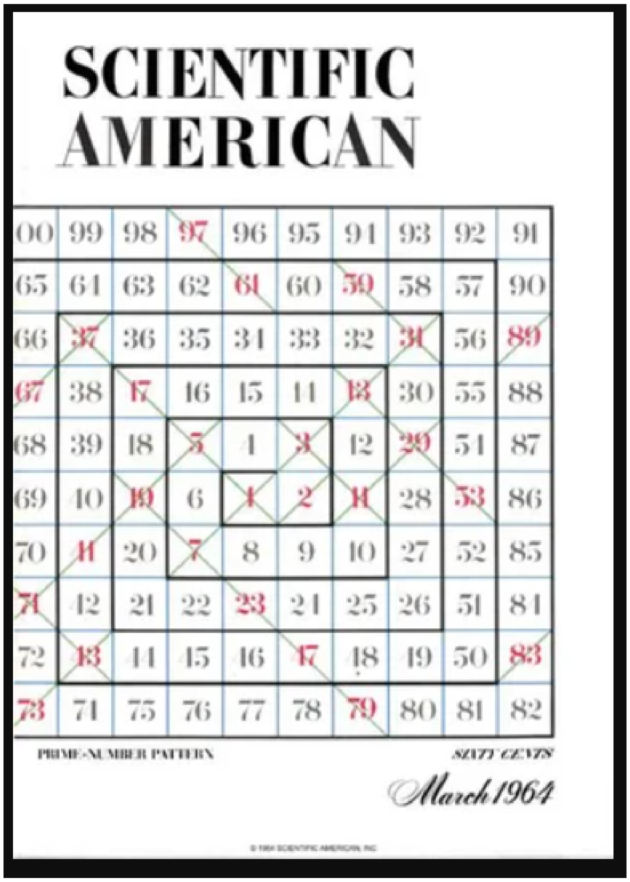
The Prime Directive
Mathematical Programming with Python

https://people.sc.fsu.edu/~jburkardt/classes/python_2025/primes/primes.pdf



SCIENTIFIC AMERICAN

00	99	98	97	96	95	94	93	92	91
65	64	63	62	61	60	59	58	57	90
66	37	36	35	34	33	32	31	56	89
67	38	17	16	15	14	13	30	55	88
68	39	18	5	4	3	12	29	54	87
69	40	19	6	1	2	11	28	53	86
70	41	20	7	8	9	10	27	52	85
71	42	21	22	23	24	25	26	51	84
72	43	44	45	46	47	48	49	50	83
73	74	75	76	77	78	79	80	81	82

PRIME-NUMBER PATTERN

SIXTY CENTS

March 1964

© 1964 SCIENTIFIC AMERICAN, INC

Stanislaw Ulam plotted the integers in a spiral pattern, and noticed that prime numbers seemed to fall into diagonal patterns.

Prime Numbers

- *Division is the only tool we need to study prime numbers.*
- *The sieve of Erastosthenes is the basic (expensive) tool for listing primes.*
- *Primes provide a unique factorization of every integer.*
- *Primes seem to occur irregularly, and yet to follow larger patterns.*
- *Gauss studied approximations to $\pi(n)$, the number of primes up to n;*
- *Prime numbers also generate many functions useful in number theory and cryptography.*
- *Determining the prime factors of an arbitrary number is extremely expensive.*
- *Cryptography relies on keys that are the products of enormous prime numbers.*

1 Overview

We will start off looking at prime numbers, as the Greeks did, because you don't need any special mathematical tools for this investigation except for division. To decide whether the number 37 is prime, you simply have to answer the following 35 questions:

```
is 37 divisible by 2?
is 37 divisible by 3?
...
is 37 divisible by 36?
```

If every answer is "no", then 37 is a prime number.

Prime numbers fascinated the Greeks, because they seemed to behave the way chemical elements do. Just like every substance can be broken down into some set of the chemical elements, every integer can be uniquely factored into a product of prime numbers. And just like Mendeleev's table of chemical elements, a Greek named Erastosthenes showed how to start a sort of table of prime numbers using a method he called the "sieve". You write down a long (but finite) list of numbers from 1 to n. Cross out every second number (eliminating even values), then cross out every third number (eliminating numbers divisible by 3), and so on. If you do all the possible crossing outs, the remaining values are the start of the prime number table.

