must be odd. Setting $k = 2a \pm 1$ in each case, and continuing the regression for both, will show for 6k+5 either $2^{p-x} - 3 = 3a$ or $2^{p-x} = 3a$ at each stage. For $2^{p-x} = 3a$ the left side is even (a power of 2), thus not divisible by 3, and $2^{p-x} - 3 = 3a \Rightarrow 2^{p-x} = 3(a + 1)$, and again 2^{p-x} is not divisible by 3, thus 6k + 5 cannot contain Mersenne Primes.

At this point, by mere deduction then, we know all Mersenne Primes must be of form 6k+1. Because, for $2^{p-1} - 1 = 3k$, setting $k = 2a \pm 1$, and performing both regression, creates an alternating series that ends with 1 = x in both case, thus showing we ultimately achieve equality. Thus we see, for example, the first three odd prime exponents 3, 5, 7 correspond to k values of 1, 5, 21, etc.

Using this same process we can prove for P4, Mersenne Primes have the form 4k+ 3.

Proof: $2^p - 1 = 4k + 3$

$$2^p - 1 = 4k + 3$$
$$2^p = 4(k + 1)$$
$$2^{p-2} = k + 1$$
$$2^{p-2} - 1 = k$$

Proof: $2^p - 1 \neq 4k + 1$

$$2^p - 1 = 4k + 1$$
$$2^p = 2(2k +)$$
$$2^{p-1} = 2k + 1$$
$$2^{p-1} - 1 \neq 2k$$

Here for 4k + 3, for prime exponents 2, 3, 5, 7 we can directly compute k to be 0, 1, 7, 31, etc.

It can also be shown all the k values for each generator are an odd number, which follow a specific sequence, and additionally, not only must p be prime but each prime must have form p = 2n + 3.

While P4 reduces the Mersenne Prime number space to 1/4 of all integers P3|6 reduces it even further to 1/6. But we can do even better by combining them to create a bigger PG with more bandwidth.

The fact that Mersenne Primes reside along a single restrack for P4 and P6 means we can combine them. These primes will exist when their bandwidths intersect, i.e. when the modulus is 4 * 6 = 24. There are 8 residues for modpg = 24, so P24 = 24k + {1, 5, 7, 11, 13, 17, 19, 23}, which is an in-order prime generator which generates all primes > 3. We want only one of these residues to be a generator for Mersenne Primes, i.e. $2^p - 1 = 24k + r$ for only one r ε {1, 5, 7, 11, 13, 17, 19, 23}. To find where they intersect we numerate their values up to 24 and find where their values are equal.

k:	0	1	2	3	4	5
4k + 3:	3	**7**	11	15	**19**	23

k:	0	1	2	3	4
6k + 1:	1	**7**	13	**19**	25

So now we have two possibilities, 24k + 7 or 24k + 19, and use the same proof technique as before.

Proof: $2^p - 1 = 24k + 7$ for p > 3

$$2^p - 1 = 24k + 7$$
$$2^p = 8(3k + 1)$$
$$2^{p-3} = 3k + 1$$
$$2^{p-3} - 1 = 3k$$

Proof: $2^p - 1 \neq 24k + 19$

$$2^p - 1 = 24k + 19$$
$$2^p = 24k + 20$$
$$2^p = 4(6k + 5)$$
$$2^{p-2} - 5 \neq 6k$$

Because $2^{p-2} - 5$ is odd, and 6k even, they can't equate, thus r must be 7, therefore Mp = P24 = 24k + 7.

The number space has now been reduced from 1/4 to 1/6 to now 1/24 (4.167%) of all integers. It is also possible now to build a family of unique generators based on a general pattern for their construction.

By observation, there appears to be an Mp generator pattern. P6 and P24 can be rewritten such that

P6 = 6k + 1 = $(2^1 \cdot 3k) + 1$ P24 = 24k + 7 = $(2^3 \cdot 3k) + 7$ => Mp = $(2^n \cdot 3k) + (2^n - 1)$ for n odd

For P6, n = 1 and r = $2^1 - 1 = 1$, and for P24, n = 3 and r = $2^3 - 1 = 7$.

It can be shown for n odd, a family of declining number space generators can be constructed with form

$$Mp = 2^p - 1 = 2^n \cdot 3k + (2^n - 1), \text{ n odd } (1, 3, 5, 7\ldots)$$

They are used in the following manner. Once you find a Mp exponent prime you can construct the next larger Mp generator to find the next one, reducing the number space to $1/(2^n \cdot 3k)$ % of all integers.

Use P4 Mp = 4k + 3 to find Mp = 3 (p = 2; k = 0)
Use P6, n = 1, Mp = 6k + 1 to find Mp = 7, (p = 3; k = 1)
Use P24, n = 3, Mp = 24k + 7 to find Mp = 31, (p = 7; k = 1)
Use P96, n = 5, Mp = 96k + 31 to find Mp = 8191, (p = 13; k = 85)
Use P384, n = 7, Mp = 384k + 127 to find Mp = 131071, (p = 17; k = 341), and so on.

Again, each Mp generator will find all the Mp primes (past their base primes), but by using the next larger generator past a found prime exponent the number of Mp candidates are successively squeezed within a decreasing percentage of the integer number space, increasing the search process efficiency.

Prime-counting Function $\pi(x)$

While I focus herein how to use prime generators to create fast sieves, from a purely mathematical perspective, they can also be used to identify all the primes without the need to perform a sieve, e.g.:

Axiom 1: *we can always (theoretically) construct a prime generator modulus which forms the basis for a deterministic primality test for every integer up to some finite N*.

Axiom 2: *a prime generator can always (theoretically) be constructed that will directly generate all the primes up to some finite N without the need of performing a sieve*.

To be clear, when I say *theoretically* it's 100% certain you can do this, but space-time limitations will restrict the number sizes you can process using physical devices. (While the mind may be limitless, not so for our time, hardware, money, etc.) Theses axioms merely functionally separate two distinct prime generator operations, where Axiom 1 is required for Axiom 2, but can stand alone independently.

Again, lets look at the P5 residues tables and *visually extract* these axioms.

k	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
rt0	7	37	67	97	127	157	187	217	247	277	307	337	367	397	427	457	487	517
rt1	11	41	71	101	131	161	191	221	251	281	311	341	371	401	431	461	491	521
rt2	13	43	73	103	133	163	193	223	253	283	313	343	373	403	433	463	493	523
rt3	17	47	77	107	137	167	197	227	257	287	317	347	377	407	437	467	497	527
rt4	19	49	79	109	139	169	199	229	259	289	319	349	379	409	439	469	499	529
rt5	23	53	83	113	143	173	203	233	263	293	323	353	383	413	443	473	503	533
rt6	29	59	89	119	149	179	209	239	269	299	329	359	389	419	449	479	509	539
rt7	31	61	91	121	151	181	211	241	271	301	331	361	391	421	451	481	511	541

The reason to use a PG's *functional form* residues table representation is because the first element in the table will always be the first prime that's not a excluded|base prime that forms its modulus **modpg**. For SP Pgs, its **modpg = $\prod(p_i)$**, again, structurally eliminates all the prime multiples of the base primes.

Thus, starting with **rt_0 = 7**, we observe all the pcs in the table less than its square, 49, are in fact primes. Thus, all the **pc_i** from **rt_0** < **$(rt_0)^2$** have the property **gcd(pc_i, modpg) = 1**, and are prime, for every PG.

Ex: To find all the primes < N = 1,000,000, we take its sqrt(N) = 1000. The largest prime < 1000 is 997. We construct **P997**'s modulus, **modp997** = 2 * 3 *…. * 991 * 997 (168 primes), with **rt_0 = 1009**, and since 1009^2 = 1,018,081, then all the pcs from 1009...1,000,000 are prime, i.e. **gcd(pc_i, modp997) = 1**. This also means **modp997** is the modulus for a deterministic primality test for all integers in the range (997...1,018,081), so that for any integer **n** inside the range, if **gcd(n, modp997) != 1** it's not a prime!

The beauty of these mathematical properties is that they are all visually intuitive, and provable by mere observation of a PG's residues tables. *One residues table is worth a 1000 postulates!*

As a practical (versus theoretical) matter sieving for primes as N becomes larger is more physically realizable, as you can use much smaller PG's to achieve the same results faster. But all is not lost in using this technique though, especially if you just want a **fast deterministic primality test**.

Say you wanted to find|test for primes within the 64-bit range, i.e. (0...18,446,744,073,709,551,615). The largest generator modulus would be for **P4294967291**. Now this is a really! large number, much larger than 64-bits, but we can be clever in creating it. We can precompute and store the multiplications in groups e.g. they fit within 64-bits. Then the full value would be modpg1 * modpg2 .. * modgpn. We could also partition this e.g. different desired ranges would need a minimal set of groups. Then to test an integer would entail performing the gcd(n, modpgi) tests for the least number of groups in its range. If for an **n** all the gcd tests are 1 then **n** is prime, if any one isn't, then it's not prime. A significant speed benefit of this approach is that the gcd tests can be done in parallel, as they are all independent.

