Fast Random Integer Generation in an Interval

Lukas Geis ✉
Goethe Universität Frankfurt am Main

Abstract

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Praesent convallis orci arcu, eu mollis dolor. Aliquam eleifend suscipit lacinia. Maecenas quam mi, porta ut lacinia sed, convallis ac dui. Lorem ipsum dolor sit amet, consectetur adipiscing elit. Suspendisse potenti.

2012 ACM Subject Classification Mathematics of computing → Random number generation

Keywords and phrases Random Number Generation, Rejection Method, Randomized Algorithms, Algorithm Engineering

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

Supplementary Material Sources of implementations and experiments can be accessed at `https://github.com/lukasgeis/FRIGiaI-Review`

1 Introduction

1.1 Motivation

The study of randomized algorithms goes as far back as the 1910s [15]. While initially small, this study is now scientifically pursued in almost all parts of computer science. Not only in theory, but also in practice, randomized algorithms gain even more importance today to not only solve specific algorithmic problems but also to create large, scalable, and unbiased data sets that can be used in a variety of fields.
The core of such algorithms is most often the pseudo-random number generator (PRNG). There are many efficient PRNGs such as XorShift [23], MersenneTwister [24], linear congruential generators [19, 26, 27], and younger generators such as the PCG-Family [25] and ChaCha [5]. These generators produce 32-bit or 64-bit words which we can interpret as unsigned integers in $[0, 2^{32})$ and $[0, 2^{64})$ respectively. However, for most applications we do not want to have such a large range of possible random values but instead want to confine the range to a given interval $[a, b]$. This can not only be used to uniformly sample an element from an array but also for more complex algorithms

- The Fisher-Yates random shuffle [17] (see Algorithm 1) uniformly permutes an array of n elements in time $\Theta(n)$. We have to draw a random integer in an interval for every shuffled element.
- A random graph model describes a distribution over a (parametrized) family of graphs. There exist an uncountable number of such models and almost all of them require some sort of integer sampling in an interval. For example, the $\mathcal{G}(n, m)$ model [10] uniformly samples a (directed) graph with n nodes and m edges. A possible algorithm for this problem is Reservoir-Sampling [28](see Algorithm 2) which is a general technique that can sample k elements without replacement from an input stream. Another model is the BA model [3, 4] which iteratively builds the graph by adding nodes with edges to previously added nodes. Both models require sampling of an integer in an interval for each of its edges (or more).
- An Alias-Table [29] can be used to simulate static discrete distributions in practice. After creating and initializing the table, we can sample in (expected) time $\Theta(1)$ by drawing a uniform random row and offset in each round. Hence, we need to sample an integer uniformly at random in $[0, n)$ where n is the number of rows in the table.

Algorithm 1 Fisher-Yates-Shuffle: uniformly shuffle an array of size n

Require: source of uniformly-distributed random integers in $[0, i]$ for parameter i
Require: array A of size n
for $i = n-1, \ldots, 1$ **do**
  $j \leftarrow$ random integer in $[0, i]$
  swap $A[i]$ and $A[j]$

Algorithm 2 Sampling of (directed) $\mathcal{G}(n, m)$ graphs using Reservoir Sampling

Require: source of uniformly-distributed random integers in $[0, i]$ for parameter i
Require: number of nodes n and number of edges $m \leq n^2$
$E \leftarrow$ array (of edges) of size m
for $i = 0, \ldots, m-1$ **do**
  $E[i] \leftarrow (i \div n, i \mod n)$
for $i = m, \ldots, n^2$ **do**
  $j \leftarrow$ random integer in $[0, i]$
  if $j < m$ **then**
    $E[j] \leftarrow (i \div n, i \mod n)$
return E

1.2 Preliminaries

We denote by $x \div y$ the integer division operation, namely $\lfloor x/y \rfloor$, and the remainder operation by $x \bmod y \equiv x - (x \div y)y$. Divisions by a power of two can be efficiently implemented by Bit-RightShift denoted by $\gg$: $x \div 2^W \equiv x \gg W$. Similarly, we denote a Bit-LeftShift by $\ll$. Additionally, we can efficiently compute the remainder of a division py a power of two by a logical And-operation, denoted by &: $x \bmod 2^W \equiv x \ \& \ (2^W - 1)$. Typically, PRNGs generate uniform random numbers in an interval $[0, 2^W)$ where $W \in \{32, 64\}$, namely a uniform random W-bit number.

