Rogue random number generator

Jeffrey Goldberg

April 17, 2024

[83]:
```
from collections.abc import Generator
from typing import Optional, NamedTuple
from scipy.stats import binomtest
```

1 Rogue background

Rogue was originally written during 1980–1983 by some of my school mates at UC Santa Cruz for BSD Unix running on, if I recall correctly, was a VAX-11.

The code that I will be referring to in this discussion is from sometime later, and is taken from https://github.com/Davidslv/rogue with the copyright notice including the text

> Copyright (C) 1980-1983, 1985, 1999 Michael Toy, Ken Arnold and Glenn Wichman

The 1985 and 1999 code appear include getting this to run on Windows and use of the Gnu autoconf build tools. All of the code that I will be discussing is almost certainly from what I remember playing when I should have been doing my homework during my first years at university.

2 The PRNG

The rogue PRNG an attempt at linear congruential generator (LCG). These are not appropriate for security or cryptographic purposes, but they are really fast and easy to implement.

Following the notation from Hull and Dobell (1962)

$x_n \equiv ax_{n-1} + c \pmod{m}$

where the n-th output of the RNG is x_n depends on the previous value, x_{n-1}, and the parameters a, c, and m. The x_0 is the initial seed. m is called the modulus, a is the multiplier, and c is the increment. When c isn't zero, this really should be called an *affine* congruential generator instead of linear, but as noted by Steele and Vigna (2021) more than half a century of tradition leaves us with the name *linear* congruential generator.

The Rogue RNG is *similar* to that, but it differs in two ways, one of which is intended. The intended difference is that it makes a stronger distinction between internal state of the RNG, S_n, and the output, X_n. In doing so, solves a particular problem, which we will get to later.

The rogue RNG can be described as

$$\begin{aligned} s_n &\equiv as_{n-1} + c \pmod{m} \\ x_n &= \lfloor s_n / 2^{16} \rfloor \end{aligned}$$

The output of the RNG, x_n, is the 16 most significant bits of the internal state, s_n. The internal state will be a 32 bit integer on a VAX.

The parameters are

- The multiplier: $a = 11109$
- The increment: $c = 13849$
- The modulus is (intended to be) $m = 2^{31} = 2\,147\,483\,648$
- The seed, S_0, is set as the sum of current time (seconds since the beginning of the epoch) and the process ID.

The code itself for the this RNG is a one line macro in the source

```
#define RN (((seed = seed*11109+13849) >> 16) & 0xffff)
```

[85]:
```
a = 11109
c = 13849
m = 2**31
```

In analyzing these parameters we will often want to do things with the factorization of various numbers. See Sage math Factorization for details of the object returned by the `factor()` function, but as a simple introduction, here are some examples of what we will be using.

2.0.1 Sageless

I would like this notebook to run in pure Python without having to use a Sagemath kernal. So I will be including my own (not available on PyPi) `math-utils` package. It has a `factor` function that mimicks SageMath factor well enough for the small numbers that will be used here.

[86]:
```
from math_utils import factor, gcd  # my local package
```

[87]:
```
n = 18
f_n = factor(n)
print(f'Factorization of {n} is {f_n}.')
print(f'Factorization as a list of (prime, exponent) tuples: {list(f_n)}')

# The radical_value of a factorization is the product of each prime factor
# each used only once.
print(f'Radical value of {n} is {f_n.radical_value()}')
```

```
Factorization of 18 is 2 * 3^2.
Factorization as a list of (prime, exponent) tuples: [(2, 1), (3, 2)]
Radical value of 18 is 6
```

Some of the criteria for good parameters for the kind of LCG aimed at here are that

1. m and c are relatively prime,
2. $a \equiv 1 \pmod{p}$ if p is a prime factor of m;
3. $a \equiv 1 \pmod{4}$ if 4 is a factor of m;
4. $m > 0$, $c \not\equiv 0 \pmod{m}$;
5. Beyond what is needed to satisfy the above conditions there are no extra common factors between a and m.

6. $a \not\equiv 1 \pmod{m}$.

The Hull-Dobell Theorem states that if conditions 1 through 4 are met then the internal state of the generator will cycle through m possibilities. The other conditions improve the apparent randomness of the generator.

We can look at the factorizations of our m, $a-1$, and c to see by inspection that these conditions hold. This is made particulary easy by the fact that m is simply a power of 2 and that the parameters have been crafted to meet these conditions.