Prime-counting Function Pn(1000) and Pn(1000000) in Ruby

Enough theory! The Ruby code here, if entered in a **irb** terminal session, will work as shown. It is provided to functionally show how to perform the Pn(x) counting function (not for speed). Ruby will work with arbitrary size numbers, until your system memory (or time and/or patience) runs out.

```
$ irb
> n = 1000; Integer.sqrt n  => 31
> base_primes = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31]
> modp31 = base_primes.reduce :*  => 200560490130
> rescntp31 = base_primes.reduce(1){ |prod, p| prod * (p-1) }  => 30656102400
> range_primes = (32..1000).select { |n| n if n.gcd(modp31) == 1 }  => [37...997]
> pn1000 = base_primes.size + range_primes.size  => 168
> primes1000 = base_primes + range_primes  => [2, 3, 5...991, 997]
> modp997 = primes1000.reduce :*
 =>
195903406449990834312625081982063810461239723905893682238826053289686663163798706618519516487894823215962295591154360191491895297252152667282922829908526490233627313924040179391420109582613936349594714837571967216722434100671185162276611331351924888489899148921571883086798968751374395193389039680949055497503864071060338365866606835392010116359179000399044950652032997495429859931346698148053184740805812078911259100
> range_cnt = (998..1000000).sum { |n| n.gcd(modp997) == 1 ? 1 : 0 }  => 78330
> pn1000000 = range_cnt + primes1000.size  => 78498
```

Here also, using a **PGS** will greatly reduce the number of values checked in the ranges to create a much more efficient|faster search process, with P3's short sequence alone reducing the range values by 2/3.

Thus, the first 11 primes found the next 157 ($N = 10^3$); those first 168 found the next 78,330 ($N = 10^6$); and those first 78,498 primes can find the next 37,607,833,520 (I encourage you to continue the process for $N = 10^{12}$). With one more squaring for $N = 10^{24} > 2^{79}$ we're well past the 64-bit range, showing it's conceptually quite feasible to identify|count every prime within 64 bits using this process.

This example clearly shows the dynamic range affect of the square-law property. As the number of base primes increase they find a power-law increase in the number of additional primes, and subsequently they constitute a power-law decrease of their percentage of the total primes up to N. Thus, for $N = 10^3$ the base primes are 6.548% of the total; for $N = 10^6$, 0.214%; and for $N = 10^{12}$ they are 2.087e-04%.

Prime Generators and Riemann Hypotheses

Because it's theoretically possible to create a prime generator that can generate all the primes ≤ N we can then create an exact prime-counting function, i.e. $\pi(x)$ = Pn(x) + count(base primes).

To perform a sieve with a **suboptimal** PG we use the primes ≤ $\sqrt{x}$, to mark off their prime multiples. We know now that the exact minimum upper bound to compute $\pi(x)$ are the primes ≤ $\sqrt{x}$, which can construct an **optimal generator** which can directly identify and count exactly all the primes ≤ x.

An alternative expression of the Riemann Hypotheses (all Zeta zeroes lie on critical line x = 1/2 ± iy) is

$$\pi(x) = Li(x) + O(\sqrt{x}\ log\ x) \quad \text{or} \quad |\pi(x) - Li(x)| < K \cdot \sqrt{x} \cdot \log x$$

This expression invokes a picture, as shown in **Prime Obsession**, page 243 [19] of a sideways parabola symmetric about the x-axis bounding the growth curve of $\sqrt{x} \cdot \log x$. But this is the error term to *Li(x)*, which is an approximation to $\pi(x)$, which we now know can be computed exactly by some Pn(x), with the primes ≤ $\sqrt{x}$ for x. In fact, for the example of N = 1,000,000, we could also use the generator **P991**, with **rt_0 = 997**, which requires reducing the count for 997^2, but otherwise still provides an exact count of the primes ≤ 1,000,000. Backtracking for suboptimal PGs in this manner, we can also create an error term expression we would subtract from the Pn count to give an exact value for $\pi(x)$ for any PG.

Finally, as noted, the residues (canonical form) exist as **modular compliment pairs**, symmetric around the value **modpg div 2**. This is strikingly similar to the property that the Zeta zeroes come as **complex conjugates** (pairs) on the critical line, symmetric to the x-axis. We can also represent the residues in a manner that more mimic the Zeta zeroes complex conjugate forms, as shown below for P5.

1, 7, 11, 13 | 17, 19, 23, 29 <=> 1, 7, 11, 13 | -13, -11, -7, -1 mod 30

Now the **modular compliment pairs** sum to 0, vs 30, and lie on a line symmetric to the x-axis (x = 0). Thus, every PG can be written to be normalized around 0. This conjures a picture of **negative primes** balanced out by **positive primes**, on their respective residue tracks. If we use a physics metaphor of charged particles, the primes exist as **bonded modular pairs** (with zero net charge) that tend to attract (or repel) causing their spacing to be non-random. (See **Prime Obsession** chapter 18.)

It seems too coincidental for these similarities to randomly exist. So while Riemann employed the tools of complex analytic calculus to characterize the primes, there must exist a discrete analogue that can employ the tools of prime generator math to reveal and achieve similar results. But to paraphrase Riemann, proof of this is not necessary in the presentation of the major topic of discussion in this paper.

The Sieve of Zakiya (SoZ)

The SoZ [2][3] first computes the maximum number of prime candidates (pcs) up to N. To identify the non-primes, each prime ≤ sqrt(N) in a resgroup multiplies each member in its resgroup (including self). These rescnt products (first multiples) map uniquely to one residue track in some resgroup. From these values, every primenth resgroup (column) along each restrack (row) is a prime multiple and marked as such, to the end of the table, leaving the unmarked values in the table as the primes ≤ N.

SoZ Algorithm

To find all the primes ≤ N

1. choose a Prime Generator (PG)
2. determine for PG maxpcs = number of pcs ≤ N
3. create (boolean) array prms[maxpcs] to represent the value|location of each pc ≤ N
4. set prime = primes from r1 .. rn ≤ sqrt(N) (unmarked locations in prms)
5. set primestep = prime * rescnt
6. perform inner sieve loop with current prime value:
    - multiply prime by each pc value of the residue group it belongs to
    - for each product compute its location in pc table, and from there
    - successively mark each primestep location in prms as non-prime until end of list
7. repeat from 4, set prime to next prime (unmarked pc in prms list) ≤ sqrt(N)
8. count and/or numerate and store unmarked locations in prms as primes

Table 1. Residues 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

Table 1 depicts the SoZ using P5. ***Everything you need to know is encapsulated in this picture!***

Here for P5, we see all the possible integers (pcs) that can be prime ≤ 541. To find the non-primes, we start with 7, multiply all its resgroup (k = 0) members, getting all the first blue numbered cells on each restrack|row. From them, mark every 7th row column (blues), until end. Then use next prime 11, and start from 77 (11 * 7) and mark every 11th row column (reds), etc. (The table colors show the first prime value to mark each column.) We perform this process for all the primes ≤ sqrt(541) (7, 11, 13, 17, 19, 23). When the sieve process finishes, all the unmarked integers will be the primes ≤ 541.

The Segmented Sieve of Zakiya (SSoZ)

As with the SoZ, we conceptually process similar tables as Table 1 for a given prime generator. As N becomes larger the number of resgroups (cols) in the table increases. So instead of serially sieving a long array, we slice the table into chunks of resgroups and sieve out all the multiples of each prime from each slice, process the primes (count, display, or store them), then repeat with the next slice, until the full residues table is processed. This uses much less memory, is much more hardware resource efficient (especially for large N), and is inherently conducive to parallel processing techniques.

SSoZ Algorithm

To find all the primes ≤ N

1. choose a Prime Generator (PG)
2. set the maximum (bit|byte) size B for the segment (desirable to fit in hardware cache memory)
3. determine system parameters:
    - bprg, the number of bytes per residues group for the PG
    - KB, the number of residues groups of bprg size that fits into B bytes
    - B, reset the segment byte size to be just the KB * bprg bytes
    - seg[B], the segment array to have length of B bytes
    - Kmax, max number of residues groups for N
4. generate the r1..pn primes ≤ sqrt(N), and their pcnt number of primes
5. create and initialize nextp[rescnt * pcnt] array of the primes first multiples resgroups
6. perform segmented sieve for all Kmax residues groups:
    - determine the byte size for the current segment
    - perform segmented sieve of all primes ≤ sqrt(N) on the segment
    - count the number of primes found in the segment
    - display and/or store the primes (optional)
    - repeat until the Kmax residues groups processed
7. determine the total prime count and value of last prime ≤ N

A key benefit of the SSoZ is that it can be conceptualized in various forms to best meet a design's goal. This allows it to be implemented with different architectural structures to optimize its use for a given software environment and hardware system.

