1 The Primes Library

A prime number is any integer greater than one that has no exact integer divisors other than one and itself.

Prime numbers have increasingly important practical applications in cryptography, and are also useful in hashing, besides being of fundamental importance in number theory.

The primes library ('`primes.el`') is a small collection of functions for:

- testing integers for primality,
- generating nearby primes,
- finding the n-th prime,
- generating lists of primes in a given range,
- factoring a number into a product of primes,
- finding the greatest common divisor of two numbers, and
- finding the least common multiple of two numbers.

The modest collection of functions implemented in the library is likely to grow, and perhaps may even be improved algorithmically. The core of these functions is the primality test, `(prime-p n)`, whose running time is *O(sqrt(n))*, which becomes excessive for large n.

Note that *sqrt(n) == 2^((lg n)/2)*, where *lg n*, the base-2 logarithm of n, is the number of bits in n. Thus *O(sqrt(n))* means *O(2^(bits in n))*, or *O(10^(digits in n))*. That is, the running time increases **exponentially** in the number of digits of n.

Because knowledge of the cost of these functions may be critical to the caller, each function's documentation string ends with a bracketed cost estimate as a final paragraph.

From the time of the Greek Eratosthenes (ca. 276–195 BCE) to 1974, the only known provably-correct algorithm for testing primality and factoring arbitrary integers was the Sieve of Eratosthenes, which involves the brute force trying of all possible divisors.

Without additional storage to record primes already found or known, this means that *O(sqrt(n))* divisions are needed to test an integer n for primality. [Actually, that the divisions could stop early, at *sqrt(n)*, was only discovered by Fibonacci (a.k.a. Leonardo Pisano) about 1200 CE.]

Faster algorithms capable of dealing with larger integers are known. For example, Maple V Release 5 (1997) implements a probabilistic function, `isprime(n)`, that is

> "very probably" prime — see Knuth "The Art of Computer Programming", Vol 2, 2nd edition, Section 4.5.4, Algorithm P [Addison-Wesley, Reading, MA, 1969, ISBN 0-201-03802-1] for a reference and H. Reisel, "Prime numbers and computer methods for factorization" [Birkhäuser, Boston, 1994, 2nd edition, ISBN 0-8176-3291-3] . No counter example is known and it has been conjectured that such a counter example must be hundreds of digits long.

The goal of the function implementations in the first release of the primes library is to provide demonstrably correct, small, and straightforward, GNU Emacs Lisp code for them, using pre-1974 algorithms. They should not be expected to be speedy for large arguments.

Even if faster algorithms were implemented, their applicability would be limited, because GNU Emacs Lisp does not provide a big integer type. And finally, the significant computational resources needed to apply these algorithms to big integers means that a compiled, rather than byte-code interpreted, implementation language is essential.

Obviously, since the functions have integer arguments, and the function result is always the same for a given argument, they could all be implemented by a fast *O(1)* table lookup operation, except that the storage required would be *O(n)*, which is unacceptably large.

Perhaps a future version of the library might offer a limited table-lookup implementation, reverting to computation for numbers beyond the tabulated range. Preliminary experiments show that the primality test could thus be sped up by as much as a factor of five, factorization by a factor of three, and the expensive *n*-th prime computation reduced to a simple table lookup for common cases (there are only 9592 primes less than 100,000, and 78,498 primes less than 1,000,000).

As a measure of the programming complexity of recent improved algorithms, the body of the primality test function in this library is only 12 lines of code, and the factoring function is 15 lines of code, while similar functions (and several others) in the PARI library (a collaborative project to implement high-quality fast algorithms in number theory) amount to about 3600 lines of code embedded in a library of more than 97,000 lines.

1.1 Prime Number Functions

To use the primes library in your own code, simply include this line near the beginning of your GNU Emacs Lisp file:

```
(require 'primes)
```

The functions provided by the GNU Emacs primes library all take integer arguments; invalid arguments provoke a silent `nil` return value.

gcd *m n* Function

Return the *greatest common divisor* of integers *m* and *n*, or `nil` if they are invalid.

Example: `(gcd 1024 768)` returns `256`.

[cost: *O((12(ln 2)/pi^2)ln max(m,n)) == 0.8427659... max(m,n)*]

lcm *m n* Function

Return the *least common multiple* of integers *m* and *n*, or `nil` if they are invalid, or the result is not representable (e.g., the product *m*n* overflows).

Example: `(lcm 1024 768)` returns `3072`.

[cost: *O((12(ln 2)/pi^2)ln max(m,n)) == 0.8427659... max(m,n)*]

prime-factors *n* Function

Return a list of prime factors of *n*.

