1. Introduction. This is GB_FLIP, the module used by GraphBase programs to generate random numbers.

To use the routines in this file, first call the function *gb_init_rand*(*seed*). Subsequent uses of the macro *gb_next_rand*( ) will then return pseudo-random integers between 0 and $2^{31} - 1$, inclusive.

GraphBase programs are designed to produce identical results on almost all existing computers and operating systems. An improved version of the portable subtractive method recommended in *Seminumerical Algorithms*, Section 3.6, is used to generate random numbers in the routines below. The period length of the generated numbers is $2^{85} - 2^{30}$. The low-order bits of the generated numbers are just as random as the high-order bits.

2. Changes might be needed when these routines are ported to different systems, because the programs have been written to be most efficient on binary computers that use two's complement notation. Almost all modern computers are based on two's complement arithmetic, but if you have a nonconformist machine you might have to revise the code in sections that are listed under 'system dependencies' in the index.

A validation program is provided so that installers can tell if GB_FLIP is working properly. To make the test, simply run `test_flip`.

```
⟨ test_flip.c 2 ⟩ ≡
#include <stdio.h>
#include "gb_flip.h"     /* all users of GB_FLIP should do this */
  int main( )
  { long j;
    gb_init_rand(−314159L);
    if (gb_next_rand( ) ≠ 119318998) {
      fprintf(stderr, "Failure␣on␣the␣first␣try!\n");
      return −1;
    }
    for (j = 1; j ≤ 133; j++) gb_next_rand( );
    if (gb_unif_rand(#55555555L) ≠ 748103812) {
      fprintf(stderr, "Failure␣on␣the␣second␣try!\n");
      return −2;
    }
    fprintf(stderr, "OK,␣the␣gb_flip␣routines␣seem␣to␣work!\n");
    return 0;
  }
```

3. The C code for GB_FLIP doesn't have a main routine; it's just a bunch of subroutines to be incorporated into programs at a higher level via the system loading routine. Here is the general outline of `gb_flip.c`:

```
⟨ Private declarations 4 ⟩
⟨ External declarations 5 ⟩
⟨ External functions 7 ⟩
```

4. The subtractive method. If m is any even number and if the numbers a_0, a_1, ..., a_{54} are not all even, then the numbers generated by the recurrence

$$a_n = (a_{n-55} - a_{n-24}) \bmod m$$

have a period length of at least $2^{55} - 1$, because the residues $a_n \bmod 2$ have a period of this length. Furthermore, the numbers 24 and 55 in this recurrence are sufficiently large that deficiencies in randomness due to the simplicity of the recurrence are negligible in most applications.

Here we take $m = 2^{31}$ so that we get the full set of nonnegative numbers on a 32-bit computer. The recurrence is computed by maintaining an array of 55 values, $A[1] \ldots A[55]$. We also set $A[0] = -1$ to act as a sentinel.

⟨ Private declarations 4 ⟩ ≡
  static long $A[56] = \{-1\}$; /* pseudo-random values */
This code is used in section 3.

5. Every external variable should be declared twice in this CWEB file: once for GB_FLIP itself (the "real" declaration for storage allocation purposes), and once in `gb_flip.h` (for cross-references by GB_FLIP users).

The pointer variable *gb_fptr* should not be mentioned explicitly by user routines. It is made public only for efficiency, so that the *gb_next_rand* macro can access the private A table.

⟨ External declarations 5 ⟩ ≡
  long $*gb\_fptr = A$; /* the next A value to be exported */
This code is used in section 3.

6. The numbers generated by *gb_next_rand*( ) seem to be satisfactory for most purposes, but they do fail a stringent test called the "birthday spacings test," devised by George Marsaglia. [See, for example, *Statistics and Probability Letters* **9** (1990), 35–39.] One way to get numbers that pass the birthday test is to discard half of the values, for example by changing '*gb_flip_cycle*( )' to '(*gb_flip_cycle*( ), *gb_flip_cycle*( ))' in the definition of *gb_next_rand*( ). Users who wish to make such a change should define their own substitute macro.

Incidentally, we hope that optimizing compilers are smart enough to do the right thing with *gb_next_rand*.

