31 AUGUST 2014

Predicting the next Math.random() in Java

A while back a friend of mine hosted a programming competition where you were given 9 random numbers one at a time and you had to guess the rank (the sorted position amongst the 9) of each as they are coming in. The goal was to submit a strategy in Java that will maximize the number of these games won. The best solution can guess right about 3.3% of the time but I figured if he was going to let me execute arbitrary code I might as well cheat and guess correctly 100% of the time. So I set out to see if I can predict the rest of the numbers in the stream after seeing just one.

So I started digging through `java.util.Random`'s source and found that it is just a linear congruential generator which can be easily to cracked to obtain the original seed. The rest of this post will show how (if you just want to get the code, you can find it here.

A quick refresher on how LCGs work: starting with a seed `x₀`, the next pseudorandom number generated is given by `x₁ = a * x₀ + c mod m`, for some fixed constants `a`, `c`, and `m`. So for example let's say `a = 3`, `c = 5`, `m = 7` and we start with the seed `x₀ = 0`, then the

`= 3 * 5 + 5 mod 7` , `2 = 3 * 6 + 5 mod 7` , `4 = 3 * 2 + 5 mod 7` , etc.

It should be clear that if I gave you a "random" number generated from this process (e.g., $x_i = 2$), you can predict the next number by applying the formula yourself (e.g, $x_{i+1} = 3 * 2 + 5 \bmod 7 = 4$).

The relevant method in `java.util.Random` looks like this (where $m=2^{48}$ or in other words, a 48-bit number is generated with each iteration):

```
public
class Random implements java.io.Serializable {
    private final AtomicLong seed;

    private static final long multiplier = 0x5DEECE66DL;
    private static final long addend = 0xBL;
    private static final long mask = (1L << 48) - 1;

    protected int next(int bits) {
        long oldseed, nextseed;
        AtomicLong seed = this.seed;
        do {
            oldseed = seed.get();
            nextseed = (oldseed * multiplier + addend) & mask;
        } while (!seed.compareAndSet(oldseed, nextseed));
        return (int)(nextseed >>> (48 - bits));
    }
    /* ... */
}
```

Unlike our toy example, each call to `next` only returns `bits` number of the top bits instead of the whole 48-bit number generated so there's a bit more work to do.

Now going back to the post title, how is `next(bits)` used for `Math.random()` ? If you dig into the java source code you will find that `Math.ra`

And `nextDouble` calls `next(bits)` like this:

```
public double nextDouble() {
    return (((long)(next(26)) << 27) + next(27))
        / (double)(1L << 53);
}
```

Which means it is concatenating the top 26 and 27 bits of two iterations of `next()` to make a 53 bit number which it normalizes back into a double with a value between 0 and 1.

We are interested in the original values returned by `next(26)` and `next(27)` which can be recovered by reversing the operations done above. So assuming we have a value that we know was generated from calling `nextDouble()`, we can do the following:

```
long numerator = (long)(nextDoubleValue * (1L << 53));
int next26 = (int)(numerator >>> 27);
int next27 = (int)(numerator & ((1L << 27) - 1));
```

Now we have the top 26 and 27 bits of two previous seeds used by `next()`. But only having the top bits is not sufficient for generating future values. Fortunately, it is fairly quick to brute force the unknown remaining lower 48-26=22 bits (2^{22} possibilities). You can do this by trying all 48-bit seeds that has the same top 26 bits as `next26` and apply the LCG formula to see if the next seed has the same top 27 bits as `next27`. Like so:

```
long upper27Of48Mask = ((1L << 27) - 1) << (48 - 27);
long oldSeedUpper26 = ((long)next26 << (48 - 26)) & mask;
long newSeedUpper27 = ((long)next27 << (48 - 27)) & mask;
ArrayList<Long> possibleSeeds = new ArrayList<Long>();
for (long oldSeed = oldSeedUpper26;
     oldSeed <= (oldSeedUpper26 + ((1L << (48 - 26)) - 1));
     oldSeed++) {
            possibleSeeds.add(newSeed);
        }
    }
```

It is possible that out of the 2^{22} seeds brute-forced there is a next value with the same top 27 bits purely by chance. Though if we make the (questionable) assumption that each `newSeed` is a random independent number, this probability is fairly small: $1 - (1 - 2^{-27})^{2^{22}} = 0.03076676574$.

