Some Faster Algorithms for Finding Large Prime Gaps

Arne Maus (em), Thomas Gulli and Eric B. Jul
(arnem,thomgull,ericbj@ifi.uio.no)
Programming Technology Group
Dept. of Informatics, University of Oslo, Norway

Abstract

This paper investigates the problem of finding large prime gaps (the difference between two consecutive prime numbers, $p_{i+1} - p_i$) and on the development of a small, efficient program for generating such large prime gaps for a single computer, a laptop or a workstation. In Wikipedia [1], one can find a table of all known record prime gaps less than 2^{64}, the record is a 20 decimal digit number. We wanted to go beyond 64 bit numbers and demonstrate algorithms that do not needed a huge number of computers in a grid to produce useful results.

After some preliminary tests, we found that the Sieve of Eratosthenes, SE, from the year 250 BC was the fastest for finding prime numbers and it could also be made space efficient. Each odd number is represented by one bit and when storing 8 odd numbers in a single byte (representing 16 consecutive numbers ignoring the even numbers), we found that we should not make one long SE table, but instead divide the SE table into segments (called *SE segments*), each of length 10^8 or 10^9 and dynamically generate the necessary SE segments as to find prime numbers. First, we made a basic segment of all prime numbers < 10^8 (in less than a second). We also relied heavily on the old observation [2] that when using SE to find all prime numbers $\leq N$, we cross out all numbers using the prime numbers $p \leq \sqrt[2]{N}$, and that the first number crossed off when crossing out for prime number p is p^2. When we want to find prime gaps, we first create one or more consecutive SE in that range, say starting on 2^{74} and ending with the value M – initially these big segments are crossed out by our first basic set of primes < 10^8 , To find all prime number in these big segments, we next need the rest of prime numbers $p \leq \sqrt[2]{M}$. These can be all be constructed by using our first set of prime numbers to generate segments of consecutive SE from 10^8. The primes in these segments are used to cross out in the big SE segment and can then be discarded (each prime used only once). Our most significant algorithm was to find a simple formula for using primes from a range 3 – 2^{36} to cross out the non-primes in any SE segment without crossing out in all the numbers between 2^{36} and 2^{72}. This leads to an exponential saving in both space and execution time. In addition to this, we created a small package Int3 to represent numbers > 2^{64} by storing 8 decimal values in each of 3 integer variables together with the necessary mathematical operations. The Int3 package can handle numbers up to 24 decimal digits and is significantly faster than the BigInteger package in the Java library. We also created a faster algorithm for finding all record prime gaps.

The results presented in this paper are some tables of prime gaps for primes significantly larger than 2^{64} and data supporting an observation that big prime gaps in these segments are much more frequent than the ones we find in the Wikipedia table where the search starts at prime number 3. Our combined set of algorithms is also sufficiently fast to test every entry in the Wikipedia table in less than 5 minutes. We conclude by reflecting on the use of brute force (more computers) versus smarter algorithms.

Keywords: Prime numbers, prime gaps, algorithms.

1. Introduction

Prime numbers have been studied since antiquity. We take as our starting point a well-known algorithm from Eratosthenes (276 – 194 BC), a true genius of that time. He invented Geography as a science, was head of the library of Alexandria, measured the diameter of the Earth and the distance from Earth to the Sun with great precision and, of special interest for us, invented the Sieve of Eratosthenes, a tabular method for finding all prime numbers < N.

There are numerous research themes connected to primes, such as finding an even greater prime number (there are an infinite number of them as shown in Euclid's elegant proof), finding the largest Mersenne prime, trying to solve the Goldbach conjecture (any even number is a sum of two primes).

This paper was presented at the NIK 2020 conference. For more information see http://www.nik.no/

For an introduction, see the Wikipedia articles on 'Prime number' 'Prime Gap' and 'Sieve of Eratosthenes' [1] and their references.

To be clear, neither the Sieve of Eratosthenes (SE), nor any other method or formula, can find the prime numbers directly – SE first makes a table of all odd numbers (one bit for each number) < *M*, some maximum number, assuming they are all prime numbers. Then SE efficiently cross out in this table all numbers that are non-prime. We do not need to cross off any even number as we ignore them from the beginning because, except for 2, we know they are all non-primes. The odd numbers not crossed out in SE, are then the prime numbers greater than 2 and less than *M*.