Of course, having a computer at hand means we can automate these repetitive calculations. It also means we can examine much larger values of n than the Greeks did. Because the prime numbers really are fundamental to fields like number theory and cryptography, a number of related functions and questions have arisen over the years, and we will look at some simple examples.

2 Primes

We have:

- an infinite number of primes (thanks, Euclid!).
- arbitrarily large gaps between successive primes. For a gap of 4, consider 5!+2, 5!+3, 5!+4, 5!+5.
- estimates for the number of primes less than a given integer n (thanks, Gauss!).
- a "sieve" procedure (sure but expensive) to produce all primes up to n (thanks, Eratosthenes!)
- tests that can report that n is "probably" prime (thanks, Fermat and others!)

We **don't** have:

- a simple formula $p = f(n)$ for the n-th prime.
- a simple formula $n = f(p)$ for the index of a given prime.
- a simple formula $p = f(i)$ which will only produce prime values.
- a cheap, simple, guaranteed test to determine whether an arbitrary number n is prime.

Some mathematical structures:

- $\pi(n)$ number of primes less than or equal to n.
- $\psi(n)$ number of integers less than n and relatively prime to it.
- $\tau(n)$ number of divisors of n.
- $\sigma(n)$ sum of divisors of n.

For a perfect number, like $n = 28$, we have $\sigma(n) = n$: 1+2+4+7+14=28.

3 Is n prime?

To answer the question of whether an integer n is prime, we can simply check whether n is divisible by any integer $2 \leq i < n$. To do this in Python, we use the Python version of the `modulus()` or `modulo()` function. Mathematically, `a modulo b` evaluates the remainder after integer division, so that $\frac{a}{b} = d$ with remainder r, or more concisely, $a = d * b + r$. Here, the remainder r satisfies $0 \leq r < b$. In particular, if `a modulo b` is zero, then a is exactly divisible by b.

In Python, the modulo function is expressed as `a % b`; frequently, such an expression is surrounded by parentheses, as in `(a % b)`, for clarity, and to avoid accidental interfence from neighboring numerical expressions. Now we can start to ask whether 37 is a prime number by evaluating the series of expressions “37 mod 2”, ‘“37 mod 3”, ..., “37 mod 36”. If we encounter any zero result in this series, then 37 is not prime.

We may represent our answer as the logical variable `is_prime`. We might start by initializing this variable to `True`, that is, we might begin by assuming 37 is prime. Now we check each divisor, and if we come up with a zero remainder, we will reset `is_prime` to `False`.

Each question can be expressed by posing an `if()` statement, followed by a possible action to be taken. For instance, one of our questions will ask if 37 is divisible by 6. We would make this check, and carry out the appropriate response, as follows:

```
if ( ( 37 % 6 ) == 0 ):
  is_prime = False
```

The rules for an `if()` statement require us to follow the word `if` by a condition, that is, a variable or a statement that is true or false. Immediately following the `if()` statement, we are to list one or more statements to be carried out if the condition turns out to be true. So here, we are saying, “If 37 can be evenly divided by 6, then 37 is not a prime”.

I will make a comment on programming style here. We could just as well have written our check with no parentheses:

```
if 37 % 6 == 0:
  is_prime = False
```

You may prefer to do less typing; I find the parenthesis helpful in breaking expressions up into understandable chunks.

Of course, to be sure that 37 is a prime, we have to check every potential divisor, from 2 to 36, and then make a final report. We can start by setting a variable `is_prime`, which will report the result of our investigation. At the beginning, we have no evidence that 37 is not prime, so we let `is_prime` begin with the value `True`. Then we check every possible divisor, and if *any* division results in a zero remainder, then we know 37 is not a prime, so we reset `is_prime` to `False`.