So it is likely that we will find exactly one seed with the brute force. Using that seed we can now simulate the return value for all future calls to `next()` and thus `nextDouble()` and `Math.random()` also. So we just managed to predict all future values of this generator!

It is well known that `Math.random()` is insecure so none of this is news to anyone. I just didn't expect it to be this easy and that you can do it by just observing a single previous value!

Complete code for this can be found here.

As for the original competition this was written for, I actually ended up losing anyway. This only managed to guess a couple thousand out of 100000 correctly before exceeding the time limit. The other cheaters did a much better job by using java reflection to modify local variables to win.

Franklin Ta

Read more posts by this author.

Read More

Franklin Ta Share this

Join the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

Sort by Best

King Amada • 5 years ago

How can I generate the initial seed that produced a random number?
For example java produces the pseudorandom sequence: 1553932502 by using 12345 as a seed, how can I do the inverse? i.e. getting 12345 out of the sequence 1553932502.
Reading above says it is possible, I try to implement using your code, but I couldn't.
Thanks in advance

Reply • Share ›

Алекс Фатеев King Amada • 5 years ago • edited

If you brute-force it, there probably would be millions of seeds, each giving you first number 1553932502, but they will all differ in the next generated values. Random function isn't designed to be bijective, and seed can't be actually gained from just single value. You can only get the original seed by generating sequence from it and then repeat the method presented in the topic until the amount of possible seeds is just one.

Reply • Share ›

GeoffZoref • 5 years ago • edited

Interesting. I'm here just because I was just using the Random api for a project and was surprised to see that the first number, 52, was followed by 51. So I wrote an iteration to print 100 randoms, and on like the third try the first number was 1. I kept playing with it and after about 10 tries the last digit was 75 twice in a row!
I'm thinking of a way to calculate the percentage of the first digit of random numbers between 1-1000 that start with a 1,2 or 3. See if Zipf Law applies. Any suggestions>?

Reply • Share ›

Greg • 6 years ago

Great article, I am trying to use the replicating next int option, for 2 ints I already know, and trying to find the seed. I have increased the iterations significantly, but still am having no luck. Any suggestions? This is what I did, just a minor modification. Min and Max values are 1 and 80.

```
System.out.println("Replicating from nextInt");
System.out.println("Starting at: "+new java.util.Date());
for (int i = 0; i < iterations; i++) {
```

```
if (rr.replicateState(41, 46)) {
for (int j = 0; j < 5; j++)
System.out.println(rr.nextInt() + "\t" + r.nextInt(max-min)+min);
System.out.println();
}
}
System.out.println("Ended at: "+new java.util.Date());
```

^ | ˅ • Reply • Share ›

GoGoJJTech • 7 years ago

I feel bad for intruding and reviving this, but it's pretty interesting!
My question is:
How impossible is it to grab the possible seeds from a value that was subject to a ".toFixed(n)"?

Example from using actual Math.Random().nextFloat() calls: 0.88820076 and 0.49951583

If these floats were presented as 0.8882 and 0.4995, there would be many possible seeds, but how would we get the possible seeds in the first place? (ヮ°)☞

^ | ˅ • Reply • Share ›

Franklin Ta ➦ GoGoJJTech • 7 years ago

I actually had code for a similar scenario. I will try to clean it up and put it on github when I have time.

Basically due floating point representation, rounding a float will (usually) leave the top bits of the bitwise representation intact. Then you just follow along with the same logic outline in this post to bruteforce except this time with fewer known bits. For the example above where you know the first number to within 1/10000, you gain log_2(10000) = 13 bits of info. The random state is 48 bits so you still have to bruteforce 2^35 but might be feasible.

^ | ˅ • Reply • Share ›

sunny • 7 years ago

and how did 6 came...in eg 6 = 3 * 5 + 5 mod 7..can you plz help calculate.

^ | ˅ • Reply • Share ›

Franklin Ta ➦ sunny • 7 years ago

You should read the wikipedia article on linear congruential generator.

