PRNG

Generating random numbers
Version: 3.0.1
Date: 12 March 2001

Otmar Lendl
Josef Leydold

Table of Contents

PRNG – Pseudo-Random Number Generator

PRNG is a collection of algorithms for generating pseudorandom numbers as a library of C functions, released under the GPL (`http://www.gnu.org/copyleft/gpl.html`). It has been written by Otmar Lendl (`lendl@cosy.sbg.ac.at`) and is now maintained by Josef Leydold (`leydold@statistik.wu-wien.ac.at`).

The current version of this package can always be found on the ARVAG (Automatic Random VAriate Generation) project group (`http://statistik.wu-wien.ac.at/arvag/`) in Vienna, or the pLab server (`http://random.mat.sbg.ac.at/`) in Salzburg.

In the case of any troubles, bug reports or need of assistance please contact the maintainer via `prng@statistik.wu-wien.ac.at`. Please let us also know about your experiencies with the library.

Features

- Portability. This library should compile on any computer with an ANSI C compiler. A verification program is included.
- General Implementations. This library does not implement certain fixed generators like RANDU or `rand`, but implements the general PRNG algorithms to which all parameters can be supplied by the user.
- Consistent and object-oriented interface. This interface simplifies the PRNG handling inside the main application.
- Possibility of independent copies of the same generator.
- Extensibility. New generators are easily integrated into the framework of this library.
- Fully supported Pseudorandom number generating methods: (free parametrization)
  - LCG (linear congruential generator)
  - ICG (inversive congruential generator)
  - EICG (explicit inversive congruential generator)
  - mEICG (modified explicit inversive congruential generator)
  - DICG (digital inversive congruential generator)
  - QCG (quadratic congruential generator)

  Fixed parameter PRNG (external generators):
  - MT19937 (Mersenne Twister by M. Matsumoto)
  - TT800 (a large TSFR by M. Matsumoto)
  - CTG (Combined Tausworthe Generator by P. L'Ecuyer)
  - MRG (Multiple Recursive Generator by P. L'Ecuyer)
  - CMRG (Combined (Multiple Recursive Generator by P. L'Ecuyer)

  plus the following methods (meta-generators):
  - C (Compound generator)
  - SUB (Subsequences)
  - ANTI (antithetic random variables)
  - CON (Consecutive blocks)
  - AFILE (Ascii file)
  - BFILE (Binary file)

1 Installing PRNG

While the code is plain ANSI C and thus quite portable, the following adaptions might be neccessary for compile this library.

All configurations are done in the file '`src/prng.h`'. Each option is extensively commented there. Here is a quick rundown on what to expect there:

- Definition of the basic numeric data-type `prng_num`. It is not recommended to change this. For 32 and 64 bit computers all neccessary auxiliary definitions will be made automatically. For other architectures, please edit '`prng.h`' according to the comments.
- Various constants. See comments on the exact meanings.
- Definition of `prng_inverse`. In previous versions, there was no algorithm which was fastest on all architectures, thus is was necessary configure the library for the each platform. Now `prng_inverse_own`, which combines the speedups of all old algorithms is the fastest one on all tested architectures and thus *no* configuration is necessary any more.

The code is optimized for GNU CC (gcc). If your compiler supports the type (`long long int`), too, you can use this feature by defining `HAVE_LONGLONG` in '`prng.h`'.

Then do:

```
./configure --prefix=<prefix_path>
make
```

This should compile the library ('`libprng.a`') and example programs.

To install the library (see also GNU generic installation instructions in file '`INSTALL`') type:

```
make install
```

which installs '<prefix_path>/lib/libprng.a', '<prefix_path>/include/prng.h', and '<prefix_path>/info/prng.info'. If `--prefix` is omitted, then `/usr/local` is used as default.

It is possible to remove these files by

```
make uninstall
```

I could not test this code in many environments, thus it might be necessary to tweak the code to compile it. Please mail me any changes you made, so that I can include them in the next official release.

Documentation

A manual can be found in directory '`doc`' in various formats, including PS, PDF, HTML, Info and plain text.

Profiling and Verification

Do

```
make check
```

to make and run the following executables:

- `iter`
  This program counts the number of iterations in the `euclid_table` algorithm. It's NOT kept up to date. Use at own risk.
- `validate`
  Using the supplied file tests.dat, this program tests the generator library for correct operation. On 32-bit computers it will fail on generators requiring 64-bit arithmetic.