2. The Problem

This paper is about the problem of finding large prime number gaps. A prime number gap, or prime gap for short, is an integer interval $[p_i, p_{i+1}]$ where p_i and p_{i+1} are two consecutive prime numbers. The size of the gap is $p_{i+1} - p_i$. An interesting problem is finding a prime gap larger than all prime gaps that precede the gap. We report on the development of a small, efficient program for generating such large prime gaps using a single computer, a laptop or a workstation. In Wikipedia, one can find a table all known record prime gaps less than 2^{64} - a 20 decimal digit number. We wanted to go beyond 64 bit and demonstrate algorithms that do not need a huge number of computers in a grid to be useful.

We first observe that Horsley in his paper from 1772 [2] in the section: "The Operation of the Sieve" summarises the way to remove all non-primes for his table of the odd numbers < *M*:

1. Set $p = 3$.
2. Start at number p^2 and cross out all multiple of $2p$ after p^2 until we reach a number > *M*, the end of the table (3 will cross out 9, 15, 21, 27, 33, …)
3. Find new *p* as the next uncrossed number after last used prime (now 5 after 3) and repeat point 2 until the next prime *p* has a $p^2 > M$ (5 will cross out 25, 35, 45, …).
4. When the next picked *p* has a $p^2 > M$, the table of all primes ≤ *M* is finished.

While point 4 above is used and acknowledged today, the observation 2 seemed to be partly forgotten. It is only a few years since the demo graphical app at Wikipedia [1], to find all primes less 100, let p=5 started by crossing out 15, and p=7 start by crossing out 21 and 35. The reason that we, for all primes *p*, can start at p^2 is that all smaller non-prime numbers have already been crossed out by smaller primes, e.g., in the example above 15 was already crossed out by 3, and 21 and 35 were already crossed out by 3 and 5 respectively.

Others, such as Jacobsen, Andersen and Oliveira e Silva [3, 4] has also produced prime numbers beyond 64 bits. These prime gaps come either as a continuation from 0 as in the Wikipedia table with best record gap so far is 1550 starting at 18 361 375 334 787 046 697 – a 20-digit number, or they come from free standing segments far beyond 64 bit where the record so far seems to be a gap of length 5 103 138. They also use a metric, the merit of a gap, starting at prime *p*, defined as: $size\ of gap / \ln p$, to measure how 'fine' or unexpected such a large prime gap is. A merit above 10 is considered good and the best merit so far is 38. This is obviously inspired by 'the prime number theorem' that states that there are approximately: $n / \ln n$ prime numbers < *n*; such that the average prime gap length < *n* is: $\ln n$. We here comment on two factors. Their methods or algorithms are not clearly disclosed in their published results, and they distinguish between prime numbers and probable prime numbers (PPN), 'prime gaps' (a gap that does not contain a prime number)

and ‘proven prime gaps’ (a prime gap that has a prime at both ends). Further below, we sketch out how one might find larger gaps by finding PPNs.

3. Problems with the Traditional Data Structure

The traditional data model for finding prime numbers < M with the Sieve of Eratosthenes is one long bit-table representing the odd numbers 1, 3, 5, …, $M = 2^{64}$ ($10^{19} \approx 2^{64}$))

Figure 1. *A traditional’ Sieve of Eratosthenes’ for finding record prime gaps up to 2^{64}. All odd numbers from 1 to the maximum value M are represented as one continuous table.*

We first tried this data model and discovered that when trying to reproduce the results in the Wikipedia table, the execution time of course went up exponentially. All prime gaps, up to and including gap 31, a 10-digit number, took a little more than 1 minute, while all gaps up to gap 58, a 14-digit number, took more than 2 days. Because the last gap in the table, gap 80, is a 20 decimal digit number, our space (the table) and time approximately needs $10^{20}/10^{14} = 10^6$ times more time and space than our gap 58 – or a million times more time than 2 days – that is some 5 500 years. And in addition, one such long table needs 0,625 Exabytes for representing all numbers < 10^{19} even when we pack 16 numbers into each byte. Our two computers had 8 GB and 16 GB, so when we wanted to go beyond 2^{64}, we certainly had to find better algorithms.