The key computational components to perform are:

- process to generate primes|cnt ≤ sqrt(N)
- process to parametize a PG
- process to initialize nextp array of first prime multiples
- process to perform the prime sieve on a segment
- process to control parallel processing
- main process to control sieve parameter setup and output display

Probably the most conceptually unique of these, and necessary to perform the SSoZ, is the purpose|role of the **nextp** array, and how it's created. What follows is an explanation of the purpose and role it plays in the operation of the SSoZ and how it's constructed.

The nextp array

nextp is a table of the resgroups for the first prime multiples for each prime ≤ sqrt(N) along each residue track. Table 3 is what this looks like for the P5 prime generator, for a number of primes.

Table 2. Residues Tracks Table for P5(541).

k	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
rt0	7	37	67	97	127	157	187	217	247	277	307	337	367	397	427	457	487	517
rt1	11	41	71	101	131	161	191	221	251	281	311	341	371	401	431	461	491	521
rt2	13	43	73	103	133	163	193	223	253	283	313	343	373	403	433	463	493	523
rt3	17	47	77	107	137	167	197	227	257	287	317	347	377	407	437	467	497	527
rt4	19	49	79	109	139	169	199	229	259	289	319	349	379	409	439	469	499	529
rt5	23	53	83	113	143	173	203	233	263	293	323	353	383	413	443	473	503	533
rt6	29	59	89	119	149	179	209	239	269	299	329	359	389	419	449	479	509	539
rt7	31	61	91	121	151	181	211	241	271	301	331	361	391	421	451	481	511	541

Table 3. nextp array values for P5.

rt	res	List of resgroup values for the first prime multiples – prime * (modk + ri) – for the primes shown.																	
		7	11	13	17	19	23	29	31	37	41	43	47	53	59	61	67	71	73
0	7	7	6	8	6	8	22	22	7	75	64	70	64	104	104	75	203	182	192
1	11	5	11	7	7	18	5	18	11	65	83	67	67	65	96	83	185	215	187
2	13	4	8	13	16	4	8	16	13	60	72	87	92	72	92	87	176	196	221
3	17	2	2	12	17	14	14	12	17	50	50	84	95	86	84	95	158	158	216
4	19	1	10	5	9	19	17	10	19	45	80	61	73	93	80	99	149	210	177
5	23	6	4	4	10	10	23	6	23	72	58	58	76	107	72	107	198	172	172
6	29	3	6	9	3	6	9	29	29	57	66	75	57	75	119	119	171	186	201
7	31	2	3	2	12	11	12	27	31	52	55	52	82	82	115	123	162	167	162

To construct Table 3, each prime in Table 2 multiplies each of its regroup members, whose products are other Table 2 values. Their row|col cell locations are entries into **nextp**. Thus starting with first prime 7:

$$7 * [7, 11, 13, 17, 19, 23, 27, 29, 31] = [49, 77, 91, 119, 133, 161, 203, 217]$$

We see from Table 2, 49 occurs in resgroup k = 1 for residue value 19, which is residue track 4, rt4. Similarly for the remaining multiples of 7, we see their placement in the table. Repeating this process for each prime, we compute their first multiples, then determine their resgroup value for each restrack.

These first prime multiple locations are used to start marking off successive prime multiples in Table 2 along each restrack|row. The SoZ computes each prime's multiples *on the fly* once and doesn't need to store them for later use. The SSoZ computes an initial **nextp** to process the first segment, whose values are updated at the end of each segment to indicate the first prime multiples in the next segment(s).

The creation of **nextp** is the only real computationally intense SSoZ component. For the parallel SSoZ implementation presented herein, each thread only computes **nextp** for the twin pair residues it is processing, i.e just two rows from Table 3. A faster implementation could theoretically be achieved by precomputing Table 3 for a number of primes, then computing at run time any more that are needed. (This would have to be done for each PG that may be used, and would require much more memory than the present implementation, but if speed is the ultimate objective this could be a consideration.)

Computing nextp for SSoZ

In the SoZ we multiplied **r** by some **ri** and **(r * ri) mod modpg** fell on some restrack value **rt**, which was one starting point to mark off that prime's multiples. Now we want to multiply **r** by the **ri** that makes the **(r * ri)** product be on a specific **rt**, for each prime. This requires a little more modular math.

If for some **ri**, **(r * ri) mod modpg = rt**, then to find the **ri** that maps each **r** to a specific **rt** we do:

$$
\begin{aligned}
r * ri &= rt && \bmod modpg \\
(ri * r) / r &= rt / r && \bmod modpg \\
ri * r * r^{-1} &= rt * r^{-1} && \bmod modpg \\
ri * 1 &= rt * r^{-1} && \bmod modpg \\
ri &= rt * r^{-1} && \bmod modpg
\end{aligned}
$$

Now **kn = (r * ri) / modpg**, and **k = prime / modpg**, so again: **nextp[j] = k * (prime + ri) + kn**
If **r_inv** is a prime's residue inverse, and **rt** the restrack we want, then: **ri = rt * r_inv mod modpg**

We now know all the math needed to compute **nextp**, which is done by `nextp_init` in each thread. Below is prototype code for generating **nextp** for twin primes, where the specific **rt** values **r_lo** and **r_hi** are the upper and lower residues for each twin pair restrack. With no loss of generality, this technique can be applied to construct **nextp** for any architecture for any number of specified restracks.

```
proc nextp_init(indx)
  (row_lo, row_hi) = (0, pcnt) # nextp addrs for upper|lower twin pair
  nextp = newSeq[uint](pcnt*2) # create 1st mults array for twin pair
  r_hi = restwins[indx]        # upper twin pair residue value
  r_lo = r_hi - 2              # lower twin pair residue value
  for j, prime in primes:      # for each prime r1..sqrt(N)
    k = prime div modpg        # determine the resgroup it's in
    r = prime mod modpg        # and its residue value
    r_inv = modinv(r)          # and its residue inverse
                               # then compute its nextp val for the pair
    ri = (r_lo * r_inv) mod modpg   # compute the ri for lower twin pair
    nextp[row_lo + j] = k * (prime + ri) + (r * ri) div modpg

    ri = (r_hi * r_inv) mod modpg   # compute the ri for upper twin pair
    nextp[row_hi + j] = k * (prime + ri) + (r * ri) div modpg

  return nextp
```

Initializing **nextp** takes increasingly more time for bigger inputs. To speed it up I precompute an array **resinvrs** of the residues inverses, e.g. residues[i] * resinvrs[i] mod modpg = 1. A faster customized **modinv** function (using the special residues inverse properties) could be created for run time use to eliminate it, to reduce the executable size, as a possible implementation option for use on low memory systems, in lieu of optimal speed.

Twin Primes Generation

Let's now construct a process to find twin primes ≤ N with a segmented sieve, using our P5 example. Twin primes are consecutive odd integers that are prime, the first two being [3:5], and [5:7]. Thus from our original P5 residues table, we use just the consecutive pc restracks, whose residues table is below. A twin prime occurs in the table only when both twin pair pc values in a column are prime (not marked).

Table 4. Twin Primes Residues Tracks Table for P5(541).

k	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
rt1	11	41	71	101	131	161	191	221	251	281	311	341	371	401	431	461	491	521
rt2	13	43	73	103	133	163	193	223	253	283	313	343	373	403	433	463	493	523
rt3	17	47	77	107	137	167	197	227	257	287	317	347	377	407	437	467	497	527
rt4	19	49	79	109	139	169	199	229	259	289	319	349	379	409	439	469	499	529
rt6	29	59	89	119	149	179	209	239	269	299	329	359	389	419	449	479	509	539
rt7	31	61	91	121	151	181	211	241	271	301	331	361	391	421	451	481	511	541

We see from the table the twin pair residue tracks for [11:13] has 10 twin primes ≤ 541, [17:19] has 6, and [29:31] has 7. Thus, the total twin prime count ≤ 541 is 23 + [3:5] + [5:7] = 25, with the last being [521:523]. Twin primes are usually referenced to the mid (even) number between the upper and lower consecutive odd primes pair, e.g. the last (largest) twin pair ≤ 541, [521:523], is written as 522 ± 1.

P5's 3 twin pair residues can identify every twin prime that exists, and will search through 20% (6/30) of all integers to do so. Using the next larger SP PG P7, with 15 twin pair residues, reduces the number space to 14.2857% (30/210) of all integers. Thus, as the PGs become larger they have more twin pair residues and reduce the number space accordingly. At the time of writing, the largest known twin prime is $2996863034895 \cdot 2^{1290000} \pm 1$ [8] (2016), which resides on the restracks P5[29:31] and P7[29:31].

Fig 3. Prime Generators Twin Primes Characteristics

P_n	P3	P5	P7	P11	P13	P17
modulus (modpg)	6	30	210	2310	30030	510510
residues count (rescnt)	2	8	48	480	5760	92160
twin pairs count	1	3	15	135	1485	22275
twin count factoring	1	3	$3 \cdot 5$	$3^3 \cdot 5$	$3^3 \cdot 5 \cdot 11$	$3^4 \cdot 5^2 \cdot 11$
% of number space	33.333	20.000	14.286	5.844	4.945	4.363

We see P3 provides no advantage for twin primes, using the same number space to find all the primes. And as the SP prime generators get bigger their number space size decrease, but at a smaller rate. Thus, it would be good to be able to adaptively select the *best* one to use based on the input size.

