Good Practice in (Pseudo) Random Number Generation for Bioinformatics Applications

David Jones, UCL Bioinformatics Group

(E-mail: d.jones@cs.ucl.ac.uk)

(Last revised May 7th 2010)

This is a very quick guide to what you should do if you need to generate random numbers in your bioinformatics code. Random numbers are being used more and more in computational biology (e.g. for bootstrapping tests or stochastic simulations of biological systems) and unlike computational physicists, for example, many bioinformaticians don't seem to appreciate the potential pitfalls in generating random numbers (this is apparent from looking at many popular software packages in the field). Hey, look - there's a standard random number function in Perl – surely that must be a decent choice?

Wrong!

The field of pseudo random number generation is huge and complex (and the field of finding faults in random number generators is probably even larger). This short text is not a definitive guide to the field by any stretch of the imagination, but just a set of pointers to ensure that you don't fall into the most obvious traps. It's worth pointing out that many applications tolerate poor quality random number generators quite well. In sequence shuffling experiments for example, reasonable statistics can be obtained with even quite bad generators. Nevertheless, these days, there is simply no reason to use anything but excellent RNGs even in simple applications, and there are just three simple rules to follow.

Rule 1

Do not use system generators. The easiest (laziest) option is to make use the standard library "rand" function e.g. those found in the standard C library, or the standard Perl rand for example. Almost all of these generators are badly flawed. Even when they are not, there is no guarantee that they were not flawed in earlier releases of the library (e.g. see note on Python below) or will not be flawed in future releases. ALWAYS USE YOUR OWN RANDOM NUMBER GENERATOR. That way you can ensure your code is portable and you can try different RNGs if you suspect the one you are using is causing a problem.

Note that all of these standard generators have been shown to have serious defects:

Standard Perl rand

Python random() (versions before V2.3; V2.3 and above are OK)

Java.util.Random

C-library rand(), random() and drand48()

Matlab's rand

Mathematica's SWB generator

ran0() and ran1() in the original Numerical Recipes book

This is not an exhaustive list by any means, but if you are using any of the above for serious work – STOP NOW and think about finding a better generator!

Rule 2

Use a good RNG and build it into your code. From software alone, it is impossible to generate truly random numbers – unless there is a fault in your computer. As the numbers are generated by an algorithm, they are by definition NON-random. What these algorithms generate are PSEUDO-random numbers. The practical definition of pseudo randomness is that the numbers should not be distinguishable from a source of true random numbers in a given application. So one generator may be good enough for one application, but fail badly in another application. True random numbers should not fail in any applications.

There are libraries of tests that can be applied to RNGs – these identify obvious flaws in generators e.g. the classic C rand() function will generate alternately odd and even numbers – clearly non-random behaviour and this, along with many other issues, causes it to fail many of the standard tests. This paper by L'Ecuyer and Simard describes the intensive testing of many different RNGs:

http://www.iro.umontreal.ca/~lecuyer/myftp/papers/testu01.pdf

If you look at that paper you will see how many well-known RNGs fail one or more of the simple tests. The standard Java RNG, for example, is "defective" in 21 different ways!

Another very useful set of tests is the Diehard test suite developed by G. Marsaglia, which have been recently updated and extended by R.G. Brown as the Dieharder test suite:

http://www.phy.duke.edu/~rgb/General/dieharder.php

Probably the simplest RNG to pass all of the above tests (and many others) is the KISS generator proposed by G. Marsaglia (a slightly older version is called KISS99 in the L'Ecuyer paper):

```
static unsigned int x = 123456789,y = 362436000,z = 521288629,c = 7654321; /* Seed variables
*/
unsigned int KISS()

{
   unsigned long long t, a = 698769069ULL;

   x = 69069*x+12345;
   y ^= (y<<13); y ^= (y>>17); y ^= (y<<5); /* y must never be set to zero! */
   t = a*z+c; c = (t>>32); /* Also avoid setting z=c=0! */

   return x+y+(z=t);
}
```

This remarkably short and fast generator combines 3 different simple generators and has a period of around 10^{37} i.e. it will start repeating the same sequence of numbers after that many calls. Plenty

for pretty much any conceivable bioinformatics application, but there are other generators which have much longer periods (see later). Note that you need to set the seed values (at least x, y and z need to be changed) to random starting values else you will always generate the same sequence of numbers in your program (see below). Avoid setting the seeds to zero or small numbers in general – try to choose large "complex" seed values (see discussion below on warming up RNGs).