Since we can map every integer in an interval $[a, b)$ to the interval $[0, b-a)$ by subtracting a, we want to generate a uniform random integer in an interval $[0, n)$ for a given n. If we map all integers x in $[0, 2^W)$ with the function $x \bmod n$, we get the following sequence:

$$\overbrace{\underbrace{\overbrace{0, 1, \ldots, n-1}^{n \text{ values}}, \overbrace{0, 1, \ldots, n-1}^{n \text{ values}}, \ldots, \overbrace{0, 1, \ldots, n-1}^{n \text{ values}}}_{\left(2^W \div n\right) \cdot n \text{ values}}, \underbrace{0, 1, \ldots, \left(2^W \bmod n\right) - 1}_{2^W \bmod n \text{ values}}}^{2^W \text{ values}} \tag{1}$$

This motivates the following lemma:

▶ **Lemma 1** (Remainder Multiples). *For given integers $a, b, n > 0$ with $a < b$, there exist exactly $(b-a) \div n$ integers $x \in [a, b)$ for every $0 \leq r < n$ such that $x \bmod n \equiv r$ whenever n divides $b - a$.*

2 Existing Techniques

In this section, we will review a number of different techniques used for sampling a uniform random integer in a given range $[0, n)$. Throughout this section, we use `rand()` to indicate the drawing of a uniform random W-bit number.

2.1 Biased

While our goal is to generate *uniform* random numbers, there are also some techniques that generate biased - thus not uniform - random integers in a given range in exchange for much less complexity.

2.1.1 Modulo Reduction

The most common and under-egineered algorithm is the modulo reduction, namely

$$\texttt{rand()} \bmod n$$

Clearly, this method returns a random integer in the interval $[0, n)$. However, as seen in equation (1), for n that do not perfectly divide 2^W, we have a *leftover* interval at the end that is biased. Thus, not all integers in $[0, n)$ are equally likely and this reduction does not produce uniform numbers.

Furthermore, while this approach is probably the most intuitive and simplistic, it also illustrates why drawing numbers within a range is typically slow. We require a remainder operation which can be implemented through the division (`div`) instruction. For 32-bit registers, this instruction has a latency of 26 cycles and for 64-bit registers, the latency lies within the range of 35 to 88 cycles [11].

2.1.2 Floating-Point Conversion

A common method to not rely on the costly `div` instruction is using a fixed-point floating-point conversion which consists of the following steps:

1. Draw $x =$ `rand()` and convert x to a floating-point number.
2. Multiply x with 2^{-W} to generate a floating point number $y \in [0, 1)$.
3. Multiply y with n and round it down to an integer before returning the result.

Again, this method clearly generates a random integer in $[0, n)$ because y is less than 1 and we multiply with n before rounding down. However, in the typical floating-point standard (IEEE 754) with 32 bits, we have a mantisse of 24 bits and hence can at best represent all values in $[0, 2^{24})$ divided by 2^{24}. Thus, we do not get the same accuracy as we would have with a 32-bit number and hence can not generate all numbers in $[0, n)$ if $n > 2^{24}$. A solution would be to instead use double precision floating-point numbers that have a mantisse of 53 bits and thus can represent all values in $[0, 2^{53})$ divided by 2^{53}. But this again is not enough if $W = 64$ where we require a mantisse of at least 64 bits which is not accessible on all systems (long double). Furthermore, the conversion also comes at a price: the corresponding instructions have a latency of at least 6 cycles on Skylake processors [11].

2.1.3 Multiply-and-Shift

We can adjust the prior floating-point conversion method to only use fixed-point arithmetic rather than floating-point arithmetic:

$$(\texttt{rand()} \cdot n) \gg W$$

where $\gg W$ is equivalent to dividing by 2^W as mentioned in section 1. Clearly, this method and the floating-point method before work the same way, hence creating the same bias - but here, we multiply first before dividing by 2^W [20]. On the other hand, by multiplying with n first, we get a number in $[0, n \cdot 2^W)$ instead. If we now divide by 2^W, we map all multiples of n in $[0, 2^W)$ to 0, all multiples of n in $[2^W, 2 \cdot 2^W)$ to 1, and so on such that we map all multiples of n in $[i \cdot 2^W, (i+1) \cdot 2^W)$ (the $(i+1)^{\text{th}}$ 2^W-interval) to i.

