Twin Primes Segmented Sieve of Zakiya (SSoZ) Explained

Jabari Zakiya © June 12, 2022
jzakiya@gmail.com

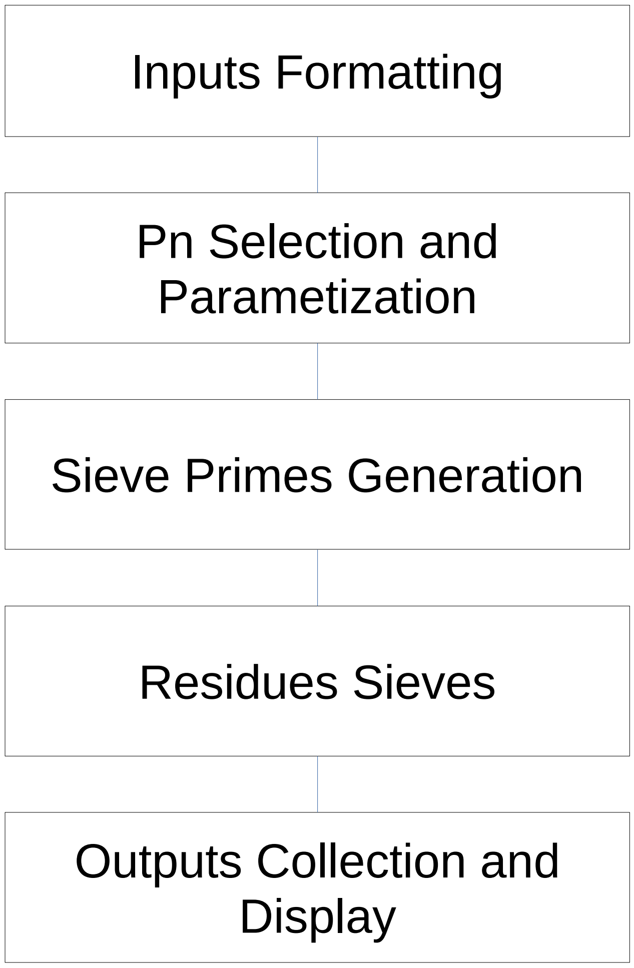
Introduction

In 2014 I released ***The Segmented Sieve of Zakiya (SSoZ)*** [1]. It described a general method to find primes using an efficient prime sieve based on Prime Generators (PG). I expanded upon it, and in 2018 I released ***The Use of Prime Generators to Implement Fast Twin Primes Sieve of Zakiya (SoZ), Applications to Number Theory, and Implications for the Riemann Hypotheses*** [2]. The algorithm has been improved and now also used to find *Cousin Primes*. This paper explains in detail the what, why, and how of the algorithm and shows its implementation in 6 software languages, and performance data for these 6 languages run on 2 different cpu systems with 8 and 16 threads.

General Description

The programs count the number of *Twin|Cousin Primes* between two numbers within a 64-bit range, i.e. 0 – 18,446,744,073,709,551,615 (2**64 – 1), and also returns the largest twin|cousin value within it. The algorithm has no mathematical limits, but [hard|soft]ware does, so its coded to run on commonly available 64-bit multi-core systems containing a *reasonable* amount of memory (the more the better).

Below is a diagram and description of the major functional components of the algorithm and software.


Inputs Formatting
Pn Selection and Parametization
Sieve Primes Generation
Residues Sieves
Outputs Collection and Display

Inputs Formatting
One or two values are entered (order doesn't matter) specifying the numerical range. They're converted to odd values, and|or defaults, after conditional checks.

Pn Selection and Parameterization
The inputs numerical range is used to select the Pn generator used to perform the residues sieve. Once determined, its generator parameters are created.

Sieve Primes Generation
The sieving primes ≤ sqrt(`end_num`) for the range are generated, but only those with multiples within the numerical range are used for the Pn generator.

Residues Sieves
In parallel for each twin|cousin residues pair for Pn, the sieve primes are used to create the `nextp` array of start locations for marking their multiples for each segment size the input numerical range is split into.

Outputs Collection and Display
The prime pairs count and largest value is collected for each residue pair thread, and their final greatest values displayed, along with timing data.

Math Fundamentals

Prime numbers do not exist randomly! When we break the number line into even sized groups of integers (the group ***numerical bandwidth*** and prime generator ***modulus*** value), the primes are evenly distributed along the ***residues*** in each group, i.e. the ***coprime*** values to the modulus (their ***greatest common divisor (gcd)*** with the modulus is 1). Thus a modulus, and its associated residues, form a ***Prime Generator (PG)***, a ***mathematical expression*** and ***framework*** for ***generating and identifying*** every prime not a modulus prime factor.

While a PG modulus can be any even number, the ***most efficient moduli*** are ***strictly prime primorials***. These prime generators have the smallest ratios of (# of residues)/modulus and make the ***number space*** primes exist within the smallest possible for a given number of residues. As more primes are used to form the PG moduli they systematically squeeze the primes into smaller and smaller number spaces.

The S|SoZ algorithms are based on the structure and framework of *Prime Generators*, whose math and properties are formalized in ***Prime Generator Theory (PGT)***. For an extensive review read [1], [2], [3] and see the video – ***(Simplest) Proof of the Twin Primes and Polignac's Conjectures.*** https://www.youtube.com/watch?v=HCUiPknHtfY&t=940s [4].

Below is a list of the major properties of Prime Generators that comprise the mathematical foundation for the S|SoZ algorithms and code.

Major Properties of Prime Generators

- a prime generators has form: $\mathbf{Pn = modpn * k + \{r_0 \ldots r_n\}}$
- the modulus for prime generator with last prime value p_n has primorial form: $\mathbf{modpn} = \boldsymbol{p_n}\mathbf{\#}$
- the number of residues are even, with counts: $\mathbf{rescntpn = (}\boldsymbol{p_n}\mathbf{-1)\# =}\ \boldsymbol{p_n^{-1}\#}$
- the residues occur as ***modular complement pairs*** to its modulus: $\mathbf{modpn = r_i + r_j}$
- the last two residues of a generator are constructed as: **(modpn - 1) (modpn + 1)**
- the residues, by definition, will include all the coprime primes < **modpn**
- the first residue $\mathbf{r_0}$ is the next prime > $\boldsymbol{p_n}$
- the residues from $\mathbf{r_0}$ to $\mathbf{r_0^2}$ are consecutive primes
- each generator has a characteristic ***Prime Generator Sequence (PGS)*** of even residue gaps
- the last 3 sequence gaps have form: $\mathbf{(r_0 - 1)\ \ 2\ \ (r_0 - 1)}$
- the gaps are distributed with a ***symmetric mirror image*** around a pivot gap size of **<u>4</u>**
- the residue gaps sum from $\mathbf{r_0}$ to $\mathbf{(r_0 + modpn)}$ equals the modulus: $\mathbf{modpn} = \boldsymbol{\Sigma a_i \cdot 2i}$
- the coefficients $\boldsymbol{a_i}$ are the frequency of each gap of size $\boldsymbol{2i}$
- the sum of the coefficients $\boldsymbol{a_i}$ equal the number of residues: $\mathbf{rescntpn} = \boldsymbol{\Sigma a_i}$
- coefficients $\boldsymbol{a_1 = a_2}$ are odd and equal with form: $\boldsymbol{a_1 = a_2 = (p_n - 2)\# = p_n^{-2}\#}$
- the coefficients $\boldsymbol{a_i}$ are even for $i > 2$
- the number of nonzero coefficients $\boldsymbol{a_i}$ in a sequence for **Pn** is of order $\boldsymbol{p_{n-1}}$

Residues have ***canonical form*** values (1...modpn-1), as 1 is always coprime to any modulus, but for coding|math efficiency their ***functional form*** values (r_0…modpn+1) are used, with r_0 defined above, and **modpn+1 ≡ 1 modpn** is the permuted first ***congruent value*** for 1. Also, as the residues exist as ***modular complement pairs*** the code determines their first half values and their 2nd half values come for FREE. To find the residues for a Pn, we can use the PGS of a smaller generator (in the code for P3), to reduce the number space of the ***residue candidates (rc)*** in larger moduli that need to be checked.

Shown here is the ***primes candidates (pcs) table*** for P5 up to the 100th prime 541. It shows the only possible pc values that can be primes for 30 integer groupings. Each of the **k** columns is a ***residue group (resgroup)*** of prime candidates. The colored pc values are nonprime composites, and can be sieved out by the *SoZ* (Sieve of Zakiya), leaving only the prime values shown.

P5 = 30 * k + {7, 11, 13, 17, 19, 23, 29, 31}

k	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
r0	7	37	67	97	127	157	187	217	247	277	307	337	367	397	427	457	487	517
r1	11	41	71	101	131	161	191	221	251	281	311	341	371	401	431	461	491	521
rt	13	43	73	103	133	163	193	223	253	283	313	343	373	403	433	463	493	523
r3	17	47	77	107	137	167	197	227	257	287	317	347	377	407	437	467	497	527
r4	19	49	79	109	139	169	199	229	259	289	319	349	379	409	439	469	499	529
r5	23	53	83	113	143	173	203	233	263	293	323	353	383	413	443	473	503	533
r6	29	59	89	119	149	179	209	239	269	299	329	359	389	419	449	479	509	539
r7	31	61	91	121	151	181	211	241	271	301	331	361	391	421	451	481	511	541

Table 1.

Every PG represents a pcs table like this, which visually display all their properties. To identify all the Twin Primes we merely observe the residue pair values that differ by 2, (11, 13), (17, 19), (29, 31), and for Cousins those that differ by 4, (7, 11), (13, 17), (19, 23). These ***residues gaps*** form the basis for the Twins|Cousins SSoZ implementations, and other k-tuples of interest.

To find larger constellations of prime pairs, et al, we merely identify the residue pairs of desired size. For Sexy Primes (*p*, *p*+6), we just use the pairs (7, 13), (11, 17), (13, 19), (17, 23), (23, 29), (31, 37). Using them, we easily see and count there are 47 Sexy Primes (with [5:11]) within the first 100 primes. Larger generators have more residues and larger gaps and enable identifying more desired size k-tuples.

In my video [4], I define the residue gaps as the gaps between consecutive residues, and thus I refer to prime gaps as consecutive prime (2, n) tuples, where n is an even integer. Thus in the video I state there are 25 Sexy Primes in the table above, i.e. 25 pairs of consecutive primes that differ by 6. However in the academic math world Sexy and Cousin primes are defined as any (2, 6) and (2, 4) tuple, thus [7:13] is a Sexy Prime even though we see 11 is between them. So [5:11] is defined as the first Sexy Prime and [3:7] the first Cousin, and [3:103] would be the first (2, 100) tuple, i.e. 2 primes that differ by 100.

However, if you want to know and understand the true distribution of primes, what you want to know is the distribution of the ***gaps between consecutive primes***, which I'll define as ***prime gap k_{pg}-tuples***. So the actual first (2, 100) k_{pg}-tuple is [396,733: 396,833], *a very big difference*. It's from the k_{pg}*-tuples* that inform you where the ***prime deserts*** are (long number stretches without primes), and characterize the true average thinning (density) of primes as the integers grow larger. And as shown and explained in [3] and [4], there are an infinity of consecutive prime gaps of any even size.

Thus the PGS for the Pn's provide a deterministic floor (minimum) value of the number of k_{pg}-tuples of any size, and their prime values, over any range of numbers, which we can (in theory) create an SSoZ residues sieve to identify and count.

Shown here are the PG parameters for the first 9 Pn generators P2 – P23 where modpn = $\prod_{i=1}^{m} p_i$

Here pn = p_m is the prime value of the mth prime, thus: p2 = p_1, p3 = p_2, p5 = p_3, p7 = p_4,, etc.
Pn's modulus value modpn: $(p_n - 0)\# = p_n^{-0}\# = \Pi\,(p_n - 0) = (2 - 0) * (3 - 0) * (5 - 0) \ldots * (p_m - 0)$
Number of residues rescnt: $(p_n - 1)\# = p_n^{-1}\# = \Pi\,(p_n - 1) = (2 - 1) * (3 - 1) * (5 - 1) \ldots * (p_m - 1)$
# of twins|cousins pairscnt: $(p_n - 2)\# = p_n^{-2}\# = \Pi\,(p_n - 2) = (2 - 2) * (3 - 2) * (5 - 2) \ldots * (p_m – 2)$

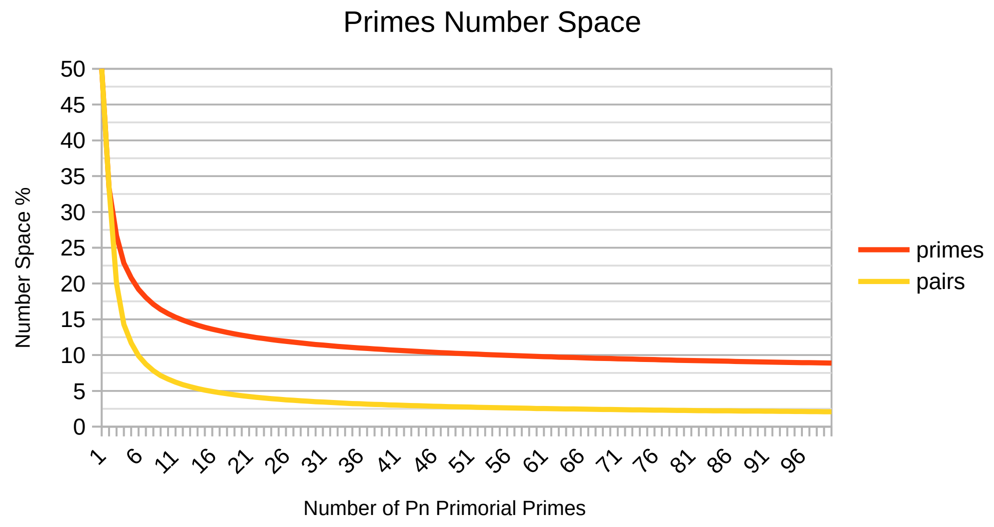
For P23 modulus: modp23 = 2 * 3 * 5 * 7 * 11 * 13 * 17 * 19 * 23 = 223092870
For P23 residues: rescount = 1 * 2 * 4 * 6 * 10 * 12 * 16 * 18 * 22 = 36495360
For P23 twins|cousin: pairs = 1 * 1 * 3 * 5 * 9 * 11 * 15 * 17 * 21 = 7952175

The primes number space % is: (rescntpn/modpn) * 100 = $(p_n^{-1}\# / p_n\#) * 100$
The pairscnt number space % is: (pairscntpn*2/modpn) * 100 = $(p_n^{-2}\# / p_n\#) * 200$

P_n	P2	P3	P5	P7	P11	P13	P17	P19	P23
modulus (modpg)	2	6	30	210	2310	30030	510510	9699690	223092870
residues count (rescnt)	1	2	8	48	480	5760	92160	1658880	36495360
twins|cousins pairscnt	0	1	3	15	135	1485	22275	378675	7952175
primes % number space	50.00	33.33	26.67	22.86	20.78	19.18	18.05	17.10	16.36
pairs % number space	50.00	33.33	20.00	14.29	11.69	9.89	8.73	7.81	7.13

Table 2.