Coding Twin Primes SSoZ

Using our example to find the twin primes ≤ 541 with P5, let's see how to processes the first twin pair [11:13], in our example. This process is the same for every thread's twin pair for any prime generator. In the code, `twin_sieve` performs the sieve for each twin pair in a thread. In our example **Kmax = 18**.

The code selects the max segment size, which here I'll set to **KB = 6**. Thus, the **seg** array will represent 6 resgroups. Below I've taken the twin pair table for [11:13] and separated it into 3 segment slices of 6 resgroups. Underneath it is what each **seg** array will look like after processing each segment.

k	0	1	2	3	4	5
rt11	11	41	71	101	131	161
rt13	13	43	73	103	133	163

6	7	8	9	10	11
191	221	251	281	311	341
193	223	253	283	313	343

12	13	14	15	16	17
371	401	431	461	491	521
373	403	433	463	493	523

k	0	1	2	3	4	5
seg	0	0	0	0	1	1

0	1	2	3	4	5
0	1	1	0	0	1

0	1	2	3	4	5
1	1	0	0	1	0

Performing twins_sieve

As part of the SSoZ setup, the **primes** array of the primes ≤ 541 is created by `sozpg` (with **pcnt = 6)**. It's then used to initialize **nextp**, and perform the segmented sieve in `twins_sieve`, for each twin pair.

To begin the process, `nextp_init` initializes **netxp** for that thread's twin pair residues. Using data from Table 3, the initial **nextp** is shown below. We also create|init the **seg** array to be all primes (0). Thus, before segment processing starts, the **nextp** and **seg** array are initialized as shown below for [11:13].

j	0	1	2	3	4	5
primes	7	11	13	17	19	23

Initial nextp[11:13]						
j	0	1	2	3	4	5
rt_11	5	11	7	7	18	5
rt_13	4	8	13	16	4	8

k	0	1	2	3	4	5
seg	0	0	0	0	0	0

```
proc twins_sieve(Kmax, indx)
  (sum, Ki, Kn) = (0, 0, KB)
  (hi_tp, k_hi, upk) = (0, 0, 0)
  r_hi = restwins[indx]
  seg = newSeq[uint8](KB)
  nextp = nextp_init(indx)
  while Ki < Kmax:
    if KB > (Kmax–Ki): Kn = (Kmax–Ki)
    for b in 0..Kn-1: seg[b] = 0
    for j, prime in primes:

      k = nextp[j]
      while k < Kn:
        seg[k] = 1
        k += prime
      nextp[j] = k - Kn

      k = nextp[pcnt + j]
      while k < Kn:
        seg[k] = 1
        k += prime
      nextp[pcnt + j] = k - Kn

    for k in 0..Kn-1:
      if seg[k] == 0: sum.inc
    Ki += KB
```

We now start the sieve loop. For each prime in **primes** at index **j**, **nextp[j]** for each twin pair gives the first resgroup **k** in **seg** to begin marking that prime's multiples, by incrementing **k** by the prime's value. When **k** exceeds **seg**'s last index value, either initially in **nextp** or after incrementing, it's reduced by **seg**'s size (here 6), and that value is stored back in **nextp[j]** for each prime, updating **nextp** to the first prime multiple locations **k** in the next segment(s).

Presented here is the output for each segment for **nextp** and **seg**, produced by `twins_sieve`. (It's purely coincidental here that the index size for **primes**|**nextp** is the same as the segment length.)

Start for Segment 1 nextp[11:13]						
j	0	1	2	3	4	5
rt_11	5	11	7	7	18	5
rt_13	4	8	13	16	4	8

seg 1

k	0	1	2	3	4	5
seg	0	0	0	0	1	1

Start for Segment 2 nextp[11:13]						
j	0	1	2	3	4	5
rt_11	6	5	1	1	12	22
rt_13	5	2	7	10	17	2

seg 2

k	0	1	2	3	4	5
seg	0	1	1	0	0	1

Start for Segment 3 nextp[11:13]						
j	0	1	2	3	4	5
rt_11	0	10	8	12	6	16
rt_13	6	7	1	4	11	19

seg 3

k	0	1	2	3	4	5
seg	1	1	0	0	1	0

Start for Segment 4 nextp[11:13]						
j	0	1	2	3	4	5
rt_11	1	4	2	6	0	10
rt_13	0	1	8	15	5	13

It will be very instructive to perform the code to verify these results for yourself. After sieving ends, `twins_sieve` determines the true twins sum for the largest twin ≤ N, which is performed as follows.

After performing each segment's sieve, we count the twins locations in **seg**, and add it to the running total **sum**. Then starting from the end of **seg**, we backtrack to find **upk**, the resgroup of the largest twin in the current segment, and use it to form **hi_tp**, its total table regroup value. But first we save **hi_tp** from the previous segment in **k_hi**. When we finish the last segment, we numerate **hi_tp** for its largest twin. We then check if that upper twin prime is > N. If not we store it, and the twins sum up to it, in **lastwins** and **cnts**. If its > N, from **upk** we reduce the **sum** for it and backtrack to find a smaller twin, and its **sum**, that's in range. If none in range in the final segment, we use **k_hi** to numerate the largest twin from the previous segment, and store it's sum value. We need to do this to find the last twin prime ≤ N for the twin pair to accurately determine the sum, and we also get the largest twin prime value for free in the process.

```
  cnt = 0
  for k in 0..Kn - 1:
    if seg[k] == 0: cnt.inc
  if cnt > 0:
    sum += cnt
    for k in 1..Kn:
      if seg[Kn - k] == 0:
        upk = Kn - k; break
     k_hi = hi_tp
     hi_tp = Ki + upk
  Ki += KB

hi_tp = hi_tp * modpg + r_hi

if hi_tp > num:
  prev = true
  for k in 0..upk:
    if seg[upk - k] == 0:
      if hi_tp <= num:
        prev = false
        break
      sum.dec
    hi_tp -= modpg

  if prev: hi_tp = r_hi > num ?
    0 : k_hi * modpg + r_hi
lastwins[indx] = hi_tp
cnts[indx] = sum
```

To visualize this from Table 4, for N = 541 we see for [11:13] its largest twin prime occurs in the last resgroup in its last segment, but no backtracking or **sum** reduction is needed since its < 541. For [17:19], it has no last (6 cols) segment twins, so **hi_tp** stays set from the previous segment and is used as the largest twin value|sum for it. Finally for [29:31], its last twin is in the final segment but its value is < 541, so no backtracking is needed.

Now let's analyze Table 4 for N = 420. Here **Kmax** is 14 resgroups (k = 13). For twin pair [11:13] it has 2 twin primes in its final segment > 420, so it backtracks the sum by 2, and since no other twin in the final segment is in range it uses the last val|sum from the previous segment (311:313). For [17:19], again there are no twin primes in its final segment and it too uses the values from the previous segment. Now for twin pair [29:31], the last segment twin upper prime, 421, is > 420 and outside the range (both primes must be inside) so we reduce **sum** for it. In backtracking, we find no other twins in the final segment, and thus use the largest twin value|sum from the previous segment.

Main twinprimes_ssoz

Main routine `twinsprimes_ssoz` has a pretty straightforward job. It receives input values, makes sure **num** is odd, calls `selectPG` with it to choose the PG and seg size factor **Bn**, then computes the max **seg** size **KB**. It also parametizes **num**, and finds the **Kmax** residues table size. It then calls `sozpg` to find the **pcnt** and **primes** r1..rn ≤ sqrt(num), and initializes **twinscnt** for the PG. It then starts a loop to invoke `twins_sieve` in parallel for every twin pair, to perform the segmented sieve. When they all finish, the **sum**s in **cnts** for each twin pair are added to **twinscnt**, and the largest twin prime in **lastwins** is found and stored in **last_twin**. These values, timing results, and other information, is then displayed.

I also generate and display some diagnostic data, though while not necessary, was helpful in debugging, and it provides a basis to assess some numerical characteristics of the process. These can be eliminated or altered to satisfy users preferences.

Implementation Details and Considerations

The Nim source code **twinprimes_ssoz.nim** in this paper is also available (and updates) at [C1].

Compile time vs run time actions

To maximize speed, at compile time **genPGparameters** is used to generate a set of constant SP PG parameters, which includes for each PG its: 1) **modpg** value, 2) **residues** array, 3) **rescnt** value, 4) **restwins** array of upper twin pairs residues, 5) **pairscnt** value, and 6) **resinvrs** inverses array. (For Nim, increasing a compiler config variable, and rebuilding the compiler, was needed to compile P17.)