2 Usage of PRNG

2.1 Interface Description

The interface has changed dramatically in version 2.0. As more and more generator types were added to this package, a new generic interface was needed. While still plain Ansi C, the architecture is now object-oriented.

All generators are identified by a textual description. This description is either of the form "type(parameter1,parameter2, ...)" or is a shortcut name for a common PRNG as defined in 'src/prng_def.h'.

Calling prng_new with such a description as the only argument will allocate a new generator object, initialize it, and return its handle (struct prng *).

All further calls need this handle as the first argument. They are best explained by example:

```
#include <prng.h>   /* make sure that the compiler can find this file. */

main()
{
   struct prng *g;
   prng_num seed, n, M;
   double next, *array;
   int count;

   g = prng_new("eicg(2147483647,111,1,0)");

   if (g == NULL) /* always check whether prng_new has been successful */
   {
      fprintf(stderr,"Initialisation of generator failed.\n");
      exit (-1);
   }

   printf("Short name: %s\n",prng_short_name(g));
                               /* definition as in call to prng_new */
   printf("Expanded name: %s\n",prng_long_name(g));
                               /* Shortcuts expanded                */

   next = prng_get_next(g);       /* get next number 0 <= next < 1     */
   prng_get_array(g,array,count); /* fill array with count numbers     */
   prng_reset(g);                 /* reset the generator */
   prng_free(g);                  /* deallocate the generator object   */

}
```

These functions work with all generators. For certain generators, the following functions are available, too:

```
if (prng_is_congruential(g))
{
   n = prng_get_next_int(g);       /* return next *unscaled* number    */
   M = prng_get_modulus(g);        /* return the modulus of the prng   */
}
```

```
if (prng_can_seed(g))
   prng_seed(g,seed);              /* reseed the generator             */

if (prng_can_fast_sub(g))
   puts(prng_get_sub_def(g,20,0)); /* Get subsequence definition       */

if (prng_can_fast_con(g))
   puts(prng_get_con_def(g,20,1)); /* Get block definition             */
```

NOTE:
`prng_new` performs only a rudimentary check on the parameters. The user is responsible for enforcing all restrictions on the parameters, such as checking that the modulus of an [E]ICG is prime, or that LCG and ICG are maximum period generators.

Most of these functions are implemented as macros, so be careful with autoincrements (`++`) in parameters.

2.2 PRNG Functions

`struct prng` **prng_new** (`char *`*str*) Library Function

Create a new generator object. If initialisation of the generator object fails then `NULL` is returned. Thus the pointer returned by this routine **must** be checked against `NULL` **before** using it. Otherwise the program aborts with a segmentation fault.

`void` **prng_reset(struct** `prng *`*g*) Library Function

Reset random number generator.

`double` **prng_get_next** (`struct prng *`*g*) Library Function

Sample from generator (get next pseudo-random number from stream).

`void` **prng_get_array** (`struct prng *`*g*, `double *`*array*, `int` *count*) Library Function

Sample array of length *count*.

`prng_num` **prng_get_next_int** (`struct prng *`*g*) Library Function

Sample integer random number from generator.

`void` **prng_free** (`struct prng *`*g*) Library Function

Destroy generator object.

`char*` **prng_short_name** (`struct prng *`*g*) Library Function

Get name of generator as in call to `prng_new`.

`char*` **prng_long_name** (`struct prng *`*g*) Library Function

Get name of generator with shortcuts expanded.

`int` **prng_is_congruential** (`struct prng *`*g*) Library Function

`TRUE` if *g* is a congruential generator.

`prng_num` **prng_get_modulus** (`struct prng` **g*) Library Function

Return modulus of generator.

`int` **prng_can_seed** (`struct prng` **g*) Library Function

`TRUE` if generator *g* can be reseeded.

`void` **prng_seed** (`struct prng` **g*, `prng_num` *next*) Library Function

Reseed generator.

`int` **prng_can_fast_sub** (`struct prng` **g*) Library Function

`TRUE` if subsequences of the random stream can computed directly.

`char*` **prng_get_sub_def** (`struct prng` **g*, `int` *s*, `int` *i*) Library Function

Get definition for the generator of the subsequence stream of *g* with starting point *i* and stepwidth *s*. It returns a character string that can be used a argument for `prng_new`. For generators where `prng_can_fast_sub` is `TRUE`. (see also Section 3.2.10 [SUB], page 10).