As the Pn primorial primes p_m increase, the number space containing primes and twins|cousins steadily decreases, and can be made an arbitrarily small value ε > 0 of the total number space as m → ∞.



This graph shows the decreasing prime number space for Pn using the first 100 primes. Once past the knee of the curve, the differential change becomes smaller for each additional p_m. For many common use cases we can effectively limit usable Pn generators to the first 10 primes or so. However, for prime searches in large number values ranges, using the largest generator possible for a system is desirable, to make the maximum searchable number space as small as possible.

Generating Sieve Primes

The SSoZ uses the *necessary* sieving primes ≤ $\sqrt{end\_num}$ (i.e. only those with multiples within the inputs range) to sieve out their nonprime multiples. An efficient coded P5 Sieve of Zakiya (SoZ) generates them at runtime (though other means can be used). Below is its algorithm.

SoZ Algorithm

To find all the primes ≤ N = $\sqrt{end\_num}$

1. for Prime Generator P5, create its generator parameters
2. determine **kmax**, the number of residue groups (resgroups) up to N
3. create byte array **prms[kmax]** to represent the value|residue of each resgroup pc
4. perform outer sieve loop:
    - starting from the first resgroup, determine where each pc bit location is prime
    - if a bit location a prime, keep its residue value in **prm_r**; numerate its **prime** value
    - exit loop when **prime > sqrt(N)**

    5. perform inner sieve loop with each residue **ri**:
        - create cross-product **(prm_r * ri)**
        - determine the resgroup **kn** it’s in, and its residue **rn**
        - compute first prime multiple resgroup **kpm** for the prime with **ri**
        - mark in **prms** each primenth **kpm** resgroup **bitn[rn]** as non-prime until its end

    6. repeat from 4 for next resgroup

7. when sieve ends, numerate|store from each prms resgroup the needed sieving primes ≤ N

P5’s primes candidates (pcs) table up to 541 (the 100th prime) is shown below.

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 function `sozpg` performs the P5 sieve exactly as shown. An array `prms` of **kmax** bytes is created to represent each resgroup|column of 8 pc values|rows up to the resgroup that covers the input value.

Each row represents a residue value|bit position|residue track. **`prms`** is initialized to ‘0’ to make all bit positions be prime. The sieve computes for each prime ≤ $\sqrt{end\_num}$ its first prime multiple resgroup **kpm** on each row, and starting from these, sets each primenth resgroup bit on each row to ‘1’, to mark its multiples (colors), to eliminate the nonprimes. The process is explained in greater detail as follows.

Performing SoZ Sieve

To sieve the nonprimes from P5's pcs table up to 541 we use the primes ≤ isqrt(541)=23. They are the first 6 primes|residues: 7, 11, 13, 17, 19, 23, whose first unique multiples are shown with 6 different colors. The value 541 resides in residue group k=17, so **kmax=18** is the number of resgroups up to it.

Starting with the first prime in regroup k=0, 7 multiplies each pc in the resgroup, whose multiples are in blue: 7 * [7, 11, 13, 17, 19, 23, 29, 31] = [49, 77, 91, 119, 133, 161, 203, 217]. Each 7th resgroup|col along each restrack|row from these start values are 7's multiples. Thus 7 * 7 = 49 in resgroup k=1, on rt4|r=19 is 7's first multiple. Every 7th regroup starting there (k=1, 8, 15) < kmax on rt4 is a multiple of 7 and set to '1' to mark as nonprime. We repeat for 7's other first multiples 77, 91, etc, on their rows.

We then use the next prime location in resgroup k=0 after 7, which is 11, and repeat the process with it. 11 * [7, 11, 13, 17, 19, 23, 29, 31] = [77, 121, 143, 187, 209, 253, 319, 341], whose first *unique* multiples are red. Note, the first *unique* multiple for each prime is its square, which for 11 is 121. The first multiples with smaller primes, e.g. 11* 7 = 77, are colored with those primes colors (here 7|blue). Also note, *each prime must multiply each member in its resgroup, whether prime or not*, to map its starting first prime multiple onto each distinct row in some **`kpm`** resgroup.

As shown, this process is very simple and fast, and we can perform the multiplications very efficiently. We can also perform the sieve and primes extraction process in parallel, making it even faster.

Extracting Sieve Primes

To extract the primes from `prms` in sequential order, we start at resgroup k=0 and iterate over each byte bit, then continue with each successive byte. A '0' bit position represents a prime value in each byte, and if '1' we skip to the next bit. The prime values are numerated as: **`prime = modpg * k + ri`**, with `k` the resgroup index, `ri` the residue for the bit position, and `modpg` = 30 for P5's modulus.

Alternatively we can reverse the order, and for each bit row, iterate over each resgroup byte and find the primes along them. This may provide certain software computational advantages, but the primes will no longer be extracted in sequential order (though if necessary they could be sorted afterwards). For the purposes of the SSoZ algorithm, it's not necessary the primes be used in sequential order.

To optimize performance of the SSoZ, during the prime sieve extraction process, primes which don't have multiples within the inputs range are discarded. This significantly increases SSoZ performance for small input ranges between large input numbers, by reducing the work the residues sieves do.

The algorithm described here is generic to all Pn generators, where only their parameters change for each. Implementations may vary based on hardware|software particulars, but the work performed is the same. Larger generators systematically reduce the primes number space, by having larger modulus sizes and more residues, but we generally want to pick the smallest Pn generator that optimizes the system resources for given input values and ranges.

For the implementations provided, whose inputs range are constrained to 64-bits, using P5 to perform the SoZ with was the overall most efficient choice, as it's straightforward to code, and as we'll see, can also be done in parallel to increase its performance.

Efficient residue multiplications

To find the resgroup (column) 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 each pc regroup value has parameters: **k = pc div modpg**, with residue value: **ri = pc mod modpg**.

Multiplying two regroup pcs e.g. (17 * 19) = 323 gives: **k, ri = (17 * 19).divmod 30 –>** k = 10, ri = 23. From P5's pc table, we see pc = 323 is in resgroup k=10 with residue 23 on restrack rt5.

Each prime can be parameterized by its residue **r** and resgroup **k** values e.g.: **prime = modk + r,** where **modk = modpg * k**, for each resgroup, and each resgroup **pc_i** has form: **pc_i = modk + ri**. Thus the multiplication – **(prime * pc_i)** – translates into the following parameterized form:

$$\begin{gathered} prime * pc\_i \\ (modk + r) * (modk + r_i) \\ modk * modk + modk * (r + r_i) + (r * r_i) \\ modk * (modk + r + r_i) + (r * r_i) \\ (modpg * k) * (prime + r_i) + (r * r_i) \\ modpg * [k * (prime + r_i)] + [r * r_i] \end{gathered}$$

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

where **kk = k * (prime + ri)** and **rr = r * ri**, which also 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). We extract from it its resgroup value: **kn = (r * ri) / modpg**, and residue: **rn = (r * ri) % modpg**, which maps to a restrack bit value as **rt_n = residues.index(rn)**. Thus for P5, r = 7 is at residues[0], so that its rt_i row value is: i = residues.index(7) = 0, whose bit mask is: **bit_r = 2^i = (1 << i)** in the code.

Thus, the product of two members in resgroup **k** maps to a higher resgroup: **kp = kk + kn** on **rt_n**, comprised of two components; **kn** (their cross-product resgroup), and **kk** (their k resgroup component).

To describe this verbally, to find the product resgroup **kp** of any two resgroup members, numerate one member (for us a prime), call its residue **r**, add the other's residue **ri** to it, multiply their sum by the resgroup value **k**, then add it to their residues cross-product resgroup. For (97 * 109) with k = 3 gives:

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

For each Pn the last resgroup pc value is: **(modpg + 1) ≡ 1 mod modpg**, so for P5, its **modpg*k + 31**. To ensure **pc / modpg = k** always produces the correct k value, 2 is subtracted before the division. Thus the resultant residue value is 2 less than the correct one, so 2 is added back to get the true value. In `sozpg:` **`kn, rn = (prm * ri - 2).divmod md`**; **kn** is the correct resgroup and (**rn + 2**) the correct residue. The code uses **rn** without the addition sometimes when doing memory addressing. (In the code, the `posn` array performs the mapping at address (r – 2) into restrack rtn indices 0 – 7).

Ex: (7 * 43) / 30 = 301 / 30 = 10, but 301 is the last pc in resgroup 9, so (301 – 2) / 30 is correct value. Also 301 % 30 = 1, but 299 % 30 = 29, and when 2 is added we get the correct residue 31 for pc 301.

sozpg

```
def sozpg(val, res_0, start_num, end_num)
  # Compute the primes r0..sqrt(input_num) and store in 'primes' array.
  # Any algorithm (fast|small) is usable. Here the SoZ for P5 is used.
  md, rscnt = 30u64, 8                  # P5's modulus and residues count
  res  = [7,11,13,17,19,23,29,31]       # P5's residues
  bitn = [0,0,0,0,0,1,0,0,0,2,0,4,0,0,0,8,0,16,0,0,0,32,0,0,0,0,0,64,0,128]

  kmax = (val - 2) // md + 1            # number of resgroups upto input value
  prms = Array(UInt8).new(kmax, 0)      # byte array of prime candidates, init '0'
  modk, r, k = 0, -1, 0                 # initialize residue parameters

  loop do                               # for r0..sqrtN primes mark their multiples
    if (r += 1) == rscnt; r = 0; modk += md; k += 1 end # resgroup parameters
    next if prms[k] & (1 << r) != 0     # skip pc if not prime
    prm_r = res[r]                      # if prime save its residue value
    prime = modk + prm_r                # numerate the prime value
    break if prime > Math.isqrt(val)    # exit loop when it's > sqrtN
    res.each do |ri|                    # mark prime's multiples in prms
      kn,rn = (prm_r * ri - 2).divmod md # cross-product resgroup|residue
      bit_r = bitn[rn]                  # bit mask for prod's residue
      kpm = k * (prime + ri) + kn       # resgroup for 1st prime mult
      while kpm < kmax; prms[kpm] |= bit_r; kpm += prime end
  end end
  # prms now contains the nonprime positions for the prime candidates r0..N
  # extract only primes that are in inputs range into array 'primes'
  primes = [] of UInt64                 # create empty dynamic array for primes
  prms.each_with_index do |resgroup, k|  # for each kth residue group
    res.each_with_index do |r_i, i|     # check for each ith residue in resgroup
      if resgroup & (1 << i) == 0       # if bit location a prime
        prime = md * k + r_i            # numerate its value, store if in range
        # check if prime has multiple in range, if so keep it, if not don't
        n, rem = start_num.divmod prime  # if rem 0 then start_num is multiple of prime
        primes << prime if (res_0 <= prime <= val) && (prime * (n + 1) <= end_num || rem == 0)
  end end end
  primes
end
```

Inputs:

`val` – integer value for $\sqrt{end\_num}$

`res_0` – first residue for selected SSoZ Pn

`end_num` – inputs high value

`start_num` – inputs low value

Output:

`primes` – array of sieving primes within inputs range

`sozpg` sieves the prime multiples ≤ `val` to create P5's pcs table held in byte array `prms,` as described. To extract only the necessary primes for the SSoZ it uses inputs: `res_0, start_num, end_num`

`res_0` is the first residue of the selected Pn for the SSoZ. For P5 it's 7, but when Pn is larger, e.g. P7, P11, P13 etc, their `res_0` are greater, i.e. 11, 13, 17, etc, so only the primes ≥ `res_0` are kept. The last byte `prm[kmax-1]` may also have bit positions for primes > val, which aren't needed and are discarded.

We thus perform two checks for each found prime, the first being: `(res_0 <= prime <= val)`
This filters out from P5's pcs table the primes outside the SSoZ inputs range for the selected Pn.

The second check determines the primes with multiples within the SSoZ inputs range that are needed. For small input ranges, primes > the range size can be discarded if they don't have multiples within it. This is done by the check: `(prime * (n + 1) <= end_num || rem == 0)`

All the primes ≤ $\sqrt{end\_num}$ are used if their values are ≤ range = (end_num – $start\_num$). But if range = (end_num – $start\_num$) < $\sqrt{end\_num}$ some sieving primes may be discarded, i.e. when (end_num – $\sqrt{end\_num}$) < $start\_num$ some primes may not have multiples within the range.

Example: end_num = 4,000,000; $\sqrt{end\_num}$ = 2,000
(end_num – $\sqrt{end\_num}$) < $start\_num$
(4,000,000 – 2,000) < $start\_num$
3,998,000 < $start\_num$

If $start\_num$ ≤ 3,998,000; say 500,000; the input range is ≥ 1999, the largest prime less than 2000, and all the primes < $\sqrt{end\_num}$ will have at least one multiple in the range, and must be used.