At run time **selectPG** chooses the PG parameters to use, based on profiling of input values vs segment sizes. This can be greatly improved, as larger values took more time to test, so at best for values > 10^{12} the settings used in the code are approximations of optimal segment sizes and PG cross over values. I'm sure better use of cache (fast system) memory, et al, will also likely increase performance.

The Nim binaries compile to 1.9MB+ (most of it for P17). To make it smaller the **resinvrs** array, which is only used in **nextp_init**, can be eliminated, and the inverses computed at run time. However, a customized **modinv**, using the special inverse residue properties, is advisable, for better performance.

Computing Resgroup Values

To compute a pc's resgroup value k in a resgroup table, we would normally do: **k_pc = pc div modpg**. However, in using a PG's *functional form*, where a pc with $\mathbf{r_0 = 1}$ is placed at the end of its resgroup, we have to slightly adjust the operation. For P5, 31 is the last pc in the first resgroup, i.e. k = 0, and dividing it by 30 would give k = 1. So to compute its correct value of k = 0, I just subtract 2 from every pc value before dividing and thus: k = (31 – 2) div 30 = 29 div 30 = 0. (I could have used any value from 2..7 for P5 to subtract with.) Thus in the code you'll see, **k = (prime - 2) div modpg**, etc.

Computing Residue and Restrack Values

Similarly, to find a pc's residue value we can always just do, **k_pc = pc mod modpg**. However, using the *functional form* I need $\mathbf{r_0}$ to be 31 and not 1. Thus for the code for **nextp_init**, I do the same trick as above so, **r = (prime - 2) mod modpg + 2**, where the + 2 gives back the correct residue value. Some languages provide a **divmod** instruction e.g. **k, r = (prime - 2) divmod modpg**, and then **r += 2**. (Since we first find prime's k, then **r = prime - modpg * k** always provides its correct residue value too.)

In **nextp_init** and **sozpg** I need to know the restrack|row a pc value is on. Conceptually for a given residue **r**, **rt = residues.index(r)**. For a fast and universal implementation for this I create the position array **pos**, which contains the **residues** index **rt** for each **r**. A hash|dictionary|associative array can also be used, but an array is more standard across languages, and faster too. As implemented in **selectPG**, because I first do these **r = (pc - 2) mod modpg** operations anyway, I use **(r - 2)** as the **r** inputs for **pos**, to alleviate adding 2 each time to r, to reduce source code noise. Both ways work, with no discernible performance or code size difference when compiled.

Static Typing Issues

The code is designed to work on 64 bit systems, and take 64 bit inputs. Nim's universal numeric types **uint** and **int**, default to the system bit size (whereas in D, they are 32 bit values). In translating, be aware of intermediate value sizes. The Nim code required copious explicit type recasting to compile.

Manually Outsmarting the Compiler

Modern compilers (gcc, clang, llvm) have now decades of knowledge embedded in them, that allow them to produce highly optimized machine code from source code. A big benefit to programmers of this is they can focus on writing readable code and let the compiler best determine how to implement it. However in at least two instances in the Nim (and D version) code, writing the source code differently in `twins_sieve` had noticeable performance affects.

The first instance occurs in the inner sieving loop, and exists for this single slight coding difference:

```
          seg[k] = 1                          seg[k] = seg[k] or 1
movb   $0x1,(%rbx,%rdi,1)              orb  $0x1,(%rbx,%rdi,1)
```

In the first instance a **movb** instruction is compiled while in the second an **orb** is compiled, but the second form is noticeably faster. This is what is used in the reference code. (See discussion at [18].)

The second instance involves more extensive code, and since it happens at the end of the routine its performance affect is less drastic, and doesn't really reveal itself until using large inputs.

```
  var cnt = 0
  for k in 0..Kn - 1: (if seg[k] == 0: cnt.inc)
  if cnt > 0:
    sum += cnt
    for k in 1..Kn:
      if seg[Kn - k] == 0: upk = Kn - k; break
    ki_hi = hi_tp
    hi_tp = Ki + upk
  Ki += KB

hi_tp = hi_tp * modpg + r_hi
if hi_tp > num:
    prev = true
    for k in 0..upk:
      if seg[upk - k].int == 0:
        if hi_tp <= num:
          prev = false; break
        sum.dec
      hi_tp -= modpg
    if prev: hi_tp = if r_hi > num: 0
                     else: k_hi * modpg +  r_hi
lastwins[indx] = hi_tp
cnts[indx] = sum
```

```
  var cnt = 0
  for k in 0..Kn - 1: (if seg[k] == 0: cnt.inc)
  if cnt > 0:
    sum += cnt
    for k in 1..Kn:
      if seg[Kn - k] == 0: upk = Kn - k; break
    lastwins[indx] = hi_tp
    modk = (Ki + upk) * modpg
    hi_tp = modk + r_hi
  Ki += KB

if hi_tp > num:
    for k in 0..upk:
      if seg[upk - k].int == 0:
        if hi_tp <= num:
          lastwins[indx] = hi_tp; break
        sum.dec
      hi_tp -= modpg
else: lastwins[indx] = hi_tp
cnts[indx] = sum
```

Both Nim code snippets compile to the same size binary but the right snippet, though shorter, is a wee bit slower. So here, breaking the code into more pieces seems to provide more optimization options. It is an interesting question to answer which is faster using different languages, hardware, and compilers.

Segment Implementations

In the reference code the **seg** segments are byte arrays as it's the smallest addressable size memory unit for most cpus. Since only the **lsb** of each byte is used boolean arrays can also be used. In Nim (and D) byte arrays tested faster. A fast **bitarray** library may be faster, by allowing larger **seg** sizes to fit in the fastest system (usually cache) memory. I was not able to test for this.

A segment can also be conceptualized as a **set** of the unique **k** resgroup values for the prime multiples. Subtracting their number from the segment's length gives the number of twins. I implemented a version using Nim's **sets** to do this, but (as expected) it was much slower than arrays, but it did work.

There is also room for possibly vast performance improvements in the implementation structure for the segmentation process, as used in [10] and [11], especially in tuning for the use of cache for different hardware and memory architectures, as done in [11].

Sieving Over Ranges

In [1] and [5] I show and explain how to perform the S/SoZ over a range that starts at any position in a residues table. `nextp_init` would need modification to determine the first resgroup values for the beginning of the range (not from the beginning of the table). Also `twin_sieve` would have to find and account for the smallest in range prime|sum, similarly done for the largest prime|sum in the range.

Architectural Options

There is great deal of architectural flexibility that can be used to decide on a *best* implementation. I originally created global **nextp|seg** arrays in main routine, immediately after finding the primes ≤ N. Each thread then used the restracks for it, and just updated them in the process. However by doing this, the sieving process couldn't start until all of **nextp** was created, and the memory was slower to access. By pushing their creation into each thread, they could start faster, increasing macro performance, with the added benefit that overall memory use decreased, because as each thread finished it's memory was released by Nim's garbage collector (gc). (In the first case, compiling with gc off was quicker). It's important to note though, you need to be able to recover memory for each thread or system memory will steadily be used up, which you can see happening using a tool like **htop**, especially for large N.

It's also possible to initialize **nextp** when performing `sozpg`, as the same multiplications are done in it. But **nextp** would then have to be global again, and this may be too slick for what it's worth.

Another possibility is to use the results from `sozpg` as the outputs for N below some threshold value, instead of performing the segmented sieve with its results. Again in the big picture, it's probably not worth it just to save a few (m, u, n)secs at the low end, while increasing code size and complexity.

Performance Comparisons

To assess the SSoZ performance I compared it to **primesieve** (https://primesieve.org/) [11], an open source project which states on its homepage: "*primesieve is a free software program and C/C++ library that generates primes using a highly optimized sieve of Eratosthenes implementation.*" It is mature and continually updated, and times here are produced compared to version 7.2 (latest atow). I used the console version and ran it as: **$ ./primesieve -c2 <number>**

Testing System

Tests were run on a System76, Gazelle (gaze 10, 2016) laptop, Intel Core i7-6700HQ, 2.6 – 3.5 GHz, 4 cores, 8 threads, 32K L1|256K L2 cache, with 16 GBs of system memory. PCLinuxOS-2014 KDE 64-bit was the host Linux OS. The binaries for **twinprimes_ssoz** were generated in a VirtualBox instance of Manjaro KDE-64 using **gcc 8.2.1**, and run on the host OS to use the full 8 threads.

Software

The reference **twinprimes_ssoz** implementation was written in **Nim** (https://nim-lang.org/)[16]. Nim's homepage describes it as: "*Nim is a systems and applications programming language. Statically typed and compiled, it provides unparalleled performance in an elegant package.*" The latest version (atow) is 0.19.0 (from 0.18.0), however I found the results using 0.18.0 to be a *smidgen* faster and the binaries smaller, so all times listed here are from using Nim 0.18.0.