A nice consequence of combining different RNGs is that a statistical flaw in any one of the component generators is likely to be covered up by the other generators. Combining different RNGs is now considered sound practice in designing good RNGs by many experts in the field. I would propose the KISS RNG as representing the minimum acceptable standard in random number generation.

IMPORTANT – the use of "unsigned long long" in the above code is vital for the multiply-with-carry component of the function (HINT from reader comments: "long long" can be replaced by "_int64" in some compilers) – the standard version of KISS depends on unsigned 64-bit arithmetic (easy for modern C compilers).

By following the basic design principles, many different KISS-like generators can be constructed. My own KISS generator is shown below. It slightly improves upon the original by tuning the component generators so that when any one of the three component generators is excluded, the remaining two will still pass all of the Dieharder tests. The period of JKISS is $\approx 2^{127} = 1.7 \times 10^{38}$ (2^{32} x (2^{32}-1) x (1/2 * 4294584393 x 2^{32} - 1)) and it passes all of the Dieharder tests and the complete BigCrunch test set in TestU01.

```
/* Public domain code for JKISS RNG */
static unsigned int x = 123456789,y = 987654321,z = 43219876,c = 6543217; /* Seed variables */

unsigned int JKISS()
{
   unsigned long long t;

   x = 314527869 * x + 1234567;
   y ^= y << 5; y ^= y >> 7; y ^= y << 22;
   t = 4294584393ULL * z + c; c = t >> 32; z = t;

   return x + y + z;
}
```

If your language only handles 32-bit integers or even has difficulties with unsigned integers (e.g. Fortran), or you just want a faster generator that avoids multiplication, Marsaglia has proposed using an *add*-with-carry generator rather than the superior multiply-with-carry. Although I prefer to use the full multiplicative KISS, I do really like this 32-bit generator because it is the simplest and fastest RNG that I know of which passes all of the Dieharder tests and the BigCrunch tests in TestU01. The period is $\approx 2^{121} = 2.6 \times 10^{36}$.

```
/* Implementation of a 32-bit KISS generator which uses no multiply instructions */
static unsigned int x=123456789,y=234567891,z=345678912,w=456789123,c=0;

unsigned int JKISS32()
{
  int t;

  y ^= (y<<5); y ^= (y>>7); y ^= (y<<22);
  t = z+w+c; z = w; c = t < 0; w = t&2147483647;
  x += 1411392427;

  return x + y + w;
}
```

A frequent question that gets asked is which is the *best* random number generator. It's obvious that there can be no such thing, as different applications may need specific algorithms. For example, an algorithm which makes use of a large array of integers will not be very useful if you are writing code for specialised hardware with little available RAM. Nevertheless, it's clear that the RNG that many people believe to be the state-of-the-art at the present time (2007) for simulation work is the so-called “Mersenne Twister” (MT):

http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html

In my view this generator is a bit over-elaborate for day-to-day use, and it does fail 2 of the Crunch tests, but the tests it fails are linear complexity tests which pure shift-register-type generators like MT are doomed to fail (and are unlikely to be an issue in a simulation). It’s hard to imagine any real bioinformatics application failing with this generator, however, and it has a HUGE period (2^{19937}-1). It has also been ported to many languages, and is the default RNG in R (and *recent* releases of Python). Nevertheless, you can still say that it does fail some empirical tests and so I still generally prefer the much simpler KISS generators, which pass all of the empirical tests I've come across (please do let me know if you find a valid statistical test that KISS fails convincingly) and need just 10 lines of code. However, if you need a very long period RNG or just want state-of-the-art, then by all means use the Mersenne Twister. I suspect that nobody was ever fired for choosing to use MT in their code!

Note that new RNGs which have even better theoretical properties than MT have been created by Panneton, L’Ecuyer and Matsumoto (one of the authors of MT). The WELL RNGs may ultimately take the “best RNG” crown from MT. See here for more details:

http://www.iro.umontreal.ca/~panneton/WELLRNG.html

Don't forget, that for cryptographic work, most standard RNGs (including MT) are worthless - but if you are working in cryptography or security then hopefully you already know everything about secure random number generation!

Rule 3

Properly seed your generator. Even the state-of-the-art Mersenne Twister ran into problems early on because the authors had neglected the issue of proper seeding. Even now, correctly seeding MT is a non-trivial problem (MT effectively has 624 integer seed variables which need to be properly initialised at the start!). This rule is VITAL if you are going to run parallel simulations on a Beowulf cluster for example.