`int` **prng_can_fast_con** (`struct prng` **g*) Library Function

`TRUE` if blocks of the random stream can computed directly.

`int` **prng_get_con_def** (`struct prng` **g*, `int` *l*, `int` *i*) Library Function

Get definition for the generator of the blocked stream of *g* with position *i* and block length *l*. It returns a character string that can be used a argument for `prng_new`. For generators where `prng_can_fast_con` is `TRUE`. (see also Section 3.2.12 [CON], page 10).

2.3 Examples

'`examples/pairs.c`' is an example how to generate overlapping pairs of PRN using this package.

'`examples/tuples.c`' is a more general version of pairs.

3 The Theory behind PRNG

This chapter lists the implemented generators plus a few recommendations on the parameters.

3.1 General Remarks

- On a b-bit computer, the size of the modulus is limited by 2^{b-1}, that is 2147483648 on a 32 bit machine or 9223372036854775808 on a 64 bit architecture. As of version 1.3 the library will reject larger moduli.
- The library relies on controlled overflow. If you feel uncomfortable with that, restrict your choice of moduli to numbers $< 2^{b-2}$,

  and disable the check for power of two moduli in `mult_mod_setup` (‘`support.c`’). Run the supplied validate program if you have doubts about this.
- The library does **NOT** test if the parameters are valid for the chosen generator. The user is responsible for ensuring that the modulus of an inversive generator is a prime, or that the choice of parameters will lead to an optimal period length.

  IT IS THUS NOT A GOOD IDEA TO JUST USE ARBITRARY NUMBERS.

  This chapter contains recommended values for all implemented generator types.
- Do not base your simulation on a single generator. Even if you picked a good one you should verify the results using a completely different generator. There is no generator whose output does not exhibit an intrinsic structure, so it is in theory possible that this structure correlates to the simulation problem and thus leads to a skewed result. Do not use just other parameters for the verification but use a different generator type.
- Small (< 32767) factors will be faster than larger ones.

3.2 Generator Definitions and Parameters

TeX notation is used.

Most generators operate in the group (or field) Z_p and generate a sequence y_n, n >= 0 of numbers in Z_p. p is called modulus. In order to generate U([0,1[) distributed numbers, the y_n are scaled: x_n = y_n / p.

Notice: If p is prime, one can define the inversion inv() so that

```
inv(a)*a mod p = 1   (a != 0)
inv(0) = 0
```

Generator types

3.2.1 EICG (explicit inversive congruential generator)

- Definition:

  ```
  y_n = inv(a*(n_0 + n) + b) (mod p)      n >= 0
  ```
- Name (as given to `prng_new`): `"eicg(p,a,b,n_0)"`
- Properties:
  - Period length = p.
  - Strong non-linear properties. (e.g. no lattice)
  - Parameter selection not sensitive.
  - `prng_is_congruential` is `TRUE`

  - `prng_can_seed` is `TRUE`.
    The parameter of `prng_seed` will be used as `"n"` in the next call to `get_next`.
  - `prng_can_fast_sub` and `prng_can_fast_con` are `TRUE`.
- Parameter selection: Besides a != 0, no restrictions or even suggestions are known.
- Introduced in: Eichenauer-Hermann, J. "Statistical independence of a new class of inversive congruential pseudorandom numbers", Math. Comp. 60:375-384, 1993

3.2.2 ICG (inversive congruential generator)

- Definition:

```
y_n = a * inv(y_{n-1}) + b (mod p)     n > 0
```

- Name (as given to `prng_new`): `"icg(p,a,b,y_0)"`
- Properties:
  - Period length = p. (for suitable parameters)
  - Strong non-linear properties. (e.g. no lattice)
  - Parameter selection not sensitive.
  - `prng_is_congruential` is `TRUE`.
  - `prng_can_seed` is `TRUE`.
    The parameter of `prng_seed` will be used as `y_{n-1}` in the next call to `get_next`.
  - `prng_can_fast_sub` and `prng_can_fast_con` are `FALSE`.
- Parameter selection: To ensure that the period length is p, a and b must be chosen in a way that x^2 - bx -a (\in F_p[x]) is a primitive polynomial over F_p.