So one way to determine if 37 is prime is suggested as follows:

```
  is_prime = True
  if ( ( 37 % 2 ) == 0 ):
    is_prime = False
  if ( ( 37 % 3 ) == 0 ):
    is_prime = False
  ... more statements...
  if ( ( 37 % 36 ) == 0 ):
    is_prime = False

  print ( 'is_prime(37) = ', is_prime )
```

4 Using a `for` statement

Of course, it is tedious to type such a long series of statements. Worse, if we want to test whether 39 is prime, we have to start typing all over again. For this reason, we should be grateful that Python includes a single statement that allows us to generate the sequence of test divisors in a compact way. To generate a sequence of divisors, we will use a *for()* statement:

```
for i in range ( low, high+1 ):
  one or more indented statements involving i
```

The variable, which here we have called `i`, is termed the *loop index*. The statements which immediately follow the `for()` statement, and which are indented, will be executed repeatedly, once for each value of `i`. These values will start with `low`, and continue up to `high`. Notice the very strange fact that in Python, the loop variable stops just before reaching the upper limit of the `for()` loop. Keep this strange fact in mind. It does not correspond to how humans think, and it will cause you endless trouble. Also notice that the `for()` statement ends with a colon. Forgetting this colon is one of the most common mistakes I make.

To see how this statement helps us, here is how we can now check whether 37 is prime:

```
is_prime = True
for d in range ( 2, 37 ):
  if ( ( 37 % d ) == 0 ):
    is_prime = False
print ( 'is_prime(37) = ', is_prime )
```

5 Efficiency Step 1

Our code as written is a little stupid. Suppose we were checking whether 36 is prime. Our first test divisor of 2 evenly divides 36, and so we know we don't have a prime. Nonetheless, the code will still go on, checking divisors 3, 4, 5, ..., 35. The `for()` loop doesn't know that it's wasting our time. Luckily, if we are working inside a `for()` loop and we suddenly realize we have no need to execute any more of the steps, we can use a `break` statement, which jumps out of the loop, and moves on to the next statement.

To make the point, we will change our test value of n to 36:

```
is_prime = True
for d in range ( 2, 36 ):
  if ( ( 36 % d ) == 0 ):
    is_prime = False
    break
print ( 'is_prime(36) = ', is_prime )
```

Notice that it's very important that the `break` statement is indented to be directly under the `is_prime = False` line, so that it is only carried out when the *if()* condition is true, and after the value of *is_prime* has been changed. What would happen if we switched the order of these two statements? What would happen if we changed the indentation of `break` so that it started in the same column as the `if()` statement? Or with the `for()` statement?

6 Using a function

If we are only interested in studying 37, then our work is done. But it's also likely that we could want to know whether other numbers are prime. So it would be convenient to turn our script into a function, so that we can use it any time as a sort of formula. To do this, we need to make a sandwich out of our code, wrapping it with a function definition statement at the beginning, and a `return` statement at the end.

```
def is_prime1 ( n ):
  value = True
  for d in range ( 2, n ):
    if ( ( n % d ) == 0 ):
      value = False
  return value
```

And our slightly more efficient code would be:

```
def is_prime2 ( n ):
  value = True
  for d in range ( 2, n ):
    if ( ( n % d ) == 0 ):
      value = False
      break
  return value
```

We could use it like this:

```
  n = 37
  value = is_prime2 ( n )
  print ( 'is_prime2(', n, ') = ', value )
```

What would happen if we now gave the command:

```
for n in range ( 1, 50 ):
  value = is_prime2 ( n )
  print ( 'is_prime2(', n, ') = ', value )
```

7 A Big Efficiency Step

Since determing whether an integer is prime is a useful operation, we would like to make sure that our algorithm runs efficiently. We would like to make sure that, if possible, it doesn't waste time. Notice that, because 37 is prime, our code has to do 35 successive divisions. We hardly notice any lag in time for this small problem. But what happens when we check the primality of 27644437? If it is prime, our code will take 27,644,435 steps to verify this. At some point, there will be so much arithmetic that we will not be willing to wait for an answer. So it's important to try to improve our code if we can.