If $start\_num$ > 3,998,000, say 3,999,300, the primes < 700 (the input range) will have multiples in the range; 122 for P5. But some of the 178 primes between 700 < *p* < 2,000 will not, and can be discarded. The second test finds 103 are needed. So for P5 only 75% (225 of 300) of the primes < 2000 are used. Described below is the process to determine if a prime *p* has at least one multiple in the inputs range.

```
                   |—— p ——|
                   |rem|     | np+p
1p…2p…3p…..np….|-------+-----------------------|
                   start_num                end_num
```

For a given `prime` value do: `n = start_num // prime; rem = start_num % prime`
In Crystal, et al, can just do: `n, rem = start_num.divmod prime`
Then do the following test: `prime * (n + 1) <= end_num || rem == 0`

Here, `n`**p* + `rem` = $start\_num$, where `n` is the number of prime's multiples e.g. n*p* ≤ $start\_num$.
If `rem` is 0 then $start\_num$ is a multiple of *p*, otherwise 0 < `rem` < *p*. If *p* > $start\_num$, `n` = 0.
Thus (`n`**p* + *p*) = *p**(`n` + 1) is the next multiple of *p* whose value is > $start\_num$.
If *p**(`n` + 1) ≤ end_num *p* is in range, if not, but `rem` = 0, then *p**`n` = $start\_num$, so *p* is in range.

Also, when performing: `kn, rn = (prm_r * ri - 2).divmod md`, `rn`'s true value is reduced by 2, but we need to know its true residue bit position to mark the prime multiples for those bit positions.

Conceptually, given residue `rn`, its bit index is: `posn[rn] = res.index(rn)`, for P5 a value from 0..7. Because the `rn` values are 2 less than their real values, (`rn` – 2) is used as their addresses into the array `posn` used to map them, coded as: `posn=[];(0..rscnt-1).each { |n| posn[res[n]-2] = n }`
Then `posn`[7-2] = 0, `posn`[11-2] = 1, etc, and each `rn` bit value is: `bit_r = 1 << posn[rn]`, which are OR'd into `prms` to mark the prime multiples as: `prms[kpm] |= bit_r`. The shift values 2^i can be converted to their bit position values directly using array `bitn`[] e.g. now: `bit_r= bitn[rn]`

```
posn =[0,0,0,0,0,0,0,0,0,1,0,2,0,0,0,3,0, 4,0,0,0, 5,0,0,0,0,0, 6,0,  7]
bitn =[0,0,0,0,0,1,0,0,0,2,0,4,0,0,0,8,0,16,0,0,0,32,0,0,0,0,0,64,0,128]
```

In both cases byte arrays can be used to store the values, as they all can be represented by just 8 bits. This is an implementation detail to decide.

Because the processing of each row is independent from the others we can perform both the sieve and prime extraction processes in parallel. Below shows Rust code using the Rayon crate to do this.

```
fn atomic_slice(slice: &mut [u8]) -> &[AtomicU8] {
    unsafe { &*(slice as *mut [u8] as *const [AtomicU8]) }
}

fn sozpg(val: usize, res_0: usize, start_num : usize, end_num : usize) -> Vec<usize> {
  // Compute the primes r0..sqrt(input_num) and store in 'primes' array.
  // Any algorithm (fast|small) is usable. Here the SoZ for P5 is used.
  let (md, rscnt) = (30, 8);              // P5's modulus and residues count
  static RES: [usize; 8] = [7,11,13,17,19,23,29,31];
  static BITN: [u8; 30] = [0,0,0,0,0,1,0,0,0,2,0,4,0,0,0,8,0,16,0,0,0,32,0,0,0,0,0,64,0,128];

  let kmax = (val - 2) / md + 1;          // number of resgroups upto input value
  let mut prms = vec![0u8; kmax];         // byte array of prime candidates, init '0'
  let sqrt_n = val.integer_sqrt();        // compute integer sqrt of val
  let (mut modk, mut r, mut k) = (0, 0, 0 );

  loop {                                  // for r0..sqrtN primes mark their multiples
    if r == rscnt { r = 0; modk += md; k += 1 }
    if (prms[k] & (1 << r)) != 0 { r += 1; continue } // skip pc if not prime
    let prm_r = RES[r];                   // if prime save its residue value
    let prime = modk + prm_r;             // numerate the prime value
    if  prime > sqrt_n { break }          // exit loop when it's > sqrtN
    let prms_atomic = atomic_slice(&mut prms); // share mutable prms among threads
    RES.par_iter().for_each (|ri| {       // mark prime's multiples in prms in parallel
      let prod = prm_r * ri - 2;          // compute cross-product for prm_r|ri pair
      let bit_r = BITN[prod % md];        // bit mask for prod's residue
      let mut kpm = k * (prime + ri) + prod / md; // 1st resgroup for prime mult
      while kpm < kmax { prms_atomic[kpm].fetch_or(bit_r, Ordering::Relaxed); kpm += prime; };
    });
    r += 1;
  }
  // prms now contains the nonprime positions for the prime candidates r0..N
  // numerate the primes on each bit row in prms in parallel (won't be in sequential order)
  // return only the primes necessary to do SSoZ for given inputs in array 'primes'
  let primes = RES.par_iter().enumerate().flat_map_iter( |(i, ri)| {
    prms.iter().enumerate().filter_map(move |(k, resgroup)| {
      if resgroup & (1 << i) == 0 {
        let prime = md * k + ri;
        let (n, rem) = (start_num / prime, start_num % prime);
        if (prime >= res_0 && prime <= val) && (prime * (n + 1) <= end_num || rem == 0) {
          return Some(prime);
      } } None
  }) }).collect();
  primes
}
```

Here the primes are extracted from each row in parallel using 8 threads, thus not kept in sequential order. Reversing the loops, as in the Crystal code, will extract them in order but will be slower as the number of resgroups increase. Since sequential order isn't necessary to do the SSoZ this is optimal.

For systems with more than 8 threads, using P7 with 48 residues may be faster, especially for large input values, if P7's smaller number space can be processed faster with those threads than using P5.

We can see the performance gain that's achieved between using all the sieving primes upto `end_num`, to only using those with multiples within the inputs ranges, to then generating them in parallel in `sozpg`. The following examples using Rust show the three cases and the progressive performance increases.

This is the Rust output of the original unoptimized `sozpg` using these two 63-bit number as inputs. It shows (in `nextp[2 x 129900044]`) 129,900,044 sieving primes were generated, which accounted for most of the setup time. The times shown are for the i7 6700HQ 4C|8T and AMD 5900HZ 8C|16T cpus.

```
$ echo 7200011140000000000 7200011139993250000 | ./twinprimes_ssoz157
threads = 8                        // 16
using Prime Generator parameters for P5
segment size = 65536 resgroups; seg array is [1 x 1024] 64-bits
twinprime candidates = 675003; resgroups = 225001
each of 3 threads has nextp[2 x 129900044] array
setup time = 13.098702568 secs     // 7.089318922 secs
perform twinprimes ssoz sieve
3 of 3 twinpairs done
sieve time = 9.731177018 secs      // 4.944145598 secs
total time = 22.829885781 secs     // 12.033471504 secs
last segment = 28393 resgroups; segment slices = 4
total twins = 4711; last twin = 7200011139999998808+/-1
```

These are the result from filtering out the unnecessary primes (no multiples in inputs range), using 49x fewer primes – 2,636,377. Though there's some setup time increases for 8 threads, there's a massive decrease in the sieve time, as each thread now does significantly less work (and use less memory).

```
$ echo 7200011140000000000 7200011139993250000 | ./twinprimes_ssoz158
threads = 8                        // 16
using Prime Generator parameters for P5
segment size = 65536 resgroups; seg array is [1 x 1024] 64-bits
twinprime candidates = 675003; resgroups = 225001
each of 3 threads has nextp[2 x 2636377] array
setup time = 13.743127493 secs     // 6.987116498 secs
perform twinprimes ssoz sieve
3 of 3 twinpairs done
sieve time = 0.175270322 secs      // 0.107544045 secs
total time = 13.918427314 secs     // 7.094673324 secs
last segment = 28393 resgroups; segment slices = 4
total twins = 4711; last twin = 7200011139999998808+/-1
```

Finally, when `sozpg` performs the prime generation and filtering process in parallel the setup times drops from 13.7|6.9 to 5.3|4.7 secs, with a total time drop from 22.8|12.0 to ~5.5|4.9 secs.

```
$ echo 7200011140000000000 7200011139993250000 | ./twinprimes_ssoz159
threads = 8                        // 16
using Prime Generator parameters for P5
segment size = 65536 resgroups; seg array is [1 x 1024] 64-bits
twinprime candidates = 675003; resgroups = 225001
each of 3 threads has nextp[2 x 2636377] array
setup time = 5.296482074 secs      // 4.74022821 secs
perform twinprimes ssoz sieve
3 of 3 twinpairs done
sieve time = 0.180924203 secs      // 0.116552963 secs
total time = 5.477426691 secs      // 4.856791579 secs
last segment = 28393 resgroups; segment slices = 4
total twins = 4711; last twin = 7200011139999998808+/-1
```

Constructing nextp

nextp is a table of the resgroups for the first prime multiples for the sieving primes along each restrack. From P5's pcs table we can look at each row and create Table 3 of their first prime multiples resgroups.

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.

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

Note on each row, when two primes have the same resgroup table value they were multiplied. When only one value occurs, its either for a prime square, or a (prime * nonprime) value. Also, for a prime in any resgroup k, its first prime multiple resgroup value on its own row is just: `prime * (k + 1) + k`
For P5's pcs table this is equivalent to: **k * (prime + 31) + ((prm_r * 31) - 2) / modpg**
(This is a property for every pc member in a resgroup for every Pn, for its first multiple on its row).

To construct Table 3, each prime in P5's pcs table multiplies each regroup member, whose products are other table 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 in P5's pcs table, 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 in Table 3 are used to start marking off successive prime multiples 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`** for the inputs range first segment, which is updated at the end of each segment slice to set the first prime multiples for the next segment(s).

For each sieve prime we compute its first multiple resgroup `k` for the restracks of interest, e.g. for twin pair residues. We then determine its regroup `k'`≥ kmin, where **`kmin`** is the resgroup for the `start_num`, input value (kmin = 1 if one input given). Thus `k'`≥ 0 is the number of resgroups starting from `kmin`.

In the picture below, `k` is a prime's 1st multiple resgroup on a row, and `k'`its projection relative to `kmin`. If `k` ≥ kmin, then `k'`= k - kmin. Thus if kmin = 3 and k = 7, `k'`=4 is its first resgroup inside the segment starting at kmin. If `k` = kmin then `k'`= 0, i.e. that first prime multiple starts at the segment's beginning.

```
                    |—— p ——|
k                   |rem|     k'
|…….……..……….|…...|--------|----------------------
                      kmin
```

If `k` < `kmin`**,** we compute prime's multiple closest to `kmin`, i.e. where `k'`= 0...prime-1 resgroups ≤ `kmin`:

`k'` = (`kmin` - k) % prime –> value of rem in picture
`k'` = prime - `k'` if `k'` > 0 –> translated `k'` value > `kmin`

Ex: for prime 7 on `rt0`, let `k` = 7, `kmin` = 21: then `k'` = (21 - 7) % 7 = 0; to start from (multiple of 7).
Ex: for prime 7 on `rt0`, let `k` = 7, `kmin` = 25: then `k'` = (25 - 7) % 7 = 4; `k'` = 7 - 4 = 3; to start from.

In software, we can reassign the variable `k` to use for `k'`, so the (Crystal, et al) code just becomes:

`k` < `kmin` ? (`k` = (`kmin` - `k`) % prime; `k` = prime - `k` if `k` > 0) : `k` -= `kmin`

It should be noted, while the sieve primes have at least 1 multiple within the inputs range, some may not have multiples on each restrack, especially for small ranges, and for them `k` > `kmax`. If this happens for both residue pairs, those primes could be discarded from the primes lists for those residues sieves. For general purposes though, it won't happen enough to increase performance to justify the extra code.

To make the process|code simple, the `k` values for each sieve prime are generated and stored in `nextp`, without worry if they're > `kmax`. If a prime's `k` is larger than a segment size its skipped for it (not used to mark prime multiples) and reduced|updated by `kn` with smaller values for the next segment(s). When less than a segment size, it's used in the residues sieve to mark prime multiples. Thus in `twins_sieve`, only primes with multiples in a segment for each restrack are used to mark prime multiples, or skipped.

A unique `nextp` array is created for each residues pair in each thread for the sieving primes. Thus for twin|cousin primes, `nextp` holds their first prime multiples resgroups values for each segment slice for both residue pairs restracks. Thus its memory increases with inputs values (more sieving primes) and larger generators (more residue pairs), though active memory use will be determined by the number of parallel threads holding onto memory. How different languages manage memory affects the size and throughput they can achieve for various inputs and ranges, for a system's memory size and profile.

Creating nextp for SSoZ

In the SoZ, a prime's residue **r** multiplies each Pn residue $\mathbf{r_i}$ and **(r * $\mathbf{r_i}$) mod modpg** maps to a unique restrack $\mathbf{r_t}$ in some resgroup **k**, is the starting point to mark off that prime's multiples for that $\mathbf{r_i}$. We now want to multiply **r** by the $\mathbf{r_i}$ that makes **(r * $\mathbf{r_i}$)** be on a given restrack $\mathbf{r_t}$, for each sieving prime.