  If ICG(p,a,1) has period length p, then ICG(p,a*c^2,c) will have period length p, too. For recommended parameters see Section A.2 [Table_ICG], page 17.
- Introduced in: Eichenauer, J. and J. Lehn. "A non-linear congruential pseudo random number generator", Stat. Papers 27:315-326, 1986

3.2.3 LCG (linear congruential generator)

- Definition:

```
y_n = a * y_{n-1} + b (mod p)      n > 0
```

- Name (as given to `prng_new`): `"lcg(p,a,b,y_0)"`.
- Properties:
  - Period lengths up to p are possible.
  - Strong linear properties.
  - The quality of the PRN depends very strongly on the choice of the parameters.
  - `prng_is_congruential` is `TRUE`.
  - `prng_can_seed` is `TRUE`. The parameter of `prng_seed` will be used as `y_{n-1}` in the next call to `get_next`.
  - `prng_can_fast_sub` and `prng_can_fast_con` are `TRUE`.
    Requesting these subsequence may be slow if large skips are involved and b is not 0.
- Parameter selection: If p is a power of 2, then a mod 4 = 1 and b odd will guarantee period length = p.

  If p is prime and b = 0 then any prime-root modulo p as a will guarantee period length p-1. (y_0 != 0)

  For recommended parameters see Section A.1 [Table_LCG], page 12.

  See also the file '`src/prng_def.h`' for a list of frequently used LCGs.

Hint: A rule of thumb suggests not to use more than sqrt(p) random numbers from an LCG.

References:

Fishman, G.S. "Multiplicative congruential random number generators ..." Math. Comp. 54:331-344 (1990);

L'Ecuyer, P., "Efficient and portable combined random number generators" Comm. ACM 31:742-749, 774 (1988)

L'Ecuyer, P., Blouin, F. and Couture R. "A search for good multiple recursive random number generators" ACM Trans. Modelling and Computer Simulation 3:87-98 (1993)

- Introduced by D. H. Lehmer in 1948.

  The LCG is *the* classical method. I refer to: Knuth, D. E. "The Art of Computer Programming, Vol. 2 Seminumerical Algorithms", Addison-Wesley, second edition, 1981

3.2.4 QCG (quadratic congruential generator)

- Definition:

  ```
  y_n = a * y_{n-1}^2 + b *y_{n-1} + c (mod p)      n > 0
  ```
- Name (as given to `prng_new`): `"qcg(p,a,b,c,y0)"`.
- Properties:
  - Period lengths up to p are possible.
  - Weaker linear properties (tuples fall into union of lattices)
  - Reasonable distribution in dimension 2, but not that good in dimension 3.
  - `prng_is_congruential` is `TRUE`.
  - `prng_can_seed` is `TRUE`.
    The parameter of `prng_seed` will be used as `y_{n-1}` in the next call to `get_next`.
  - `prng_can_fast_sub` and `prng_can_fast_con` are `FALSE`.
- Parameter selection:

  If p is a power of 2, then a even, b == a + 1 mod 4, and c odd will guarantee period length = p.

  No table of good parameters has been published.
- Introduced in:

  Knuth, D. E. "The Art of Computer Programming, Vol. 2 Seminumerical Algorithms", Addison-Wesley, second edition, 1981

3.2.5 MT19937 (Mersenne Twister)

- Name (as given to `prng_new`): `"mt19937(seed)"`.
- Properties:
  - Period lengths is 2^19937-1.
  - `prng_is_congruential` is `TRUE`.
  - `prng_can_seed` is `TRUE`.
    The parameter of `prng_seed` will be used to seed the array of coefficients.
  - `prng_can_fast_sub` and `prng_can_fast_con` are `FALSE`.
- Introduced in:

  Matsumoto, M. and Nishimura, T., "Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator", ACM Transactions on Modeling and Computer Simulation, Vol. 8, No. 1, January 1998, pp 3–30.

3.2.6 MEICG (modified explicit inversive congruential generator)

- Definition:

  ```
  y_n = n * inv(a*(n_0 + n) + b) (mod p)     n >= 0
  ```
- Name (as given to `prng_new`): `"meicg(p,a,b,n_0)"`.
- Properties:
  - Period length = p.
  - `prng_is_congruential` is `TRUE`.
  - `prng_can_seed` is `TRUE`. The parameter of `prng_seed` will be used as "n" in the next call to `get_next`.
  - `prng_can_fast_sub` and `prng_can_fast_con` are `FALSE`.