The simplest way to seed a RNG is to take something like the current time e.g. using the time() function found in Unix and most C libraries, which returns a 32-bit integer giving the number of seconds since 1st Jan 1970. Early versions of Perl used this idea for example. What’s wrong with this? Well, it’s more or less acceptable if you are running a program once a day – but think of what happens when you submit 1000 jobs to a Beowulf cluster in one batch. Hundreds of jobs on different nodes will be starting at almost exactly the same time – therefore many of your jobs will be starting with exactly the same seed and therefore those that have the same seed will generate exactly the same results (assuming your code has no bugs in it). Not only is it a waste of CPU time to repeat the exact same calculation many times, but if you don’t know it has happened, any statistics you carry

out on the final data will be biased by the repetitions. Think of molecular dynamics as a good example – hundreds of simulations could be running – all generating exactly the same trajectory because the RNGs were unknowingly started in exactly the same state.

Note that even if you manage to come up with a way of generating truly random 32-bit seed values you are still not necessarily safe. Remember the old Birthday Paradox i.e. how many people do you need in a room for it to be more likely than not that at least two of them have the same birthday? Remember the answer is surprisingly small (n=23). The same thing applies to random number seeds. If you generate just 2^{16} (=65536) random 32-bit seeds you would expect there to be at least one repetition. So if you submit many thousands of jobs to a cluster even with completely random 32-bit seeds you run a risk that two or more of the simulations will start with identical seeds.

To avoid this, you firstly need more state bits in your RNG i.e. your RNG has to use seeds summing to at least 64-bits of information to avoid repeating the same seed (and to reduce the chance of the sequences of random numbers overlapping in two simulations). In the case of the KISS generator above, if you randomise the 3 main seed variables (x,y,z) you will get 96 bits of seed in total. But how do you set these values initially?

You could use a combination of time of day and the microsecond system clock (both from gettimeofday()) plus say process ID (getpid()) and host ID (gethostid() - to avoid generating the same seeds on two different compute nodes) or any other varying system values you can think of. Some of these values won't change much from run to run and so the Birthday Paradox may still catch you out, but the above values are probably good enough for many applications. Don't fall into the trap of just combining these variables into a single 32-bit seed (e.g. by adding them together) - you need to use them separately to form 64 or more bits of seed information.

Another simple trick is to pipe some system data that frequently changes into md5sum e.g.:

```
ps -ef | md5sum
```

In many operating systems today, however, there is an ideal solution available:

```
/dev/random
```

If you read 4 bytes from this device then you will get a nicely random and unpredictable 32-bit seed value, and you can read as many bytes as you need. This device uses system hardware timings (e.g. hard disk access times and intervals between key presses) to produce entropy and is as close as you will get to a source of true random numbers without a radioactive source and a Geiger counter!

Note that /dev/random will block until it thinks it has accumulated enough entropy. This could delay your program by several seconds or even minutes, so you will probably be better off using **/dev/urandom** instead. This always returns random data immediately, but with a risk that the numbers might be more “predictable” than from /dev/random. Unless you are into cryptography this shouldn’t worry you, and the results from /dev/urandom are still going to be much better than from any of the simpler methods described above. The /dev/urandom device could even be used as the sole source of random numbers in your program, but is probably best used just to set the seeds for something like the KISS generator as reading from these devices can be slow on some systems.

For Windows systems, it is possible to use the Cryptographic Application Programming Interface to generate secure random numbers in place of /dev/random). The simplest option if using Microsoft C++ is to use the rand_s() library function (no relation to the infamous rand() function - careful you don't mistype!) to generate good quality secure random numbers for seeding purposes.

As a footnote to this, it's worth pointing out that another simple solution to the seeding problem is to allow the seed values to be set externally e.g. through a command-line parameter or configuration file. It is then the responsibility of the user to ensure that a unique seed is used every time the program is run. A danger here is that the user could specify small low complexity seed values e.g. 1, 2, 3, 4 etc. To counter this, the input seed values should be randomised e.g. using a secure hash such as MD5, or if the seeds are used directly, a long "warm up" of the generator should be carried out first (see later).