If *n* is prime, there are no factors, except the trivial one of *n* itself, so the return value is the list (*n*). Thus, if `(length (prime-factors n))` is 1, *n* is prime.

Otherwise, if *n* is not an integer greater than 1, the return value is `nil`, equivalent to an empty list.

Example: `(prime-factors 1023)` returns `(3 11 31)`.

[cost: *O(n)*]

next-prime *n* Function

Return the next prime number after *n*.

Example: `(next-prime 9)` returns `11`.

[cost: *O(sqrt(n))*]

nth-prime *n* Function

Return the *n*-th prime, where the first prime is 2.

Example: `(nth-prime 100)` returns `541`.

[cost: *O(n*sqrt(n))*]

prev-prime *n* Function

Return the previous prime, the largest one less than *n*.

Example: `(prev-prime 7)` returns `5`, and `(prev-prime 2)` returns `nil`.

[cost: *O(sqrt(n))*]

prime-p *n* Function

Test whether *n* is prime, and return *n* if so, and otherwise, *nil*.

Example: `(prime-p 117)` returns `nil`.

[cost: *O(sqrt(n))*]

primes-between *from to* Function

Return a list of primes in the range (*from*, *to*), inclusive.

Example: `(primes-between 0 10)` produces `(2 3 5 7)`.

[cost: *O((to - from + 1)*sqrt(n)/2)*]

this-or-next-prime *n* Function

Return *n* if it is prime, else return the next prime number after *n*.

Example: `(this-or-next-prime 7)` returns `7`.

[cost: *O(sqrt(n))*]

this-or-prev-prime *n* Function

Return *n* if it is prime, else return the prime number before (i.e., less than) *n*.

Example: `(this-or-prev-prime 7)` returns `7`.

[cost: *O(sqrt(n))*]

Because Emacs integers are usually more limited in size than the host word size would suggest, e.g.,

```
[-2^27, 2^27 - 1] == [-134217728, 134217727]
```

on a 32-bit machine, avoid passing excessively large integers to these functions, otherwise you may experience a failure like this one:

```
(next-prime 134217689)
Arithmetic domain error: "sqrt", -134217728.0
```

While you may be able to use larger integers on some 64-bit machines, the required run time for these functions is then likely to be excessive.

The `lcm` function is particularly sensitive to overflow, since it is computed from the relation *lcm(m,n) = (m*n)/gcd(m,n)*: the intermediate product *(m*n)* can overflow for values as small as *2^14*, even if the final result would be representable. Consequently, `lcm` is written to use double-precision floating-point arithmetic until the final division is completed. Even this will fail for values near the overflow limit, such as *2^27 - 1 - 2^25 = 100663295*, and worse, the failure will not be detected: a non-`nil` incorrect answer will be returned. This blemish needs to be remedied in a future version of this library.

To complete this section, it is instructive to examine how certain special cases are handled in two important functions. Recall first the important definition that began this chapter:

> *A prime number is any integer greater than one that has no exact integer divisors other than one and itself.*

Here is a table of results of primality testing from recent releases of several important algebra programming systems, and this package:

		Primality Testing						
Program	Function	-10	-2	-1	0	1	2	10
Maple V5	isprime	false	false	false	false	false	true	false
Matlab 5.2.1.1420	isprime	0	0	0	0	0	1	0
Mathematica 2.2	PrimeQ	False	True	False	False	False	True	False
Reduce 3.6	primep	nil	(list)	nil	nil	nil	(list)	nil
primes.el	prime-p	nil	nil	nil	nil	nil	2	nil

Mathematica and Reduce incorrectly ignore the argument sign, reporting that -2 is a prime.

Reduce returns a list of the first 500 primes instead of `t`, but the two are equivalent for logical tests, so that behavior is acceptable, if perhaps unexpected.

Maple, Matlab, and this package are consistent with the standard definition of a prime number.

This package's `prime-p` function returns its argument when it is prime, because that is more useful than just `t`, and yet can still be treated equivalently in logical tests.