Failing to meet all of these conditions means that the generator is not good, but meeting all of the conditions not not guarantee that it is good. It only guarentees that it doesn't fail in some known ways.

2.1 m and c relatively prime

We could simply compute the greatest common divisor (GCD) of m and c and check that it is 1. A GCD of 1 would mean that the two numbers are relatively prime.

[88]:
```
f_m = factor(m)
print(f'factorization of "m" ({m}) is {f_m}')
```

```
factorization of "m" (2147483648) is 2^31
```

Of course when m is a power of 2 it only has 2s as prime factors. We really don't need to factor it to know that 2 is its only prime factor. As c is an odd number, so we know that it does not have 2 among its prime factors. And so we only need to see that c is an odd number to know that c and m are relatively prime.

For the record, the factorization of c is

[89]:
```
f_c = factor(c)
print(f'factorization of "c" ({c}) is {f_c}')
```

```
factorization of "c" (13849) is 11 * 1259
```

2.2 $a-1$ divisible by all unique prime factors of m

Note that saying that $a \equiv 1 \pmod{p}$ is (almost) the same as saying that $a-1$ is divisible by p. I will switch back and forth between the two forms as I see fit.

It is important to note that this condition only needs $a-1$ to be divisible by each prime factor of m only once. For example, suppose that m is divisible by 9 we do not need $a-1$ to be divisible by 9, we just need it to be divisible by 3. That is, $a-1$ only needs to be divisible by 3 once even if 3 occurs multiple times in the factorization of m. So if some prime p appears at least once in the factorization of m, we only need a - 1 to be divisible by a single p.

Another way of saying this (which will be explained in a bit more detail later) is that we want $a-1$ to be divisible by the radical value of m.

In our case the only prime factor we have for m is 2. We have it a whole bunch of times (31 times in our case), but it is the only prime factor. So we merely need a to leave a remainder of 1 when divided by 2.

We see that a is odd, so we know it leaves a remainder of 1 when divided by 2, and thus this condition holds in our case.

As we will use it later, it will be useful to have the factorization of $a - 1$.

[90]:
```
a_dim = a - 1
f_adim = factor(a_dim)
print(f'Factorization of {a} - 1 is {f_adim}.')
```

```
Factorization of 11109 - 1 is 2^2 * 2777.
```

2.3 Rule of 4

The third condition says that if m is divisible by 4 then $a - 1$ must also be divisible by 4. It places no conditions on a if m is not divisible by 4.

Our m is very definitely divisible by 4. 2^{31} is going to be divisible by 2^2, so we do need to check on $a - 1$.

We've already factored $a - 1$ and we see that it has two 2s among its prime factors, so it is divisible by 4.

2.4 Simple conditions on m and c

The condition that $c \not\equiv 1 \pmod{m}$ actually follows from the first condition that $\gcd(c, m) = 1$, as the greated common divisor is not defined for an argument of 0.

The condition that m is positive (almost) follows from implications of notation, and it really just simplies the statements of the other conditions. I have explicitly added these conditions because they are quick to check and make the other checks easier.

2.5 No extra common factors

It's easier to describe this 5th condition with a couple of examples.

If, say, m is divisible by 9 we actually want $a - 1$ to be divisible by 3 due to the second condition, but we don't want $a - 1$ to be divisible by 9. That is need $a - 1$ to be divisible by 3 but only once. So if m' were something like $2 \cdot 3^{20}$ then we could need $a - 1$ to be divisible by 2 exactly once and divisible by 3 exactly once. So an $a' = 2 \cdot 3 \cdot 17^4 + 1$ would be fine. But an $a'' = 2 \cdot 3^2 \cdot 17^4 + 1$ would not be.

Our previous condition, the rule of 4, is the one execption to this condition about extra factors. When m is divisible by 4 we have an exception. When this rule is combined with the rule of 4, it says that if we have at *least two* 2s as factors of m we must have *exactly* two 2s as factors of $a - 1$.

This rule is not needed to ensure that there are m possible states of the generator, but instead is used to reduce certain patterns in the output of the generator.

We see from the factorization of $a - 1$ that it has exactly two 2s, which is what we need in this case. There are no other distinct prime factors of m so we only have to check the number of 2s in the factorization of $a - 1$. This condition is satisfied for our parameters.