To run the tests, I used a *quiet* system, turning off laptop, rebooting, and ran the programs in a terminal with nothing else open except a spreadsheet to record the times. For values < 1 trillion (1×10^{12}) I ran multiple tests, and used the lowest time (usually there were just small deviations). For larger values the I ran fewer iterations, as times became longer, and the variation in significant digits became smaller.

Fig 4. twinprimes_ssoz vs primesieve times in seconds.

N	Last Twin ± 1	Number of Twins	twinprimes_ssoz	primesieve
1×10^9	999,999,192	3,424,506	0.042	0.049
5×10^9	4,999,999,860	14,618,166	0.219	0.248
1×10^{10}	9,999,999,702	27,412,679	0.398	0.525
5×10^{10}	49,999,999,590	118,903,682	2.331	2.878
1×10^{11}	99,999,999,762	224,376,048	4.476	5.992
5×10^{11}	499,999,999,062	986,222,314	26.578	33.494
1×10^{12}	999,999,999,960	1,870,585,220	57.895	70.801
5×10^{12}	4,999,999,999,878	8,312,493,003	383.231	406.676
1×10^{13}	9,999,999,998,490	15,834,664,872	853.453	875.369

It should be noted this is not an exact apples-to-apples comparison because **twinprimes_ssoz** tracks the largest segment twin prime to output, while **primesieve** doesn't, but it does provides a visual progress indicator, which is good to have for large values (which I couldn't figure out how to do in Nim). These differences are most likely negligible, and don't affect the overall performance figures shown here.

Performance Notes

The overall macro performance of **twinprimes_ssoz** comes from its ability to adaptively select the *best* prime generator to use based on the input size. Because generators have the same structure, and operate the same, they only differ by their operating parameters. In an ideal (on paper) world you could just pick as large a generator as you want, but as you do they have more residues, which means they need to use more threads and memory. So hardware limitations affect which are physically realizable to use. However, using specialized hardware, e.g. GPUs, et al, make larger PGs more realizable attractive.

In my original paper [1] (2014) I used C++ for SSoZ coding (after development in Ruby) because it was the main compiled language with a developed parallel processing environment (OpenMP [17]) my OS supported, and I could get documentation on, and I saw examples on how to use. The C++ code in that paper leave a lot to be desired.

Since that time period, a slew of other compiled languages with true parallel processing capabilities have matured to become useful. I played around with some on-and-off, as my time and interest and their utility for me provided. I finally settled in on Nim, and stuck working with it, to produce this working reference implementation of **twinsprimes_ssoz**. I believe as Nim matures as a language it's entirely likely to be able to construct code in it to run faster.

I *discovered* and used Nim because: it's statically typed (causing me lots of problems reconciling type differences, especially coming from Ruby), it has a simple|usable parallel program paradigm, the code is easy to read (Python like syntax) with low code noise (braces {}, etc), and creates highly performant C code to use with either gcc|clang. It's still young(ish), with lots of development decisions to make and issues to resolve (feature stability, and lack of documentation and examples being big), but for my purposes, it was a *good enough* tool to implement this algorithm.

It would be extremely interesting to see other language implementations, and their performance results. In particular, I would like to see an *expertly done* C++ and Rust versions, as they claim the same coding space of being *system* (low level) programming languages. It would also be intriguing to see Functional and Object Oriented programming implementations.

Ultimately, the highest performing software implementations will be for GPUs, or other massive cores hardware. The algorithm, and its components, are so simple, it would surely be easier to implement on them than the average video game currently is.

Conclusion

Prime Generators are very simple mathematical expressions that structurally limit the integer space domain that contains primes. Most of their properties are discernible by visual observation of their residues tables. These contain the minimal set, and number, of prime candidate integers that can be prime up to some N, or within a range, for a given prime generator.

Their unique mathematical properties provide the basis for creating a small simple set of computational components for finding primes. These can be used to create very efficient, fast, and scalable prime sieves, with suboptimal generators. They can also create optimal generators, which can directly identify all primes ≤ N exactly. They can also characterize various types of primes. For example, I show that Mersenne Primes = 2^p - 1, p prime, all have the form $M_p = 6*k + 1$ or $4*k + 3$, et al, for some k.

In this paper I focused primarily on a process to generate **twin primes**, but its components and techniques are applicable to other subsets. The same code can find **cousin primes** (primes that differ by four) and **sexy primes** (primes that differ by six) by merely changing the residue pair (restrack) values. With slightly more modification it can also easily identify **triplets** and **quadruplets**, et al.

Also I show prime generators have applications to various fields in pure Number Theory. I show how to create an exact Prime-counting Functions Pn(x), and its implication to the Riemann Hypotheses. Another implication of Pn(x) is it establishes there are an infinite number of twin primes, and other k tuples. Additional PG properties provide revealing information of other prime characteristics, such as their distribution, frequency, spacing, et al. Their study also provides the basis for alternative analytic techniques to understanding these prime characteristics.

The S/SoZ differs from other methods, such as the Sieve of Eratosthenes [9], by working strictly within a PG's number space, as no other values outside it are relevant. It also enables parallel processing along a PG's residue tracks to mark prime multiples, making them particularly conducive to implement on multi-cores|threads hardware. Their structure is also conducive to distributed systems implementations, such as used for the Grand Internet Mersenne Prime Search (GIMPS) [12].

Greater possible performance can be achieved through developing hardware assisted architectures which can be replicated to accommodate whatever sized generator desired. Because the components are so simple this is very easy to do using small single board computers (sbc) and/or alterable hardware (field programmable arrays, etc) or specifically designed parallel processors such as [14]. The ultimate hardware architecture would be a full custom design, with lots of fast memory and copious independent computing elements, to parallel process individual residue tracks within a system-on-chip (soc) design.

Hopefully the methodology shown here using prime generators will inspire others to investigate using them, in various forms and systems, to advance understanding the properties and characteristic of primes, and the means to create faster and more efficient methods for finding and counting them.

References

[1] The Segmented Sieve of Zakiya (SSoZ) , Jabari Zakiya, 2014
http://www.scribd.com/doc/73384039/Ultimate-Prime-Sieve-Sieve-Of-Zakiya
https://www.academia.edu/7583194/The_Segmented_Sieve_of_Zakiya_SSoZ
[2] Ultimate Prime Sieve – Sieve of Zakiya, Jabari Zakiya, 2008
http://www.scribd.com/doc/73384039/Ultimate-Prime-Sieve-Sieve-Of-Zakiya
[3] The Sieve of Zakiya, Jabari Zakiya, 2008
http://www.scribd.com/doc/73385696/The-Sieve-of-Zakiya
[4] Improved Primality Testing and Factorization in Ruby revised, Jabari Zakiya, June 2013
http://www.scribd.com/doc/150217723/Improved-Primality-Testing-and-Factorization-in-Ruby-revised
[5] PRIMES-UTILS HANDBOOK, Jabari Zakiya, January 2016
https://www.scribd.com/document/266461408/Primes-Utils-Handbook
[6] An Introduction to Prime Number Sieves, Jonathan Sorensen, January 1990,
http://minds.wisconsin.edu/handle/1793/59248, TR909.pdf.
[7] primegen.c, Daniel J. Bernstein (DJB), http://cr.yp.to/primegen.html
[8] Twin Prime,https://en.wikipedia.org/wiki/Twin_prime
[9] Sieve of Eratosthenes, https://en.wikipedia.org/wiki/Sieve_of_Eratosthenes
[10] Tomás Oliveira e Silva, http://sweet.ua.pt/tos/software/prime_sieve.html
[11] Kim Walisch, https://primesieve.org/
[12] Grand Internet Mersenne Prime Search, http://mersenne.org/
[13] Optimizing_cpp.pdf, Anger Fog, http://agner.org/optimize/#manuals
[14] Parallela Board, Adapteva, http://www.adapteva.com/, http://www.parallella.org/
[15] Ruby language, https://www.ruby-lang.org/en/
[16] Nim language, https://nim-lang.org/
[17] OpenMP, https://www.openmp.org/
[18] D forum post, https://forum.dlang.org/post/rfoplhydaavsdilgemsp@forum.dlang.org
[19] Prime Obsession, John Derbyshire, 2003, https://en.wikipedia.org/wiki/Prime_Obsession
[20] Prime-counting Function, https://en.wikipedia.org/wiki/Prime-counting_function
[21] Riemann Hypotheses, https://en.wikipedia.org/wiki/Riemann_hypothesis