Thus if for some $\mathbf{r_i}$, **(r * $\mathbf{r_i}$) mod modpg = $\mathbf{r_t}$**, to find the $\mathbf{r_i}$ that maps each **r** to a specific $\mathbf{r_t}$ we do:

$$\begin{aligned} r * r_i &= r_t & mod\ modpg \\ (r * r_i) * r^{-1} &= r_t * r^{-1} & mod\ modpg \\ r_i * \left(r * r^{-1}\right) &= r_t * r^{-1} & mod\ modpg \\ r_i * 1 &= r_t * r^{-1} & mod\ modpg \\ r_i &= r_t * r^{-1} & mod\ modpg \end{aligned}$$

Where for $\mathbf{r^{-1}}$, **r_inv** = **modinv(r, modpg)** in the code, with **r** being the residue for a sieve prime.
(A property of prime generators is that every residue has an inverse, either itself or another residue.)
Now **kn = (r * $\mathbf{r_i}$ - 2) / modpg**, and **k = (prime - 2) / modpg**, so again: **kpm = k * (prime + $\mathbf{r_i}$) + kn**
If **r_inv** is a prime's residue inverse, and $\mathbf{r_t}$ the desired restrack: **$\mathbf{r_i}$ = (r_inv * $\mathbf{r_t}$ - 2) mod modpg + 2**

For each residues pair, **`nextp_init`** creates the **`nextp`** array of the sieve primes first resgroup multiples relative to **kmin**, for the $\mathbf{r_t}$ values **r_lo** and **r_hi**, the upper|lower twinpair residues. With no loss of generality, it can be used to construct `nextp` for any architecture for any number of specified restracks.

nextp_init

```
def nextp_init(rhi, kmin, modpg, primes, resinvrs)
  # Initialize 'nextp' array for twinpair upper residue rhi in 'restwins'.
  # Compute 1st prime multiple resgroups for each prime r0..sqrt(N) and
  # store consecutively as lo_tp|hi_tp pairs for their restracks.
  nextp = Slice(UInt64).new(primes.size*2) # 1st mults array for twinpair
  r_hi, r_lo = rhi, rhi - 2              # upper|lower twinpair residue values
  primes.each_with_index do |prime, j|   # for each prime r0..sqrt(N)
    k = (prime - 2) // modpg             # find the resgroup it's in
    r = (prime - 2) %  modpg + 2         # and its residue value
    r_inv = resinvrs[r].to_u64           # and residue inverse
    rl = (r_inv * r_lo - 2) % modpg + 2  # compute r's ri for r_lo
    rh = (r_inv * r_hi - 2) % modpg + 2  # compute r's ri for r_hi
    kl = k * (prime + rl) + (r * rl - 2) // modpg # kl 1st mult resgroup
    kh = k * (prime + rh) + (r * rh - 2) // modpg # kh 1st mult resgroup
    kl < kmin ? (kl = (kmin - kl) % prime; kl = prime - kl if kl > 0) : (kl -= kmin)
    kh < kmin ? (kh = (kmin - kh) % prime; kh = prime - kh if kh > 0) : (kh -= kmin)
    nextp[j * 2] = kl.to_u64             # prime's 1st mult lo_tp resgroup val in range
    nextp[j * 2 | 1] = kh.to_u64         # prime's 1st mult hi_tp resgroup val in range
  end
  nextp
end
```

Inputs:
`rhi` – hi residue value for this twinpair
`kmin` – resgroup value for start_num
`modpg` – modulus value for chosen pg
`primes` – array of sieving primes
`resinvrs` – array of residues modular inverses

Output:
`nextp` – array of primes 1st mults for given residues

Twins|Cousins SSoZ

Let’s now construct the 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 pcs table, we use just the consecutive pc residue tracks, whose residues table is below. A twin prime occurs when both twin pair pc values in a column are prime (not colored), e.g. [191:193].

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, so the last (largest) twin pair ≤ 541 for [521:523] is written as 522 ± 1.

As shown before, the number of twin|cousin residue pairs are equal to: $\mathbf{(p_n - 2)\# = p_n^{-2}\# = \Pi\ (p_n - 2)}$
Thus P5 has 3 residue pairs for each. Below are the three Cousin Prime pairs taken from P5’s pcs table.

Table 5. Cousin 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
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

The SSoZ algorithm is the same for both, with their coding only differing to deal with accounting for low input values ranges, as the first cousin prime is defined as [3:7] and first twins are [3:5], [5:7].

Up to 541, there are 25 twin and 27 cousin primes. Their ratio over increasingly larger input ranges remains close to unity, as their pairs count, and pair prime values, infinitely increase, [3], [4].

Residues Sieve Description

The Segmented Sieve of Zakiya (SSoZ) is a memory efficient way to find the primes using a given Pn. For an input range defined by a **`start_num`** and **`end_num`**, it divides the range into **segments**, which are efficiently sized to fit into usable memory for processing. This allows the reuse of the same memory to process long number ranges that otherwise would require more memory than a system has to use.

A standard segment slice is **`ks`** resgroups, with last one **`ks’`** usually less. For a given Pn and range size **`set_sieve_parameters`** determines its optimal memory size, which is set to be a multiple of 64 (bits).

```
               |   ks   |   ks   |   ks    |   ks   |   ks   |   ks   |   ks   |  ks’ |
Fig. 1     kmin                                                                   kmax
               |............|............|............|............|...........|............|............|..........|
        start_num                                                                  end_num
```

Here `start|end_num` are the lo|hi values that define a number range of interest. They also define the absolute values for **`kmin`** and **`kmax`** for a given Pn generator, as these resgroups cover these input values.

When only one input is given it becomes `end_num`, whose resgroup determines `kmax`, and `start_num` is set to 3 (low prime for first twin [3:5]), and `kmin` set to 1 (min number of resgroups). The SSoZ sieve adjusts `kmin|kmax` for each residues pair when necessary, to ensure only their pc values within the inputs range are processed.

For example, if **start_num** = **342** and **end_num** = **540**, we see below the valid in-range pc values. Here `kmin` = 12 and `kmax` = 18 are the global resgroup values, which are adjusted as needed in **`twins_sieve`** for each residues pair. For [11:13], 341 < 342, so its `kmin` is increased to 13, whose values are all in the range. Conversely for twinpair [29:31], pc 541 > 540 is outside the range, so its kmax is reduced to 17, whose resgroup values are now all in the range. For twinpair [17:19] no adjustment is needed (done).

Thus for each residues pair, we check if the numerated r_lo pc value in `kmin` is < `start_num`, and if so increment `kmin`, and check if the numerated r_hi pc value in `kmax` is > `end_num`, and decrement `kmax` if so. In `twins_sieve` the adjusted `kmin|kmax` values are determined then used in **`nextp_init`** to create **`nextp`** for the sieving primes to begin performing the residues sieve for the first segment in the range.

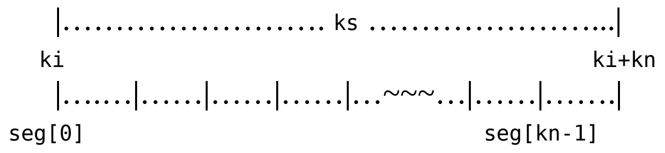
Table 6. Twin Primes Residues Tracks Table for range 342 – 540.

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

In `twins_sieve` segment array **seg,** its resgroups size **ks** is a multiple of 64-bit mem elements, where each bit represents a residues pair resgroup. Thus a resgroup k maps to bit: (k mod 64) in mem elem seg[k / 64], where (k mod 64) masks k's lower 6 bits: (k & 0x3F), and (k / 64) right shifts k by 6 bits. This is coded as: seg[(kn - 1) >> 6], bit value: 1 << ((kn - 1) & 63), (>>|<< are right|left bit-shift opts).

Ex: for ks = 131072 resgroups, seg size is 2048 64-bit mem elements
for resgroup k = 89257, it maps to seg[1394], bit 2^{40}, mem value = 1 << 40 = 1099511627776

Fig. 2



`ki` is the absolute resgroup value to start each segment slice (in Fig. 1) initialized to kmin-1 (0 indexed arrays). **`kn`** is the resgroups size for each segment slice. It's initialized to **`ks`**, but if the last segment slice **`ks'`** < **`ks`** resgroups it's set to its slice size.

To sieve for twin primes, etc, each instance of `twins_sieve` processes a unique twinpair for the entire inputs range split into `ks` resgroup size segments. It first determines the adjusted kmin|kmax values for the twinpair residues, then creates their initial `nextp` array of first resgroup sieve prime multiples k values. Using them, it iterates over the sieve primes, computes|updates their prime multiples k values, and sets them to '1' in `seg` for each residues pair, until k > kn, the k value past the end of the current segment. When k > kn it updates it to: k = k – kn, which is the first k multiple value into the next segment, and stores it back into `nextp` for that prime to update it to use for the next segment(s).

This is the Crystal code to mark a prime's resgroup multiples in `seg` to '1'. This is done for the lo|hi residues pair, and if either resgroup member is a prime's multiple that resgroup isn't a twinprime.

```
k = nextp.to_unsafe[j * 2]          # starting from this resgroup in seg
while k < kn                        # mark primenth resgroup bits prime mults
  seg[k >> s] |= 1_u64 << (k & bmask)
  k += prime  end                   # set resgroup for prime's next multiple
nextp.to_unsafe[j * 2] = k - kn     # save 1st resgroup in next eligible seg
```

When the residues sieve finishes `seg` contains the resgroup bit positions for the twin primes. Because `seg` is set to all '0's to start each segment, we need to set to '1' any unused hi bits in its last mem elem `ks'` is in when it's not a multiple of 64. Algorithmically this only needs to be done for the last segment. However, doing it after every segment is faster in software, as it eliminates the branching code to check for the last segment, and is more efficient to compile|run. Below is the Crystal code to perform this.

```
seg.to_unsafe[(kn - 1) >> s] |= ~1u64 << ((kn - 1) & bmask)
```

If kn = 89257 for the last segment, only the first 1395 64-bit `seg` mem elems are used, up to the 41st bit in the last elem, so we need to set to '1' its bit values $2^{41}..2^{63}$, because (89257-1 & 63) = 40, for bit 2^{40}. Thus we invert 1 to be: 11111111..1110 and left-shift it 40 bits, which is ORed with the last mem elem. If kn is a multiple of 64, (kn – 1) & bmask = 63, shifts the bits to be all 0s, and thus when ORed doesn't change `seg's` last mem value. Thus left shifts of n = 0..62 bits mask all the upper bit values: $2^{63}... 2^{n+1}$.

Once all the nonprime bits are set we can count|numerate the primes. We read each `seg[0..kn-1]` and invert the bits, and use `popcount` to count the '1's (as primes) for each `seg[i]` (the Rust code counts the '0's directly), and sum their segment count in variable `cnt`.

If `cnt` > 0 we find the largest prime resgroup in the segment. We first update the total pairs count with `sum += cnt`. Then `upk` is set to the last resgroup value in the segment, then loops backward checking for the first bit that's prime ('0'), and then `upk` holds the largest|last prime pair resgroup in the segment. Its absolute resgroup value in the inputs range is then: `hi_tp = ki + upk`. For each segment slice its value is updated to a larger value, and at the end holds the largest absolute resgroup for these residues pair in the inputs range. The `r_hi` prime value is numerated and returned as: `hi_tp * modpg + r_hi`, along with the total prime pairs count in the range, in variable `sum`.

```
seg.to_unsafe[(kn - 1) >> s] |= ~1u64 << ((kn - 1) & bmask)
cnt = 0                      # count the twinprimes in the segment
seg[0..(kn - 1) >> s].each { |m| cnt += (~m).popcount }
if cnt > 0                   # if segment has twinprimes
  sum += cnt                 # add segment count to total range count
  upk = kn - 1               # from end of seg count back to largest tp
  while seg[upk >> s] & (1_u64 << (upk & bmask)) != 0; upk -= 1 end
  hi_tp = ki + upk           # set its full range resgroup value
end
```

`twins_sieve` can be modified for different purposes. The code to find the largest prime pair can be removed if all you want is their count. I also originally had code to print out the `r_hi` primes in each segment as a validity check (only for small ranges). However, if you really wanted to see|record the twins, a better way may be to return `ki|seg` for each segment and externally store|process them later for any desired range of interest. (This, of course, would be very memory intensive.)

Twin Primes Example

Using our example to find the twin primes ≤ 541 with P5, let's see how to processes the first twin pair residues [11:13] with **`kmax = 18`**. `twin_sieve` can perform the sieve for each pair in a separate thread.

`set_sieve_parameters` sets the segment size, but here I'll set it to **ks = 6**. Thus, the `seg` array will represent 6 resgroups. Below is the twin pair table for [11:13] separated it into 3 segment slices of 6 resgroups each. Underneath it is what each `seg` array will look like after processing for each slice. (`seg` conceptually is a bitarray, so each `seg[i]` is just 1 bit. I later show an implementation using a bitarray, which makes the code simpler|shorter, and faster, depending on a language's implementation.)

Table 7.

k	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
rt11	11	41	71	101	131	161	191	221	251	281	311	341	371	401	431	461	491	521
rt13	13	43	73	103	133	163	193	223	253	283	313	343	373	403	433	463	493	523

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

`nextp_init` initializes **`netxp`** for the sieve primes [7, 11, 13, 17, 19, 23] for residues 11 and 13, taking the values shown in Table 3. For each lo|hi residue, their k values are stored as consecutive pairs in `nextp` and `seg` is created and initialized to all primes ('0').

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

Initial nextp[11:13]						
2j	0	2	4	6	8	10
2j+1	1	3	5	7	9	11
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

For each prime **j** in **primes**, **nextp[2j|2j+1]** give the pairs **k's** to start marking off prime's multiples (by incrementing k by prime's value). When **k** > **kn**, (here kn is always 6), it's reduced by it: **`k = k - 6`**, and updates `nextp` with the new **k** values for the next segment. Below shows the changes to `nextp` and `seg` in `twins_sieve`. (It's coincidental here the index size for `primes` and `nextp` are the segment size.)

