Revised SoZ algorithm.
prime candidates are: r1 = 30k+7, r2 = 30k+11, ... r8 = 30k+31 for k = 1..t
the k values are the residue groups

Table 1.

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

Now generate the non-primes by $P'_5 = (p_j*30)k + p_j*(7, 11, 13, 17, 19, 23, 29, 31)$, for k = 0..t, and

$$r1_k = (p_j*30)k+p_j*7, \quad r2_k = (p_j*30)k+p_j*11, \quad \ldots \quad r8_k = (p_j*30)k+p_j*31$$

Shaded values in Table 1 are the non-primes shown in Table 2.
Table 2.

| $p_j$ | $r1_0$ | $r2_0$ | $r3_0$ | $r4_0$ | $r5_0$ | $r6_0$ | $r7_0$ | $r8_0$ | $r1_1$ | $r2_1$ | $r3_1$ | $r4_1$ | $r5_1$ | $r6_1$ | $r7_1$ | $r8_1$ | $r1_2$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 7 | 49 | 77 | 91 | 119 | 133 | 161 | 203 | 217 | 259 | 287 | 301 | 329 | 343 | 371 | 413 | 427 | 469 |
| 11 | 77 | 121 | 143 | 187 | 209 | 253 | 319 | 341 | 407 | 451 | 473 | 517 | 539 | 583 | 649 | 671 | 737 |
| 13 | 91 | 143 | 169 | 221 | 247 | 299 | 377 | 403 | 481 | 533 | 559 | 611 | 637 | 689 | 767 | 793 | 871 |
| 17 | 119 | 187 | 221 | 289 | 323 | 391 | 493 | 527 | 629 | 697 | 731 | 799 | 833 | 901 | 1003 | 1037 | 1139 |
| 19 | 133 | 209 | 247 | 323 | 361 | 437 | 551 | 589 | 703 | 779 | 817 | 893 | 931 | 1007 | 1121 | 1159 | 1273 |
| 23 | 161 | 253 | 299 | 391 | 437 | 529 | 667 | 713 | 851 | 943 | 989 | 1081 | 1127 | 1219 | 1357 | 1403 | 1541 |
| 29 | 203 | 319 | 377 | 493 | 551 | 667 | 841 | 899 | 1073 | 1189 | 1247 | 1363 | 1421 | 1537 | 1711 | 1769 | 1943 |
| 31 | 217 | 341 | 403 | 527 | 589 | 713 | 899 | 961 | 1147 | 1271 | 1333 | 1457 | 1519 | 1643 | 1829 | 1891 | 2077 |
| 37 | 259 | 407 | 481 | 629 | 703 | 851 | 1073 | 1147 | 1369 | 1517 | 1591 | 1739 | 1813 | 1961 | 2183 | 2257 | 2479 |
| 41 | 287 | 451 | 533 | 697 | 779 | 943 | 1189 | 1271 | 1517 | 1681 | 1763 | 1927 | 2009 | 2173 | 2419 | 2501 | 2747 |

The rows are the primes $p_n < p_j \leq \text{sqrt}(N)$, the columns are the residues [mk+($r_i$, m+1)], and the table entries are their products. The first product (row1:col1) is always $p_1^2 = r_1^2$.

Generic Python code using P5 generator.

```
1  def SoZP5(val):
2    # all prime candidates > 5 are of form 30*k+(1,7,11,13,17,19,23,29)
3    num = val-1 | 1              # if val even number then subtract 1
4    mod=30; rescnt=8             # modulus value; number of residues
5    maxprms= 2+(rescnt*num)//mod # max number of prime candidates
6    prms = [True]*maxprms        # set all prime candidates to True
7    primes = [2,3,5]

8    # array of residues to compute primes and candidates values
9    residues = [1,7,11,13,17,19,23,29,31]
10   # hash of residues offsets to compute nonprimes positions in prms
11   pos = {1:-1, 7:0, 11:1, 13:2, 17:3, 19:4, 23:5, 29:6}

12   # sieve to eliminate nonprimes from prms
13   limit = int(ceil(sqrt(num)))
14   modn=r=0
15   for prm in prms:
16      r += 1
17      if r > rescnt: r = 1; modn += mod
18      if not prm: continue
19      prime = modn + residues[r]
20      if limit < prime: break
21      prmstep = prime*rescnt  # prime multiples position inc in prms
22      for ri in residues[1:]:
23         product = prime*(modn+ri)
24         if product > num: break
25         # compute product position index in prms
26         qq,rr  = divmod(product,mod)
27         nonprmpos = (qq*rescnt) + pos[rr]
28         for nprm in xrange(nonprmpos,maxprms,prmstep): prms[nprm] = False
29   # the prms array now has all the positions for primes 7..N
30   if num < 7: return primes[:1+num//2]
31   modn=r=0
32   for prime in prms:
33      r += 1
34      if r > rescnt: r = 1; modn += mod
35      if prime: primes.append(modn+residues[r])
36   while primes[-1] > num: primes.pop()
37   return primes
```

