13

The Sieve of Eratosthenes – How Fast Can We Compute a Prime Number Table?

Rolf H. Möhring and Martin Oellrich

Technische Universität Berlin, Berlin, Germany
Beuth Hochschule für Technik Berlin, Berlin, Germany

A **prime number**, or just **prime**, is a natural number that is not divisible without remainder by any other natural number but 1 and itself. Primes are scattered irregularly among the set of natural numbers. This fact has fascinated and occupied mathematicians throughout the centuries.

A **prime number table up to *n*** is a list of all primes between 1 and n. It begins as follows:

$$2 \quad 3 \quad 5 \quad 7 \quad 11 \quad 13 \quad 17 \quad 19 \quad 23 \quad 29 \quad 31 \quad 37 \quad 41 \quad \ldots .$$

Over time, many problems involving primes were found. Not all of them have been solved. Here are two examples.

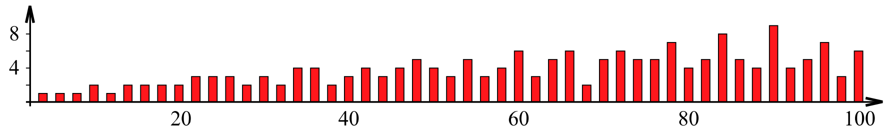
Christian Goldbach (1694–1764) formulated an interesting observation in 1742:

Every even number greater than or equal to 4 can be written as the sum of two primes.

For instance, we find:

$$4 = 2 + 2, \quad 6 = 3 + 3, \quad 8 = 3 + 5, \quad 10 = 3 + 7 = 5 + 5, \quad \text{etc.}$$

This proposition demands that there be at least one such representation. In fact, there are several for most numbers. The following diagram, based on a prime table, shows the number of different prime sums. On the x-axis, we see the partitioned (even) numbers.



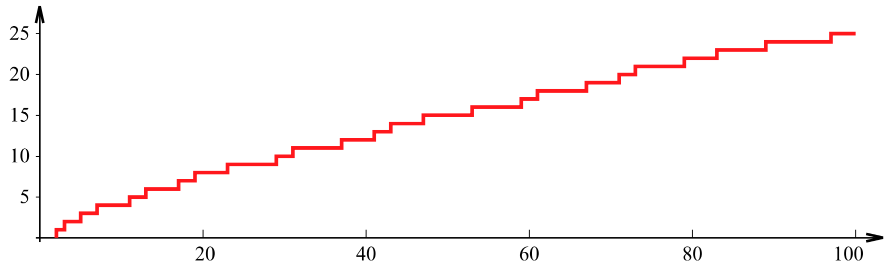
The slight trend upward in the columns continues with increasing n. No even number was found for which the proposition fails. Nevertheless, no proof is known that it holds *for all* of them.

B. Vöcking et al. (eds.), *Algorithms Unplugged*,
DOI 10.1007/978-3-642-15328-0_13, © Springer-Verlag Berlin Heidelberg 2011

Carl Friedrich Gauß (1777–1855) examined the distribution of the primes by counting them. He considered the function

$$\pi(n) := \text{ number of primes between 1 and } n.$$

A diagram of this function looks like this:



$\pi(n)$ is called a *step function*, for obvious reasons. Gauß constructed a "continuous" curve clinging as close as possible to $\pi(n)$, no matter how large n grows. In order to picture his plan and to check his results later, he needed a prime table. (This problem has since been solved. Deeper coverage exceeds the scope of this book.)

Today, primes are not only a challenge for mathematicians, but are of very practical value. For instance, 100-digit primes play a central role in electronic cryptography.

From the Idea to a Method

As far as we know today, an ancient Greek introduced the first algorithm for the computation of primes: **Eratosthenes of Kyrene** (276–194 BC). He was a high-ranking scholar in Alexandria and a director of its famous library, containing the complete knowledge of ancient mankind. He and others studied the essential astronomical, geographical, and mathematical questions of their time: What is the perimeter of the Earth? Where does the Nile come from? How can one construct a cube containing twice the volume of another given one? We will follow his steps below from a simple basic idea to a practical method. Even in his time, it could be executed well on papyrus or sand. We will also investigate how fast it is in computing a fairly large prime table. As a measure for "large," we let $n = 10^9$, i.e., *one billion.*

A Simple Idea

According to the definition of prime numbers, for any m *not* a prime there are two natural numbers i, k with the property that

$$2 \leq i, k \leq m \quad \text{and} \quad i \cdot k = m.$$

We use this fact and formulate a very simple prime table algorithm:

- write down all numbers between 2 and n into a list,
- form all products $i \cdot k$, where i and k are numbers between 2 and n, and
- cross out all reoccurring results from the list.