Here is how they handle factorization:

		Factorization						
Program	Function	-10	-2	-1	0	1	2	10
Maple V5	ifactor	-2,5	-2	-1	0	1	2	2,5

```
Maple V5          ifactors      -1(2,5) -1(2,1) -1()  0() 1() 1(2,1) 1(2,5)
Matlab 5.2.1.1420 factor        ERROR   ERROR   ERROR 0   1   2      2,5
Mathematica 2.2   FactorInteger -1,2,5  -1,2    -1,1  0   ()  2      2,5
Reduce 3.6        factorize     2,5,-1  2,-1    1,-1  ()  1   2      2,5
primes.el         prime-factors nil     nil     nil   nil nil 2      2,5
================================================================================
```

Maple has two related functions: according to help inside the program, `ifactor` returns the complete integer factorization of its integer argument, and `ifactors` returns the complete integer factorization of its integer or fractional argument. However, the Maple Handbook which accompanies the package claims that `ifactors` returns the *prime* integer factors.

Matlab handles arguments 0 and 1 anomalously, and raises an uncatchable error for negative arguments, aborting processing. By contrast, the consistent `nil` return from the Emacs `prime-factors` function for invalid arguments makes it possible to handle the exception gracefully.

Reduce takes the absolute value of the argument, then for negative arguments, appends an additional factor of -1. Argument 1 is handled anomalously.

Like Reduce, Mathematica takes the absolute value of the argument, and then, for negative arguments, prefixes an additional factor of -1. However its handling of arguments -1 and 1 is inconsistent, and the handling of arguments 0 and 1 is anomalous.

Evidently, all of those other packages could profitably reexamine their prime number support for consistency, correctness, and usability!

1.2 Testing and Profiling the Primes Library

The primes library file, '`primes.el`', is accompanied by a thorough test package, '`test-primes.el`', a practice that we hope other GNU Emacs Lisp package writers will follow.

The test package contains a test function for each public function in the primes library, plus two driver programs that the (human) tester can invoke interactively with the usual `M-x` prefix, and two interfaces to those functions, to be used in batch mode as part of an automated package validation test:

test-primes Function

Run all of the validation tests. The test output log is stored in a buffer named '`*test-primes*`' (or whatever you have set *test-primes-buffer* to); an existing buffer of that name is made unique by addition of a numeric suffix. If all of the tests are successful, the buffer just contains a list of the test names, something like this:

```
There should be no output here other than the test names

test of gcd ...
test of lcm ...
test of prime-p ...
test of next-prime ...
```

```
test of nth-prime ...
test of prev-prime ...
test of primes-between ...
test of this-or-next-prime ...
test of this-or-prev-prime ...
```

The tests are hierarchically ordered, since, for example, the primality test is needed in all of the other functions.

Any errors detected would appear following the corresponding '`test of ...`' line; there should be none.

test-primes-with-profile Function

Run `test-primes` with function profiling turned on. This produces the normal test log in the '`*test-primes*`' buffer, and in addition, produces a second temporary buffer, '`*profile*`' (or whatever you have set *profile-buffer* to), to hold the run-time profile showing counts and execution times for each function profiled. An existing buffer of that name is made unique by addition of a numeric suffix. A fragment of the profile looks something like this (slightly reformatted to reduce line width):

```
Function                  Calls  Total time (sec)  Avg time
                                                   per call
========================  ======  ================  =========
gcd                         5304         73.327100  0.013825
lcm                         2754         46.438376  0.016862
next-prime                    11          0.462194  0.042018
nth-prime                      5         15.425060  3.085012
prev-prime                    11          0.539797  0.049072
prime-factors                242          3.093837  0.012784
prime-p                     4984          9.984730  0.002003
primes-between                 4          0.186052  0.046513
this-or-next-prime            10          0.353138  0.035314
this-or-prev-prime            10          0.371206  0.037121
...
Profile by decreasing average time
Function                  Calls  Total time (sec)  Avg time
                                                   per call
========================  ======  ================  =========
nth-prime                      5         15.425060  3.085012
prev-prime                    11          0.539797  0.049072
primes-between                 4          0.186052  0.046513
next-prime                    11          0.462194  0.042018
this-or-prev-prime            10          0.371206  0.037121
this-or-next-prime            10          0.353138  0.035314
lcm                         2754         46.438376  0.016862
gcd                         5304         73.327100  0.013825
prime-factors                242          3.093837  0.012784
prime-p                     4984          9.984730  0.002003
...
```

The first page of the buffer contains a summary of the environment in which the test was run, so that the user can readily distinguish profiles run on different systems.

The second page contains profile data with the function names in alphabetical order. The third, and last, page, contains profile results sorted by descending cost.

test-primes-and-kill-emacs Function

This function is a wrapper for `test-primes`, except that it saves the test results in a file, and exits Emacs with a status code indicating the number of test failures.

test-primes-with-profile-and-kill-emacs Function

This function is a wrapper for `test-primes-with-profile`, except that it saves the test results and profile in files, and exits Emacs with a status code indicating the number of test failures.