#define *gb_next_rand*() (*gb_fptr* ≥ 0 ? *gb_fptr*−− : *gb_flip_cycle*( ))

⟨ `gb_flip.h` 6 ⟩ ≡
#define *gb_next_rand*( ) (*gb_fptr* ≥ 0 ? *gb_fptr*−− : *gb_flip_cycle*( ))
  extern long $*gb\_fptr$; /* the next A value to be used */
  extern long *gb_flip_cycle*( ); /* compute 55 more pseudo-random numbers */
See also sections 11 and 13.

7. The user is not supposed to call *gb_flip_cycle* directly either. It is a routine invoked by the macro *gb_next_rand*( ) when *gb_fptr* points to the negative value in $A[0]$.

The purpose of *gb_flip_cycle* is to do 55 more steps of the basic recurrence, at high speed, and to reset *gb_fptr*.

The nonnegative remainder of $(x - y)$ divided by 2^{31} is computed here by doing a bitwise-and with the constant `#7fffffff`. This technique doesn't work on computers that do not perform two's complement arithmetic. An alternative for such machines is to add the value 2^{30} twice to $(x - y)$, when $(x - y)$ turns out to be negative. Careful calculations are essential because the GraphBase results must be identical on all computer systems.

The sequence of random numbers returned by successive calls of *gb_next_rand*( ) isn't really a_n, a_{n+1}, ..., as defined by the basic recurrence above. Blocks of 55 consecutive values are essentially being "flipped" or "reflected"—output in reverse order—because *gb_next_rand*( ) makes the value of *gb_fptr* decrease instead of increase. But such flips don't make the results any less random.

```
#define mod_diff(x,y) (((x)-(y))&#7fffffff)      /* difference modulo 2^31 */
⟨ External functions 7 ⟩ ≡
  long gb_flip_cycle( )
  { register long *ii,*jj;
    for (ii=&A[1],jj=&A[32]; jj<=&A[55]; ii++,jj++) *ii=mod_diff(*ii,*jj);
    for (jj=&A[1]; ii<=&A[55]; ii++,jj++) *ii=mod_diff(*ii,*jj);
    gb_fptr=&A[54];
    return A[55];
  }
```

See also sections 8 and 12.

This code is used in section 3.

8. Initialization. To get everything going, we use a scheme like that recommended in *Seminumerical Algorithms*, but revised so that the least significant bits of the starting values depend on the entire seed, not just on the seed's least significant bits.

Notice that we jump around in the array by increments of 21, a number that is relatively prime to 55. Repeated skipping by steps of 21 mod 55 keeps the values we're computing spread out as far from each other as possible in the array, since 21, 34, and 55 are consecutive Fibonacci numbers (see the discussion of Fibonacci hashing in Section 6.4 of *Sorting and Searching*). Our initialization mechanism would be rather poor if we didn't do something like that to disperse the values (see *Seminumerical Algorithms*, exercise 3.2.2–2).

```
⟨ External functions 7 ⟩ +≡
  void gb_init_rand(seed)
      long seed;
  { register long i;
    register long prev = seed, next = 1;
    seed = prev = mod_diff(prev, 0);      /* strip off the sign */
    A[55] = prev;
    for (i = 21; i; i = (i + 21) % 55) {
      A[i] = next;
      ⟨ Compute a new next value, based on next, prev, and seed 9 ⟩;
      prev = A[i];
    }
    ⟨ Get the array values "warmed up" 10 ⟩;
  }
```

9. Incidentally, if `test_flip` fails, the person debugging these routines will want to know some of the intermediate numbers computed during initialization. The first nontrivial values calculated by *gb_init_rand* are $A[42] = 2147326568$, $A[8] = 1073977445$, and $A[29] = 536517481$. Once you get those right, the rest should be easy.

An early version of this routine simply said '$seed \gg 1$' instead of making *seed* shift cyclically. This method had an interesting flaw: When the original *seed* was a number of the form $4s + 1$, the first 54 elements $A[1]$, ..., $A[54]$ were set to exactly the same values as when *seed* was $4s + 2$. Therefore one out of every four seed values was effectively being wasted.

```
⟨ Compute a new next value, based on next, prev, and seed 9 ⟩ ≡
  next = mod_diff(prev, next);
  if (seed & 1) seed = #40000000 + (seed ≫ 1);
  else seed ≫= 1;      /* cyclic shift right 1 */
  next = mod_diff(next, seed);
```

This code is used in section 8.