  Experimental generator: **USE AT OWN RISK**
- Parameter selection:
  - For prime moduli, a != 0 suffices.
  - It's possible to use a powe of 2 as modulus, which requires a = 2 (mod 4) and b = 1 (mod 2).
- Introduced in:

  Eichenauer-Hermann, J. "Modified explicit inverive congruential pseudorandom numbers with power of 2 modulus" Statistics and Computing 6:31-36 (1996)

3.2.7 DICG (digital inversive congruential generator)

- Definition:

  ```
  y_n = a * inv(y_{n-1}) + b (mod p)      n > 0
  ```

  All operations are in the field `F_{2^k}` !!
- Name (as given to `prng_new`): `"dicg(k,a,b,y_0)"`.
- Properties:
  - Period length = 2^k.
  - Strong non-linear properties.
  - Parameters seem not to be sensitive
  - `prng_is_congruential` is `TRUE`.
  - `prng_can_seed` is `TRUE`. The parameter of `prng_seed` will be used as `y_{n-1}` in the next call to `get_next`
- `prng_can_fast_sub` and `prng_can_fast_con` are `FALSE`.
- Parameter selection:

  Tricky.
- Introduced in:

  Eichenauer-Herrmann and Niederreiter, "Digital inversive pseudorandom numbers", ACM Transactions on Modeling and Computer Simulation, 4:339-349 (1994)

3.2.8 EXTERNAL (Interface to fixed-parameter generators)

These generators are included to provide a uniform interface to a wider range of PRNG. The only enhancements from the published code is the support for multiple streams of these generators, as the original code used global variables.

See the file '`src/external.c`' for the references. Included are

- TT800 (a large TSFR by M. Matsumoto)

- CTG (Combined Tausworthe Generator by P. L'Ecuyer)
- MRG (Multiple Recursive Generator by P. L'Ecuyer)
- CMRG (Combined (Multiple Recursive Generator by P. L'Ecuyer)

3.2.9 COMPOUND

- Definition:

  This is a "meta"-generator which combines a number of PRNG into one single generator by adding the respective numbers modulo 1.
- Name (as given to `prng_new`): `"c(generator1,generator2, ...)"`.

  Up to `PRNG_MAX_COMPOUNDS` generators are permitted. *generatorX* may be any valid generator definition, including a compound generator.
- Properties:
  - Period length: Least common multiple of the period length's of the component generators.
  - Generally speaking, most properties of PRNG are preserved if combining generators of the same type.
  - `prng_is_congruential` is `FALSE`.
  - `prng_can_seed` is `TRUE`.
  - The parameters of `prng_seed` is used to seed all seedable component generators.
  - `prng_can_fast_sub` and `prng_can_fast_con` depend on the underlying generators.

3.2.10 SUB

- Definition: This is a "meta"-generator which takes a subsequence out of another generator.
- Name (as given to `prng_new`): `"sub(gen,s,i)"`.

  The output of "gen" is spliced into s streams, and the i-th is used. (0 <= i < s)
- Properties:
  - Period length: Typically the period of "gen".
  - Generally speaking, most properties of PRNG are preserved when taking subsequence.
  - `prng_is_congruential` and `prng_can_seed` depend on "gen".
  - `prng_can_fast_sub` and `prng_can_fast_con` are `FALSE`.

3.2.11 ANTI

- Definition:

  This is a "meta"-generator wich returns 1-U instead of U as random number.
- Name (as given to `prng_new`): `"anti(gen)"`.

  The output of gen (U) is changed to 1-U.

3.2.12 CON

- Definition:

  This is a "meta"-generator which takes a block of numbers out of the output of another generator.
- Name (as given to `prng_new`): `"con(gen,l,i)"`.

  The output of "gen" is divided into blocks of length l, and the i-th is used. (0 <= i < l)

- Properties:
  - Period length: The period of "gen".
  - prng_is_congruential and prng_can_seed depend on "gen".
  - prng_can_fast_sub and prng_can_fast_con are FALSE.