For the latter two functions, the filenames chosen are of the form

```
test-primes.results.HOSTNAME.YYYY-MM-DD-hh-mm-ss
test-primes.profile.HOSTNAME.YYYY-MM-DD-hh-mm-ss
```

so that tests can be run on multiple machines without filename collisions, and the test results can readily be distinguished by the filenames.

Not only does profiling reveal hot spots in the code, but non-zero function invocation counts also verify that each function has been exercised by the tests.

The exact results of a profile clearly depend on test data, on the compiler and optimization level used to build Emacs, on algorithms in the Emacs kernel, on the operating system, on the timer granularity, and on the host architecture.

Nevertheless, this table of relative performance (larger is slower), sorted by function names on the left, and by decreasing relative cost on the right, may be a useful guide. It was produced on a late 1995-vintage Sun UltraSPARC 170 workstation with Sun Solaris 2.6 running GNU Emacs 20.3.6 at the package author's site, and all Emacs code was byte-compiled:

`gcd`	6.90	`nth-prime`	1540.20
`lcm`	8.42	`prev-prime`	24.50
`next-prime`	20.98	`primes-between`	23.22
`nth-prime`	1540.20	`next-prime`	20.98
`prev-prime`	24.50	`this-or-prev-prime`	18.53
`prime-factors`	6.38	`this-or-next-prime`	17.63
`prime-p`	1.00	`lcm`	8.42
`primes-between`	23.22	`gcd`	6.90
`this-or-next-prime`	17.63	`prime-factors`	6.38
`this-or-prev-prime`	18.53	`prime-p`	1.00

The execution time of `nth-prime` depends on its argument: the largest value passed by the test program was 1000.

The functions in the primes library depend heavily on integer arithmetic, and it is worth observing that some RISC architectures lack a full complement of integer instructions, sometimes relegating multiply and divide to software implementations. Older Sun SPARC systems, and all HP PA-RISC systems, are widely-used examples. Some supercomputers handle integer multiply and divide in floating-point hardware, necessitating a conversion from integer to floating-point and back.

The primes library author's site has systems from seven major UNIX vendors representing more than ten different UNIX architectures, and about four times as many models. Until the development of this library, GNU Emacs was normally built on these systems with vendor compilers using default optimizations. However, the primes library profiling turned up unexpected anomalies, with some architectures being notably slower than others, when such differences were not expected from other benchmarks.

As an experiment, therefore, Emacs was rebuilt on the Sun SPARC systems with a high optimization level and options to generate code for the latest architecture versions. The profiles showed a dramatic improvement: overall speedups by factors of 5 to 11, depending on the model, and speedups of up to 24 on the `gcd` test. Serendipitously, the largest speedups were seen on the oldest and slowest models, whose users most need the performance increase.

On Sun SPARC systems, it is possible to use this optimized Emacs on all models, because unknown hardware instructions met by an older model are silently trapped and emulated in software. That may not be possible on some other systems.

Similar rebuilds with optimization were carried out on the other architectures at the development site, and speedups of as little as 1.1, to as much as 16, were obtained. Evidently, for compute-bound functions, compiler optimizations of Emacs can be extremely profitable!

1.3 Background Reading

For an interesting historical review of number theory, and a list of outstanding unsolved problems, see Leonard M. Adleman, *Algorithmic Number Theory — The Complexity Contribution*, Proc. 35th IEEE Symposium on the Foundations of Computer Science (FOCS'94), Shafi Goldwasser (Ed.), IEEE Computer Society Press (Silver Spring, MD), pp. 88–113, 1994, ISBN 0-8186-6582-3, ISSN 0272-5428.

For more detail, and recommended computational algorithms, see the book by Eric Bach and Jeffrey Shallit, *Algorithmic Number Theory. Volume I: Efficient Algorithms*, MIT Press (Cambridge, MA), 1996, ISBN 0-262-02405-5.

For even more detail on, and complexity analysis of, the older methods, see Donald E. Knuth, *Seminumerical algorithms, The Art of Computer Programming, Volume 2*, Third edition, Addison-Wesley (Reading, MA), 1997, ISBN 0-201-89684-2.

The book by Steven S. Skiena, *The Algorithm Design Manual*, Springer-Verlag (New York, NY), 1998, ISBN 0-387-94860-0, contains in Section 8.2.8 only a brief overview of the factoring and primality testing problem, but it has pointers to important recent literature, and to excellent freely-available software packages (including the aforementioned PARI system); that practice is continued throughout the book, making it an outstanding reference volume for combinatorial algorithms.