Code and Projects

[C1] twinprimes_ssoz.nim, https://gist.github.com/jzakiya/6c7e1868bd749a6b1add62e3e3b2341e

```
#[
 This Nim source file is a multiple threaded implementation to perform an
 extremely fast Segmented Sieve of Zakiya (SSoZ) to find Twin Primes <= N.

 Based on the size of input value N, it dynamically selects the best PG to
 use from PGs P5, P7, P11, P13, P17 and then sets the optimum segment size.

 This code was developed on a System76 laptop with an Intel I7 6700HQ cpu,
 2.6-3.5 GHz clock, with 8 threads, and 16GB of memory. I suspect parameter
 tuning may have to be done on other hadware systems (ARM, PowerPC, etc) to
 achieve optimum performance on them. It was tested on various Linux 64 bit
 distros, native and in Virtual Box, using 8 or 4 threads, or 16|4GB of mem.

 To compile for nim versions <= 0.19.0 (latest at time of writing) do:
 1) in file: <path to here>/nim-0.19.0/compiler/vmdef.nim
 2) set variable: MaxLoopIterations* = 1_000_000_000 (1 Billion or >)
 3) then rebuild system: ./koch boot -d:release
 Compile with --cc:gcc and -cc:clang to compare which is faster.

 $ nim c --cc:gcc --d:release --threads:on twinprimes_ssoz.nim

 Then run executable: $ ./twinprimes_ssoz <cr>, and enter value for N.
 Or alternately: $ echo <number> | ./twinprimes_ssoz
 As coded, input values cover the range: 13 -- 2^64-1

 This source file, and updates, will be available here:
 https://gist.github.com/jzakiya/6c7e1868bd749a6b1add62e3e3b2341e

 Related code, papers, and tutorials, are available here:
 https://www.scribd.com/doc/228155369/The-Segmented-Sieve-of-Zakiya-SSoZ
 https://mega.nz/#!yJxUEQgK!MY9dwjiWheE8tACtEeS0szduIvdBjiyTn4O6mMD_aZw
 https://www.scribd.com/document/266461408/Primes-Utils-Handbook

 Use of this code is free subject to acknowledgment of copyright.
 Copyright (c) 2017-19 Jabari Zakiya -- jzakiya at gmail dot com
 Version Date: 2019/1/8

 This code is provided under the terms of the
 GNU General Public License Version 3, GPLv3, or greater.
 License copy/terms are here:  http://www.gnu.org/licenses/
]#

import math                   # for sqrt, gcd, mod functions
import strutils, typetraits   # for number input
import times, os              # for timing code execution
import osproc                 # for getting threads count
import threadpool             # for parallel processing
import algorithm              # for sort
{.experimental.}              # required to use 'parallel' (<= 0.19.x)

proc modinv(a0, b0: int): int =
  ## Compute modular inverse a^-1 of a to base b, e.g. a*(a^-1) mod b = 1.
  var (a, b, x0) = (a0, b0, 0)
  result = 1
  if b == 1: return
  while a > 1:
    result -= (a div b) * x0
    a = a mod b
    swap a, b
    swap x0, result
  if result < 0: result += b0
```

```
proc genPGparameters(prime: int): (int, int, int, seq[int], seq[int], seq[int]) =
  ## Create constant parameters for given PG at compile time.
  echo("generating parameters for P", prime)
  let primes = [2, 3, 5, 7, 11, 13, 17, 19, 23]
  var modpg = 1
  for prm in primes: (modpg *= prm; if prm == prime: break)  # PG's mdoulus

  var residues: seq[int] = @[]    # generate PG's residue values here
  var (pc, inc) = (5, 2)          # use P3's PGS to reduce pcs to check
  while pc < modpg div 2:         # find a residue, then modular complement
    if gcd(modpg, pc) == 1: residues.add(pc); residues.add(modpg - pc)
    pc += inc; inc = inc xor 0b110
  residues.sort(system.cmp[int]); residues.add(modpg - 1); residues.add(modpg + 1)
  let rescnt = residues.len       # PG's residues count

  var restwins: seq[int] = @[]    # extract upper twinpair residues here
  var j = 0
  while j < rescnt - 1:
    if residues[j] + 2 == residues[j + 1]: restwins.add(residues[j + 1]); j.inc
    j.inc
  let twinpairs = restwins.len    # twinpairs count

  var inverses: seq[int] = @[]    # create PG's residues inverses here
  for res in residues: inverses.add(modinv(res, modpg))

  result = (modpg, rescnt, twinpairs, residues, restwins, inverses)

# Generate at compile time the parameters for PGs.
const parametersp5  = genPGparameters(5)
const parametersp7  = genPGparameters(7)
const parametersp11 = genPGparameters(11)
const parametersp13 = genPGparameters(13)
const parametersp17 = genPGparameters(17)

# Global parameters
var
  pcnt = 0             # number of primes from r1..sqrt(N)
  num = 0'u64          # adjusted (odd) input value
  twinscnt = 0'u64     # number of twinprimes <= N
  primes: seq[int]     # list of primes r1..sqrt(N)
  KB = 0               # segment size for each seg restrack
  cnts: seq[uint]      # hold twinprime counts for seg bytes
  lastwins: seq[uint]  # holds last twin <= num in each thread
  pos: seq[int]        # convert residue val to its residues index val
                       # faster than `residues.find(residue)
  modpg:     int       # PG's modulus value
  rescnt:    int       # PG's residues count
  pairscnt:  int       # PG's twinpairs count
  residues:  seq[int]  # PG's list of residues
  restwins:  seq[int]  # PG's list of twinpair residues
  resinvrs:  seq[int]  # PG's list of residues inverses
  Bn:        int       # segment size factor for PG and input number

proc selectPG(num: uint) =
  ## Select at runtime best PG and segment size factor to use for input value.
  ## These are good estimates derived from PG data profiling. Can be improved.
  if num < 10_000_000:
    (modpg, rescnt, pairscnt, residues, restwins, resinvrs) = parametersp5
    Bn = 16
  elif num < 1_100_000_000'u:
    (modpg, rescnt, pairscnt, residues, restwins, resinvrs) = parametersp7
    Bn = 32
```

```
  elif num < 35_500_000_000'u:
    (modpg, rescnt, pairscnt, residues, restwins, resinvrs) = parametersp11
    Bn = 64
  elif num < 15_000_000_000_000'u:
    (modpg, rescnt, pairscnt, residues, restwins, resinvrs) = parametersp13
    if   num > 7_000_000_000_000'u: Bn = 384
    elif num > 2_500_000_000_000'u: Bn = 320
    elif num >   250_000_000_000'u: Bn = 196
    else: Bn = 96
  else:
    (modpg, rescnt, pairscnt, residues, restwins, resinvrs) = parametersp17
    Bn = 384
  cnts = newSeq[uint](pairscnt)                    # twins sums for seg bytes
  pos  = newSeq[int](modpg)                        # create modpg size array to
  for i in 0..rescnt - 1: pos[residues[i] - 2] = i # convert residue val -> indx
  lastwins = newSeq[uint](pairscnt)                # holds last twin per thread

proc sozpg(val: int | int64) =
  ## Compute the list of primes r1..sqrt(input_num), and store in global
  ## 'primes' array, and store their count in global var 'pcnt'.
  ## Any algorithm (fast|small) is usable. Here the SoZ for the PG is used.
  let md = modpg                    # PG's modulus value
  let rscnt = rescnt                # PG's residue count
  let res = residues                # PG's residues list

  let num = (val - 1) or 1          # if val even then subtract 1
  var k = num div md                # compute its residue group value
  var modk = md * k                 # compute the resgroup base value
  var r = 0                         # from 1st residue in num's resgroup
  while num >= modk + res[r]: r.inc # find last pc val|position <= num
  let maxpcs = k * rscnt + r        # max number of prime candidates <= num
  var prms = newSeq[bool](maxpcs)   # array of prime candidates set False

  let sqrtN =int(sqrt float64(num)) # compute integer sqrt of input num
  modk = 0; r = -1; k = 0           # initialize sieve parameters

  # mark the multiples of the primes r1..sqrtN in 'prms'
  for prm in prms:                  # from list of pc positions in prms
    if (r.inc; r) == rscnt: (r = 0; modk += md; k.inc)
    if prm: continue                # if pc not prime go to next location
    let prm_r = res[r]              # if prime save its residue value
    let prime = modk + prm_r        # numerate the prime value
    if  prime > sqrtN: break        # we're finished when it's > sqrtN
    let prmstep = prime * rscnt     # prime's stepsize to mark its mults
    for ri in res:                  # mark prime's multiples in prms
      let prod = prm_r * ri - 2     # compute cross-product for r|ri pair
      # compute resgroup val of 1st prime multiple for each restrack
      # starting there, mark all prime multiples on restrack upto end of prms
      var prm_mult = (k * (prime + ri) + prod div md) * rscnt + pos[prod mod md]
      while prm_mult < maxpcs: prms[prm_mult] = true; prm_mult += prmstep

  # prms now contains the nonprime positions for the prime candidates r1..N
  # extract primes into global var 'primes' and count into global var 'pcnt'
  primes = @[]                      # create empty dynamic array for primes
  modk = 0; r = -1                  # initialize loop parameters
  for prm in prms:                  # numerate|store primes from pcs list
    if (r.inc; r) == rscnt: (r = 0; modk += md)
    if not prm: primes.add(modk + res[r]) # put prime in global 'primes' list
  pcnt = primes.len                      # set global count of primes
```

```
#[