3.2.13 AFILE (ASCII file)

- Definition :

  This generator takes its numbers from the named file. It expects each number in plain ascii (atof must be able to parse it) and on its own line. If EOF is reached, a warning is printed to stderr and reading continues at the beginning of the file.
- Name (as given to prng_new): "afile(some_file_name)".
- Properties:
  - prng_is_congruential is FALSE.
  - prng_can_seed is FALSE.
  - prng_can_fast_sub and prng_can_fast_con are FALSE.

3.2.14 BFILE (Binary file)

- Definition:

  This generator takes its numbers from the named file. In order to get good numbers, the file should contain random bytes. If EOF is reached, a warning is printed to stderr and reading continues at the beginning of the file.
- Name (as given to prng_new): "bfile(some_file_name)".

  WARNING: The conversion between bytes and numbers in [0,1) is NOT guaranteed to yield the same results on different computers.
- Properties:
  - prng_is_congruential is FALSE.
  - prng_can_seed is FALSE.
  - prng_can_fast_sub and prng_can_fast_con are FALSE.

3.3 Recommended Reading

Niederreiter, H. "New developments in uniform pseudorandom number and vector generation" in "Monte Carlo and Quasi-Monte Carlo Methods in Scientific Computing", Lecture Notes in Statistics, Springer.

Hellekalek, P. "Inversive pseudorandom number generators: Concepts, Results and Links"

Eichenauer-Herrmann, J. "Pseudorandom Number Generation by Nonlinear Methods" Int. Statistical Review 63:247-255 (1995)

L'Ecuyer, P. "Uniform random number generation" Ann. Oper. Res. 53:77-120 (1994)

Wegenkittl, S. "Empirical testing of pseudorandom number generators" Master's thesis, Universitaet Salzburg, 1995

Appendix A Tables of Parameters

This chapter lists the implemented generators plus a few recommendations on the parameters.

A.1 Parameters for LCG (linear congruential generators)

```
y_n = a * y_{n-1} + b (mod p)      n > 0
```

Hint: A rule of thumb suggests not to use more than sqrt(p) random numbers from an LCG.

Notice that moduli larger than 2^32 require a computer with `sizeof(long)>32`.

Generators recommended by Park and Miller (1988), "Random number generators: good ones are hard to find", Comm. ACM 31, pp. 1192-1201 (Minimal standard).

modul p		multiplicator a
2^31 - 1 =	2147483647	16807 (b = 0)

Generators recommended by Fishman (1990), "Multiplicative congruential random number generators with modulus 2^β: An exhaustive analysis for $\beta = 32$ and a partial analysis for $\beta = 48$", Math. Comp. 54, pp. 331-344.

modul p		multiplicator a
2^31 - 1 =	2147483647	950706376 (b = 0)

Generators recommended by L'Ecuyer (1999), "Tables of linear congruential generators of different sizes and good lattice structure", Math.Comp. 68, pp. 249-260. (constant b = 0.)

Generators with short periods can be used for generating *quasi-random numbers* (*Quasi-Monte Carlo methods*). In this case the *whole* period should be used.

(These figures are listed without warranty. Please see also the original paper.)

modul p		multiplicator a
2^8 - 5 =	251	33
		55
2^9 - 3 =	509	25
		110
		273
		349
2^10 - 3 =	1021	65
		331

2^11 - 9 =	2039	995 328 393
2^12 - 3 =	4093	209 235 219 3551
2^13 - 1 =	8191	884 1716 2685
2^14 - 3 =	16381	572 3007 665 12957
2^15 - 19 =	32749	219 1944 9515 22661
2^16 - 15 =	65521	17364 33285 2469
2^17 - 1 =	131071	43165 29223 29803
2^18 - 5 =	262139	92717 21876
2^19 - 1 =	524287	283741 37698 155411
2^20 - 3 =	1048573	380985 604211 100768 947805 22202 1026371
2^21 - 9 =	2097143	360889 1043187 1939807
2^22 - 3 =	4194301	914334 2788150 1731287 2463014