Start for Segment 1 nextp[11:13]						
2j	0	2	4	6	8	10
2j+1	1	3	5	7	9	11
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]						
2j	0	2	4	6	8	10
2j+1	1	3	5	7	9	11
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]						
2j	0	2	4	6	8	10
2j+1	1	3	5	7	9	11
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

Below is the Crystal code to perform the residues sieve (here for twins) for a given residues pair.

twins_sieve

```
def twins_sieve(r_hi, kmin, kmax, ks, start_num, end_num, modpg, primes, resinvrs)
  # Perform in thread the ssoz for given twinpair residues for kmax resgroups.
  # First create|init 'nextp' array of 1st prime mults for given twinpair,
  # stored consequtively in 'nextp', and init seg array for ks resgroups.
  # For sieve, mark resgroup bits to '1' if either twinpair restrack is nonprime
  # for primes mults resgroups, and update 'nextp' restrack slices acccordingly.
  # Return the last twinprime|sum for the range for this twinpair residues.
  s = 6                                              # shift value for 64 bits
  bmask = (1 << s) - 1                               # bitmask val for 64 bits
  sum, ki, kn  = 0_u64, kmin-1, ks                   # init these parameters
  hi_tp, k_max = 0_u64, kmax                         # max twinprime|resgroup
  seg = Slice(UInt64).new(((ks - 1) >> s) + 1)       # seg array of ks resgroups
  ki += 1    if ((ki * modpg) + r_hi - 2)  < start_num # ensure lo tp in range
  k_max -= 1 if ((k_max - 1) * modpg + r_hi) > end_num # ensure hi tp in range
  nextp = nextp_init(r_hi, ki, modpg, primes,resinvrs) # init nextp array
  while ki < k_max                        # for ks size slices upto kmax
    kn = k_max - ki if ks > (k_max - ki) # adjust kn size for last seg
    primes.each_with_index do |prime, j| # for each prime r0..sqrt(N)
                                          # for lower twinpair residue track
      k = nextp.to_unsafe[j * 2]         # starting from this resgroup in seg
      while k < kn                        # mark primenth resgroup bits prime mults
        seg.to_unsafe[k >> s] |= 1_u64 << (k & bmask)
        k += prime  end                   # set resgroup for prime's next multiple
      nextp.to_unsafe[j * 2] = k - kn    # save 1st resgroup in next eligible seg
                                          # for upper twinpair residue track
      k = nextp.to_unsafe[j * 2 | 1]     # starting from this resgroup in seg
      while k < kn                        # mark primenth resgroup bits prime mults
        seg.to_unsafe[k >> s] |= 1_u64 << (k & bmask)
        k += prime  end                   # set resgroup for prime's next multiple
      nextp.to_unsafe[j * 2 | 1]= k - kn # save 1st resgroup in next eligible seg
    end                                   # set as nonprime unused bits in last seg[n]
                                          # so fast, do for every seg[i]
    seg.to_unsafe[(kn - 1) >> s] |= ~1u64 << ((kn - 1) & bmask)
    cnt = 0                               # count the twinprimes in the segment
    seg[0..(kn - 1) >> s].each { |m| cnt += (~m).popcount } # invert to count ‘1’s
    if cnt > 0                            # if segment has twinprimes
      sum += cnt                          # add segment count to total range count
      upk = kn - 1                        # from end of seg, count back to largest tp
      while seg.to_unsafe[upk >> s] & (1_u64 << (upk & bmask)) != 0; upk -= 1 end
      hi_tp = ki + upk                    # set its full range resgroup value
    end
    ki += ks                              # set 1st resgroup val of next seg slice
    seg.fill(0) if ki < k_max             # set next seg to all primes if in range
  end                                     # when sieve done, numerate largest twin
                                          # for ranges w/o twins set largest to 1
  hi_tp = (r_hi > end_num || sum == 0) ? 1 : hi_tp * modpg + r_hi
  {hi_tp.to_u64, sum.to_u64}              # return largest twinprime|twins count
end
```

Inputs:
`ks` – resgroups segment size
`rhi` – hi residue value for this twinpair
`modpg` – modulus value for chosen pg
`kmin` – total number resgroups upto for start_num
`kmax` – total number resgroups upto for end_num
`primes` – array of sieving primes
`resinvrs` – array of modular inverses for residues
`end_num` – inputs high value
`start_num` – inputs low value

Outputs:
`sum` – count of twinpairs for input range
`hi_tp` – hi prime for largest twinprime in range

Starting with Crystal 1.4.0 (April 7, 2022) its `bitarray` implementation was highly optimized, making it faster than the 64-bit mem array for `seg` on the AMD 5900HX, while making the code substantially simpler to read|write and shorter. Below is the Crystal version using a `bitarray` for the `seg` array.

```
def twins_sieve(r_hi, kmin, kmax, ks, start_num, end_num, modpg, primes, resinvrs)
  # Perform in thread the ssoz for given twinpair residues for kmax resgroups.
  # First create|init 'nextp' array of 1st prime mults for given twinpair,
  # stored consequtively in 'nextp', and init seg array for ks resgroups.
  # For sieve, mark resgroup bits to '1' if either twinpair restrack is nonprime
  # for primes mults resgroups, and update 'nextp' restrack slices acccordingly.
  # Return the last twinprime|sum for the range for this twinpair residues.
  sum, ki, kn  = 0_u64, kmin-1, ks                   # init these parameters
  hi_tp, k_max = 0_u64, kmax                         # max twinprime|resgroup
  seg = BitArray.new(ks)                             # seg array of ks resgroups
  ki += 1    if ((ki * modpg) + r_hi - 2)  < start_num # ensure lo tp in range
  k_max -= 1 if ((k_max - 1) * modpg + r_hi) > end_num # ensure hi tp in range
  nextp = nextp_init(r_hi, ki, modpg, primes,resinvrs) # init nextp array
  while ki < k_max                        # for ks size slices upto kmax
    kn = k_max - ki if ks > (k_max - ki) # adjust kn size for last seg
    primes.each_with_index do |prime, j| # for each prime r0..sqrt(N)
                                          # for lower twinpair residue track
      k = nextp.to_unsafe[j * 2]          # starting from this resgroup in seg
      while k < kn                        # until end of seg
        seg.unsafe_put(k, true)           # mark primenth resgroup bits prime mults
        k += prime  end                   # set resgroup for prime's next multiple
      nextp.to_unsafe[j * 2] = k - kn     # save 1st resgroup in next eligible seg
                                          # for upper twinpair residue track
      k = nextp.to_unsafe[j * 2 | 1]      # starting from this resgroup in seg
      while k < kn                        # until end of seg
        seg.unsafe_put(k, true)           # mark primenth resgroup bits prime mults
        k += prime  end                   # set resgroup for prime's next multiple
      nextp.to_unsafe[j * 2 | 1]= k - kn # save 1st resgroup in next eligible seg
    end
    cnt = seg[...kn].count(false)         # count|store twinprimes in segment
    if cnt > 0                            # if segment has twinprimes
      sum += cnt                          # add segment count to total range count
      upk = kn - 1                        # from end of seg, count back to largest tp
      while seg.unsafe_fetch(upk); upk -= 1 end
      hi_tp = ki + upk                    # set its full range resgroup value
    end
    ki += ks                              # set 1st resgroup val of next seg slice
    seg.fill(false) if ki < k_max         # set next seg to all primes if in range
  end                                     # when sieve done, numerate largest twin
                                          # for ranges w/o twins set largest to 1
  hi_tp = (r_hi > end_num || sum == 0) ? 1 : hi_tp * modpg + r_hi
  {hi_tp.to_u64, sum.to_u64}              # return largest twinprime|twins count
end
```

The code to find the largest twinprime in the range comes for FREE, and removing it has no detectable increase in speed, and for Crystal may even be a wee tad bit slower.

```
    sum += seg[...kn].count(false)        # count|store twinprimes in segment
    ki += ks                              # set 1st resgroup val of next seg slice
    seg.fill(false) if ki < k_max         # set next seg to all primes if in range
  end
  sum.to_u64                              # return twinprimes count in range
end
```

In general, a `bitarray’s` performance depends on the language’s implementation (test to determine), but should make the code simpler|shorter to read|write, while the memory array model should be more ubiquitous, and implementable for languages without (native of external) bitarrays.

gcd

```
def gcd(m, n)
  while m|1 != 1; t = m; m = n % m; n = t end
  m
end
```

Inputs:	Output:
`n` – even pg modulus value	`m` – gcd of inputs; (m, n) are coprime if 1
`m` – an odd pc value < pg modulus n	

This is a customized gcd (greatest common divisor) function that uses residue properties to shorten the time of the Euclidean gcd algorithm (https://en.wikipedia.org/wiki/Euclidean_algorithm). Here `m` is an odd residue candidate < `n`, the even modulus value. Some of the language implementations just use the gcd function provided with them.

modinv

```
def modinv(a0, m0)
  return 1 if m0 == 1
  a, m = a0, m0
  x0, inv = 0, 1
  while a > 1
    inv -= (a // m) * x0
    a, m = m, a % m
    x0, inv = inv, x0
  end
  inv += m0 if inv < 0
  inv.to_u64
end
```

```
def modinv1(r, m)
  r = inv = r.to_u64
  while (r * inv) % m != 1
    inv = (inv % m) * r
  end
  inv % m
end
```

Inputs:	Output:
`a0` – odd pc value < modulus `m0`	`inv` – inverse of, `a0` mod `m0`, e.g. `a0*inv` ≡ 1 mod `m0`
`m0` – even pg modulus value	

The function on the left is the standard modular inverse function (taken from Rosetta Code).

The code on the right uses the residue property that – $\mathbf{r_i * r_i^n \equiv 1 \bmod modpg}$ – for some n ≥ 1, i.e. the modular inverse of residue r_i is itself raised to some power n. This is faster for generators P3 and P5, with small number of residues, but becomes comparatively slower for generators with more residues.

For P5’s residues: [7, 11, 13, 17, 19, 23, 29, 31]
It’s inverses are: [13, 11, 7, 23, 19, 17, 29, 1]
Inverse power n: [3, 1, 3, 3, 1, 3, 1, 1]

For a chosen Pn generator, `gen_pg_parameters` produces its parameters used to perform the SSoZ. It uses `gcd` to determine the residues and `modinv` to compute their inverses.

gen_pg_parameters

```
def gen_pg_parameters(prime)
  # Create prime generator parameters for given Pn
  puts "using Prime Generator parameters for P#{prime}"
  primes = [2, 3, 5, 7, 11, 13, 17, 19, 23]
  modpg, res_0 = 1, 0                   # compute Pn's modulus and res_0 value
  primes.each { |prm| res_0 = prm; break if prm > prime; modpg *= prm }

  restwins = [] of Int32                # save upper twinpair residues here
  inverses = Array.new(modpg + 2, 0)    # save Pn's residues inverses here
  pc, inc, res = 5, 2, 0                # use P3's PGS to generate pcs
  while pc < (modpg >> 1)               # find PG's 1st half residues
    if gcd(pc, modpg) == 1              # if pc a residue
      mc = modpg - pc                   # create its modular complement
      inverses[pc] = modinv(pc, modpg)  # save pc and mc inverses
      inverses[mc] = modinv(mc, modpg)  # if in twinpair save both hi residues
      restwins << pc << mc + 2 if res + 2 == pc
      res = pc                          # save current found residue
    end
    pc += inc; inc ^= 0b110             # create next P3 seq pc: 5 7 11 13 17...
  end
  restwins.sort!; restwins <<(modpg + 1) # last residue is last hi_tp
  inverses[modpg+1] = 1; inverses[modpg-1] = modpg - 1 # last 2 are self inverses
  {modpg, res_0, restwins.size, restwins, inverses}
end
```

Inputs:
`prime` – Pn prime value 5, 7… 17

Outputs:
`res_0` – first residue of selected Pn (next prime > Pn `prime`)
`modpg` – modulus for generator Pn; value = (`prime`)#
`inverses` – array of the pg residue inverses, size = (`prime`-1)#
`restwins` – ordered array of the hi pg twinpair (tp) values
`restwins.size` – the number of pg twinpairs = (`prime`-2)#

For a given prime number, it generates its primorial value for `modpg`, and keeps its r_0 value in `res_0`. It then generates all the residues. It uses P3's PGS to generate Pn's first half rcs. It checks if they're coprime to `modpg` to identify the residues. For each residue it creates its modular complement (`mc`) and stores both inverses at their address values. It then determines if the residue is part of a twin (cousin) pair, and if so, then so is its complement, and stores both `hi` pair values in `restwins`.

Upon generating all the residues, and storing their inverses and twin (cousin) pairs `hi` residues, the `restwins` array is sorted to put them in sequential order, then the last `hi` residue for the last twin pair `modgp±1` are included as the last ones. (For cousin primes, we include the `hi` residue for the pivot pair (`modpg/2 + 2`)and then sort the array).

Finally, the inverses for the last two residues `modgp±1` are added at their address locations, and the outputs are returned for use in `set_sieve_parameters`.