Example routine to return random 32-bit integers from /dev/urandom and seed KISS:

```
unsigned int devrand(void)
{
    int fn;
    unsigned int r;

    fn = open("/dev/urandom", O_RDONLY);
    if (fn == -1)
        exit(-1); /* Failed! */

    if (read(fn, &r, 4) != 4)
       exit(-1); /* Failed! */

    close(fn);

    return r;
}

/* Initialise KISS generator using /dev/urandom */
void init_KISS()
{
   x = devrand();
   while (!(y = devrand())); /* y must not be zero! */
   z = devrand();

   /* We don't really need to set c as well but let's anyway… */
   /* NOTE: offset c by 1 to avoid z=c=0 */
   c = devrand() % 698769068 + 1; /* Should be less than 698769069 */
}
```

Miscellaneous Tips

Generating random floating point numbers

The following C code generate a random (double precision) floating point number 0 <= x < 1:

```
double x;

x = JKISS() / 4294967296.0;
```

Works fine – and in fact will often produce reasonable results even with somewhat defective random number generators. However, 32-bits are not enough to generate all possible double precision float values between 0 and 1 (more than enough for single precision floats though). This

may not be a problem in your application, but if it is, you will need to call the RNG twice and combine the results to generate a full 53-bit precision double e.g. as follows:

```
double uni_dblflt()
{
        double x;
        unsigned int a, b;

        a = JKISS() >> 6; /* Upper 26 bits */
        b = JKISS() >> 5; /* Upper 27 bits */
        x = (a * 134217728.0 + b) / 9007199254740992.0;

        return x;
}
```

If speed *really* matters, then it is possible to avoid the floating point scaling by directly manipulating the bits that make up the significand (mantissa) of the floating point variable. Unless every cycle counts, this is just not worth the hassle (or loss of portability) in my book. However, here are examples for generating single or double precision floats quickly:

```
/* Quickly generate random single precision float 0<=x<1 by type punning */
float uni_qsingflt()
{
        float x;
        unsigned int a;

        a = JKISS() >> 9; /* Take upper 23 bits */
        *((unsigned int *)&x) = a | 0x3F800000; /* Make a float from bits
*/

        return x-1.0F;
}

/* Quickly generate random double precision float 0<=x<1 by type punning */
double uni_qdblflt()
{
        double x;
        unsigned long long a;

        a = ((unsigned long long)JKISS()<<32) + JKISS();
        a = (a >> 12) | 0x3FF0000000000000ULL; /* Take upper 52 bits */
        *((unsigned long long *)&x) = a; /* Make a double from bits */

        return x-1.0;
}
```

Rather than generating two 32-bit integers and combining them, you could also use a 64-bit RNG and take the top 52 bits (e.g. see the JLKISS64 RNG below). There are also some fairly decent RNGs which work directly with double precision floating point numbers e.g. the uni64 RNG described in Statistics and Probability Letters vol. 66, no. 2, pp183-187 (again by Marsaglia and colleagues).

Generating random integers

With a *good* RNG, you can generate random integers like this:

ri = JKISS() % 10;

This will generate uniformly distributed random integers between 0 and 9 inclusive. Actually, the integers will not be perfectly uniformly distributed unless the divisor is a factor of 2^{32}, but the bias is negligible if the divisor is much smaller than 2^{32} as it is in this example.

If you are stuck with a system generator or some other defective generator – NEVER use this method. Use something like this:

ri = (int)(10.0 * random() / (RAND_MAX + 1));

In other words, convert the 32-bit random integer to a floating point number 0 <= x < 1 and use this to generate your final integer. This way all of the bits are used. If you used the first approach with the old C-library rand() you would generate odd numbers followed by even numbers every time! It's still better to just use a good RNG in the first place (and avoid the floating point arithmetic).

Generating Gaussian Deviates

In some applications, random numbers need to be generated according to the normal distribution rather than uniformly over the range 0.0-1.0. The most commonly used method for this is the Box-Mueller transform, which returns numbers with a mean of 0.0 and standard deviation of 1.0:

```
#include <math.h>

/* Generate gaussian deviate with mean 0 and stdev 1 */
double gaussrnd()
{
    double x, y, r;

    do {
        x = 2.0 * uni_dblflt() - 1.0;
        y = 2.0 * uni_dblflt() - 1.0;
        r = x * x + y * y;
    } while (r == 0.0 || r >= 1.0);

    r = sqrt((-2.0 * log(r)) / r);

    return x * r;
}
```

It should be noted that a much faster (though more elaborate) way of generating Gaussian deviates (the Ziggurat method) has been proposed by Marsaglia & Tsang (see http://www.jstatsoft.org/v05/i08). A Fortran90 implementation can be found here:

http://www.netlib.org/random/ziggurat.f90.