Using the formula x_{i+1} = a * x_i + c mod m I chose a = 3, c = 5, m = 7 just because they were small enough to work out by hand. If x_i is the i-th number then starting with x_0 = 0 you get:

x_0 = 0

x_2 = 6 = 3 * 5 + 5 mod 7

x_3 = 2 = 3 * 6 + 5 mod 7

x_4 = 4 = 3 * 2 + 5 mod 7

and so on.

^ | ˅ • Reply • Share ›

Pablo ➦ Franklin Ta • 5 years ago

To avoid discussions on operator precedence one should write the formula as x_{i+1} = (a*x_i + c) mod m; that way it is clear why one gets 6 for x_2.

^ | ˅ • Reply • Share ›

sunny • 7 years ago

why you used a=3 mode = 7 and C=5 ..why not other numbers

^ | ˅ • Reply • Share ›

Buddha • 8 years ago

This is pretty smart! I didn't know how random numbers were generated, so I would say this article was pretty informative with clear illustration and the limitation of generating random numbers by the LCG approach. Is there a better way to generate random numbers which makes it computationally very expensive to guess the next random number based on the current one?

If I were to use Java Random and didn't use the single static instance of Random, but used a new seed each time using a new instance of java.util.Random, how hard would that be to crack? I guess the question to answer then is, how does java determine the very first value of its seed?

^ | ˅ • Reply • Share ›

Franklin Ta ➦ Buddha • 8 years ago

Yea any cryptographically secure PRNG would work (such as SecureRandom).

For the second part of your comment, I think you are suggesting to create a new instance of Random each time you need a random number. That would prevent this approach (since it can only predict the state of one instance of Random) but I don't suggest it since you won't be guaranteed anything anymore about the properties of that stream you constructed. For example someone gave this example on hacker news:

```
for(int i = 0; i < 256; i++) {
System.out.println(new Random(i).nextInt(8)); // prints 5 every time
}
```

Wim Vander Schelden (@wvdschel • 8 years ago

Nice work.

Just a quick remark: Firefox does not use Rhino. Rhino was a separate project at Mozilla, and was never used in any of Mozilla's browsers as far as I know. Firefox uses SpiderMonkey, a C++ implementation of JavaScript. Rhino was adapted by Oracle (in a project called Nashorn) and included as a JavaScript runtime starting with Java 8.

• Reply • Share ›

Justin L • 8 years ago

> I was surprised when I saw that this also work on Firefox(but not on Chrome) for the
> Math.random() in JavaScript. I am guessing it is because they use Rhino which
> is written in Java?

Firefox doesn't use that code, but you're right that the Firefox Math.random is...let's say "influenced" by Java.util.random. http://dxr.mozilla.org/mozi...

• Reply • Share ›

Franklin Ta Justin L • 8 years ago

Thanks, fixed!

• Reply • Share ›

Alexandru Cobuz • 8 years ago

Amazing work, and this could be considered as a major security issue if you think of how many reliable sources are using the random function as really "random".
Incredible, but I think there's another way to predict within a margin > 3% what could be next number using neural network.

• Reply • Share ›

Jesper Alexandru Cobuz • 8 years ago

"major security issue": Not really. The API docs of java.util.Random explain that it is not cryptographically secure and that you should not use it for security sensitive operations. So it's not as if we've discovered a major security hole here. If you need secure random numbers, use java.secure.SecureRandom instead.

• Reply • Share ›

Franklin Ta Alexandru Cobuz • 8 years ago

Pretty sure neural networks don't apply here. But to clarify the 3% part, the 1 – (1 – (2 ^ -27)) ^ (2 ^ 22) = 0.0307 probability is the chance that you will find more than one seed. The actual seed you want is still going to be in the list of seeds found, you just don't know which one because another stream also matched by chance. This situation doesn't require anything fancy to solve (and definitely nothing like neural nets). You just need a few more values from the Random to cross reference with to narrow down

And also we aren't working with truly random numbers so I admit that probability

Computing CSS matrix3d transforms

I was cleaning out some old notes from my previous job and found some math scribbles for computing CSS transforms and thought I would share it. For some context, I was working on

6 MIN READ

Solving CAPTCHAs on Project Euler

A couple weeks ago I started doing Project Euler problems. Unfortunately this was during their security breach so login was disabled so I couldn't get credit for any of the problems I

4 MIN READ