Given the input values, `set_sieve_parameters` determines which prime generator to use, generates its parameters, then determines the range parameters and segment size to use. Here I use a rudimentary tree algorithm to determine for my laptops the switch points for using different generators. This can be made much more sophisticated and adaptable by also accounting for the number of system threads and cache and ram memory size, to pick better segment size values and generators for a given inputs range.

set_sieve_parameters

```
def set_sieve_parameters(start_num, end_num)
  # Select at runtime best PG and segment size parameters for input values.
  # These are good estimates derived from PG data profiling. Can be improved.
  nrange = end_num - start_num
  bn, pg = 0, 3
  if end_num < 49
    bn = 1; pg = 3
  elsif nrange < 77_000_000
    bn = 16; pg = 5
  elsif nrange <  1_100_000_000
    bn = 32; pg = 7
  elsif nrange < 35_500_000_000
    bn = 64; pg = 11
  elsif nrange < 14_000_000_000_000
    pg = 13
    if    nrange > 7_000_000_000_000; bn = 384
    elsif nrange > 2_500_000_000_000; bn = 320
    elsif nrange >   250_000_000_000; bn = 196
    else  bn = 128
    end
  else
    bn = 384; pg = 17
  end
  modpg, res_0, pairscnt, restwins, resinvrs = gen_pg_parameters(pg)
  kmin = (start_num-2) // modpg + 1      # number of resgroups to start_num
  kmax = (end_num - 2) // modpg + 1      # number of resgroups to end_num
  krange = kmax - kmin + 1               # number of resgroups in range, at least 1
  n = krange < 37_500_000_000_000 ? 4 : (krange < 975_000_000_000_000 ? 6 : 8)
  b = bn * 1024 * n                      # set seg size to optimize for selected PG
  ks = krange < b ? krange : b           # segments resgroups size

  puts "segment size = #{ks} resgroups for seg bitarray"
  maxpairs = krange * pairscnt           # maximum number of twinprime pcs
  puts "twinprime candidates = #{maxpairs}; resgroups = #{krange}"
  {modpg, res_0, ks, kmin, kmax, krange, pairscnt, restwins, resinvrs}
end
```

Inputs:
`end_num` —— high input value (min of 3)
`start_num` – low input value (min of 3)

Outputs:
`ks` – number of residue groups set for segment size
`res_0` – first residue of selected Pn (next prime > Pn `prime`)
`modpg` – modulus value for chosen pg
`kmin` – number resgroups to start_num
`kmax` – number resgroups to end_num
`krange` – number of resgroups for inputs range (at least 1)
`pairscnt` – number of twinpairs for selected pg
`resinvrs` – modular inverses array for the residues
`restwins` – hi residue values array for each twinpair

Finally, shown below is the Crystal version of the main routine `twinprimes_ssoz`. It accepts the inputs, performs the residues sieve, times the different parts of the process, and generates the program outputs.

twinprimes_ssoz

```
def twinprimes_ssoz()
  end_num   = {ARGV[0].to_u64, 3u64}.max
  start_num = ARGV.size > 1 ? {ARGV[1].to_u64, 3u64}.max : 3u64
  start_num, end_num = end_num, start_num if start_num > end_num
  start_num |= 1                        # if start_num even increase by 1
  end_num = (end_num - 1) | 1           # if end_num even decrease by 1
  start_num = end_num = 7 if end_num - start_num < 2

  puts "threads = #{System.cpu_count}"
  ts = Time.monotonic                   # start timing sieve setup execution
                                        # select Pn, set sieving params for inputs
  modpg, res_0, ks, kmin, kmax, krange,
    pairscnt, restwins, resinvrs = set_sieve_parameters(start_num, end_num)

  # create sieve primes <= sqrt(end_num), only use those whose multiples within inputs range
  primes = end_num < 49 ? [5] : sozpg(Math.isqrt(end_num), res_0, start_num, end_num)

  puts "each of #{pairscnt} threads has nextp[2 x #{primes.size}] array"

  lo_range = restwins[0] - 3            # lo_range = lo_tp - 1
  twinscnt = 0_u64                      # determine count of 1st 4 twins if in range for used Pn
  twinscnt += [3, 5, 11, 17].select { |tp| start_num <= tp <= lo_range }.size unless end_num == 3

  te = (Time.monotonic - ts).total_seconds.round(6)
  puts "setup time = #{te} secs"        # display sieve setup time
  puts "perform twinprimes ssoz sieve"
  t1 = Time.monotonic                   # start timing ssoz sieve execution

  cnts = Array(UInt64).new(pairscnt, 0)  # number of twinprimes found per thread
  lastwins = Array(UInt64).new(pairscnt, 0) # largest twinprime val for each thread
  done = Channel(Nil).new(pairscnt)

  threadscnt = Atomic.new(0)            # count of finished threads
  restwins.each_with_index do |r_hi, i|  # sieve twinpair restracks
    spawn do
      lastwins[i], cnts[i] = twins_sieve(r_hi, kmin, kmax, ks, start_num, end_num, modpg, primes,
                                         resinvrs)
      print "\r#{threadscnt.add(1)} of #{pairscnt} twinpairs done"
      done.send(nil)
  end end
  pairscnt.times { done.receive }       # wait for all threads to finish
  print "\r#{pairscnt} of #{pairscnt} twinpairs done"

  last_twin = lastwins.max              # find largest hi_tp twinprime in range
  twinscnt += cnts.sum                  # compute number of twinprimes in range
  last_twin = 5 if end_num == 5 && twinscnt == 1
  kn = krange % ks                      # set number of resgroups in last slice
  kn = ks if kn == 0                    # if multiple of seg size set to seg size
  t2 = (Time.monotonic - t1).total_seconds       # sieve execution time

  puts "\nsieve time = #{t2.round(6)} secs"      # ssoz sieve time
  puts "total time = #{(t2 + te).round(6)} secs" # setup + sieve time
  puts "last segment = #{kn} resgroups; segment slices = #{(krange - 1)//ks + 1}"
  puts "total twins = #{twinscnt}; last twin = #{last_twin - 1}+/-1"
end

twinprimes_ssoz
```

Program Output

Below is typical program output, shown here for Rust, for single and two input values (order doesn't matter), run on an Intel i7-6700HQ Linux based laptop. The programs is run in a terminal with the command-line interface (cli) shown, and display the output shown.

```
$ echo 5000000000 | ./twinprimes_ssoz
threads = 8
using Prime Generator parameters for P11
segment size = 262144 resgroups; seg array is [1 x 4096] 64-bits
twinprime candidates = 292207905; resgroups = 2164503
each of 135 threads has nextp[2 x 6999] array
setup time = 0.000796737 secs
perform twinprimes ssoz sieve
135 of 135 twinpairs done
sieve time = 0.184892352 secs
total time = 0.185704753 secs
last segment = 67351 resgroups; segment slices = 9
total twins = 14618166; last twin = 4999999860+/-1

$ echo 100000000000 200000000000 | ./twinprimes_ssoz
threads = 8
using Prime Generator parameters for P13
segment size = 524288 resgroups; seg array is [1 x 8192] 64-bits
twinprime candidates = 4945055940; resgroups = 3330004
each of 1485 threads has nextp[2 x 37493] array
setup time = 0.003883411 secs
perform twinprimes ssoz sieve
1485 of 1485 twinpairs done
sieve time = 3.819838338 secs
total time = 3.823732178 secs
last segment = 184276 resgroups; segment slices = 7
total twins = 199708605; last twin = 199999999890+/-1
```

The program output is described as follows:

Line 0 is the cli input command. When 2 inputs are given their hi|lo order doesn't matter.
Line 1 shows the number of available system threads,.
Line 2 shows the Pn generator selected based on the inputs.
Line 3 shows the selected resgroup segment size ks, and number of 64-bit memory elements (ks / 64) for the segment array.
Line 4 shows the number of twinprime candidates for the number of resgroups spanning the inputs range. In the second example, (kmax – kmin + 1) = 3,330,004 resgroups x 1485 (number of P13 twinpairs) = 4,945,055,940 twinprime candidates.
Line 5 shows the number of twinpairs for the selected PG (here 1485 for P13) and the size of the nextp array, which shows the number of sieving primes used (6999 and 37493 for theses examples.
Line 6 shows the time to select and generate Pn's parameters and the sieve primes.
Line 7 announces when the residues sieve process starts.
Line 8 is a dynamic display showing in realtime how many twinpair threads are done, until finished.
Line 9 shows the runtime for the residues sieve.
Line 10 shows the combined setup and residues sieve times.
Line 11 shows how many resgroups were in the last segment slice and the number of segment slices.
Line 12 shows the number of twinprimes for the inputs range, and the value of the largest one.

Performance

The SSoZ performs optimally on multi-core systems with parallel operating threads. The more available threads the higher the possible performance. To show this, I provide data from two systems.

System 1: Intel i7-6700HQ, 2.6 – 3.5 GHz, 4C|8T, 16 MB, System76 Gazelle (2016) laptop.
System 2: AMD 5900HX, 3.3 – 4.6 GHz, 8C|16T, 16 MB, Lenovo Legion slim 7 (2022) laptop.

For a reference I used **Primesieve** 7.4 [5] – https://github.com/kimwalisch/primesieve – described as "a command-line program and C/C++ library for quickly generating prime numbers...using the segmented sieve of Eratosthenes with wheel factorization." It's a well maintained open source project of highly optimized C/C++ code libraries, which also takes inputs over the 64-bit range (but doesn't produce results for cousin primes). Below are sample outputs for the Rust version of `twinprimes_ssoz` and `Primesieve` performed on both systems.

```
$ echo 378043979 1429172500581 | ./twinprimes_ssoz        $ ./primesieve -c2 378043979 1429172500581
threads = 8                      // 16                     Sieve size = 128 KiB     // 256 KiB
using Prime Generator parameters for P13                   Threads = 8              // 16
segment size = 802816 resgroups; seg array is [1 x 12544]  100%
twinprime candidates = 70654672440; resgroups = 47578904   Seconds: 101.873         // 33.781
each of 1485 threads has nextp[2 x 92610] array            Twin primes: 2601278756
setup time = 0.006171322 secs    // 0.005839409 secs
perform twinprimes ssoz sieve
1485 of 1485 twinpairs done
sieve time = 55.836745969 secs   // 18.062863872 secs
total time = 55.842928445 secs   // 18.068715224 secs
last segment = 212760 resgroups; segment slices = 60
total twins = 2601278756; last twin = 1429172500572+/-1
```

```
$ echo 378043979 14291725005819 | ./twinprimes_ssoz        $ ./primesieve -c2 378043979 14291725005819
threads = 8                       // 16                     Sieve size = 128 KiB    // 256 KiB
using Prime Generator parameters for P17                    Threads = 8             // 16
segment size = 1572864 resgroups; seg array is [1 x 24576]  100%
twinprime candidates = 623572052400; resgroups = 27994256   Seconds: 1218.502       // 471.776
each of 22275 threads has nextp[2 x 268695] array           Twin primes: 22078408103
setup time = 0.036543755 secs     // 0.025222812 secs
perform twinprimes ssoz sieve
22275 of 22275 twinpairs done
sieve time = 675.667368646 secs   // 235,003460103 secs
total time = 675.703922948 secs   // 235.027696883 secs
last segment = 1255568 resgroups; segment slices = 18
total twins = 22078408103; last twin = 14291725004982+/-1
```

I implemented both the twins|cousins ssoz in the 6 programming languages listed here. Again, these are *reference implementations*, and are not necessarily optimum for each language. The Rust versions are the most optimized, and generally the fastest, as they performs the soz algorithm in parallel. The code for each is < 300 ploc (programming lines of code), which highlights the simplicity of the algorithm.

The next page shows tables of benchmark results for the 6 languages implementations, and Primesieve. They are the best times for both systems from multiple runs under different operating conditions. Their code was developed on System 1, and those binaries also run on System 2. Their source code was then compiled on System 2 to compare performance differences, and those were used for the benchmarks. The 6 languages, and their development environments and versions are: C++, Nim 1.6.4 (gcc 11.3.0), D (ldc2 1.28.0, LLVM 12.0.1), Crystal 1.4.1 (LLVM 10.0.0), Rust 1.60, and Go 1.18. They most likely can be improved, and I hope others will create more versions, especially for other compiled languages.

Twin Prime Benchmark Comparisons – Intel i7 6700HQ									
N	Rust	C++	D	Nim	Crystal	Go	Prmsv	Twins Count	Largest in Range
1x10^10	0.35	0.45	0.46	0.53	0.48	0.61	0.51	27,412,679	9,999,999,703|-2
5x10^10	1.67	2.14	2.19	2.27	2.40	2.76	2.81	118,903,682	49,999,999,591|-2
1x10^11	3.41	4.24	4.31	4.34	4.69	5.51	5.91	224,376,048	99,999,999,763|-2
5x10^11	18.15	21.42	21.37	21.69	23.81	28.11	32.76	986,222,314	499,999,999,063|-2
1x10^12	37.67	44.48	44.25	44.71	49.05	58.08	69.25	1,870,585,220	999,999,999,961|-2
5x10^12	219.67	253.62	256.30	253.69	279.49	319.84	395.16	8,312,493,003	4,999,999,999,879|-2
1x10^13	482.51	543.74	542.23	541.35	602.63	678.61	825.71	15,834,664,872	9,999,999,998,491|-2

Cousin Prime Benchmark Comparisons – Intel i7 6700HQ								
N	Rust	C++	D	Nim	Crystal	Go	Cousins Count	Largest in Range
1x10^10	0.36	0.45	0.46	0.53	0.48	0.62	27,409,998	9,999,999,707|-4
5x10^10	1.69	2.11	2.18	2.26	2.41	2.81	118,908,265	49,999,999,961|-4
1x10^11	3.35	4.20	4.46	4.32	4.64	5.52	224,373,159	99,999,999,947|-4
5x10^11	18.08	21.34	21.35	21.76	23.36	28.21	986,220,867	499,999,999,901|-4
1x10^12	37.17	44.57	44.44	44.51	49.14	58.25	1,870,585,457	999,999,998,867|-4
5x10^12	220.05	250.63	251.86	252.18	278.76	320.15	8,312,532,286	4,999,999,999,877|-4
1x10^13	478.96	534.17	541.85	540.81	597.89	678.48	15,834,656,001	9,999,999,999,083|-4

Twin Prime Benchmark Comparisons – AMD Ryzen 9 5900HX									
N	Rust	C++	D	Nim	Crystal	Go	Prmsv	Twins Count	Largest in Range
1x10^10	0.12	0.12	0.12	0.19	0.13	0.15	0.16	27,412,679	9,999,999,703|-2
5x10^10	0.54	0.49	0.58	0.59	0.66	0.67	0.92	118,903,682	49,999,999,591|-2
1x10^11	1.12	0.97	1.13	1.08	1.23	1.32	1.95	224,376,048	99,999,999,763|-2
5x10^11	5.85	4.88	5.75	5.22	6.22	6.92	11.17	986,222,314	499,999,999,063|-2
1x10^12	12.14	10.03	12.01	11.12	13.06	14.61	23.71	1,870,585,220	999,999,999,961|-2
5x10^12	68.04	65.41	69.24	73.54	74.29	81.23	132.99	8,312,493,003	4,999,999,999,879|-2
1x10^13	145.01	155.45	156.57	172.68	170.77	185.25	307.78	15,834,664,872	9,999,999,998,491|-2

Cousin Prime Benchmark Comparisons – AMD Ryzen 9 5900HX								
N	Rust	C++	D	Nim	Crystal	Go	Cousins Count	Largest in Range
1x10^10	0.12	0.11	0.13	0.19	0.13	0.15	27,409,998	9,999,999,707|-4
5x10^10	0.55	0.49	0.57	0.59	0.63	0.66	118,908,265	49,999,999,961|-4
1x10^11	1.12	0.96	1.13	1.07	1.22	1.32	224,373,159	99,999,999,947|-4
5x10^11	5.87	4.89	5.78	5.25	6.18	6.92	986,220,867	499,999,999,901|-4
1x10^12	12.25	10.14	12.14	11.06	12.56	14.67	1,870,585,457	999,999,998,867|-4
5x10^12	67.69	68.51	68.74	74.68	74.86	80.29	8,312,532,286	4,999,999,999,877|-4
1x10^13	145.02	157.68	156.01	173.16	170.06	179.07	15,834,656,001	9,999,999,999,083|-4

Enhanced Configurations

The software provided is designed to work on readily available 64-bit systems, and serve as *reference* implementations, to demonstrate how Prime Generators can be used to efficiently identify and count primes. They can be enhanced to take advantage of more hardware resources when available.