3. Our Data Structure

We discovered that there are four types of prime numbers in the traditional sieve when we want to find prime gaps between some big numbers, say between: 10^{22} and $10^{22} + 10^9$ ($M = 10^{22} + 10^9$):

a. The basic set of small primes < 10^8, used many times for generating all the primes on *b* and most of the non-primes on *d.*
b. The set of primes used once, medium-sized, primes < 10^{11} used to further cross out non-primes on *d*.
c. The primes between 10^{11} and 10^{22} – not used for anything except for finding where a prime *p* will start to cross out non-primes on *d*.
d. The set of big primes, in this example between 10^{22} and $10^{22} + 10^9$ where we might find new large prime gaps.

Figure 2. F*our different types of prime numbers on the SE when we want to find big prime gaps in the top part of the table (not to scale).*

When we inspected the size requirements for these sets, we found out that:

a. The set *a* of small primes. If made 10^8 long, it would be only 6.25 Mb but will contain all the prime numbers needed for making all possible lengths of section *b* (which is at most 10^{12}) – which again contains all primes needed to create the primes on set *d*. The time needed for creating all 5.7 million primes on $a < 10^8$ was 0.7 sec.

b. Set *b* is potentially large, on the order of 10^{12} – or 62.5 Gb, too much for main memory, but possible (but much slower) if stored on disk.

c. Impossibly large and time-consuming.

d. A small set of 10^8 to 10^{10} numbers. We can afford the space and probably also time-wise.

We then had two problems, the size of set *b* and how to get rid of set *c*. They both happened to have a common solution, which we call the Sliding Sieve of Eratosthenes (SSE).

We found out that we can divide the set *b* into segments of fixed length (say 10^8). Each segment is called an SSE. They will all be created from the primes from set *a*. When one such SSE is created, all its primes will be used to cross out on the set *d*, and then deleted because each of its primes are used only once.

4. Using a Prime on a Segment with Start S and End T

When we want to cross out non-primes with a prime *p* on a segment that starts with number S and ends with T, we are posed with the following problem: Where do we start crossing out for *p* on this SSE? Answer: we want to calculate an integer *k* where we place our first cross using *p*, a relative address within this segment, ($k = 0$ means that we start at S).

The primes that we use for crossing out, we use in increasing order, and simulate that we would have done on the old, one long, continuous SE where we start at p^2 and after that at all multiples $2*p$ until we reach a number larger than T.

The total formula is then (depending on whether p^2 comes before S, between S and E, or after E):

Calculate p^2:
if $p^2 > E$, we are finished with this SSE (because no larger primes can be used)
else { if $p^2 \geq S$, $k = p^2 - S$
else $k = 2*p - (S - p^2) \%(2*p)$
if now $k \leq T-S$, cross out for k }

The explanation for the last formula is given in fig. 3 (% is the modulo operator, the remainder after having crossed out the stretch $S - p^2$ with $2*p$).

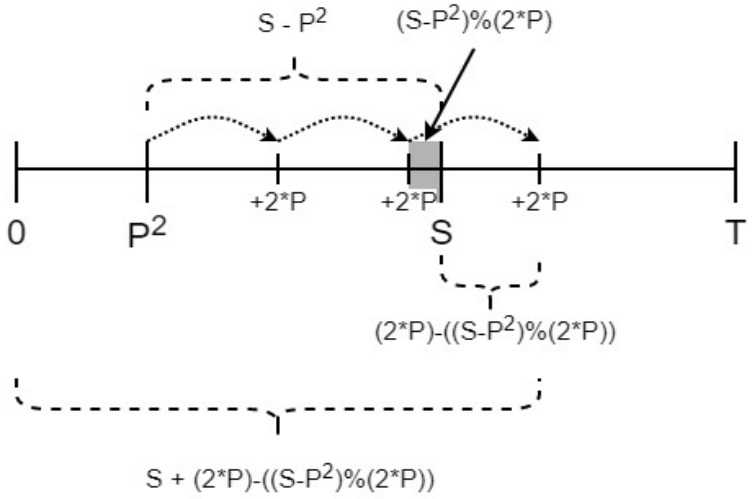

Figure 3. *The case when crossing out with p on a segment SSE stretching from S to T, and when one of the crosses hits that segment (is between S and T).*

Assume that we use the same segment size on all SSE (both *a, b* and *d*), then after having crossed out for the smaller primes *p* in section *a*, all primes *p* will be larger than the segment size, and crossing out with a step size of p*2, one can often miss placing a single cross on the SSE in question. This is a valid question for both the larger primes from *a* to *b* and *d* segments, also from all primes from *b* segments to *d* segments.