Since we might get a number much bigger than 2^W and thus need a new representation with more than W bits. For $W = 32$, we simply switch to a 64-bit number, but for $W = 64$, we have to switch to a 128-bit number which is inherently slower than 32-bit or 64-bit numbers. Alternatively, for $W = 32$, a good compiler could compile this into a 32-bit `mult` instruction which generates a 32-bit number for the first 32 bits and a 32-bit number for the last 32 bits.

2.2 Unbiased

While all prior approaches are quite simplistic and always require only a handful of instructions, they also generate a bias which can vary depending on n. But since we want to generate a *uniform* random number in $[0, n)$, we somehow need to get rid of such bias by incorporating a fail-safe: rejection sampling.

2.2.1 OpenBSD

We somehow want to get rid of the bias generated by the last interval in equation (1). The key idea is that this interval does not have to be on the right. Instead of rejecting all values generated by a random $x \geq 2^W - (2^W \bmod n)$, we could also reject all values generated by a random $x < 2^W \bmod n$ for uniformity since the interval $[2^W \bmod n, 2^W)$ has size $2^W - (2^W \bmod n)$ which is divisible by n. Due to Lemma 1, for every $r \in [0, n)$, we have exactly $(b - a) \div n$ numbers in $[2^W \bmod n, 2^W)$ with remainder r, thus having a uniform distribution over $[0, n)$ when computing the remainder. This yields Algorithm 4 which stems from the C standard library in OpenBSD and macOS (`arc4random_uniform`), and has been adopted by the Go language (`Int63n, Int31n`) with slight implementation differences [2] as well as the GNU C++ standard library [12].

Algorithm 3 The OpenBSD algorithm.

Require: source of uniformly-distributed random integers in $[0, 2^W)$ given by `rand()`
Require: size of target interval given by n

```
t ← (2^W − n) mod n                    /* (2^W − n) mod n ≡ 2^W mod n */
x ← rand()
while x < t do
    x ← rand()
return x mod n
```

In comparison to the biased techniques, we now may have to sample more than one random integer in $[0, 2^W)$ to generate an integer in $[0, n)$. Note that the number of random W-bit numbers drawn follows a geometric distribution with a success probability of $p = 1 - \frac{2^W \bmod n}{2^W}$. Thus on average, we need $\frac{1}{p}$ random W-bit numbers which is less than 2 because $p > \frac{1}{2}$, irrespective of n. We also always require two remainder operations: one to compute the

threshold that determines whether we reject a drawn random number and one to use the modulo reduction to generate the final number.

2.2.2 Java

Since we expect remainder operations to have high latency, always needing to compute two remainders can be quite costly. Hence, the Java language, in its `Random class`, uses a different approach. The key insight is that when we draw a random integer $x \in [0, 2^W)$ and compute its remainder $r = x \bmod n$, we can map all integers in $[0, 2^W)$ to a multiple of n by computing $x - r$. Then, every number in the last interval in equation (1) gets mapped to the last multiple y of n which is not greater than 2^W. Note that this implies $y > 2^W - n$. Every other integer is mapped to a different (and thus smaller) multiple z of n for which $z \leq y - n$ and thus $z \leq 2^W - n$. Consequently, if we reject x if $x - r > 2^W - n$ and otherwise return r, we get a unbiased generator. The details can be seen in Algorithm 5.

Algorithm 4 The Java algorithm.

Require: source of uniformly-distributed random integers in $[0, 2^W)$ given by `rand()`
Require: size of target interval given by n
1 $x \leftarrow$ `rand()`
2 $r \leftarrow x \bmod n$
3 **while** $x - r > 2^W - n$ **do**
4 $\quad x \leftarrow$ `rand()`
5 $\quad r \leftarrow x \bmod n$
6 **return** r

Again, the number of random draws of W-bit numbers follows a geometric distrbution with $p = 1 - \frac{2^W \bmod n}{2^W}$ and we expect to draw $\frac{1}{p} < 2$ random W-bit numbers before returning. However, this time, for every drawn random number, we also need to compute a remainder, hence we also expect $\frac{1}{p} < 2$ remainder operations. For small n, this is great because with high probability, we only need to compute one remainder operation in expectation as opposed to two.

2.2.3 Fast-Dice-Roller

For all prior unbiased algorithms, we need to use at least one remainder operation to debias the algorithm. Jérémie Lumbroso however devised an algorithm for sampling uniform random integers in $[0, n)$ for a given n by iteratively building up the final number bit-by-bit [22]. In comparison to previous algorithms, we do not use the whole W bits generated by the PRNG but instead only use one bit at a time building up the final number consisting of $\lceil \log_2 n \rceil + 1$ bits. If we, at some point, overshoot and generate a number bigger than n (but smaller than $2n$) we take a step back by subtracting n and rebuilding the necessary bits. The algorithmic details can be seen in Algorithm 6.