2.6 a is not 1 mod m

If a were equivalent to 1, then the there would be no apparent randomness contributed by the multiplier.

Given that a and m statisfy condition the "each prime" condition, $a \equiv 1 \pmod{m}$, this condition is equivalent to stating that m is not square-free. Sometimes it will be easier to check $a \not\equiv 1 \pmod{m}$ and other times it may be easier to check that m is not sqaure-free.

Saying that m is not square free, requires that at least one of the non-prime factors of m is a perfect square. m is divisible by 4, which is a perfect square, and so this condition is satisfied. It would also be statisfied for our example of $m' = 2 \cdot 3^{20}$. This is because m' would be divisible by 3^2.

2.7 checking more generally

By fairly simple inspection, we could see that all of the conditions hold for the parameters given, but it would be nice to be able to test for other cases. That is what the code below does.

[91]:
```
class Conditions:
    descriptons = {
        'SIMPLE': "simple value conditions",
        'COPRIME': "rel prime",
        'EACH_PRIME': "each factor",
        'RULE4': "rule of 4",
        'NO_EXTRA': "no extra",
        'SQUARE_LADEN': "not square-free"
    }
    full_set: set[str] = set(descriptons.keys())

    def __init__(self) -> None:
        self.value = set()


    def add(self, c) -> None:
        if c not in self.full_set:
            raise ValueError(f'Unknown condition "{c}"')
        self.value |= {c}

    def contains(self, c) -> bool:
        return c in self.value

    @classmethod
    def all_conds(cls) -> set[str]:
        return cls.full_set

    def all(self) -> bool:
        return self.value == self.full_set

    def items(self) -> list[tuple[str, bool]]:
```

```
        r: list[tuple[str,bool]] = []
        for c in self.full_set:
            r.append((c, c in self.value))
        return r
```

[92]:
```
# Hull-Dobell theorem conditions (and a few others)
hd_conditions = Conditions()

# simple conditions
if m > 0 and (c % m) != 0:
     hd_conditions.add('SIMPLE')
print("c and m "
     + ("do " if hd_conditions.contains('SIMPLE') else "do not ")
     +  "meet simple value conditions")

# m and c are relatively prime
if gcd(m, c) == 1:
     hd_conditions.add('COPRIME')
print("c and m "
    + ("are " if hd_conditions.contains('COPRIME') else "are not ")
    + "relatively prime")
```

```
c and m do meet simple value conditions
c and m are relatively prime
```

We will be using the radical value of a factorization fairly often, so it is worth going over what it is and why it is useful for us.

If we take a number like 18, whose factorization is $2 \cdot 3^2$ we see that its prime factors are 2 and 3, with 3 used twice. The radicals of 18 are simply 2 and 3, which ignores the fact that 3 is used twice to get 18. The radical value is the product of the radicals.

So the radical value of 18 is 6. It is just the product of each of the prime factors used exactly once.

If we want $a-1$ to be divisible by each of the prime factors of m, then we want $a-1$ to be divisible by the radical value of m. That is what we do below.

[93]:
```
# $a - 1$ is divisible by a single instance of each prime factors of $m$
if (a - 1) % f_m.radical_value() == 0:
        hd_conditions.add('EACH_PRIME')
print("a - 1 "
        + ("is " if hd_conditions.contains('EACH_PRIME') else "is not " )
        + "divisible by all prime factors of m")
```

```
a - 1 is divisible by all prime factors of m
```

The rule of 4 holds true either if m is not divisible by 4 or if $a-1$ is divisible by 4.

[94]:
```
if  m % 4 != 0 or (a - 1) % 4 == 0:
        hd_conditions.add('RULE4')
print("a - 1 "
        + ("is " if hd_conditions.contains('RULE4') else "is not " )
        + "divisible by 4 if m is divisible by 4")
```

```
a - 1 is divisible by 4 if m is divisible by 4
```

If conditions 2 and 3 are met, then we can check condition 4 first by reducing the modulus and a - 1 by the factors used to satisfy those conditions. Then we check that the reduced m and reduced a are relatively prime.