5. Our Data structure and Method

We decided on the following data-structure (fig, 4). All SSE segments are of the same size (here 10^8). The segments *a* and the 1..100 continuous *d* segments are fixed in memory, with *M* as the maximum number represented on *d*. The *b* segments are created, one at a time as a continuation of the *a*-segment, used and then deleted until we have created all prime numbers $p \leq \sqrt[2]{M}$.

Figure 4. *The data structure used to find large prime gaps on d.*

Our method can then be summarized as:

1. Create the *a*-segment, the first *b*-segment and all the *d*-segments and determine *M*, the maximum value.
2. Create all the prime numbers < 10^8 on the *a*-segment and use them to cross out on the first *b*-segment and all the *d*-segments.
3. Cross out with the primes then found on the current *b*-segment on the *d*-segments. If the last prime on this *b*-segment < $\sqrt[2]{M}$, delete this *b*-segment and create a new as its continuation.
4. Repeat 3 until we have crossed out all the non-primes on *d*.

5. Scan the primes on the *d*-segments for 'big' prime gaps.

By leaving a continuous sieve of Eratosthenes when dropping the c-segment, we miss the numbering (but of course not the value) of the prime numbers where the prime gap starts. But more importantly, we miss the ability to claim that any 'larger' prime gap that we find is a record because we do not know if there is an even larger gap on the *c*-segment that we have skipped. However, those like Andersen and Jacobsen that are looking for the largest gap, all have these two problems. The running time of our algorithm is roughly proportional to the number of d segments and $\sqrt[2]{M}$.

6. The Int3 and Eratosthenes Classes.

To represent numbers > 2^{64}, a class Int3 was implemented. It contained a representation for numbers < 10^{24} and the mathematical operations needed in our case: The values of big numbers are kept in a small int[] array, with 8 decimal digits in each of its tree array elements. With this representation, it is then possible explore a set of numbers 10^5 times larger than can be explored by using Java's 64 bit long numbers ($10^{19} \approx 2^{64}$). It was tested against the Java library package BigInteger and was for most operations significantly faster. The implemented operations are: add, sub, mult, less, square, sqrt and two methods for getting the values for an Int3 object: getLongValue (if the value is small enough) and getStringValue (nice output with one space between groups of each 3 digits).

The following demonstrates how the formula explained in figure 3: $k = 2*p - (S - p2)\%(2*p)$, looks in Int3-land, where bigStart is the start value for a *d*-segment. bigStart is an Int3 variable.

```
long p2 = p+p;
Int3 pp = Int3.square(p);

k = (int)(p2-(bigStart.sub(pp)).modulo(p2));
```

The type-cast from an Int3-expression to a 32-bit value can only be applied after we have tested with the 'less()' method that the prime *p* will make a cross inside the limits of this *d*-segment.

The class Eratosthenes is a class representing the *a*, *b*, and the *d*-segments (with a separate constructor for each case). One object of this class contains *start* and *end* variables and the byte array for the odd numbers between these two variables – the SE table. Most importantly, this class contains operation on the byte-array such as: boolean isPrime(int *num*), void setNotPrime(int *i*), int firstIntPrime(), int lastIntPrime(), long firstLongPrime(), int numberOfPrimes() , int nextIntPrime(int *i*). In addition, this class contains two methods for applying the primes from the *a*-segment on the *b*-segment and on the *d*-segment, and one method for applying the medium-sized primes on the *b*-segments on the d-segments.

7. Notes on the implementation

This program is an exercise in using different levels of representing integer numbers: byte-array with one bit for the odd numbers on the SE within a segment, 32-bit int values for start and end on the *a*-segment, 64 bit long for start and end values on the *b*-segments, and finally Int3-values for start and end on the *d*-segments. Byte-arrays are used on all segment types: *a, b* or *d* for representing the SE.

The program is written in Java and consists of three classes and is about 700 lines of code in total. The advantages of using Java are that it provides many helpful error messages during compile- and run-time, that parallel threads are included in the language, and that it has the ability to do typecast (e.g., from long to int when needed and possible). Java is also fast and optimized – probably between almost as fast, or not less than half the speed of optimized C. One not so nice limitation is on the length of an array – a byte array cannot have more than 10^9 elements.