Table 1 represents the generation of prime candidates using the P5 generator as an example.
Table 2 represents the sieve process to identify the non-primes in the candidates list.

The following is the breakdown of the code line-by-line.

L3: I subtract 1 from the target number N (val) to make it an odd number if it's even.
L4: Set the variable 'mod' to the modulus value, and 'rescnt' to the number of residues 1..mod
L5: Compute maximum prime candidates up to N for this generator. The 2+ is for edge roundoff cases for all generators.
L6: Initialize 'pms' array to all 'True' representing all the candidates in Table 1 as primes.
L7: Initialize the output 'primes' array with primes up to the order of the generator.

L9: Initialize 'residues' array, which has length 1+rescnt elements.
L11: Initialize 'pos' hash/array; used in determining non-primes products position in prms array.
L13: Check primes up to the sqrt(num).
L14: Initialize modn and r, which form the real values the prms indices represent from residues values.
L15: Start the main sieve loop which iterates through prms candidates values (True:False).
L16|17: For each position in prms create modn,r that form that real value.
L18: If the current prms position is 'True' (a prime) continue, else increment loop to next position.
L19: Form the real value of the current prms position, which is a prime.
L20: If limit < prime, don't need to check anymore primes, terminate main loop and continue.
L21: Compute 'prmstep', the row increment in Table 1 for each prime, to find next prime multiple.
L22: Start loop to iterate from residues 1...mod+1, to multiply 'prime' by.
L23: Find the products of (prime*res-value), as shown in Table 2, starting in appropriate k-group.
L24: If 'product' > N, process is finished, terminate this loop for this prime value.
L26: Divide 'product' by mod to determine quotient and remainder to find position in prms.
L27: (qq*rescnt) gives k-group to start in and pos[rr] gives position within the group.
L28: Set to 'False' positions in prms that are multiples of current prime starting at 'product' position.
L30: Return appropriate primes array if N < 7.
L31: Initialize modn and r, which form the real values the prms indices represent.
L32: Iterate over full prms array to determine positions of every prime.
L33|34: For each position in prms create modn,r that form that real value.
L35: Form prime value and push to end of 'primes' array if prms position is 'true'.
L36:37: Remove any prime values > N from 'primes' array (edge cases) and return 'primes'.

SoZ Algorithm Features

For the sake of this discussion I will be concentrating on just the prime order generators.

In my original algorithm implementation I used a prime candidates array that represented the odd numbers up to N, so the array size was ~ ½ of N elements. The new algorithm uses a candidates array whose size becomes smaller as the generator order increases, which is much more memory efficient.

Example: For a value of 1e12 the max number of prime candidates for the following generators are:
N = 1000000000000 → num = 999999999999

P3: maxprms = 2+(num*2)/6 = 333333333335
P5: maxprms = 2+(num*8)/30 = 266666666668
P7: maxprms = 2+(num*48)/210 = 228571428673
P11: maxprms = 2+(num*480)/2310 = 207792207794
P13: maxprms = 2+(num*5760)/30030 = 191808191810
P17: maxprms = 2+(num*92160)/501501 = 180525356997

Thus, for each higher order prime generator the number of prime candidates become smaller, with P3 using about 33% of the integers (all odd) up to N as candidates, while going down to 18% for P17.

This array, 'prms', can be represented as a simple linear array of 'maxprms' boolean (True/False) values. Thus, a real implementation need only create an array maxprms bytes (bits) long for 'prms'.

For P5, the actual prime candidate values prms represents are shown in Table 1. The 8 residues start at 7 (not 1), with the last being mod+1 (30+1). The rows in Table 1 show the candidates generated for each residue by adding the modulus to each successive column, up to maxprms length. The columns represents each kth group of candidates, with each value equal to (mod*k + r[i]). In 'prms' each group is contiguously appended to the previous group as shown in the top row of Table 2.