So let's think about how many divisors we really have to check. The square root of 37 is about 6.08. If we have checked that 37 cannot be divided by 2, 3, 4, 5, or 6, then we actually already know that 37 must be a prime; if it had a divisor that was 7 or larger, than it would also have a divisor that was 6 or smaller, and we've checked those. So we can revise our `for()` loop to go as far as $\sqrt(n)$, or more precisely, the largest integer less than or equal to $\sqrt(n)$.

Here's the revised code.

```
def is_prime3 ( n ):
  from math import sqrt
  dhi = int ( sqrt ( n ) ) + 1
  value = True
  for d in range ( 2, dhi ):
    if ( ( n % d ) == 0 ):
      value = False
      break
  return value
```

If you compare the new and old versions of this function for large values of n, you should start to see a significant difference in speed.

8 Even a Little More Efficiency

Another simple improvement to the algorithm skips checking divisors that are multiples of 2 or 3. You can refer to `https://en.wikipedia.org/wiki/Primality_test` for details about the following program:

```python
def is_prime4(n: int) -> bool:
    if n <= 3:
        return n > 1
    if n % 2 == 0 or n % 3 == 0:
        return False
    limit = int(n**0.5)
    for i in range(5, limit+1, 6):     # This three-part for statement jumps by sixes
        if n % i == 0 or n % (i+2) == 0:
            return False
    return True
```

I didn't write this code, and you should notice some big differences in formatting and style. Perhaps we need to look carefully at the `for()` statement to understand what it's doing (I didn't explain "skipping" when I talked about `for()` last week.) Once we see what it's doing, we also need to convince ourselves that it is correctly skipping potential divisors that are multiples of 2 and 3.

9 Python has its own prime checker

The Python `sympy()` library includes a function `isprime()`, which does the task we have been trying to do ourselves. We can expect it is even more efficient than our codes. However, we can't see the details of this function, so to make a comparison, we will have to pick large values of n and then somehow measure how long each of our codes takes. For the codes we have written ourselves, we could also print the number of times the modulo function was called.

10 How to time a piece of code

To time a piece of code in Python, we can use the function `perf_counter()` from the `time` library. This function returns a reading from the clock, measure in seconds. By checking the clock before and after you do something, the difference in the readings gives you a measure of how long the process took.

```python
from sympy import isprime
from time import perf_counter

tic = perf_counter ( )
value = isprime ( 6700417 )
toc = perf_counter ( )
print ( 'is_prime1(6700417) required ', toc - tic, 'seconds' )

tic = perf_counter ( )
value = is_prime1 ( 6700417 )
toc = perf_counter ( )
print ( 'is_prime1(6700417) required ', toc - tic, 'seconds' )
... and so on for is_prime2(), is_prime3(), and is_prime4()
```

These results should convince you that programs can be very inefficient, that sometimes small changes can greatly speed up your code, and that a function written by the developers is often much faster than the code you might write yourself.

11 The distribution of primes

Primes seem to pop up irregularly within the natural numbers. Euclid showed that the set of primes could not be finite, and even suggested, given a finite set of primes, how to create a number which was either a new prime, or had a factor which was a new prime. But mathematicians suspected there was some law that roughly governed how many primes we could expect to see between 1 and any given integer n. This quantity was designated $\pi(n)$, the prime counting function. Gauss gave the first useful estimate:

$$\pi(n) \approx \frac{n}{\log(n)}$$

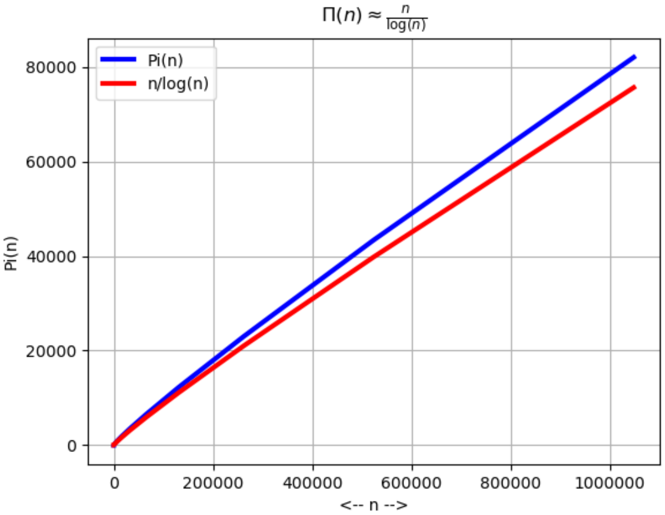
We can test this result by plotting some sample values, using for n the successive powers of 2:

```
from math import log
import matplotlib.pyplot as plt

n_list =  [ 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384 ]
pi_list = [ 1, 2, 4,  6, 11, 18,  31,  54,  97,  172,  309,  564, 1028,  1900 ]
nlogn_list = n_list / log ( n_list )

plt.plot ( n_list , pi_list , ’b–’ )
plt.plot ( n_list , nlogn_list , ’r–’ )
plt.show ( )
plt.close ( )
```

which produces the following plot (I cut some data and plotting commands):



so we have some evidence that Gauss was on the right track.

Meanwhile, other mathematicians were looking at ways of automatically generating prime numbers, or at least, of finding number sequences that were more dense with primes than the natural numbers were.

12 Fermat primes

While searching for ways to generate primes, Pierre Fermat considered *Fermat numbers*, with the simple formula

$$F_n = 2^{2^n} + 1$$

The first values in the sequence are

$$\begin{aligned} F_0 &= 2^{2^0}+1 = 2 \\ F_1 &= 2^{2^1}+1 = 5 \\ F_2 &= 2^{2^2}+1 = 17 \\ F_3 &= 2^{2^3}+1 = 257 \\ F_4 &= 2^{2^4}+1 = 65,537 \\ F_5 &= 2^{2^5}+1 = 4,294,967,297 \end{aligned}$$

Fermat was able to verify that the F_0 through F_4 were prime, and thought he might have found a way to produce an endless prime sequence. Euler, however, found a factorization of F_5, and it is generally believed that F_4 is the last Fermat prime.

This does mean that F_5 is a good test case for our `is_prime()` codes, and for any code that can look for the factors of a composite number.

13 Fermat's primality test

Fermat's little theorem states that, if p is prime, and a is not a multiple of p, then

$$a^{p-1} = -1 (\text{mod } p)$$

Because computationally, the `mod()` function will return positive remainders, we might prefer to write this as

$$a^{p-1} = p-1 (\text{mod } p)$$

Now suppose we are given a number which might or might not be prime. If we find a value a with $2 \leq a \leq p-2$ such that the condition fails, then n cannot be a prime. (Why don't we have to check $a = p-1$?). For a large n, there are many values of a to consider. It would be too expensive to check them all. What we are hoping is for a quick negative result after checking 3 or 4 values of a chosen at random. Thus, Fermat's primality test is generally probabilistic. If it fails, then n is definitely not a prime. But if it is passed, we really only can say that n was "lucky" so far; so far, it only looks like a possible prime.

14 Mersenne primes

Marin Mersenne looked at Mersenne numbers of the form

$$M_n = 2^n - 1$$

It is possible to show that, if n is not a prime, then M_n cannot be prime. In fact, if k divides n, then $2^k - 1$ divides $2^n - 1$. (This is also true for bases other than 2). Turning this around, we ask, if n is prime, will M_n be a Mersenne prime?

The sequence starts out promising:

$$\begin{aligned}
M_2 &= 2^2 - 1 = 3 \\
M_3 &= 2^3 - 1 = 7 \\
M_5 &= 2^5 - 1 = 31 \\
M_7 &= 2^7 - 1 = 127 \\
M_1 1 &= 2^{11} - 1 = 2047 \\
M_1 3 &= 2^{13} - 1 = 8191 \\
M_1 7 &= 2^{17} - 1 = 131071
\end{aligned}$$

All but one of these value is prime. (Can you spot the nonprime?) However, the sequence soon begins to return more nonprimes than primes. Nonetheless, researchers have continued to find ever higher indices for which M_n is prime. The 51st Mersenne prime, $2^{82,589,933} - 1$, is the largest prime number known.