8. Faster Finding of Larger Prime Gaps without Reading all the Prime Numbers.

When looking for a larger prime gap than previously found, one can speed up the process with the following algorithm (fig 5):

a) Say our current record gap is *Len*, and we stand at prime number *N*.
Then test if *PN=Len+N* is a prime.

b) If yes, we have at best found a prime gap just as large as our present record,
Set *N* to *PN* and repeat a.

c) If *PN* is not prime, move backwards towards *N* and find first prime *P'*
in that direction. If *N == P'*, we have found a new record. Anyway, move
rightwards from *PN* to the next prime *NP* in that forward direction.
Test if the gap *NP – P' > Len*, i.e. a new record gap. If so, update *Len*.

d) Set *N = NP*. If *N* is the last prime on this segment, then exit.
Otherwise go to a) and continue a search for an even bigger gap.

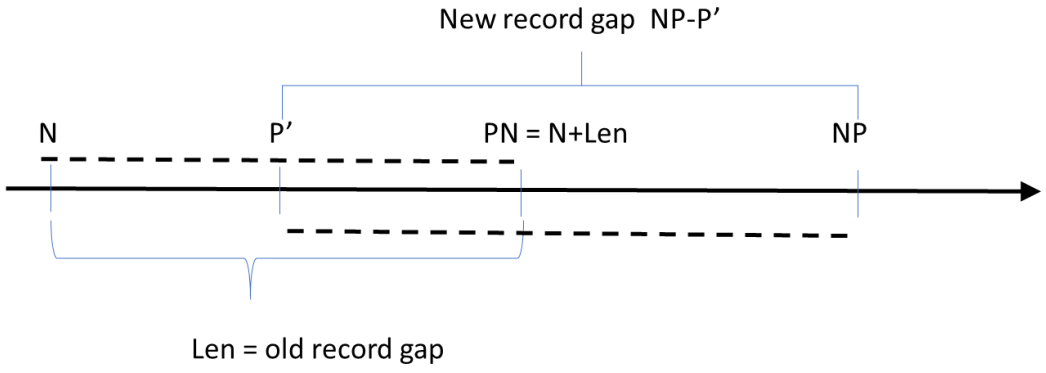

Figure 5. *Speeding up finding of record gaps, explaining two of the four different cases that can occur with this algorithm for finding record prime gaps.*

The average prime gap is *ln N*, but one soon finds a record gap of 100 or more. We then see that this algorithm will avoid reading most of the prime number while finding all the record gaps. Tests comparing this method when using the division method for deciding every prime (finding at least one factor by division with the primes to find non-primes) showed

that this method had a speedup of approximately 4 when the record gap exceeded 200, and also showed that SSE was much faster than the division method.

9. Results

As a motif for this project, we wanted to find big prime gaps above 2^{64}. The following tables 1 and 2 are typical results, we get. We note that execution time of course are longer for larger numbers but that bigger gaps appear earlier, and that the numerical differences when larger gaps appear are shorter in the 2^{70} case than in the 2^{64} case. Comparing table 3 (starting at 3) and table 1, a continuation of the Wikipedia table, we find that the longest gap 716 in table 3 occurs after inspecting 10^{14} numbers, but in table 1 we reach > 712 after inspecting 'only' 10^{10} numbers.

Table 1. for numbers > 2^{64}

From Prime	Prime gap	Time
18 446 744 073 709 551 629	24	*2,3 min*
18 446 744 073 709 551 667	30	*2,3 min*
18 446 744 073 709 551 709	48	*2,3 min*
18 446 744 073 709 551 757	166	*2,3 min*
18 446 744 073 709 554 151	178	*2,3 min*
18 446 744 073 709 556 543	180	*2,3 min*
18 446 744 073 709 558 913	186	*2,3 min*
18 446 744 073 709 559 521	222	*2,3 min*
18 446 744 073 709 564 343	224	*2,3 min*
18 446 744 073 709 573 099	252	*2,3 min*
18 446 744 073 709 604 447	266	*2,3 min*
18 446 744 073 709 612 957	294	*2,3 min*
18 446 744 073 709 631 131	390	*2,3 min*
18 446 744 073 710 729 099	432	*2,3 min*
18 446 744 073 712 569 971	486	*2,3 min*
18 446 744 073 716 572 673	488	*2,3 min*
18 446 744 073 721 938 209	492	*2,3 min*
18 446 744 073 728 573 983	546	*2,3 min*
18 446 744 073 749 613 491	576	*2,3 min*
18 446 744 073 825 492 823	580	*2,3 min*
18 446 744 073 856 141 759	592	*2,3 min*
18 446 744 073 880 768 627	622	*2,3 min*
18 446 744 073 907 622 243	638	*2,3 min*
18 446 744 073 969 423 769	712	*2,3 min*
18 446 744 076 237 171 647	740	*2,3 min*
18 446 744 077 916 002 571	792	*2,3 min*
18 446 744 081 169 869 483	834	*2,3 min*
18 446 744 104 348 707 907	882	*9,4 min*
18 446 744 134 147 887 497	900	*16,5 min*
18 446 744 233 555 656 917	920	*37,6 min*
18 446 744 955 008 014 019	950	*3 h and 29,2 min*
18 446 745 640 326 966 247	952	*6 h and 9,2 min*
18 446 747 074 982 437 271	998	*11 h and 48,5 min*
18 446 747 749 629 047 369	1062	*14 h and 26,1 min*

