True Random Number Generators vs Pseudo-Random Number Generators

Introduction

The notion of randomness is one that stitched in the very fibers of nature. Even today, despite its continual exercise of imposing order in the world through its rules and institutions, society is unable to escape the grasp of randomness in the world.

Statistics, a subject which embraces randomness in its application, then becomes our only means of finding some order in this random world we live in. By employing a variety of techniques that depend on random selection and sampling, statistics attempts to make some sense of our confusing and chaotic surroundings.

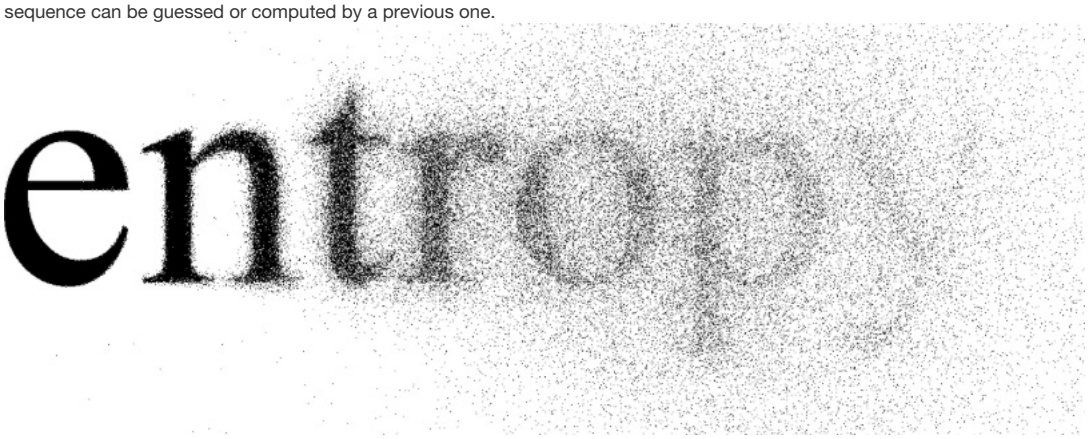
But before we can even begin to understand the functions of these time-tested methods, it becomes important to realize how exactly one might manipulate randomness to facilitate these techniques. Thus, in this post, we will delve into the topics of how random number generation is done.

Random Number Generators

First and foremost, it is important to understand that there is a difference between pseudo-random generators and true randomn generators, the first of which relies on seed-based computations to estimate randomness, and the latter of which uses fluctuations that naturally occur in the physical world around us to determine true random numbers and sequences. In the following sections, we will begin to take a much closer look at both of these different types of number generators.

True Random Number Generators

So what constitutes true randomness? Well, as mentioned in the introduction, the physical world around us constantly exhibits natural randomness due to **entropy**, whether that be in the flow of a river or the atmospheric noise of a given day. True random number generators function by measuring this random noise, and extrapolating random sequences from these samples. What makes true random number generators (TRNGs) unique is the fact that they are are nondeterministic. This is to say that no future element in a randomly progressing sequence can be guessed or computed by a previous one.



While TRNGs give us instant access to true randomness, there are some pitfalls in their usage. Lack of reproducibility, for one, make it difficult to use TRNGs during development of computer programs or software engineering. Often times, when developers want to test their programs, they will want to use random number as inputs to their programs with some expected output in mind. As you might imagine, however, it would be difficult to ever output a given outcome if the program is continuously taking on new random numbers. Because TRNGs never generate the same sequence of numbers (unless by sheer luck, of course) it would become near-impossible for a developer to actually test his/her program and see whether it functions the way its supposed to, given some known random sequence.

This reason, among others, is why most people instead opt to use a tool known as Pseudo-Random Number Generators in daily application.

Pseudo-Random Number Generators

Pseudo-random number generators (PRNGs) are essentially a set of algorithms that allow computers to compute near-random numbers given some seed value to begin the “random” sequence. Thus, PRNGs, unlike TRNGs, are not actually completely random. Instead, the sequences they output depend wholly on the set seed value.

First, let’s clarify what a seed is. A seed is essentially a value that you, as the user, set before starting the pseudo-random number generation. In R, for instance, we do this by using the function:

```
set.seed(1) #sets seed to 1
```

where the 1 can be replaced by any numeric value. The specific algorithm your pseudo-random number generator is employing will then use this seed value as an input, generate some output, and pass that output back in as input to the same algorithm.

If this still doesn’t make too much sense, hopefully this example will serve to clarify any misconceptions. Let’s say we set our seed to the value 121 in the following way:

```
set.seed(121) #sets seed to 121
```