[95]:
```
# already computed above
# f_m = factor(m)
# f_adim = factor(a-1)

print(f'factorization of m: {f_m}')
print(f'factorization of a - 1: {f_adim}')

# We will produce a reduced a - 1 by dividing each by the unique
# prime factors of m. This operation depends on condition 2 being met
reduced_a = (a - 1) // f_m.radical_value()

if m % 4 == 0: # this step assumes that condition 3 is met
    reduced_a //= 2

f_reduced_a = factor(reduced_a)
print(f'reduced a factorization: {f_reduced_a}')

if gcd(reduced_a, m) == 1:

    hd_conditions.add('NO_EXTRA')
print('a - 1 and m '
        + ('do not ' if hd_conditions.contains('NO_EXTRA') else 'do ')
        + 'share "extra" factors')
```

```
factorization of m: 2^31
factorization of a - 1: 2^2 * 2777
reduced a factorization: 2777
a - 1 and m do not share "extra" factors
```

Now to check that m is not square free. Note that because the other conditions hold we don't have to run a more expensive square_free test, as in our context if m does not have any square factors the consequence will be that $a \equiv 1 \pmod{m}$. (The proof is left as an excercise to the reader.) It is quicker to check for that condition.

[96]:
```
#hd_conditions[COND_SQUARE_LADEN] = not is_squarefree(m)
if a % m != 1:
    hd_conditions.add('SQUARE_LADEN')
```

```
print('m ' + ('is not ' if hd_conditions.contains('SQUARE_LADEN') else 'is ') +␣
 ↪'square free')
```

```
m is not square free
```

[97]:
```
if hd_conditions.all():
    print('all conditions are met')
```

```
all conditions are met
```

We can put all of those checks into a function.

[98]:
```
def dh_lcg_check(m: int, a: int, c: int) -> Conditions:
    # We are going to check simple value conditions and treat
    # them as value errors instead of a condition that fails.
    for name, value in {"m": m, "a": a, "c": c}.items():
        if value == 0:
            raise ValueError(f'"{name}" cannot be zero')

    if m < 0:
        raise ValueError(f'a ({a}) must be positive')

    # If we knew that m would either be prime or a power of primes,
    # we could perform all of these tests more simply (though perhaps)
    # more abstractly) in GF(m). But we can't assume that. So these tests
    # are messy.

    f_m = factor(m)

    conds = Conditions()
    if m > 0 and (c % m) != 0:
        conds.add('SIMPLE')
    if gcd(m, c) == 1:
        conds.add('COPRIME')
    if  a % m != 1:
        conds.add('SQUARE_LADEN')
    if (a - 1) % f_m.radical_value() == 0:
        conds.add('EACH_PRIME')
    if m % 4 != 0 or (a - 1) % 4 == 0:
        conds.add('RULE4')


    # We can only compute the "no extra common factors" test if
    # if we have passed the "each factor" test and the "rule of 4" test.
    #
    # We will consider the test a pass if the conditions for the test
    # are not met
    if not ( conds.contains('EACH_PRIME') and  conds.contains('RULE4')):
        conds.add('NO_EXTRA')
```

```
    else:
        # We test for no extra factors by first dividing a - 1 by
        # the things needed for the other conditions. If that reduced a
        # shares no factors with m there are no extra common factors.
        reduced_a = (a - 1) // f_m.radical_value()
        if m % 4 == 0: # this step assumes that condition 3 is met
            reduced_a /= 2
        if gcd(reduced_a, m) == 1:
            conds.add('NO_EXTRA')

    return conds
```

We can use this to test various LCG parameters that have been used over the ages, staring with rogue's

We are going to name a tuple type for the paramaters for any particularly LGG. 'm', 'a', 'c' are for the modulus, multiplier, and increment. 'd' and 'u' will be introduced later and have to do with which bits of the state are output.

[126]:
```
class LCGSetting(NamedTuple):
    m: int # modulus
    a: int # multipier
    c: int # increment
    d: int = 0 # discard bits
    u: int = 0 # upper bits, to discard
```

We set up a handful of LCGs to test. Many are taken from Wikipedia's list of LCG parameters in common us, but others are contrived here to specifically fail all of our conditions.

