How To Crack a Linear Congruential Generator Algorithm

Linear congruential generator (LCG) is a simple random number generator, using the following formula:

$$X_{n+1} = (aX_n + c) \bmod m$$

Under most circumstances in programming, *m* will be set to a power of 2 so that CPU can automatically do the modulo operation. According to Wikipedia, when *m* is a power of 2 and $m \geq 4$, the loop period of the random numbers will be *m* when and only when $a \bmod 4 \equiv 1$ and $c \bmod 2 \equiv 1$.

Though not strictly proved by math, here I have some fast algorithms to do the following calculations when the conditions above are met:

1. Calculate X_n when *n* and X_0 are given;
2. Calculate X_n when X_{n+1} is given;
3. Calculate *n* when $X_0 = 0$ and X_n is given.

Now I will explain how these can be done.

I. Calculate X_n when *n* and X_0 are given

When *m* is set to 2^n, the formula is simplified to:

$$X_{n+1} = aX_n + c$$

With this formula, we can get the (*n+1*)-th random number from the *n*-th number. For simplicity, let's call this formula "pushes the random number forward once". Let $F_k(x) = a_k x + c_k$ be the function that pushes the random number forward for *k* times (yes, all F_k has a same form as the LCG formula), according to the formula above, we have $F_1(x) = ax + c$, so $a_1 = a$ and $c_1 = c$.

Now we can have:

$$F_2(x) = F_1\big(F_1(x)\big) = a_1(a_1 x + c_1) + c_1 = a_1^2 x + a_1 c_1 + c_1$$

So, we have $a_2 = a_1^2$ and $c_2 = a_1 c_1 + c_1$. You can see that $F_2(x)$ is still an LCG formula, but do not have a loop period of 2^n.

We do this again and again and we will have:

$$a_4 = a_2^2, c_4 = a_2 c_2 + c_2$$
$$a_8 = a_4^2, c_8 = a_4 c_4 + c_4$$
$$a_{16} = a_8^2, c_{16} = a_8 c_8 + c_8$$
$$\dots$$
$$a_{2^{n-1}} = a_{2^{n-2}}^2,\ c_{2^{n-1}} = a_{2^{n-2}} c_{2^{n-2}} + c_{2^{n-2}}$$

Notice that in computer programs, CPU will do modulo 2^n automatically to all these a_k and c_k, but this does not matter.

Now, how to get F_k for an arbitrary *k*? We just need to write *k* into sum of the power of 2, and composite all the F_{2^i}. For example, if we want F_3, since $3 = 1 + 2$, we can have:

$$F_3(x) = F_2\big(F_1(x)\big) = a_2(a_1x + c_1) + c_2$$

You may notice that $F_2\big(F_1(x)\big) = a_2(a_1x + c_1) + c_2$ and $F_1\big(F_2(x)\big) = a_1(a_2x + c_2) + c_1$ looks different. However, they will become the same thing when you expand a_2 and c_2 into the original *a* and *c*. Though not proved, I think that for any *k*, F_k is always unique and is not related to the order we used to composite F_{2^i}.

After we get F_k, we can calculate X_k quickly using $X_k = F_k(X_0)$.

II. Calculate X_n when X_{n+1} is given

Let $F_{-1}(x)$ be the formula that pushes the random number backward once, i.e. $F_{-1}(x) = F_1^{-1}(x)$. Notice that the loop period of the original LCG algorithm is 2^n, so $F_{-1}(x) = F_{2^n-1}(x)$. We can use the way written in part I to calculate F_{2^n-1}. We can then calculate X_k from X_{k+1} using $X_k = F_{-1}(X_{k+1})$.

According to part I, F_{-1} is also an LCG formula. So, we can do the same calculation as part I to get F_{-2}, F_{-4} and so on.

III. Calculate *n* when $X_0 = 0$ and X_n is given

This is the trickiest part. First let me introduce 2 unproved lemmas: When the LCG algorithm is using 2^n as modulo and has a loop period of 2^n:

1. The *i*-th lowest bit of random number *X* in binary form has a loop period of 2^i;
2. If the *i*-th lowest bit of X_k in binary form is 0, then the *i*-th lowest bit of $X_{k+2^{i-1}}$ in binary form is 1, and vice versa.

Though they are not proved, I think they are correct. At least, no counterexamples have been found yet. With these lemmas we have an algorithm to calculate *k* quickly from X_k:

To start, we need an accumulator. We first investigate the lowest bit of X_k. If it is 1, we need to push X_k backward once and add 1 to the accumulator. Let $X_{k'} = F_{-1}(X_k)$, according to lemma 2, the lowest bit of $X_{k'}$ should be 0. If the lowest bit of X_k is 0, then Let $X_{k'} = X_k$, and do not add anything to accumulator. In either circumstances, the lowest bit of $X_{k'}$ is always 0.

We then investigate the second lowest bit of $X_{k'}$. If it is 1, we need to push $X_{k'}$ backward twice and add 2 to the accumulator. Let $X_{k''} = F_{-2}(X_{k'})$, according to lemma 2, the second lowest bit of $X_{k''}$ must be 0, and the lowest bit of $X_{k''}$ remains 0 according to lemma 1. If the second lowest bit of $X_{k'}$ is 0, then Let $X_{k''} = X_{k'}$, and do not add anything to accumulator. In either circumstances, the lowest bit and second lowest bit of $X_{k''}$ are always 0.

We can repeat this again and again. Every time this algorithm will set one certain bit to 0, and all lower bits remain 0. As X_k has finite bits, it will eventually become 0, and then we can know that k equals the total times that the number has been pushed backward, which has been recorded in the accumulator.

We can also pushing the number forward instead of pushing backward by using $F_1,\ F_2,\ F_4, \ldots$ instead of $F_{-1},\ F_{-2},\ F_{-4}, \ldots$. The algorithm will be exactly the same, except for that the number has been pushed forward for k times before becoming 0. So the actual result we want will be $2^n - k$ instead of k.