The GIMPS (Great Internet Mersenne Prime Search) at `https://www.mersenne.org/primes/` records all the known Mersenne primes, and coordinates an ongoing search for new values.

15 The Lucas-Lehmer Test

Given how large the Mersenne numbers become, we have to be extremely clever if we want to be able to determine whether a large Mersenne number is prime. Luckily, because of the special structure that these numbers have, the Lucas-Lehmer test provides an exact answer efficiently. Let M_n be a given Mersenne number. Then the Lucas-Lehmer test defines a sequence s_i which starts at $s_0 = 4$ with subsequent values defined recursively

$$s_i = s_{i-1}^2 - 2$$

Then M_n is prime if and only if $s_{n-2} = 0 \mod M_n$ Here is pseudocode from Wikipedia for this calculation:

```
Lucas{Lehmer(p)
    var s = 4
    var M = 2^p - 1
    repeat p - 2 times:
        s = ((s x s) - 2 ) mod M
    if s == 0 return PRIME else return COMPOSITE
```

Notice that when we update the value of s, we use modular arithmetic every time, rather than simply for the check at the end. It turns out that this is legal, and keeps our s values from increasing without limit.

It's not hard to turn this into a Python code:

```
def lucas_lehmer ( n ):

  if ( n == 2 ):
    return True

  Mn = 2**n - 1
  s = 4
  for _ in range ( n - 2 ):
    s = ( ( s * s - 2 ) % Mn )

  return ( s == 0 )
```

The Mersenne numbers quickily become gigantic. It is amazing that we can still determine the primality of these monster numbers within a reasonable amount of computer time. As an exercise, we can try to tabulate the size of some of these numbers versus the time it takes to run the Lucas-Lehmer test.

Note the underscore used in the `for()` loop. This is programming convention to suggest that the loop index variable is of no interest, so we are giving it the stupidest name we can think of. Surprisingly, you can give any variable the underscore name, and it will hold a value and can be used in arithmetic like any other variable. I don't really think it's useful, but you will see other people doing it, and now you don't have to be so puzzled.

16 Python integers are literally enormous

One method of encrypting messages uses a public key key which is the product of two primes $key = p_1 * p_2$. The value of key is publicly listed, but the factors are secret. The message can only be decoded if the factors are known. If p_1 and p_2 are very large primes, then it is essentially impossible to determine the factorization of key. In most computer languages, integers are limited in size. For instance, 32-bit integers typically are allowed to have values from -2,147,483,648 to +2,147,483,647. The primes used in cryptography are much larger than these limits.

Surprisingly, Python integers have an unlimited range, which means that you can easily multiply, divide and add very large integers directly. (However, it is still essentially impossible to factor a given enormous integer!) However, there are many times when it is useful to be able to work with enormous integers. To give you an idea of what "large" means, consider the following problem, (which is now considered "small"!):

The RSA public key cryptography system was described in an article by Martin Gardner in Scientific American in August 1977. The public key was given as

```
key = 114,381,625,757,888,867,669,235,779,976,146,612,010,218,296,
      721,242,362,562,561,842,935,706,935,245,733,897,830,597,123,
      563,958,705,058,989,075,147,599,290,026,879,543,541
```

The challenge was to determine the factors p_1 and p_2. Seventeen years later, a team of 600 programmers working cooperatively managed to crack the problem:

```
p_1 = 3,490,529,510,847,650,949,147,849,619,903,898,133,417,764,638,
        493,387,843,990,820,577
```

and

```
p_2 = 32,769,132,993,266,709,549,961,988,190,834,461,413,177,642,967,992,
         942,539,798,288,533
```

In the exercises, you will be asked to verify that $key = p_1 * p_2$. You will **not** be asked to try to factor key on your own, nor to verify that p_1 and p_2 are prime. Those are very very hard problems!