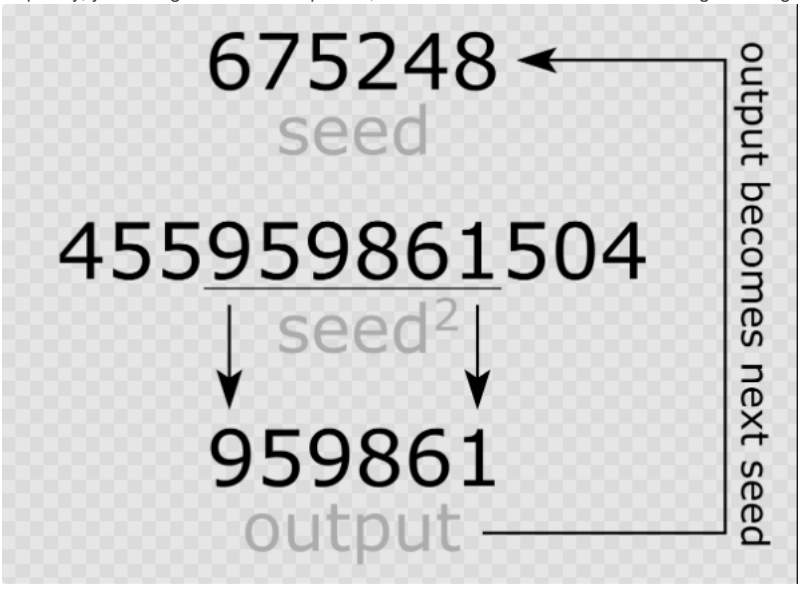
So now we have set our seed. Now let’s say that our specific PRNG employs a unique algorithm which takes an input value, multiplies the input by itself, and outputs the middle of the product. On first iteration, for instance, our input will be our seed, 121. We will then multiply $121 * 121 = 14641$, and output the middle of this result: 464.

Upon second iteration, things really begin to get interesting. Our input for this iteration is now the output of the previous iteration, 464. So let’s take this input, multiply $464 * 464 = 215296$, and output the middle of this result: 529. However this time, instead of simply outputting this result,

we will append it to the output from last time, giving us 464529.

Upon third iteration, we will again do the same thing, taking our output from last time, 529, and multiplying it by itself using $529 * 529 = 279841$, and again output the middle of this result: 984. Additionally, we will append this output to what we can call our "output sequence": 464529984.

Hopefully, you're begin to notice the pattern, which is further clarified in the following flow diagram:



Each input will result in a new output, which we will then use as our new input, to then get a new output, and so on. Additionally, on each iteration, we will append our output to our "output sequence," which is effectively our random number sequence! In this way, we can get close to generating a near-random sequence.

In fact, the algorithm described above is one that is so commonly used for PRNGs that it is given a name: the *Middle Squares method*. Notice that the entire sequence of numbers depends solely on what input value (seed) our algorithm starts with. In other words, this algorithm, as with most PRNGs, is deterministic, meaning that a future element in our random number sequence can easily be computed by a previous element. This also means that if the same seed is passed into the same PRNG, it will alawys give us the same random sequence, solving the problem of reproducibility that we were facing with TRNGs.

Well, this is perfect! Seems like with PRNGs, we have the best of both worlds. Not only do we have the random-like behavior of TRNGs, but we get to utilize the reproducibility of PRNGs to do actual program development and testing. Right? While that's true, there are some unforeseen circumstances with using PRNGs which could lead to some unexpected behavior if you're not careful.

The main problem with PRNGs is that the random sequence they generate will eventually repeat. As we have already discovered, the output sequence of PRNGs depends solely on the input value, so if we ever get to a point where our PRNG outputs a value that we have already previously passed in as an input, that means that we will end up passing in the same input value as one from before. In short, this means that our output sequence will begin to repeat itself exactly.

As one might intuitively guess, the number of random outputs we can generate before our output sequence begins to repeat is directly proportional to the length of the seed value we pass in. In other words, the larger the seed value we pass in, the more unique numbers we will get in our random output sequence before it begins to repeat. For example, with a two-digit seed value, our Middle Squares method can produce at most 100 unique random outputs in its random sequence before repeating itself. If, instead, a four-digit seed value was used, then our algorithm can produce 10000 unique random outputs in its sequence before we see repetition!

A Summary

Hopefully you now have a better understanding of the different ways in which random number generation is done in computer programs and simulations today.

To recap, we have systems called True Random Number Generators, which actually sample physical phenomenon and fluctuations and, from those measurements, generate real random sequences. While these are truly random, their disadvantage lies in the fact that they lack reproducibility, and thus have little application when it comes to program/simulation testing and development.

On the other hand, Pseudo-Random Number Generators solve for this reproducibility problem by introducing seed values and algorithms which compute near-random sequences by running computations on this seed value and subsequent input values. The problem we discovered with these PRNGs is that their near-random sequences begin to repeat when an input value is repeated, which means that their randomness is limited by the initial seed value that is passed in.

Hopefully the following chart will help elucidate any remaining questions:

Characteristic	Pseudo-Random Number Generators	True Random Number Generators
Efficiency	Excellent	Poor
Determinism	Determinstic	Nondeterministic
Periodicity	Periodic	Aperiodic

In sum, both types of generators have their own advantages and disadvantages. So which one to use? Perhaps we ought to pick at random.

References

1. https://www.khanacademy.org/computing/computer-science/cryptography/crypt/v/random-vs-pseudorandom-number-generators
2. https://www.random.org/randomness/
3. https://en.wikipedia.org/wiki/Random_number_generation#.22True.22_vs._pseudo-random_numbers

4. https://www.howtogeek.com/183051/htg-explains-how-computers-generate-random-numbers/
5. https://www.researchgate.net/post/Is_True_Random_Number_Generation_possible
6. http://mathworld.wolfram.com/PseudorandomNumber.html
/pseudorandomNumberGen.html