[100]:
```
lcgs: dict[str, LCGSetting] = {}

lcgs["rogue"] = LCGSetting(m, a, c, 16, 0)
lcgs["Knuth & Lewis"] = LCGSetting(2**32, 1664525, 1013904223)
lcgs["ANSI C"] = LCGSetting(2**31, 1103515245, 12345, 16, 0)
# The Sinclair ZX81. The UK answer to the Commodore64
lcgs["ZX81 (Bad)"] = LCGSetting(2**16 + 1, 75, 74)
lcgs["VMS"] = LCGSetting(2**32, 69069, 1)
lcgs["SV32"] = LCGSetting(2**32, 0x915f77f5, 1)

# Some specifically bad ones
lcgs["Bad 4"] = LCGSetting(m, ((a-1)//2) +1, c)
lcgs["Bad rel prime"] = LCGSetting(m , a, c * 2)
lcgs["Bad extra"] = LCGSetting(m, 1 + (a-1)*2, c)
lcgs["Bad each prime"] = LCGSetting(m*3, a, c)

# The only way I can think of to have something fail at "square free"
# while "each primne" is to make a = m + 1.
square_free_m = 2 * 3 * 5 * 7 * 11 * 13 * 17 * 19 * 23
lcgs["Bad square free"] = LCGSetting(square_free_m, square_free_m + 1, 29)
```

And now to see how each do.

[101]:
```
for (name, v) in lcgs.items():
    conditions = dh_lcg_check(v.m, v.a, v.c)
    if conditions.all():
        print(f'All conditions hold for {name}')
    else:
        fails = [k for (k, v) in conditions.items() if v == False ]
        print(f'Conditions {fails} fail for {name}')
```

```
All conditions hold for rogue
All conditions hold for Knuth & Lewis
All conditions hold for ANSI C
Conditions ['EACH_PRIME'] fail for ZX81 (Bad)
All conditions hold for VMS
All conditions hold for SV32
Conditions ['RULE4'] fail for Bad 4
Conditions ['COPRIME'] fail for Bad rel prime
Conditions ['NO_EXTRA'] fail for Bad extra
Conditions ['EACH_PRIME'] fail for Bad each prime
Conditions ['SQUARE_LADEN'] fail for Bad square free
```

3 Dealing with low order bits

There is a bit of a problem with all generators of the type described above. The three least significant bits the result are very highly predictable. Let's illustrate.

First we will create a generator that does not trim bits. (Also, I just learned the Generator/yeild construction in Python.)

[102]:
```
def unshifted_lcg(
                modulus: int,
                multiplier: int,
                increment: int,
                seed: int) -> Generator[int, None, None]:

    state = seed
    a = multiplier % modulus
    c = increment % modulus
    while True:
        state = (a * state + c) % modulus
        yield state

seed = 0xABAD5EED # well, not a particularly bad seed.
rogue_unshifted = unshifted_lcg(m, a, c, seed)
```

Now lets look at the first few outputs.

[103]:
```
for _ in range(10):
    print(next(rogue_unshifted))
```

```
1515747482
19507419
1959566720
1932450201
1342751350
203341991
1920877820
1634189701
1534125714
172340147
```

On thing you might note from the above is that the output alternates between even and odd numbers. This means that the least significant bit of the output alternates between 1 and 0.

This kind of pattern is not something we want to see in a random number generator.

[104]:
```
for _ in range(5):
    r = next(rogue_unshifted)
    lsb = r & 1
    print(lsb)
```

```
0
1
0
1
0
```

The problem isn't only the the very least significant bit, there is a pattern of repetition with the second and third least bits when m is a power of 2.

[105]:
```
for _ in range(16):
    r = next(rogue_unshifted)
    lsb3 = r & 6
    lsb3 = lsb3 >> 1
    print(f'bits 3 and 2: {lsb3:02b}')
```

```
bits 3 and 2: 10
bits 3 and 2: 01
bits 3 and 2: 01
bits 3 and 2: 00
bits 3 and 2: 00
bits 3 and 2: 11
bits 3 and 2: 11
bits 3 and 2: 10
bits 3 and 2: 10
bits 3 and 2: 01
bits 3 and 2: 01
```

```
bits 3 and 2: 00
bits 3 and 2: 00
bits 3 and 2: 11
bits 3 and 2: 11
bits 3 and 2: 10
```

The solution to this for LCGs is to keep the low order bits for the internal state of the generator, but to not include them in the output. The rogue LCG returns the the 16 high order bits.