Unbiased Shuffling

Probably the most common need for random numbers in bioinformatics applications is in shuffling e.g. shuffling the amino acids in a protein sequence. There is a correct way to shuffle, called the Fisher-Yates or Knuth shuffle, which ensures that all permutations are sampled uniformly. Here is an example of how to shuffle a string/sequence correctly:

```
/* Perform random shuffle on sequence (string) s */
void            shufseq(char *s, int length)
{
    int             i, r;
    char            temp;

    for (i = length - 1; i >= 1; i--)
    {
        /* 0 <= r <= i */
        r = (double) JKISS() * (i + 1) / 4294967296.0;
        temp = s[i];
        s[i] = s[r];
        s[r] = temp;
    }
}
```

Warm up your generator first

If you are not able to generate good quality random seeds e.g. using /dev/random or if you allow the user to specify the seed values via command-line parameters, then it is often wise to “warm up” your generator before using the random numbers it generates. In other words, set the seeds to your desired starting values and then before your program starts its real work, generate a few hundred or even a few thousand random numbers and discard them (as a rule - the longer the period of the RNG the more initial numbers you should discard at the beginning). This is important if your seed values have very low entropy (or more simply they have too many runs of 0 or 1 bits in their binary representation) – some otherwise good RNGs (especially shift-register-based generators such as MT) are not very random from their initial state when certain seed values are used. The warm up trick is generally good advice for many RNGs, particularly those with very long periods.

Don't use "too many" random numbers

Although it's clear that it's a bad idea to use a longer sequence of random numbers than the period of a given RNG (because the sequence obviously starts repeating), there are arguments in the literature that suggest that it's a bad idea to use more than even a small fraction of the RNG's period in a single simulation. At least a part of this argument is that you ideally want to avoid using overlapping sequences of random numbers from one run to the next, though there is a bit more to it than that. For a RNG with period *p*, Knuth suggests the limit should be *p*/1000, which is not a problem for decent RNGs with periods of say 2^{50} or more. However, Ripley has argued that a safer limit should be $\frac{1}{200}\sqrt{p}$, and there are some suggestions that in extreme cases an even safer limit might be $\sqrt[3]{p}$. Today, a single application running for a month on a very fast computer might

conceivably be able to use 10^{12} random numbers (that's assuming it does little more than generate random numbers!). This suggests that as a worst-case-scenario, a minimum period of at least 2^{120} might be needed for simulations of that length. All of the RNGs mentioned in these notes have sufficient periods. However, for much longer simulations, even though you would probably be wise to use MT or some other super-long-period generator (e.g. see Appendix 1), it's simple to construct a KISS generator with much longer period e.g. $\approx 2^{191}$ with little loss of speed on modern 64-bit CPUs, by upgrading the xor-shift register and linear congruential generators to 64-bit operations:

```
/* Public domain code for JLKISS RNG - long period KISS RNG with 64-bit operations */

unsigned long long x = 123456789123ULL,y = 987654321987ULL; /* Seed variables */
unsigned int z = 43219876,c = 6543217; /* Seed variables */

unsigned int JLKISS()
{
    unsigned long long t;

    x = 14900243430053362 37ULL * x + 123456789;
    y ^= y << 21; y ^= y >> 17; y ^= y << 30; /* Do not set y=0! */
    t = 4294584393ULL * z + c; c = t >> 32; z = t; /* Avoid z=c=0! */

    return (unsigned int)(x>>32) + (unsigned int)y + z; /* Return 32-bit result */
}
```

Although the above code makes use of 64-bit variables, it still produces 32-bit integers. If, for some reason, you want to produce 64-bit random integers directly, then an extra multiply-with-carry generator can be used to produce a full 64-bit result (period this time is $\approx 2^{250}$):

```
/* Public domain code for JLKISS64 RNG - long period KISS RNG producing 64-bit results */

unsigned long long x = 123456789123ULL,y = 987654321987ULL; /* Seed variables */
unsigned int z1 = 43219876, c1 = 6543217, z2 = 21987643, c2 = 1732654; /* Seed variables */

unsigned long long JLKISS64()
{
    unsigned long long t;

    x = 14900243430053362 37ULL * x + 123456789;
    y ^= y << 21; y ^= y >> 17; y ^= y << 30; /* Do not set y=0! */
    t = 4294584393ULL * z1 + c1; c1 = t >> 32; z1 = t;
    t = 4246477509ULL * z2 + c2; c2 = t >> 32; z2 = t;

    return x + y + z1 + ((unsigned long long)z2 << 32); /* Return 64-bit result */
}
```