We immediately see how this prescription does what we want: all numbers remaining in the list never occurred as a product. Consequently, they cannot be written as a product and are thus prime.

How Fast Is the Computation?

In order to analyze the algorithm, we write the individual steps of the basic idea more formally and enumerate the lines:

PRIME NUMBER TABLE (basic version)

```
1  procedure PRIME NUMBER TABLE
2  begin
3      write down all numbers between 2 and $n$ into a list
4      for $i := 2$ to $n$ do
5          for $k := 2$ to $n$ do
6              remove the number $i \cdot k$ from the list
7          endfor
8      endfor
9  end
```

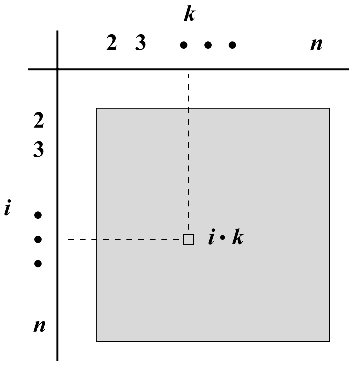
If the number $i \cdot k$ in step 6 is not present in the list, nothing happens.

This algorithm can be programmed in a straightforward fashion on a computer, and we can measure its time consumption. On a Linux PC (3.2 GHz), we get the following running times:

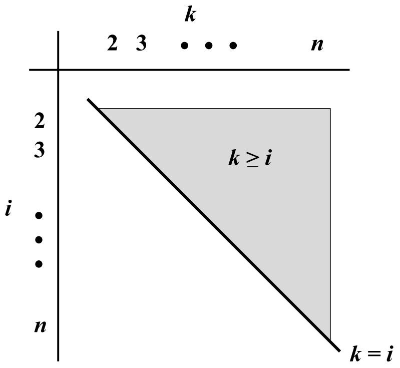
n	10^3	10^4	10^5	10^6
Time	0.00 s	0.20 s	19.4 s	1943.4 s

We clearly see how increasing n by a factor of 10 leads to a longer computation time by a factor of approx. 100. This was to be expected, since i as well as k run over a range about 10 times as large. The algorithm forms almost 100 times as many products $i \cdot k$.

From this, we can calculate the time needed for $n = 10^9$: we must multiply the time for $n = 10^6$ by a factor of $(10^9/10^6)^2 = 10^6$, resulting in $1943 \cdot 10^6$ seconds = *61 years and 7 months.* Clearly, this is of no practical use.



(a) $i \cdot k$ for all $i, k \in \{2, \ldots, n\}$



(b) $i \cdot k$ for $k \geq i$ only

Fig. 13.1. Computing products $i \cdot k$ in a certain range

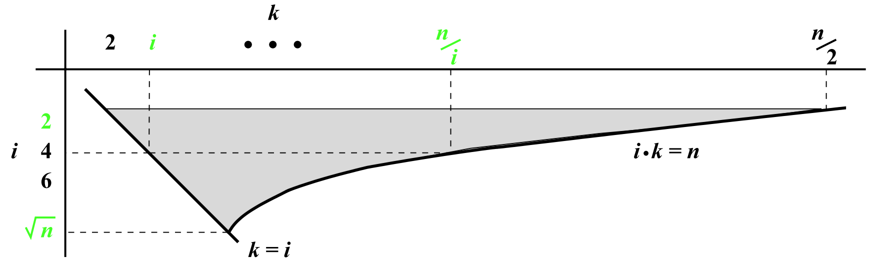
How Does the Algorithm Spend Its Time?

The algorithm generates all products in a certain range (Fig. 13.1(a)).

However, every individual result of $i \cdot k$ is needed only once. After being removed from the list, the algorithm would never have to generate it again. Where does it do surplus work? This happens, for instance, when i and k attain exchanged values, say, $i = 3$, $k = 5$ and later $i = 5$, $k = 3$. In both cases, the results of the product are identical, as is assured by the commutative rule of multiplication: $i \cdot k = k \cdot i$. For this reason, we restrict $k \geq i$ and avoid these duplications (Fig. 13.1(b)).

