Pseudo Random Number Generator Analysis

Linear Congruence Generator

When looking at the built-in pseudo random number generator (PRNG) from java.util.Random critical to recognize, that the local value that is used as *x* is of type *long*, while the returned type of the method `next()` is *int*. This means that the modulus of the linear congruence generator must be greater than $2^{31} = 2\,147\,483\,648$ to cover every positive *int.* Also, since the modulo is stored as a *long* the value of the modulo must also be smaller than $\sqrt{2^{63}} \approx 3\,037\,000\,499$ since otherwise two values could multiply to give a result greater than the largest possible *long.* Using an online random prime generator I picked the prime 2 655 067 433 as it was the first one that fulfilled the two criteria above. The reason I wanted to use a prime instead of any other number was because they have higher period. I selected the multiplier 221 255 620 as it was the first primitive root of the prime that I found. The increment I chose is 1 because it is simple and is used by many other libraries. In summary, the linear congruence generator looks like this:

$$x_{i+1} = 221\,255\,620x_i + 1\,mod\,2\,655\,067\,433$$

Randomness Analysis

The value of *x* is returned and updated by my method `next(int bits)` where bits is the number of bits wanted in the result. When tested this generator gives, like expected 2 655 067 432 different values of *x* before repeating the sequence again.

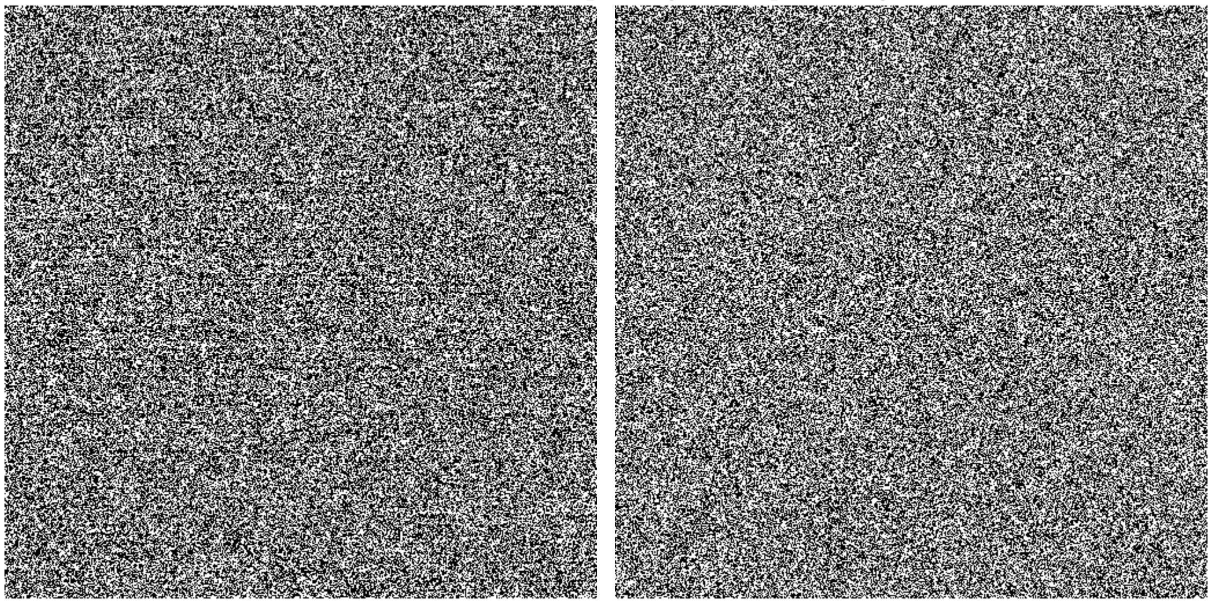

Figure 1 Bitmap Comparison

When comparing the built in PRNG (left) with the one I made (right) in figure 1, at least to my naked eye I cannot discern any pattern in the right image. This is not surprising given that the period is very long. The way the bitmap was created is that many numbers were generated in sequence with the `nextInt(256)` method and plotted them in binary with a 1 being represented with a white pixel and a 0 represented by a black pixel. The starting seed was set as the current system time, similar to how it is done in the original Java class. When looking at the frequency of the numbers it also appears to be evenly distributed. When generating 10 million numbers with the `nextInt(256)` again with the current system time as the seed the most and least frequently appearing numbers deviate less than 1.3% from the median.

Generating large numbers

One problem that I had is the case of generating larger numbers evenly. The linear congruence generator currently generates numbers below *m* in value, and when the argument is, for example 8 bits, it simply generates a random number in the range zero to m and then returns the 8lsb to fit the requested bit size. This does slightly affect the frequency at which numbers appear since the largest number that can be returned by the linear congruence generator is $m - 1$ which, when written in binary has the 8lsb of $00101000_2 = 40_{10}$. This means that the first 40 numbers have a slightly higher likelihood of appearing, with a ratio of $\left\lceil \frac{2655067432}{256} \right\rceil = 10\,371\,358$ to $\left\lfloor \frac{2655067432}{256} \right\rfloor = 10\,371\,357$. This ratio is insignificant since the difference is so small. However, when generating larger random numbers such as 31 bits, the ratio is even more skewed in favour of the lower numbers. In that case, the ratio of less frequent numbers to more frequent numbers is: $\left\lfloor \frac{2655067432}{2^{31}} \right\rfloor = 1$ to $\left\lceil \frac{2655067432}{2^{31}} \right\rceil = 2$. This means that the lower numbers appear twice as often as the higher ones. I solved this by looking at the built-in Random class. There, in the `nextInt()` the `next()` function is called until the result becomes smaller than the largest multiple less than *m* of `nextInt()` argument. This is unlikely to happen with small arguments but, for larger arguments such as 2^{31} it would happen 50% of the time. This is a potential weakness, because an attacker could use the time required to recalculate the number to find out where in the sequence the number is given that they know where there are multiple consecutive large numbers. But given that the built in Java class does it I think it should be accepted.

Conclusion

My implementation of the random class in Java generated a long sequence of pseudorandom numbers using a linear congruence generator with a period of 2 655 067 432. Using a bitmap comparison there are no obvious patters that appear. A method was also put in place to ensure that the numbers are evenly distributed when smaller numbers are inputted as the argument. Therefore, I am convinced that the PRNG fulfils the requirements for this task.