2^23 - 15 =	8388593	653276 3219358 1706325 6682268 422527 7966066
2^24 - 3 =	16777213	6423135 7050296 4408741 12368472 931724 15845489
2^25 - 39 =	33554393	25907312 12836191 28133808 25612572 31693768
2^26 - 5 =	67108859	26590841 19552116 66117721
2^27 - 39 =	134217689	45576512 63826429 3162696
2^28 - 57 =	268435399	246049789 140853223 29908911 104122896
2^29 - 3 =	536870909	520332806 530877178
2^30 - 35 =	1073741789	771645345 295397169 921746065
2^31 - 1 =	2147483647	1583458089 784588716
2^32 - 5 =	4294967291	1588635695 1223106847 279470273
2^33 - 9 =	8589934583	7425194315 2278442619 7312638624
2^34 - 41 =	17179869143	5295517759 473186378

2^35 - 31 =	34359738337	3124199165 22277574834 8094871968
2^36 - 5 =	68719476731	49865143810 45453986995
2^37 - 25 =	137438953447	76886758244 2996735870 85876534675
2^38 - 45 =	274877906899	17838542566 101262352583 24271817484
2^39 - 7 =	549755813881	61992693052 486583348513 541240737696
2^40 - 87 =	1099511627689	1038914804222 88718554611 937333352873
2^41 - 21 =	2199023255531	140245111714 416480024109 1319743354064
2^42 - 11 =	4398046511093	2214813540776 2928603677866 92644101553
2^43 - 57 =	8796093022151	4928052325348 4204926164974 3663455557440
2^44 - 17 =	17592186044399	6307617245999 11394954323348 949305806524
2^45 - 55 =	35184372088777	25933916233908 18586042069168 20827157855185
2^46 - 21 =	70368744177643	63975993200055 15721062042478 31895852118078
2^47 - 115 =	140737488355213	72624924005429 47912952719020 106090059835221
2^48 - 59 =	281474976710597	49235258628958 51699608632694 59279420901007

2^49 - 81 =	562949953421231	265609885904224 480567615612976 305898857643681
2^50 - 27 =	1125899906842597	1087141320185010 157252724901243 791038363307311
2^51 - 129 =	2251799813685119	349044191547257 277678575478219 486848186921772
2^52 - 47 =	4503599627370449	4359287924442956 3622689089018661 711667642880185
2^53 - 111 =	9007199254740881	2082839274626558 4179081713689027 5667072534355537
2^54 - 33 =	18014398509481951	9131148267933071 3819217137918427 11676603717543485
2^55 - 55 =	36028797018963913	33266544676670489 19708881949174686 32075972421209701
2^56 - 5 =	72057594037927931	4595551687825993 26093644409268278 4595551687828611
2^57 - 13 =	144115188075855859	75953708294752990 95424006161758065 133686472073660397
2^58 - 27 =	288230376151711717	101565695086122187 163847936876980536 206638310974457555
2^59 - 55 =	576460752303423433	346764851511064641 124795884580648576 573223409952553925
2^60 - 93 =	1152921504606846883	561860773102413563 439138238526007932 734022639675925522
2^61 - 1 =	2305843009213693951	1351750484049952003 1070922063159934167 1267205010812451270
2^62 - 57 =	4611686018427387847	2774243619903564593

		431334713195186118
		2192641879660214934
2^63 - 25 =	9223372036854775783	4645906587823291368
		2551091334535185398
		4373305567859904186
2^64 - 59 =	18446744073709551557	13891176665706064842
		2227057010910366687
		18263440312458789471

A.2 Parameters for ICG (inversive congruential generator)

```
y_n = a * inv(y_{n-1}) + b (mod p)     n > 0
```

Notice that moduli larger than 2^32 require a computer with `sizeof(long)>32`.

Parameters suggested by P. Hellekalek (1995), “Inversive pseudorandom number generators: Concepts, Results and Links”, in: C. Alexopoulos, K. Kang, W.R. Lilegdon, and D. Goldsman (eds.), Proceedings of the 1995 Winter Simulation Conference, pp. 255-262:

There are no results that give reason to prefer one set of parameters over another.

(These figures are listed without warranty. Please see also the original paper.)

p	a	b
1031	849	1
	345	1
	55	1
	116	1
	441	1
1033	413	1
	878	1
	595	1
	522	1
	818	1
1039	173	1
	481	1
	769	1
	1028	1
	136	1
2027	579	1
	1877	1
	390	1
	837	1
	1048	1
2147483053	858993221	1

	22211	11926380
	579	24456079
	11972	62187060
	21714	94901263
	4594	44183289
2147483647	1288490188	1
	9102	36884165
	14288	758634
	21916	71499791
	28933	59217914
	31152	48897674