This idea instantly saves half the work! Yet, even 30 years and 10 months are still too long to wait for our table. Where can we save more? In those cases when in step 6 never anything happens: for $i \cdot k > n$. The list contains numbers up to n, so there is nothing to remove beyond that.

So we need to execute the k-loop (line 5) only for those values satisfying $i \cdot k \leq n$. This condition immediately delivers the applicable k-range: $k \leq n/i$. As a side effect, we can also limit the i-range. From the two restrictions $i \leq k \leq n/i$, we conclude $i^2 \leq n$, and ultimately, $i \leq \sqrt{n}$. For larger i, there are no k-values to enumerate. The number domain generated now looks as follows:



The algorithm has now attained the following form:

Prime number table (better)

```
1  procedure Prime number table
2  begin
3      write down all numbers between 2 and $n$ into a list
4      for $i := 2$ to $\lfloor\sqrt{n}\rfloor$ do
5          for $k := i$ to $\lfloor n/i\rfloor$ do
6              remove the number $i \cdot k$ from the list
7          endfor
8      endfor
9  end
```

(The notation $\lfloor\cdot\rfloor$ means floor rounding, as i and k can attain integral values only.)

How fast have we become? The new running times:

n	10^4	10^5	10^6	10^7	10^8	$\mathbf{10^9}$
Time	0.00 s	0.01 s	0.01 s	2.3 s	32.7 s	**452.9** s

The effects are considerable, with our target of 10^9 within close sight: it is just seven and a half minutes away. We let the computer run and use this time to make some more improvements!

Do We Need Every i Value?

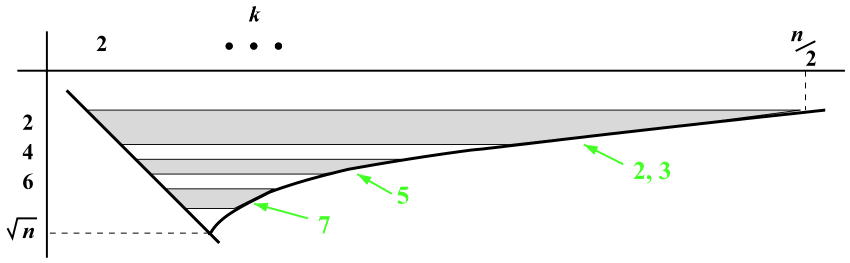
Let us consider what exactly happens within the i-loop (line 4): i remains fixed and k traverses its own loop (line 5). Doing so, the product $i \cdot k$ attains the values

$$i^2, \quad i(i+1), \quad i(i+2), \quad \ldots .$$

So when the k-loop is finished, no proper multiples of i are left in the list. The same applies to multiples of numbers less than i. They were removed earlier in the same way.

What happens if i is not a prime? Example $i = 4$: the product $i \cdot k$ attains the values $16, 20, 24, \ldots$. These numbers are all multiples of 2, since 4 itself has this property. In principle, there is nothing to do in the case $i = 4$. The same is true for all other even numbers $i > 4$.

Example $i = 9$: the product attains only multiple values of 9. Yet, those have already been enumerated as multiples of 3 and are thus redundant. This reasoning applies to all non-primes, since they possess a smaller prime divisor that was an i-value before them. Consequently, we need to execute the k-loop exclusively for prime i-values. See the following figure:

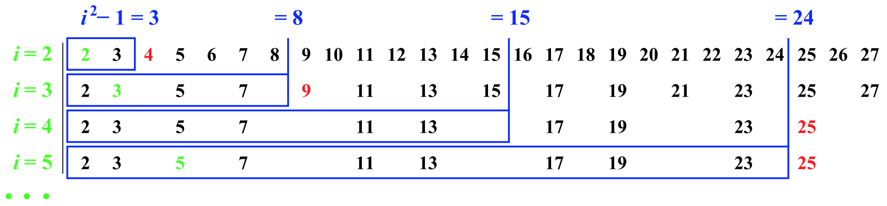


Whether i is a prime or not could be looked up in the list itself – if it were complete. We can trust it to contain primes only no earlier than termination. Or is it?