Table 2. for numbers > 2^{70}

From Prime	Prime gap	Time
1 180 591 620 717 411 303 423	26	8,1 min
1 180 591 620 717 411 303 449	42	8,1 min
1 180 591 620 717 411 303 539	74	8,1 min
1 180 591 620 717 411 303 949	120	8,1 min
1 180 591 620 717 411 304 553	170	8,1 min
1 180 591 620 717 411 305 251	198	8,1 min
1 180 591 620 717 411 306 883	204	8,1 min
1 180 591 620 717 411 309 137	252	8,1 min
1 180 591 620 717 411 313 217	342	8,1 min
1 180 591 620 717 411 395 749	450	8,1 min
1 180 591 620 717 411 865 357	454	8,1 min
1 180 591 620 717 412 221 491	518	8,1 min
1 180 591 620 717 417 889 613	534	8,1 min
1 180 591 620 717 418 271 863	540	8,1 min
1 180 591 620 717 423 200 219	624	8,1 min
1 180 591 620 717 434 792 771	626	8,1 min
1 180 591 620 717 572 279 207	684	8,1 min
1 180 591 620 717 901 172 461	720	8,1 min
1 180 591 620 718 577 721 559	780	8,1 min
1 180 591 620 719 294 224 537	802	8,1 min
1 180 591 620 720 955 194 707	806	8,1 min
1 180 591 620 724 220 277 437	870	8,1 min
1 180 591 620 724 880 139 029	914	8,1 min
1 180 591 620 821 862 275 773	916	1 h and 26,1 min
1 180 591 620 839 196 851 809	924	1 h and 41,4 min
1 180 591 620 894 134 391 499	940	2 h and 14,0 min
1 180 591 620 909 492 164 771	1020	2 h and 35,2 min
1 180 591 620 910 787 543 027	1034	2 h and 35,2 min
1 180 591 621 421 105 449 327	1134	9 h and 11,5 min

The last result in table 1, 1 062, has a Merit of 23.9, which is very good, because it also is a continuation of all known prime gaps > 0 (albeit it is not a new record). The last result in table 2 has a merit of 23.4, also good, but it is not in the continuation from 0.

Table 3. Gaps and Wall clock execution time for numbers > 3
(',' used as decimal point)

From prime	Prime gap	Time
7	4	*41,0 sec*
23	6	*41,0 sec*
89	8	*41,0 sec*
113	14	*41,0 sec*
523	18	*41,0 sec*
887	20	*41,0 sec*
1 129	22	*41,0 sec*
1 327	34	*41,0 sec*
9 551	36	*41,0 sec*
15 683	44	*41,1 sec*
19 609	52	*41,1 sec*
31 397	72	*41,1 sec*
155 921	86	*41,1 sec*
360 653	96	*41,1 sec*
370 261	112	*41,1 sec*
492 113	114	*41,1 sec*
1 349 533	118	*41,1 sec*
1 357 201	132	*41,1 sec*
2 010 733	148	*41,1 sec*
4 652 353	154	*41,1 sec*
17 051 707	180	*41,1 sec*
20 831 323	210	*41,1 sec*
47 326 693	220	*41,2 sec*
122 164 747	222	*41,3 sec*
189 695 659	234	*41,3 sec*
191 912 783	248	*41,3 sec*
387 096 133	250	*41,5 sec*