proc printprms(Kn: int, Ki: uint64, indx: int, seg: seq[uint8]) =
  ## Print twinprimes for given twinpair for given segment slice.
  ## Primes will not be displayed in sorted order, collect|sort later for that.
  var modk = Ki * modpg.uint        # base value of 1st resgroup in slice
  let res = restwins[indx]          # for upper twinpair residue value
  for k in 0..Kn - 1:               # for each of Kn resgroups in slice
    if seg[k].int == 0:             # if seg byte for resgroup has twinprime
      if modk + res.uint <= num:    # and if upper twinprime <= num
        echo(modk + res.uint - 1)   # print twinprime mid val on a line
    modk += modpg.uint              # set base value for next resgroup
]#
proc nextp_init(indx: int): seq[uint64] =
  ## Initialize 'nextp' array for given twin pair at 'indx' in 'restwins',
  ## Set each row[j] w/1st prime multiple resgroup for each prime r1..sqrt(N).
  var nextp = newSeq[uint64](pcnt * 2)   # create 1st mults array for twin pair
  let r_hi = restwins[indx]              # upper twin pair residue value
  let r_lo = r_hi - 2                    # lower twin pair residue value
  let (row_lo, row_hi) = (0, pcnt)       # nextp addr for lower|upper twin pair
  nextp = nextp_init(indx)               # init w/1st prime mults
  for j, prime in primes:                # for each prime r1..sqrt(N)
    let k = (prime - 2) div modpg        # find the resgroup it's in
    let r = (prime - 2) mod modpg + 2    # and its residue value
    let r_inv = resinvrs[pos[r - 2]]     # and its residue inverse
    var ri = (r_lo * r_inv - 2) mod modpg + 2 # compute the ri for r
    nextp[row_lo + j] = uint(k * (prime + ri) + (r * ri - 2) div modpg)
    ri = (r_hi * r_inv - 2) mod modpg + 2     # compute the ri for r
    nextp[row_hi + j] = uint(k * (prime + ri) + (r * ri - 2) div modpg)
  result = nextp

proc twins_sieve(Kmax: uint, indx: int) {.gcsafe.} =
  ## Perform in a thread, the ssoz for a given twin pair, for Kmax resgroups.
  ## First create|init 'nextp' array of 1st prime mults for given twin pair and
  ## its seg array of KB bytes, which will be gc'd|recovered at end of thread.
  ## For sieve, mark seg byte to '1' if either twin pair restrack is nonprime,
  ## for primes mults resgroups, update 'nextp' restrack slices acccordingly.
  ## Then find last twinprime|sum <= num, store sum in 'cnts' for twin pair.
  ## Optionally compile to print mid twin prime values generated by twin pair.
  {.gcsafe.}:
   var (sum, Ki, Kn) = (0'u, 0'u, KB)   # init twins cnt|1st resgroup for slice
   var (hi_tp,  upk) = (0'u, 0)         # max twin prime|resgroup val for slice
   var k_hi = 0'u                       # resgroup for largest tp in prev seg
   let r_hi = restwins[indx].uint       # twin prime hi residue value
   var seg = newSeq[uint8](KB)          # create seg byte array for twin pair
   nextp = nextp_init(indx)             # init w/1st prime mults for twin pair
   while Ki < Kmax:                     # for Kn resgroup size slices upto Kmax
     if KB > int(Kmax - Ki): Kn = int(Kmax - Ki) # set last slice resgroup size
     for b in 0..Kn - 1: seg[b] = 0     # set all seg restrack bytes to prime
     for j, prime in primes:            # for each prime index r1..sqrt(N)
                                        # for lower twin pair residue track
       var k = nextp[j].int             # starting from this resgroup in seg
       while k < Kn:                    # for each primenth byte to end of seg
         seg[k] = seg[k] or 1           # mark address as not a twin prime
         k += prime                     # compute next prime multiple resgroup
       nextp[j] = uint(k - Kn)          # save 1st resgroup in next eligible seg
                                        # for upper twin pair residue track
       k = nextp[pcnt + j].int          # starting from this resgroup in seg
       while k < Kn:                    # for each primenth byte to end of seg
         seg[k] = seg[k] or 1           # mark address as not a twin prime
         k += prime                     # compute next prime multiple resgroup
       nextp[pcnt + j] = uint(k - Kn)   # save 1st resgroup in next eligible seg
```

```
      var cnt = 0                         # initialize segment twin primes count
      for k in 0..Kn - 1: (if seg[k] == 0: cnt.inc) # sum segment twin primes
      if cnt > 0:                         # if segment has twin primes
        sum += cnt.uint                   # add the segment count to total count
        for k in 1..Kn:                   # find location of largest twin prime
          if seg[Kn - k] == 0: (upk = Kn - k; break) # count down to largest tp
        k_hi = hi_tp                      # keep largest tp resgroup from prev seg
        hi_tp = Ki + upk.uint             # keep largest tp resgroup for this seg
      #printprms(Kn, Ki, indx, seg)       # optional: display twin primes in seg
      Ki += KB.uint                       # set 1st resgroup val of next seg slice
    hi_tp = hi_tp * modpg.uint + r_hi     # numerate final seg largest twin prime
                                          # see if largest tp in range
    if hi_tp > num:                       # if outside find sum|value that's inside
      var prev = true                     # flag to use last tp from previous seg
      for k in 0..upk:                    # count down from upk resgroup to '0'
        if seg[upk - k].int == 0:         # if twin prime at seg resgroup address
          if hi_tp <= num.uint: prev = false; break # keep if in range, flip flag
          sum.dec                         # else reduce sum for too large twin
        hi_pt -= modpg.uint               # then check next lower twin pair hi val
                                          # keep from prev seg if none in range
      if prev: hi_tp = if r_hi > num: 0'u else: k_hi * modpg.uint + r_hi
    lastwins[indx] = hi_tp                # store un|adjusted final seg tp value
    cnts[indx] = sum                      # store correct|ed sum for twin pair

proc twinprimes_ssoz() =
  ## Main routine to setup, time, and display results for twin primes sieve.
  stdout.write "Enter integer number: "
  let val = stdin.readline.parseBiggestUInt # find primes <= val (13..2^64-1)

  echo("threads = ", countProcessors())
  let ts = epochTime()              # start timing sieve setup execution

  num = uint64((val - 1) or 1)      # if val even subtract 1
  selectPG(num.uint)                # select PG and seg factor Bn for input num
  let modpg = modpg.uint            # to reduce casting hell noise for Nim
  let Kmax = (num-2) div modpg + 1 # maximum number of resgroups for num
  let B = Bn * 1024                 # set seg size to optimize for selected PG
  KB = if Kmax.int < B: Kmax.int else: B  # min num of segment resgroups

  echo("each thread segment is [", 1, " x ", KB, "] bytes array")

  # This is not necessary for running the program but provides information
  # to determine the 'efficiency' of the used PG: (num of primes)/(num of pcs)
  # The closer the ratio is to '1' the higher the PG's 'efficiency'.
  let k = num div modpg             # compute num's resgroup
  let modk = modpg * k.uint         # compute num's base val
  var r = 0                         # from first residue in last resgroup
  while num.uint >= modk + restwins[r].uint: r.inc # find last tp index <= num
  let maxpairs = k * pairscnt.uint + r.uint  # maximum number of twinprime pcs
  echo("twinprime candidates = ", maxpairs, "; resgroups = ", Kmax)
  # End of non-essential code.

  sozpg(int(sqrt float64(num)))    # compute pcnt and primes <= sqrt(num)

  echo("each ", pairscnt, " threads has nextp[", 2, " x ", pcnt, "] array")
  let te = epochTime() - ts         # sieve setup time
  echo("setup time = ", te.formatFloat(ffDecimal, 3), " secs")

  twinscnt = if modpg > 30030'u: 4 elif modpg > 210'u: 3 else: 2 # 1st 4 tps

  echo("perform twinprimes ssoz sieve")
```

```
  let t1 = epochTime()             # start timing ssoz sieve execution
  parallel:                        # perform in parallel
    for indx in 0..pairscnt - 1:   # for each twin pair row index
      spawn twins_sieve(Kmax.uint,indx) # sieve selected twin pair restracks
  sync()                           # when all the threads finish
  last_twin = 0'u                  # find largest twin prime in range
  for i in 0..pairscnt - 1:        # and determine total twin primes count
    twinscnt += cnts[i]
    if last_twin < lastwins[i]: last_twin = lastwins[i]
  var Kn = Kmax.int mod KB         # set number of resgroups in last slice
  if Kn == 0: Kn = KB              # if multiple of seg size set to seg size
  let t2 = epochTime() - t1        # sieve execution time

  echo("sieve time = ", t2.formatFloat(ffDecimal, 3), " secs")
  echo("last segment = ", Kn, " resgroups; segment slices = ", (Kmax - 1) div KB.uint + 1)
  echo("total twins = ", twinscnt, "; last twin = ", last_twin - 1, "+/-1")
  echo("total time = ", (t2 + te).formatFloat(ffDecimal, 3), " secs\n")

twinprimes_ssoz()
```