The answer is: yes and no. In general *yes*; otherwise we could abbreviate the algorithm. Consider, for instance, $n = 100$: the non-prime 91 must eventually be removed from the list. It is generated close to termination, when $i = 7$, $k = 13$.

Yet in our special case, *no*. We do not attempt to recognize arbitrary numbers as prime, but just the specific value i. Also, not at an arbitrary moment, but exactly at the beginning of the k-loop for the i-value in question. In this restricted case, the list returns the correct primeness information on i! Why?

We observed above that for every fixed i all removed values satisfy $i \cdot k \geq i^2$. Put differently, the range $2, \ldots, i^2 - 1$ remains unaltered. Upon growing i-values, this range expands and comprises all previous ranges. In the following figure, these ranges are marked blue. The first "wrong" number in every table row is printed in red.



		$i^2-1=3$					$=8$							$=15$									$=24$			
$i=2$	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
$i=3$	2	3		5		7		9		11		13		15		17		19		21		23		25		27
$i=4$	2	3		5		7				11		13				17		19				23		25		
$i=5$	2	3		5		7				11		13				17		19				23		25		
• • •																										

Now all of these ranges do not change until termination. Therefore, they must be correct *before* execution of the k-loop for the i-value in question. We can say the table is completed in quadratic steps. The essential i-values – the ones whose primeness we need – are printed in green. It is obvious how they always lie within a blue range. So in order to decide whether i is a prime number or not, we may simply look it up in the current list.

In our algorithm, we can thus enhance the i-loop as follows:

```
Prime number table (Eratosthenes)
1   procedure Prime number table
2   begin
3       write down all numbers between 2 and $n$ into a list
4       for $i := 2$ to $\lfloor\sqrt{n}\rfloor$ do
5           if $i$ is present in the list then
6               for $k := i$ to $\lfloor n/i \rfloor$ do
7                   remove the number $i \cdot k$ from the list
8               endfor
9           endif
10      endfor
11  end
```

It was this version of the method that the clever Greek presented. It is called the **Sieve of Eratosthenes**: *sieve* because it does not construct the desired objects, the primes, but filters out all non-primes.

Our time measurements on his algorithm read as follows:

n	10^6	10^7	10^8	$\mathbf{10^9}$
Time	0.02 s	0.43 s	5.4 s	**66.5** s

Even with $n = 10^9$, it needs roughly one minute!

Can We Get Even Faster?

With an argument similar to that for the prime i-values above, we can further restrict the k-values needed: we take only those found in the list! If k was removed as a non-prime, it possesses a prime divisor $p < k$. In the i-loop where $i = p$, all proper multiples of p were removed. In particular, k and its multiples were among them. Nothing remains to do.

Again enhancing the algorithm, it appears to be natural to do it as follows:

```
6               for $k := i$ to $\lfloor n/i \rfloor$ do
7                   if $k$ is present in the list then
8                       remove the number $i \cdot k$ from the list
9                   endif
```

But caution! This formulation is misleading. Running the algorithm like that, we get the following list:

2 3 5 7 *8* 11 *12* 13 17 19 *20* 23 *27* *28* 29 31 *32* 37 ...

What is wrong? Let us look at the first steps more closely. After initializing the list with all numbers up to n (line 3), it reads:

2 3 4 5 6 7 8 9 10 11 ...

First, we set $i = 2$ and then $k = 2$. The number 2 stands in the list, so we remove $i \cdot k = 4$:

2 3 – 5 6 7 8 9 10 11 ...

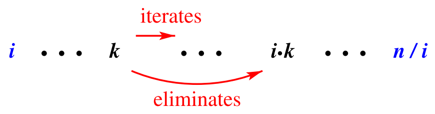
Next, we set $k = 3$. The number 3 also stands in the list and we remove $i \cdot k = 6$:

2 3 – 5 – 7 8 9 10 11 ...

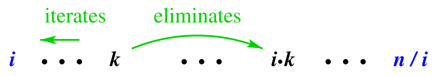
Now something happens: $k = 4$ is no longer present in the list, since we removed it. According to the new **if**-condition, we must skip the k-loop and continue with $k = 5$:

2 3 – 5 – 7 *8* 9 – 11 ...