10. After the first 55 values have been computed as a function of *seed*, they aren't random enough for us to start using them right away. For example, we have set $A[21] = 1$ in order to ensure that at least one starting value is an odd number. But once the sequence a_n gets going far enough from its roots, the initial transients become imperceptible. Therefore we call *gb_flip_cycle* five times, effectively skipping past the first 275 elements of the sequence; this has the desired effect. It also initializes *gb_fptr*.

Note: It is possible to express the least significant bit of the generated numbers as a linear combination mod 2 of the 31 bits of *seed* and of the constant 1. For example, the first generated number turns out to be odd if and only if

$$s_{24} + s_{23} + s_{22} + s_{21} + s_{19} + s_{18} + s_{15} + s_{14} + s_{13} + s_{11} + s_{10} + s_8 + s_7 + s_6 + s_2 + s_1 + s_0$$

is odd, when $seed = (s_{31} \ldots s_1 s_0)_2$. We can represent this linear combination conveniently by the hexadecimal number `#01ecedc7`; the `1` stands for s_{24} and the final `7` stands for $s_2 + s_1 + s_0$. The first ten least-significant bits turn out to be respectively `#01ecedc7`, `#dbbdc362`, `#400e0b06`, `#0eb73780`, `#da0d66ae`, `#002b63bc`, `#adb801ed`, `#8077bbbc`, `#803d9db5`, and `#401a0eda` in this notation (using the sign bit to indicate cases when 1 must be added to the sum).

We must admit that these ten 32-bit patterns do not look at all random; the number of `b`'s, `d`'s, and `0`'s is unusually high. (Before the "warmup cycles," the patterns are even more regular.) This phenomenon eventually disappears, however, as the sequence proceeds; and it does not seem to imply any serious deficiency in practice, even at the beginning of the sequence, once we've done the warmup exercises.

⟨ Get the array values "warmed up" 10 ⟩ ≡
```
(void) gb_flip_cycle( );
(void) gb_flip_cycle( );
(void) gb_flip_cycle( );
(void) gb_flip_cycle( );
(void) gb_flip_cycle( );
```
This code is used in section 8.

11. ⟨ gb_flip.h 6 ⟩ +≡
```
extern void gb_init_rand( );
```

12. Uniform integers. Here is a simple routine that produces a uniform integer between 0 and $m-1$, inclusive, when m is any positive integer less than 2^{31}. It avoids the bias toward small values that would occur if we simply calculated *gb_next_rand*( ) % m. (The bias is insignificant when m is small, but it can be serious when m is large. For example, if $m \approx 2^{32}/3$, the simple remainder algorithm would give an answer less than $m/2$ about 2/3 of the time.)

This routine consumes fewer than two random numbers, on the average, for any fixed m.

In the `test_flip` program (*main*), this routine should compute $t = m$, then it should reject the values $r = 2081307921$, 1621414801, and 1469108743 before returning the answer 748103812.

```
#define two_to_the_31  ((unsigned long) #80000000)
⟨ External functions 7 ⟩ +≡
  long gb_unif_rand(m)
       long m;
  { register unsigned long t = two_to_the_31 - (two_to_the_31 % m);
    register long r;
    do {
      r = gb_next_rand( );
    } while (t ≤ (unsigned long) r);
    return r % m;
  }
```

13. ⟨ `gb_flip.h` 6 ⟩ +≡

```
  extern long gb_unif_rand( );
```

14. Index. Here is a list that shows where the identifiers of this program are defined and used.

⟨ Compute a new *next* value, based on *next*, *prev*, and *seed* 9 ⟩ Used in section 8.
⟨ External declarations 5 ⟩ Used in section 3.
⟨ External functions 7, 8, 12 ⟩ Used in section 3.
⟨ Get the array values "warmed up" 10 ⟩ Used in section 8.
⟨ Private declarations 4 ⟩ Used in section 3.
⟨ `gb_flip.h` 6, 11, 13 ⟩
⟨ `test_flip.c` 2 ⟩

August 4, 2025 at 13:45

GB_FLIP

Preliminary work on the Stanford GraphBase project was supported in part by National Science Foundation grant CCR-86-10181.