[106]:
```
def lcg(modulus: int,
        multiplier: int,
        increment: int,
        seed: int,
        discard_bits: int) -> Generator[int, bool, None]:

    state = seed
    m = modulus
    a = multiplier % m
    c = increment % m

    # Python has int.bit_length(), but Sage doesn't. Sage has Integer.nbits()
    if not discard_bits < m.bit_length():
        raise ValueError("can't discard so many bits")

    while True:
        state = (a * state + c) % m
        reseed = yield state >> discard_bits
        if reseed:
            state = seed

# And now we can constuct the (intended) rogue RNG
rogue_lcg = lcg(m, a, c, seed, 16)
```

[107]:
```
for _ in range(10):
    print(f'{next(rogue_lcg):8}')
```

```
   23128
     297
   29900
   29486
   20488
    3102
   29310
   24935
   23408
    2629
```

4 The rogue LCG modulus

I have stated that the (intended) modulus for the rogue LCG is $m = 2^{31}$. I might be mistaken about that. It is possible that the intent was $m = 2^{32}$. Either way, the modulus is never explicitly stated in the code. Instead it is achieved through interger overflow behavior.

Let distill the relevant portions of the source code

```
int seed;
seed = /* initial seed set from time and PID */
#define RN (((seed = seed*11109+13849) >> 16) & 0xffff)
```

If `seed` had been declared as a `uint32_t` instead of `int`, then we would have $m = 2^{32}$. Of course the type `uint32_t` did not exist as part of the C language when rogue was first written, but there was a distinction between signed integers and unsigned integers that should have been made use of.

There are two issues here. One is that `int` isn't always going to be 32 bits, but the more important one is that `int` is a signed integer.

It is easier to explain how things might break in the rogue code by first describing how things would correctly work if `seed` had that type `uint32_t`.

The maximum unsigned 32 bit integer. Following conventions that came into C in the late 1990s, I will call that `UINT32_MAX`

$$\texttt{UINT32\_MAX} = 2^{32} - 1 = 4\,294\,967\,295$$

If you do interger arithmatic on unsigned 32 bit integers, all your arithmatic will be modulo 2^{32}. If `a`, `c`, and `seed` are all unsigned 32 bit integers then

```
seed = seed * a + c
```

will be done modulo 2^{32}.

When working with unsigned 32 bit integers, the following are true

- `UINT32_MAX + 1 == 0`
- `UINT32_MAX + 5 == 4`
- `2 * 2147483648 = 0`

This all comes for free as it is a consequence of how the computing system handles unsigned integers when you exceed their maximum value. And this has been part of the C programming language early days.

> For unsigned integers the C language is quite specific on the question of overflow: every operation on unsigned integers always produces a result value that is congurent mod 2^n to the true mathematical result of the operation (where n is the number of bits used to represent the unsigned result). {cite:ps}`HarbisonSteele91` [p 168]

4.1 Unsigned overflow

With signed integers, as used in the rogue LCG, the behavior not well-defined. The C reference at the time, {cite:ps}`KerninghanRitchie78`, rogue was written merely had this to say

> The handling of overflow and divide check in expression evaluation is machine-dependent.

while {cite:ps}`HarbisonSteele91` have more to say:

> Traditionally, all implementations of C have ignored the question of signed integer overflow, in the sense that the result is whatever value is produced by the machine instruction used to implement the operation.

And they continue to say

> Many computers that use a two's-complement representation for signed integers handle overflow of addition and subtraction simply by producing the low-order bits of the true two's-complement result. No doubt *many existing C programs depend on this fact*, but such code is technically not portable. [emphasis added]

It appears that the design of rogue's random number generator depends on that assumption about how overflow would be handled.

If that could be relied upon, then it works out to behaving as if integer arithematic on signed 32 bit integers is done modulo 2^{31}. Had they used an unsigned integer, they would have had a larger effective modulusof 2^{32} and would have had more portable code.

I should note that people did not think about portability back then the way they do now. There simply wheren't that many different architectures that Unix ran on. Additionally, C programmers tended to have a greater sense of machine architecture and behavior for their target systems. While it is a mistake to rely on how the machine will handle integer overflow, I am confident that the authors did think about how integer overflow was handled.

4.2 Unsigned bit shift

We have a similar story with the 16 bit leftward shift, `>> 16` when applied to a signed integer. It can behave in one of two ways according the the C standards. But in our case there is no dependence on either of the two possible behaviors because those bits are discarded.

When a negative signed 32 bit integer is shifted left by 16 bits, the filled in left-most bits could all be 0s on some implementions or all 1s on others. However, the `& 0xffff` operation switches those all to zero anyway.