The algorithm starts by initializing necessary variables and arrays (L3-L11) for a given generator. The 'pos' hash/array values are based on where you choose to start the primes candidates Table 1. Table 1 represents the most generic representation of candidates, but you can start the process in any group you wish, which changes the number of seed primes you have (the initial primes in 'primes') and also changes the values in the 'pos' hash/array, and L26.

For Table 1, the first non-prime 49 =7*7 is prms[12] (using zero indexing), which is in group 1 (k=1) and r5 =19. Thus, for k =1, rescnt*k = 8*1=8, which is the starting position for this group. To get to 49, do (49 modulo 30) = 19, which is row 5 (index 4) in group 1, so in 'pos', 19=>4. Thus, the position of a number N in Table 1 can be formed by (rescnt*(N/mod) + pos[N % mod]), where '/' returns the integer quotient and '%' returns the integer remainder, which is what the code in L26-L27 does.

Table 2 represents the sieve process to eliminate the non-primes from Table 1; the code from L13-L28. For each prime <= sqrt(N), starting at prms[0] = 7, multiply the prime by each residue in the k-group the prime starts in. If product < N, determine product position in prms, set it to False, then set multiples of this prime positions to False while they are <= maxprms position.

Once all the non-primes in prms are found L32-L37 forms the primes and puts them into the 'primes' array, which is checked to make sure the last prime found isn't > N, and then returns the 'primes' array.

In Table 1, same colored values are first eliminated by the same prime, starting with 7, 11, 13, etc. To do sozP5(541), maxprms =2+ (8*541)/30 = 146 which is Table 1.

Starting at L18 prms[0] is True, so 'prime' = modn+residues[1] = 0+7=7, which is < (limit=23) (sqr(541)=23), so prmstep =7*rescnt = 56. With prime=7, starting in L22, compute the non-primes in Table 2 by computing 'product' = 7*(modn+residues[1]) = 7*7 = 49, which is not > 541, so compute its position (prm[12]) – nonprmpos, and starting from this position (which is < maxprms) mark this position as non-prime (False) then all the multiples of 7 < maxprms by adding prmstep successively to this address. In Table 1, this corresponds to row r5, and starting at 49, mark every 7th column in that row (259, 469) as False while < maxprms. Then r=2, 'product' = 7*(modn+residues[2]) = 7*11=77, which is in row r4, prms[19], and then mark as non-prime every 7th column in the that row (287, 497). Repeat this for every residues[i] for which 'product' <= 541, then find the next prime in prms (11) and repeat until prime > (limit=23). When done, find the prime positions in prms, form the prime values, and put them in ascending order in the 'primes' array, and return the array.

Multiple Core/Thread Implementation

The sieving process L15-L28 can be done simultaneously in multiple cores/threads using a common prms array/memory. For each prime, the inner loop L21-L28 can be assigned it's own core/thread, where the product would be 'prime' *(modn+ri), where ri is the specific residue (row in Table 1) for that thread. For P5 you need 8 threads to fully unroll L21-L28. Otherwise, divide rescnt by the number of threads you have and use modular compliment pairs (mcp) as the residues for each thread.

For P5, if you use 4 threads/cores then assign the residues to the threads as:

Thread-1: [7,31]; Thread-2: [11,29]; Thread-3: [13,23]; Thread-4: [17,19]

For P5, if you use 2 threads/cores then assign the residues to the threads as:

Thread-1: [7,13,23,31]; Thread-2: [11,17,19,29]

You do this to balance the total amount of work each thread does by putting the smallest residue (which marks the most non-primes) with the largest (which mark the fewest), and successively match the pairs to even out the workload per thread.

If you use 8 threads with P7, then each thread would accommodate 48/8=6 residues values, like above. Since all the prime generators have rescnts that are divisible by 2,4,6,8, or 12, they should map nicely to Intel Core 970/980 systems, which have 6 cores and 12 possible threads. Memory management should be simpler (certainly be no harder) than generalized wheel, and should become more efficient for higher order generators.

On my single core P4 laptop, under Ruby and Python, P7 is faster than P11 for 'small' N (by a small amount) I assume because it all fits in cache, and P11 doesn't. However when coded in Forth and run with gforth-0.7.0, P11 is consistently faster than P7, as it algorithmically should be. What this shows is that the physical implementation of the algorithm affects its performance, even though if all things were equal, the higher order generator order should be faster, with the higher order generators always needing progressively less memory.

Jabari Zakiya