Ideally we want to use as many system threads as possible. So for P5, which has 3 twin|cousin residue pairs, instead of using 3 threads over an input range it may be faster to divide the range into 2 equal parts and use 6 threads (3 for each half). Even if a system has only 4 threads, this may be faster as the range increases, but should definitely be faster (for sufficiently large ranges) if a system has 6 or more threads. In fact, if a system has at least 16 threads, using P7 (15 residue pairs) as the default generator for small ranges may be more efficient than P5, as they all can run in 1 ***parallel threads time*** **(*ptt*)**.

Thus a more sophisticated algorithm can be devised for `set_sieve_parameters` to use threads count, and also cache|memory sizes, to pick the best generator and segment size for given input ranges. For best performance this would require the profiling of targeted hardware system(s), to optimize the differences between cpus and systems capabilities and resources. However, I think the algorithm would still be fairly simple to code, to dynamically compute these parameters to achieve higher performance.

Eliminating Sieving Primes

As the value for `end_num` becomes larger more|bigger sieve primes must be generated, and filtered out or kept. Generating them takes increasing time with increasing input values. This also affects the time to perform the residue sieve, by increasing the time (and memory) to create the `nextp` array, and use it. While it’s possible to use stored lists of primes to eliminate dynamically generating them, this doesn’t get around creating `nextp` with them, with the associated memory issues for it in each thread.

One simple way around this is to use a fast primality test algorithm to check each residue pair pc value in each resgroup in the threads. If one value isn’t prime the other doesn’t have to be checked. By using sufficiently large generators for a given input range, the number of resgroups over a range can be made arbitrarily small to reduce the number of primality tests to perform.

For example for P47, modp47 = 614,889,782,588,491,410 is the largest primorial value that can fit into (unsigned) 64-bits. Its 15,681,106,801,985,625 residue pairs use 5.1% of the number space to hold the twin|cousin primes > 47. Eliminating using sieving primes greatly reduces the work of the algorithm.

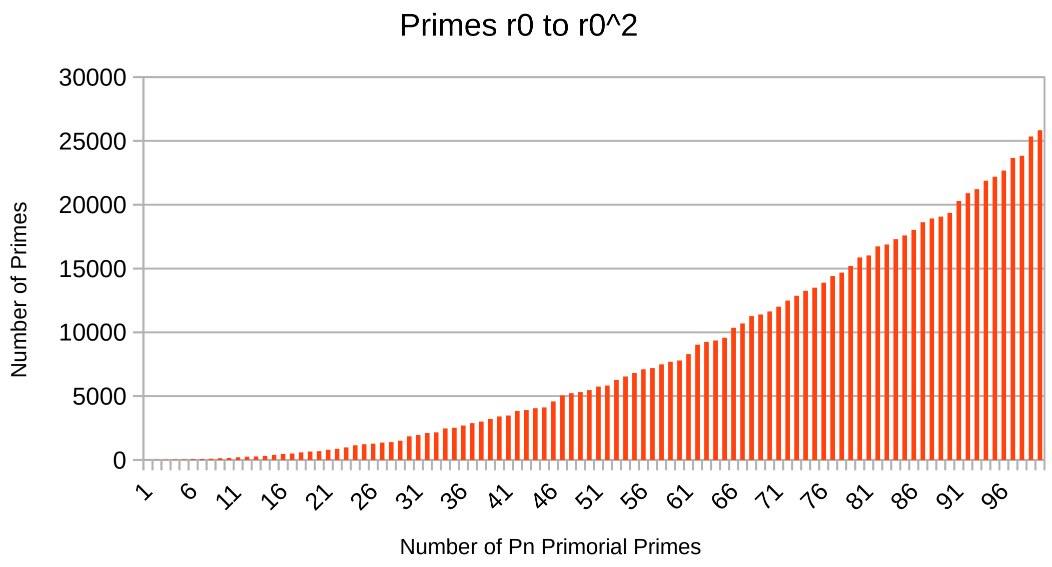
Realizable machines to perform this would use as many parallel compute engines as possible, but each would now be much simpler, eliminating `sozpg` and `nextp_init`. Now `gen_pg_parameters` just identifies the residue pair values (and no longer their inverses), needing only a (fast) `gcd` function. This could be done with massive arrays of graphic processing units (GPUs), or better, *Simple Super Computers (SSCs)*.

To search for yet undiscovered million digit primes, a distributed network can be constructed, similar to that for the Grand Internet Mersenne Prime Search (GIMPS) [7] and Twin Primes Search [8]. A benefit of creating this network, is that with all the available (free) compute power in the world, groups of residue pairs can be dedicated to machine clusters and run full time, and deterministically identify new twins|cousins (thus two primes for the price of one) forever, as there are an infinity of each [3], [4].

The Ultimate Primes Search Machine

Using just a few basic properties of Prime Generator Theory (PGT) we can construct a conceptually simpler and more efficient machine to find as many primes as physical reality and time will allow.

Because for any Pn, modpn = p_m# (primorial of first m primes), $r_0 = p_{m+1}$, and the residues from r_0 to r_0^2 are consecutive primes, we don't have to do primality tests for them, but merely gcd tests to determine which values are coprime to modpn. Thus we can arbitrarily use any prime as r_0 of a Pn whose modpn is the primorial of all the primes < r_0, to directly find the consecutive primes in $[r_0, r_0^2)$. After finding the new additional primes, we can them create a larger Pn modulus with them, and repeat the process, to continually find more primes.



This graph shows the number of consecutive primes in the regions $[r_0, r_0^2)$ for generator moduli made with the first 100 primes. Thus for the last data point for p_{100} = 541, from r_0 = 547 to r_0^2 = 299,209 there are 25,836 primes|residues, and we now know the first 25,936 primes, with 299,197 the largest prime.

Using this approach we no longer have to even identify the residue pairs, but just maintain and use the growing modulus values to perform the gcd operations with. The key here is to do the gcd operations on chunks of partial primorial values as we identify more primes and not one humongous p_m# value. Thus as we identify new primes, we make partial primorial chunks with them. To check if a value is a residue we perform repeated gcd tests with all the partial primorial chunks. If any partial gcd chunk is not 1 (coprime) then that rc value isn't a residue and we can stop testing it. Only rc values that pass all the partial chunks tests (done in parallel) are residues to the full modpn value, and thus are new primes.

The main job for this machine would be to control the creation, distribution, and storage of the gcd operations, and their results, performed by a distributed network of compute engines. For each range $[r_0, r_0^2)$ it would use the PGS for some smaller Pn, (e.g. P3's PGS in the code to reduce the residues candidates search space to 1/3 of the range values) and distribute the rcs for testing. After creating a list of new consecutive primes, it can be processed to identify new primes or k-tuples of any type.

Source Code

The SSoZ is a good algorithm to assess hardware and software multi-threading capabilities. It's very simple mathematically, needing only basic computational functions most languages have, but are easy to implement if they don't. The implementations I provide should be considered as references and not necessarily optimum for each language. They should be considered as starting points to improve upon, as they, most importantly, produce correct results that other implementations can check results against.

The code source files can be found here [6]: https://gist.github.com/jzakiya, and individually below.

twinprimes_ssoz

Crystal – https://gist.github.com/jzakiya/2b65b609f091dcbb6f792f16c63a8ac4

Rust – https://gist.github.com/jzakiya/b96b0b70cf377dfd8feb3f35eb437225

Nim – https://gist.github.com/jzakiya/6c7e1868bd749a6b1add62e3e3b2341e

C++ – https://gist.github.com/jzakiya/fa76c664c9072ddb51599983be175a3f

Go – https://gist.github.com/jzakiya/fbc77b8fdd12b0581a0ff7c2476373d9

D – https://gist.github.com/jzakiya/ae93bfa03dbc8b25ccc7f97ff8ad0f61

cousinprimes_ssoz

Crystal – https://gist.github.com/jzakiya/0d6987ee00f3708d6cfd6daee9920bd7

Rust – https://gist.github.com/jzakiya/8879c0f4dfda543eaf92a3186de554d7

Nim – https://gist.github.com/jzakiya/e2fa7211b52a4aa34a4de932010eac69

C++ – https://gist.github.com/jzakiya/3799bd8604bdcba34df5c79aae6e55ac

Go – https://gist.github.com/jzakiya/0ea756a8f6fd09f56cd9374d0dcf4197

D – https://gist.github.com/jzakiya/147747d391b5b0432c7967dd17dae124

Conclusion

Prime Generators allow for the creation of efficient, simple, and resource sparse generic algorithms that can be performed with any Pn generator. Generators can dynamically be chosen to optimize speed and memory use for given number ranges, to best use the hardware and software resources available.

The SSoZ algorithms are inherently implementable in parallel, and can be performed on any hardware or distributed system that provides multiple cores or compute engines. As shown, the more cores and threads that are available to use the higher the inherent performance will be for a given number range.

While the code to generate Twin and Cousin primes was shown here, the basic math and principles explaining the process for them can be applied similarly to find other k-tuples, and other specific prime types, such as Mersenne Primes [2].

It is hoped this detailed explanation of how the SSoZ works and performs will encourage its use in applied applications, and its inclusion in software libraries, et al, that are used in the study of primes.

References

[1] The Segmented Sieve of Zakiya (SSoZ)
https://www.academia.edu/7583194/The_Segmented_Sieve_of_Zakiya_SSoZ

[2] The Use of Prime Generators to Implement Fast Twin Prime Sieve of Zakiya (SoZ), Applications to Number Theory and Implications for the Riemann Hypotheses
https://www.academia.edu/37952623/The_Use_of_Prime_Generators_to_Implement_Fast_Twin_Primes_Sieve_of_Zakiya_SoZ_Applications_to_Number_Theory_and_Implications_for_the_Riemann_Hypotheses

[3] On The Infinity of Twin Primes and other K-tuples
https://www.academia.edu/41024027/On_The_Infinity_of_Twin_Primes_and_other_K_tuples

[4] (Simplest) Proof of Twin Primes and Polignacs' Conjectures (video):
https://www.youtube.com/watch?v=HCUiPknHtfY&t=940s

[5] Primesieve - https://github.com/kimwalisch/primesieve

[6] Twins|Cousins SSoZ software language source files: https://gist.github.com/jzakiya

[7] Grand Internet Mersenne Primes Search (GIMPS) – https://www.mersenne.org/

[8] Twins Primes Search – https://primes.utm.edu/bios/page.php?id=949

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

# Inputs are single values N, or ranges N1 and N2, of 64-bits, 0 -- 2^64 - 1.
# Output is the number of twin primes <= N, or in range N1 to N2; the last
# twin prime value for the range; and the total time of execution.

# 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. Parameter tuning
# probably needed to optimize for other hardware systems (ARM, PowerPC, etc).