5 `sizeof(int)`

The code assumes that the size of an `int` value is 32 bits. At the time it was written, 64 bit systems were far in the future. Unix did run on the 16 bit PDP-11, but I suspect that curses (the terminal graphics library) or even BSD Unix did not run on a PDP-11, so the authors may have had some reason to think that 16 bit `int`s were a thing of the past.

They did not anticipate the rise of the 8086 and 80286 16-bit architecture.

5.1 15 bit multiplier and increment

While the multiplier, 11109, and increment, 13849 meet all of the requirements stated above, they are rather small in comparison to the modulus, 2^{32}. I suspect that they were originally designed for a 16 or 15 bit modulus. Both numbers are less than 2^{15}, 32768.

The problem with this is that when the internal state, s, is small the output of the subsequent calls to LFG will increase until $s > m/a - c$. In a random sequence, you would expect that output x_{n+} to be less than x_n exactly as often as it is more than x_n.

Lets see what happens when we start with a small seed.

5.2 Ups and Downs

In earlier versions there was a bug in my code for counting how many times in a sequence current < previous. I was perplexed by the results and so I tried tinkering with generator settings to see what was going on. Therefore there is a lot of such testing in this section that is largely irrelevant.

We are going to look at the ups and downs of different generators, so lets write some code for doing that.

[108]:
```
class DirectionCounts:

    def __init__(self, first_down: Optional[int], downs: int, ups: int, ties:␣
↪int):
        self.first_down = first_down # first time it moves downward
        self.downs = downs # count of when it moves down
        self.ups = ups # count of when it moves up
        self.ties = ties # count of when next and curr are the same

    @classmethod
    def from_generator(cls, gen: Generator, trials: int):
        if trials < 1:
            raise ValueError(f"trials ({trials}) must be greater than 0")

        first_down: Optional[int] = None
        downs = 0
        ups = 0
        ties = 0
        prev = next(gen)
        curr = next(gen)
        for n in range(1, trials + 1):
            if curr < prev:
                downs += 1
                if first_down is None:
                    first_down = n + 1
            elif curr > prev:
                ups += 1
            else:
                ties += 1
            prev, curr = curr, next(gen)

        if (ups + downs + ties) != trials:
            raise Exception(f"ups ({ups}) + downs ({downs}) + ties({ties}) !=␣
↪trials ({trials})")
```

```
        return cls(
            first_down,
            downs,
            ups,
            ties
    )

    def trials(self) -> int:
        return self.ups + self.downs + self.ties

    def __str__(self) -> str:
        return f'''
            First downward change: {self.first_down}
            Number of downward steps: {self.downs}
            Number of upward steps: {self.ups}
            Number of ties: {self.ties}
        '''
```

[109]:
```
trials = 20
rogue_lcg.send(True)
counts = DirectionCounts.from_generator(rogue_lcg, trials)
print(counts)
```

```
            First downward change: 3
            Number of downward steps: 11
            Number of upward steps: 9
            Number of ties: 0
```

[110]:
```
rogue_lcg.send(True)
[next(rogue_lcg) for _ in range(20)]
```

[110]:
```
[297,
 29900,
 29486,
 20488,
 3102,
 29310,
 24935,
 23408,
 2629,
 17071,
 15263,
 23483,
 14971,
 18623,
 27579,
```

```
 3038,
 2855,
 3520,
 21738,
 20084]
```

[111]:
```
rogue15_lcg = lcg(modulus= 2 ** 15,
                  multiplier=a,
                  increment=c,
                  seed = 1,
                  discard_bits=8)
```

[112]:
```
trials = 50000
rogue_lcg.send(True)
counts = DirectionCounts.from_generator(rogue_lcg, trials)
print(counts)
```

```
First downward change: 3
Number of downward steps: 24974
Number of upward steps: 25025
Number of ties: 1
```

[113]:
```
trials = 2 ** 17
counts = DirectionCounts.from_generator(rogue15_lcg, trials)
print(counts)

# And lets re-init the generator
_ = rogue15_lcg.send(True)
```

```
First downward change: 2
Number of downward steps: 65020
Number of upward steps: 65028
Number of ties: 1024
```

If we want to do a statistical analysis of upward vs downward steps (and who wouldn't want to do that) we need to know the expected probability of each. For large sized output, it is going to be very close to 0.5, but for generators that produce smaller output, like our 8 bit case above, we need to take ties into account.