References

Usenet postings by G. Marsaglia:

http://groups.google.co.uk/group/sci.math.num-analysis/msg/eb4ddde782b17051

http://groups.google.co.uk/group/sci.math/msg/9959175f66dd138f

http://groups.google.co.uk/group/comp.lang.fortran/msg/6edb8ad6ec5421a5

http://groups.google.co.uk/group/sci.math/msg/5d891ca5727b97d2

Ziggurat method:

G. Marsaglia and W.Tsang, The Ziggurat Method for Generating Random Variables. Journal of Statistical Software, vol. 5 (2000), no. 8

Random number testing:

P. L'Ecuyer and R. Simard, `TestU01: A C Library for Empirical Testing of Random Number Generators', ACM Transactions on Mathematical Software, 33, 4, Article 22, August 2007.

G. Marsaglia and W. Tsang, Some Difficult-to-pass Tests of Randomness. Journal of Statistical Software, vol. 7 (2002), no. 3

Other miscellaneous references:

B. Ripley, Journal of Computational and Applied Mathematics Volume 31, Issue 1, 24 July 1990, Pages 153-163.

D. Knuth, The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms. Addison-Wesley Longman Publishing Co., Inc., 1997. ISBN 0-201-89684- 2.

Acknowledgements

Thanks to various readers for e-mailed comments and suggestions for improvements.

Appendix 1

Below are some other short, interesting and good RNGs that have been posted to Usenet (again by the prolific George Marsaglia). I thought I'd append them here, as despite being short and simple RNGs, they have *huge* periods (and pass all of the standard tests). They are based around the multiply-with-carry algorithm, rather than large shift registers, so they are useful alternative methods to compare against MT when you do have a need for a very long period generator. The only difficulty in using these generators (in common with other super-long-period generators) is in proper seeding i.e. how to initialise the large state arrays (Q), which should be pre-filled with random seed values. They can be filled using another RNG such as the KISS generator or using /dev/(u)random. If there are any concerns about the randomness of the starting state, then a warm-up of the generator should be carried out (at least 4 * 256 values should be discarded at the start for MWC256, at least 4 * 4096 discarded for CMWC4096 and 4 * 41790 for SuperKISS).

Firstly, the baby MWC256 achieves a period of $\approx 2^{8222}$ by using a state array Q of 256 32-bit unsigned integers.

```
/* MWC256 from Usenet posting by G. Marsaglia - Period 2^8222 */
static unsigned int Q[256], c=362436;

unsigned int MWC256(void)
{
    unsigned long long t;
    static unsigned char i=255;

    t = 809430660ULL * Q[++i] + c;
    c = (t>>32);

    return (Q[i]=t);
}
```

CMWC4096 achieves a period of $\approx 2^{131086}$ (bigger even than that of MT) by using a state array Q of 4096 32-bit unsigned integers.

```
/* CMWC4096 from Usenet posting by G. Marsaglia - Period 2^131086 */
static unsigned int Q[4096],c=362436;

unsigned int CMWC4096(void)
{
    unsigned long long t;
    unsigned int x;
    static unsigned int i=4095;

    i=(i+1)&4095;
    t=18782ULL*Q[i]+c;
    c=(t>>32);
    x=t+c;
    if (x<c)
    {
        x++;
        c++;
    }

    return (Q[i] = 0xFFFFFFFEU-x);
}
```

Super KISS achieves a period of 54767 x $2^{1337279}$ (vastly bigger than that of MT) by using a state array Q of 41790 32-bit unsigned integers.

```
/* Super KISS based on Usenet posting by G. Marsaglia - Period 54767 * 2^1337279 */
static unsigned int Q[41790],indx=41790,carry=362436,xcng=1236789,xs=521288629;

/* Fill Q array with random unsigned 32-bit ints and return first element */
unsigned int refill()
{
    int i;
    unsigned long long t;

    for (i=0;i<41790;i++)
    {
        t = 7010176ULL * Q[i] + carry;
        carry = (t>>32);
        Q[i] = ~t;
    }
    indx=1;

    return (Q[0]);
}

/* Return 32-bit random integer – calls refill() when needed */
unsigned int SuperKISS()
{
        xcng = 69069 * xcng + 123;

        xs ^= xs<<13;
        xs ^= xs>>17;
        xs ^= xs>>5;

        return (indx<41790 ? Q[indx++] : refill()) + xcng + xs;
}
```