# Compile as: $ crystal build twinprimes_ssozgist.cr -Dpreview_mt --release
# To reduce binary size do: $ strip twinprimes_ssoz
# Thread workers default to 4, set to system max for optimum performance.
# Single val: $ CRYSTAL_WORKERS=8 ./twinprimes_ssoz val1
# Range vals: $ CRYSTAL_WORKERS=8 ./twinprimes_ssoz val1 val2

# Mathematical and technical basis for implementation are explained here:
# https://www.academia.edu/37952623/The_Use_of_Prime_Generators_to_Implement_Fast_
# Twin_Primes_Sieve_of_Zakiya_SoZ_Applications_to_Number_Theory_and_Implications_
# for_the_Riemann_Hypotheses
# https://www.academia.edu/7583194/The_Segmented_Sieve_of_Zakiya_SSoZ_
# https://www.academia.edu/19786419/PRIMES-UTILS_HANDBOOK

# This source code, and its updates, can be found here:
# https://gist.github.com/jzakiya/2b65b609f091dcbb6f792f16c63a8ac4

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

# Copyright (c) 2017-2022; Jabari Zakiya -- jzakiya at gmail dot com
# Last update: 2022/05/22

# Customized gcd for prime generators; n > m; m odd
def gcd(m, n)
  while m|1 != 1; t = m; m = n % m; n = t end
  m
end

# Compute modular inverse a^-1 to base m, e.g. a*(a^-1) mod m = 1
def modinv(a0, m0)
  return 1 if m0 == 1
  a, m = a0, m0
  x0, inv = 0, 1
  while a > 1
    inv -= (a // m) * x0
    a, m = m, a % m
    x0, inv = inv, x0
  end
  inv += m0 if inv < 0
  inv
end

def gen_pg_parameters(prime)
  # Create prime generator parameters for given Pn
  puts "using Prime Generator parameters for P#{prime}"
  primes = [2, 3, 5, 7, 11, 13, 17, 19, 23]
  modpg, res_0 = 1, 0                    # compute Pn's modulus and res_0 value
  primes.each { |prm| res_0 = prm; break if prm > prime; modpg *= prm }

  restwins = [] of Int32                 # save upper twinpair residues here
  inverses = Array.new(modpg + 2, 0)     # save Pn's residues inverses here
  pc, inc, res = 5, 2, 0                 # use P3's PGS to generate pcs
  while pc < (modpg >> 1)                # find PG's 1st half residues
```

```
    if gcd(pc, modpg) == 1              # if pc a residue
      mc = modpg - pc                   # create its modular complement
      inverses[pc] = modinv(pc, modpg)  # save pc and mc inverses
      inverses[mc] = modinv(mc, modpg)  # if in twinpair save both hi residues
      restwins << pc << mc + 2 if res + 2 == pc
      res = pc                          # save current found residue
    end
    pc += inc; inc ^= 0b110             # create next P3 sequence pc: 5 7 11 13 17 19 ...
  end
  restwins.sort!;          restwins << (modpg + 1)         # last residue is last hi_tp
  inverses[modpg + 1] = 1; inverses[modpg - 1] = modpg - 1 # last 2 residues are self inverses
  {modpg, res_0, restwins.size, restwins, inverses}
end

def set_sieve_parameters(start_num, end_num)
  # Select at runtime best PG and segment size parameters for input values.
  # These are good estimates derived from PG data profiling. Can be improved.
  nrange = end_num - start_num
  bn, pg = 0, 3
  if end_num < 49
    bn = 1; pg = 3
  elsif nrange < 77_000_000
    bn = 16; pg = 5
  elsif nrange <  1_100_000_000
    bn = 32; pg = 7
  elsif nrange < 35_500_000_000
    bn = 64; pg = 11
  elsif nrange < 14_000_000_000_000
    pg = 13
    if    nrange > 7_000_000_000_000; bn = 384
    elsif nrange > 2_500_000_000_000; bn = 320
    elsif nrange >   250_000_000_000; bn = 196
    else  bn = 128
    end
  else
    bn = 384; pg = 17
  end
  modpg, res_0, pairscnt, restwins, resinvrs = gen_pg_parameters(pg)
  kmin = (start_num-2) // modpg + 1     # number of resgroups to start_num
  kmax = (end_num - 2) // modpg + 1     # number of resgroups to end_num
  krange = kmax - kmin + 1              # number of resgroups in range, at least 1
  n = krange < 37_500_000_000_000 ? 4 : (krange < 975_000_000_000_000 ? 6 : 8)
  b = bn * 1024 * n                     # set seg size to optimize for selected PG
  ks = krange < b ? krange : b          # segments resgroups size

  puts "segment size = #{ks} resgroups for seg bitarray"
  maxpairs = krange * pairscnt          # maximum number of twinprime pcs
  puts "twinprime candidates = #{maxpairs}; resgroups = #{krange}"
  {modpg, res_0, ks, kmin, kmax, krange, pairscnt, restwins, resinvrs}
end

def sozpg(val, res_0, start_num, end_num)
  # Compute the primes r0..sqrt(input_num) and store in 'primes' array.
  # Any algorithm (fast|small) is usable. Here the SoZ for P5 is used.
  md, rscnt = 30u64, 8                  # P5's modulus and residues count
  res  = [7,11,13,17,19,23,29,31]       # P5's residues
  bitn = [0,0,0,0,0,1,0,0,0,2,0,4,0,0,0,8,0,16,0,0,0,32,0,0,0,0,0,64,0,128]

  kmax = (val - 2) // md + 1            # number of resgroups upto input value
  prms = Array(UInt8).new(kmax, 0)      # byte array of prime candidates, init '0'
  modk, r, k = 0, -1, 0                 # initialize residue parameters

  loop do                               # for r0..sqrtN primes mark their multiples
    if (r += 1) == rscnt; r = 0; modk += md; k += 1 end # resgroup parameters
    next if prms[k] & (1 << r) != 0     # skip pc if not prime
    prm_r = res[r]                        # if prime save its residue value
    prime = modk + prm_r                  # numerate the prime value
    break if prime > Math.isqrt(val)      # exit loop when it's > sqrtN
    res.each do |ri|                      # mark prime's multiples in prms
      kn,rn = (prm_r * ri - 2).divmod md # cross-product resgroup|residue
      bit_r = bitn[rn]                    # bit mask for prod's residue
      kpm = k * (prime + ri) + kn         # resgroup for 1st prime mult
      while kpm < kmax; prms[kpm] |= bit_r; kpm += prime end
  end end
  # prms now contains the nonprime positions for the prime candidates r0..N
  # extract only primes that are in inputs range into array 'primes'
  primes = [] of UInt64                   # create empty dynamic array for primes
  prms.each_with_index do |resgroup, k|  # for each kth residue group
    res.each_with_index do |r_i, i|       # check for each ith residue in resgroup
      if resgroup & (1 << i) == 0         # if bit location a prime
        prime = md * k + r_i              # numerate its value, store if in range
        # check if prime has multiple in range, if so keep it, if not don't
        n, rem = start_num.divmod prime  # if rem 0 then start_num is multiple of prime
        primes << prime if (res_0 <= prime <= val) && (prime * (n + 1) <= end_num || rem == 0)
  end end end
  primes
end

def nextp_init(rhi, kmin, modpg, primes, resinvrs)
  # Initialize 'nextp' array for twinpair upper residue rhi in 'restwins'.
  # Compute 1st prime multiple resgroups for each prime r0..sqrt(N) and
  # store consecutively as lo_tp|hi_tp pairs for their restracks.
  nextp = Slice(UInt64).new(primes.size*2) # 1st mults array for twinpair
  r_hi, r_lo = rhi, rhi - 2               # upper|lower twinpair residue values
  primes.each_with_index do |prime, j|   # for each prime r0..sqrt(N)
    k = (prime - 2) // modpg              # find the resgroup it's in
    r = (prime - 2) %  modpg + 2          # and its residue value
    r_inv = resinvrs[r].to_u64            # and residue inverse
    rl = (r_inv * r_lo - 2) % modpg + 2  # compute r's ri for r_lo
    rh = (r_inv * r_hi - 2) % modpg + 2  # compute r's ri for r_hi
    kl = k * (prime + rl) + (r * rl - 2) // modpg # kl 1st mult resgroup
    kh = k * (prime + rh) + (r * rh - 2) // modpg # kh 1st mult resgroup
    kl < kmin ? (kl = (kmin - kl) % prime; kl = prime - kl if kl > 0) : (kl -= kmin)
    kh < kmin ? (kh = (kmin - kh) % prime; kh = prime - kh if kh > 0) : (kh -= kmin)
    nextp[j * 2] = kl.to_u64              # prime's 1st mult lo_tp resgroup val in range
    nextp[j * 2 | 1] = kh.to_u64          # prime's 1st mult hi_tp resgroup val in range
  end
  nextp
end

def twins_sieve(r_hi, kmin, kmax, ks, start_num, end_num, modpg, primes, resinvrs)
  # Perform in thread the ssoz for given twinpair residues for kmax resgroups.
  # First create|init 'nextp' array of 1st prime mults for given twinpair,
  # stored consequtively in 'nextp', and init seg array for ks resgroups.
  # For sieve, mark resgroup bits to '1' if either twinpair restrack is nonprime
  # for primes mults resgroups, and update 'nextp' restrack slices acccordingly.
  # Return the last twinprime|sum for the range for this twinpair residues.
  s = 6                                              # shift value for 64 bits
  bmask = (1 << s) - 1                               # bitmask val for 64 bits
  sum, ki, kn  = 0_u64, kmin-1, ks                   # init these parameters
  hi_tp, k_max = 0_u64, kmax                         # max twinprime|resgroup
  seg = Slice(UInt64).new(((ks - 1) >> s) + 1)       # seg array of ks resgroups
  ki += 1    if ((ki * modpg) + r_hi - 2)  < start_num # ensure lo tp in range
  k_max -= 1 if ((k_max - 1) * modpg + r_hi) > end_num # ensure hi tp in range
  nextp = nextp_init(r_hi, ki, modpg, primes,resinvrs) # init nextp array
  while ki < k_max                        # for ks size slices upto kmax
    kn = k_max - ki if ks > (k_max - ki) # adjust kn size for last seg
    primes.each_with_index do |prime, j| # for each prime r0..sqrt(N)
                                          # for lower twinpair residue track
      k = nextp.to_unsafe[j * 2]          # starting from this resgroup in seg
      while k < kn                       # mark primenth resgroup bits prime mults
        seg.to_unsafe[k >> s] |= 1_u64 << (k & bmask)
        k += prime  end                  # set resgroup for prime's next multiple
      nextp.to_unsafe[j * 2] = k - kn    # save 1st resgroup in next eligible seg
                                         # for upper twinpair residue track
      k = nextp.to_unsafe[j * 2 | 1]     # starting from this resgroup in seg
      while k < kn                       # mark primenth resgroup bits prime mults
        seg.to_unsafe[k >> s] |= 1_u64 << (k & bmask)
        k += prime  end                  # set resgroup for prime's next multiple
      nextp.to_unsafe[j * 2 | 1]= k - kn # save 1st resgroup in next eligible seg
    end                                  # set as nonprime unused bits in last seg[n]
                                         # so fast, do for every seg[i]
    seg.to_unsafe[(kn - 1) >> s] |= ~1u64 << ((kn - 1) & bmask)
    cnt = 0                              # count the twinprimes in the segment
    seg[0..(kn - 1) >> s].each { |m| cnt += (~m).popcount }
    if cnt > 0                           # if segment has twinprimes
      sum += cnt                         # add segment count to total range count
      upk = kn - 1                       # from end of seg, count back to largest tp
      while seg.to_unsafe[upk >> s] & (1_u64 << (upk & bmask)) != 0; upk -= 1 end
      hi_tp = ki + upk                   # set its full range resgroup value
    end
    ki += ks                             # set 1st resgroup val of next seg slice
    seg.fill(0) if ki < k_max            # set next seg to all primes if in range
  end                                    # when sieve done, numerate largest twin
                                         # for ranges w/o twins set largest to 1
  hi_tp = (r_hi > end_num || sum == 0) ? 1 : hi_tp * modpg + r_hi
  {hi_tp.to_u64, sum.to_u64}             # return largest twinprime|twins count
end

def twinprimes_ssoz()
  end_num   = {ARGV[0].to_u64, 3u64}.max
  start_num = ARGV.size > 1 ? {ARGV[1].to_u64, 3u64}.max : 3u64
  start_num, end_num = end_num, start_num if start_num > end_num
  start_num |= 1                         # if start_num even increase by 1
  end_num = (end_num - 1) | 1            # if end_num even decrease by 1
  start_num = end_num = 7 if end_num - start_num < 2

  puts "threads = #{System.cpu_count}"
  ts = Time.monotonic                    # start timing sieve setup execution
                                         # select Pn, set sieving params for inputs
  modpg, res_0, ks, kmin, kmax, krange,
    pairscnt, restwins, resinvrs = set_sieve_parameters(start_num, end_num)

  # create sieve primes <= sqrt(end_num), only use those whose multiples within inputs range
  primes = end_num < 49 ? [5] : sozpg(Math.isqrt(end_num), res_0, start_num, end_num)

  puts "each of #{pairscnt} threads has nextp[2 x #{primes.size}] array"

  lo_range = restwins[0] - 3             # lo_range = lo_tp - 1
  twinscnt = 0_u64                       # determine count of 1st 4 twins if in range for used Pn
  twinscnt += [3, 5, 11, 17].select { |tp| start_num <= tp <= lo_range }.size unless end_num == 3

  te = (Time.monotonic - ts).total_seconds.round(6)
  puts "setup time = #{te} secs"         # display sieve setup time
  puts "perform twinprimes ssoz sieve"
  t1 = Time.monotonic                    # start timing ssoz sieve execution

  cnts = Array(UInt64).new(pairscnt, 0)  # number of twinprimes found per thread
  lastwins = Array(UInt64).new(pairscnt, 0) # largest twinprime val for each thread
  done = Channel(Nil).new(pairscnt)

  threadscnt = Atomic.new(0)             # count of finished threads
  restwins.each_with_index do |r_hi, i|  # sieve twinpair restracks
    spawn do
      lastwins[i], cnts[i] = twins_sieve(r_hi, kmin, kmax, ks, start_num, end_num, modpg, primes,
                                        resinvrs)
      print "\r#{threadscnt.add(1)} of #{pairscnt} twinpairs done"
      done.send(nil)
  end end
  pairscnt.times { done.receive }        # wait for all threads to finish
  print "\r#{pairscnt} of #{pairscnt} twinpairs done"

  last_twin = lastwins.max               # find largest hi_tp twinprime in range
  twinscnt += cnts.sum                   # compute number of twinprimes in range
  last_twin = 5 if end_num == 5 && twinscnt == 1
  kn = krange % ks                       # set number of resgroups in last slice
  kn = ks if kn == 0                     # if multiple of seg size set to seg size
  t2 = (Time.monotonic - t1).total_seconds       # sieve execution time

  puts "\nsieve time = #{t2.round(6)} secs"      # ssoz sieve time
  puts "total time = #{(t2 + te).round(6)} secs" # setup + sieve time
  puts "last segment = #{kn} resgroups; segment slices = #{(krange - 1)//ks + 1}"
  puts "total twins = #{twinscnt}; last twin = #{last_twin - 1}+/-1"
end

twinprimes_ssoz
```