From prime	Prime gap	Time
436 273 009	282	*41,5 sec*
1 294 268 491	288	*42,1 sec*
1 453 168 141	292	*42,2 sec*
2 300 942 549	320	*42,3 sec*
3 842 610 773	336	*43,5 sec*
4 302 407 359	354	*43,7 sec*
10 726 904 659	382	*1,5 min*
20 678 048 297	384	*2,3 min*
22 367 084 959	394	*2,3 min*
25 056 082 087	456	*2,3 min*
42 652 618 343	464	*3,9 min*
127 976 334 671	468	*10,6 min*
182 226 896 239	474	*15,7 min*
241 160 624 143	486	*20,8 min*
297 501 075 799	490	*25,2 min*
303 371 455 241	500	*26,1 min*
304 599 508 537	514	*26,1 min*
416 608 695 821	516	*35,7 min*
461 690 510 011	532	*40,1 min*
614 487 453 523	534	*53,3 min*
738 832 927 927	540	*1 h and 4,0 min*
1 346 294 310 749	582	*1 h and 59,0 min*
1 408 695 493 609	588	*2 h and 4,4 min*
1 968 188 556 461	602	*2 h and 55,5 min*
2 614 941 710 599	652	*3 h and 55,3 min*
7 177 162 611 713	674	*11 h and 0,4 min*
13 829 048 559 701	716	*21 h and 32,9 min*

We earlier remarked that we would discuss how the size of numbers affects the result, or more specifically, if lager gaps occur more often for large numbers. From the Wikipedia table and our tables 1 and 2, we see that if one starts the search from 3, we have to investigate up to a 16-digit number to find a gap > 1 000, a 13-digit additional numbers in table 1 to find gap > 1 000, and an 11 digit numbers in table 2. In total, a factor 100 000 lower when starting at 2^{70} than at 3! As described above, almost the same factor of numbers necessary to test we find between table 3 (a subset of the Wikipedia table), tables 1 and 2 when we look how fast we reach at least 716.

Also, the running time for finding really big gaps > 1 000, we see that the running times are much smaller, and that the table 3 starting at number 3, it is not possible to reach a gap of 1 000 in any reasonable time with a traditional algorithm.

We earlier claimed that we can make really large speculative prime gaps in the sense that if we made a SSE far out, say starting at 10^{45}, and of length = 10^9, and then crossed out on this SSE with all the primes on that SSE that we were able to produce in a reasonable

amount time, we might produce really big prime gaps. We would have to make, trivially, a new class Int5 or Int6 for holding such large numbers.

Because we know that all non-prime (composite) numbers on this SSE are genuine, crossing out with more prime numbers will only make more non-primes, not invalidating those already found. It is then the remaining uncrossed numbers that are PPNs (= true primes, or composite numbers with a large first factor). Anyway, the distance between two such PPNs is either a true prime gap or part of an even larger prime gap – voila!

10. Conclusions

We have set out to make space- and time-efficient algorithms for finding large prime gap on a single computer, and we believe that we to a large extent, have succeeded. We have introduced a new algorithm: the sliding Sieve of Eratosthenes (SSE) saving both time and space, the formula for crossing out with prime numbers on such a free-standing segment (SSE), and a faster method to find record gaps on a SSE. The first two of these in combination are obviously the most space and time saving of our algorithms.

This is our answer to the question on better algorithms versus more computers. However, we have found no new record large gap so far, but it might be possible to find a new record by starting from the last known result in table 1.

References

[1] Wikipedia articles on '*Prime number', 'prime gap'* and *'Sieve of Eratosthenes*. www.en.wikipedia.org/wiki/English_Wikipedia , (visited 28.08.2020)

[2] Horsley, Samuel. "ΚΟΣ ΚΙΝΟΝ ΕΡΑΤΟΣ Θ ΕΝΟΥ Σ . Or, *The Sieve of Eratosthenes. Being an Account of His Method of Finding All the Prime Numbers*, by the Rev. Samuel Horsley, F. R. S." *Philosophical Transactions (1683-1775)* 62 (1772): 327-47. (visited 28.08.2020) www.jstor.org/stable/106053.

[3] Andersen, Jens Kruse: *The Top-20 Prime Gaps*, http://primerecords.dk/primegaps/gap⌈s20.htm#top20meritset, (visited 28.08.2020)

[4] Tomás Oliveira e Silva: *Gaps between consecutive primes*, http://sweet.ua.pt/tos/gaps.html (visited 28.08.2020)