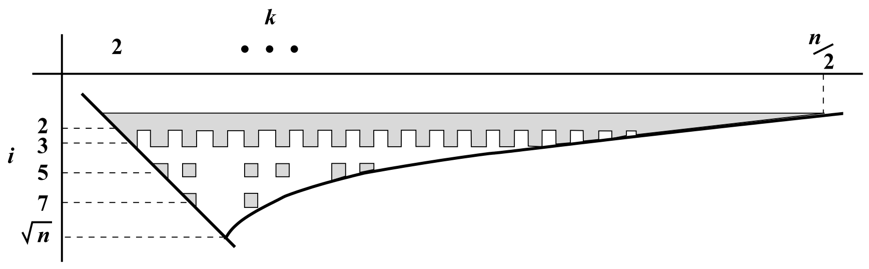
This way, $2 \cdot 4 = 8$ is erroneously never removed from the list. The problem is that k eliminates only numbers $i \cdot k > k$ and is subsequently incremented. Eventually, k attains the value of a former product $i \cdot k$ and the method unfavorably effects on itself:



The solution is to let k traverse its loop range *backwards*, thus avoiding the unwanted feedback:



According to this reasoning, only the following products $i \cdot k$ are formed:



Summarizing, we achieve the following version of the algorithm:

Prime number table (final version)

```
procedure Prime number table
begin
    write down all numbers between 2 and $n$ into a list
    for $i$ := 2 to $\lfloor\sqrt{n}\rfloor$ do
        if $i$ is present in the list then
            for $k$ := $\lfloor n/i\rfloor$ to $i$ step -1 do
                if $k$ is present in the list then
                    remove the number $i \cdot k$ from the list
                endif
            endfor
        endif
    endfor
end
```

Its running times:

n	10^6	10^7	10^8	$\mathbf{10^9}$
Time	0.01 s	0.15 s	1.6 s	**17.6** s

This result is very acceptable by today's standards. Starting out with a naive basic version, we have accelerated the method for $n = 10^9$ with a few closer looks by a **factor of 254.5 million!**

What Can We Learn from This Example?

1. Simple computation methods are not always *efficient.*
2. In order to accelerate them, we need to *understand well* how they work.
3. Often *several different improvements* are possible.
4. *Mathematical ideas* can lead very far!

Further Considerations

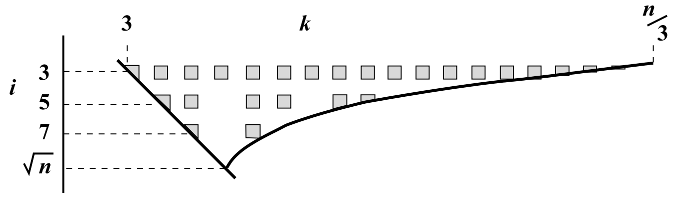
A time of 17.6 seconds is a good result for spending a few thoughts on the algorithm. Yet after all, how good is that? Have we reached the best possible?

Let us establish what the algorithm generally has to achieve. It must generate all non-primes up to n at least once and remove them from the list. Below $n = 10^9$ there are exactly 949,152,466 of them. Counting the number of products $i \cdot k$ computed, we obtain the following values for our variants above:

	Basic version	Better	Eratosthenes	Final version
Num. products	10^{18}	$9.44 \cdot 10^9$	$2.55 \cdot 10^9$	$9.49 \cdot 10^8$
Relation to nonprimes	$1.1 \cdot 10^9$	9.9	2.7	**1.0**

In fact, the final version performs just as much work as necessary. In this respect, it is optimal!

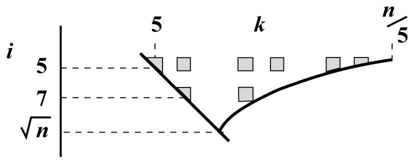
Interestingly, this does not mean we could no longer improve the algorithm. The comparison with the number of non-primes is no absolute measure, since we can still diminish them. The following trick works: the list is not initialized with *all* numbers from 2 on, but just with 2 itself and *all odd* numbers ≥ 3. We know that the even numbers ≥ 4 are never prime, so why generate them in the first place? The procedure has considerably less work to do on a list containing odd prime candidates only. The k-loop for $i = 2$, the longest one of all, is completely omitted. Only the following products are being generated now:



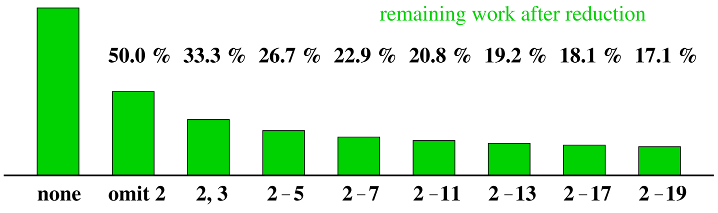
We can play more on this theme: we also omit initializing the list with all proper multiples of 3. At the start, it contains the numbers

2 3 5 7 11 13 17 19 23 25 29 31 35 37 41 43 47 49 ...

and the actual sieve work begins with $i = 5$:



This way, we can achieve ever-decreasing initial lists for the same range up to n by omitting the proper multiples of 5, 7, 11, 13, etc.:



We find the same characteristic in the running times and the memory consumptions, as the lists themselves are diminished:

		Omit							
Reduction	None	2	2, 3	2–5	2–7	2–11	2–13	2–17	2–19
Run time [s]	17.6	33.0	22.6	17.8	14.7	13.3	**12.6**	24.0	25.9
Memory [MByte]	119.2	59.6	39.7	31.8	27.3	24.8	22.9	21.6	20.4

The list is represented here by a bit array. At position i in the array, a 1 marks the primeness of i while a 0 indicates that i is a nonprime.

If we choose a linked list as the memory representation instead, we would incur two disadvantages. First, in order to look up the primeness property of some number, we would have to search for the appropriate entry first. Second, the numbers run up to 9 digits and we would need to store all 50,847,534 prime numbers below one billion in 1551.7 MByte memory.

A note on the almost double computation time after omitting the even initial numbers. Since the list is condensed, we need a transformation between the list indices $(1, 2, 3, 4, \ldots)$ and the numbers addressed (in this example: $2, 3, 5, 7, 9, \ldots$). This uses up some time. Yet altogether, we achieve an excellent 12.6 s with a list reduced by the multiples of 2 through 13. Beyond that, the preparation of the transformation data dominates over the actual list computation, rendering further reduction useless.

Further Reading

1. `http://en.wikipedia.org/wiki/Sieve_of_Eratosthenes`
   This Wikipedia article offers a concentrated introduction to the topic.
2. On the homepage of one author, we provide the `C` code with which we measured the running times in the tables:
   `http://prof.beuth-hochschule.de/oellrich/aktivitaeten-mit-schuelern.html`
3. Chapters 14 (One-Way Functions) and 16 (Public-Key Cryptography)
   In Chaps. 14 and 16, the primary topic is not prime numbers. But the treated problems essentially reduce to finding divisors of very large natural

numbers fast. Since primes possess no proper divisors, none can be found fast and thus the factorization problem cannot be broken down. Large primes are hard to recognize and therefore qualify as building bricks for encryption methods. However, primes with 100 or more digits cannot be practically handled by means of prime tables. For this purpose, other methods generate numbers of this magnitude directly.

4. Chapter 20 (Hashing)
   In Chap. 20, the main topic is also not prime numbers. Yet for most purposes, hash tables of prime length are advantageous. Hash functions of the form *key value modulo table length* generally have good distribution properties when the keys are uniformly distributed. A certain lookup method called *double hashing* can in this way assert to find any free entry. For the purpose of selecting a suitable prime for a hash table, a prime table is very useful.
5. A *twin prime* is a pair of prime numbers with a difference of exactly two. The first such pairs are: $(3, 5), (5, 7), (11, 13), (17, 19), (29, 31), (41, 43), (59, 61), (71, 73), \ldots$. The very first one $(3, 5)$ is an exception for two reasons. First, the number 5 is the only one to occur in two pairs. Second, in all other twins, the number between the primes is divisible by 6. This must be the case, since among three contiguous numbers exactly one must be divisible by 3 and at least one by 2. The two primes (except with $(3, 5)$) are neither divisible by 2 nor by 3, therefore the middle number contains both factors. Twin primes have been found in arbitrary large ranges of natural numbers. However, there is still no proof whether finitely or infinitely many of them exist. In a prime table up to 10^9, we can find 3,424,506 such twins.