[114]:
```
def outsize(modulus: int, discard_bits: int) -> int:
    return modulus // (2 ** discard_bits)

def expected_p_up(outsize) -> float:
    return 0.5 - 1.0/outsize
```

Now lets get these for our 8 bit out case.

[115]:
```
out_size = outsize(2**15, 8)
print(out_size)
exp_p = expected_p_up(out_size)

print(exp_p)
```

```
128
0.4921875
```

[116]:
```
binomtest(k = counts.ups, n=trials, p = exp_p, alternative="two-sided")
```

[116]:
```
BinomTestResult(k=65028, n=131072, alternative='two-sided',
statistic=0.496124267578125, pvalue=0.004397797351552629)
```

[117]:
```
kl_settings = lcgs["Knuth & Lewis"]
kl_lcg = lcg(kl_settings[0], kl_settings[1], kl_settings[2], seed=1,␣
 ↪discard_bits=0)
```

Now let's try with the Knuth and Lewis parameters.

[118]:
```
trials = 2 ** 15

counts = DirectionCounts.from_generator(kl_lcg, trials)
print(counts)

# And lets re-init the generator
_ = kl_lcg.send(True)
```

```
First downward change: 5
Number of downward steps: 16363
Number of upward steps: 16405
Number of ties: 0
```

Now we try with the Steele and Vigna parameters for 32 bit generators.

[119]:
```
sv32_settings = lcgs["SV32"]
sv32_lcg = lcg(sv32_settings[0], sv32_settings[1], sv32_settings[2], seed=1,␣
 ↪discard_bits=0)
```

[120]:
```
trials = 2 ** 17

counts = DirectionCounts.from_generator(sv32_lcg, trials)
print(counts)

# And lets re-initialize the generator
_ = sv32_lcg.send(True)
```

```
First downward change: 2
Number of downward steps: 65678
Number of upward steps: 65394
Number of ties: 0
```

As I loaded the statistics packages already, we will run the stats.

[121]:
```
binomtest(counts.ups, n=trials, p = 0.5, alternative="two-sided")
```

[121]:
```
BinomTestResult(k=65394, n=131072, alternative='two-sided',
statistic=0.4989166259765625, pvalue=0.4344001564044689)
```

5.3 Discarding some high bits

Note again that much of this was written when I mistakenly believed there was a strong mismatch between downward steps and upward steps.

We are already set up to discard some low (least significant) bits in what the generator returns, discarding some high (most significant) bits should mask the step direction effect.

To avoid going back and changing the name of the `lcg` function, I am going to call this `acg` for Affine Congruential Generator. (That would be the correct name for any of these where the incremenent isn't zero).

[122]:
```
def acg(modulus: int,
        multiplier: int,
        increment: int,
        seed: int,
        discard_bits: int,
        discard_high) -> Generator[int, bool, None]:

    state = seed
    m = modulus
    a = multiplier % m
    c = increment % m

    # Python has int.bit_length(), but Sage doesn't. Sage has Integer.nbits()
    if not discard_bits < m.bit_length():
        raise ValueError("can't discard so many bits")

    if discard_high > discard_bits:
        raise ValueError("Can't discard more high bits than all discarded")

    high_bit_modulus = m - (2**discard_high)

    while True:
        state = (a * state + c) % m
```

```
        reseed = yield (state % high_bit_modulus) >> (discard_bits - discard_high)
        if reseed:
            state = seed
```

Now let's try SVG32, but shave off six bits from the top

[123]:
```
sv32h_acg = acg(sv32_settings[0], sv32_settings[1], sv32_settings[2], seed=1,␣
↪discard_bits=8, discard_high=6)
```

[124]:
```
trials = 2 ** 17

counts = DirectionCounts.from_generator(sv32h_acg, trials)
print(counts)

# And lets re-init the generator
_ = sv32h_acg.send(True)
```

```
First downward change: 2
Number of downward steps: 65678
Number of upward steps: 65394
Number of ties: 0
```

[125]:
```
binomtest(k=counts.ups, n=trials, p = 0.5, alternative="two-sided")
```

[125]:
```
BinomTestResult(k=65394, n=131072, alternative='two-sided',
statistic=0.4989166259765625, pvalue=0.4344001564044689)
```

Now that my downward/upward count method is fixed, we find no problem with the number of upward vs downward steps.