▶ **Lemma 2** (Correctness of Fast-Dice-Roller [22]). *The Fast-Dice-Roller algorithms returns a uniformly distributed random integer in* $[0, n)$ *and terminates in expected time* $\Theta(\log n)$.

Proof. Consider the following loop invariant: x is uniformly distributed over $[0, b)$. At the beginning, this is trivially true as b is 1 the only value x can take (and takes) is 0. In line 4 we double b, but also double x in line 5 before adding a parity bit which ensures uniformity in

Algorithm 5 The Fast-Dice-Roller algorithm.

```
   Require: source of a uniformly-distributed random bit given by flip()
   Require: size of target interval given by n
1  b ← 1
2  x ← 0
3  while true do
4      b ← 2 · b
5      x ← 2 · x + flip()
6      if b ≥ n then
7          if x < n then
8              return x
9          else
10             b ← b − n
11             x ← x − n
```

the enlarged new range. Since we reach line 10 and 11 only when $x \geq n$, x must be uniformly distributed in $[n, b)$ at this point. Since we subtract n from both b and x, we simply shift the range from $[n, b)$ to $[0, b - n)$ maintaining uniformity. As x is always uniformly distributed in $[0, b)$ due to this loop invariant, when we return x, it must also be uniformly distributed over $[0, n)$.

For the proof of termination time, consider a less efficient variant of this algorithms that throws away all prior sampled bits when rejecting in line 9-11. Then, in each iteration, we have $\Theta(\log n)$ steps before the condition in line 6 is *true*. Since we double b in each iteration, we know that $b < 2n$ and hence, due to the previous loop invariant, we accept with probability $p = {}^{n}/_{2^{\lfloor \log_2 n \rfloor + 1}} > {}^{1}/_{2}$, thus proving the lemma. ◀

Clearly, while this algorithm completely avoids any remainder or division operation, for most n, it is too slow. In comparison, every prior algorithm needs constant time or expected constant time to generate a uniform random number in $[0, n)$. Fast-Dice-Roller, on the other hand, needs expected logarithmic time for sampling which is catastrophical for bigger n - a big price to pay for avoiding division (and multiplication if line 4 and 5 are implemented using a BitShift).

2.2.4 Bitmask

The main reason, Fast-Dice-Roller is highly impractical, is due to the fact that we build up our random number bit-by-bit instead of drawing all required bits at once. This motivates the following algorithm (see Algorithm 7) that was adopted by Apple in the macOS Sierra release when they made their own revision to the code of `arc4random_uniform` [1].

The key idea is to first compute k such that k is the smallest value with $2^k \geq n$. We then can use a Bitmask $m = 2^k - 1$ to only take the necessary k bits from our generated W-bit number and use rejection sampling to generate a uniform random number in $[0, n)$. This works in expected constant time since we accept with probability $p = {}^{n}/_{2^k} > {}^{1}/_{2}$.

Bitmask completely avoids any type of division operation. Instead, we have to compute the number of leading zeros in line 1. For C and C++, the GCC-Compiler provides the `__builtin_clz()` method to compute just that [13]. In Rust, we can use the `leading_zeros()`

Algorithm 6 The Bitmask algorithm.

Require: source of uniformly-distributed random integers in $[0, 2^W)$ given by `rand()`
Require: size of target interval given by n
1 $z \leftarrow$ number of leading zeros in the binary representation of n
2 $m \leftarrow 1 \ll (W - z)$ `/*` $1 \ll (W - z) \equiv 2^{W-z}$ `*/`
3 $m \leftarrow m - 1$
4 $x \leftarrow$ `rand()` & m
5 **while** $x \geq n$ **do**
6 $\quad x \leftarrow$ `rand()` & m
7 **return** x

method of the standard library [14]. These methods however can perform vastly different depending on the underlying architecture. For processors which have an inbuilt `clz` instruction such as ARM [9], the compiler could translate this line into such an assembly instruction. In systems without direct support, one might to rely on more algorithmic implementations such as [6].

3 Nearly Divisionless Unbiased Sampling

While completely avoiding division seems possible with the Fast-Dice-Roller and the Bitmask algorithm, it also comes at a cost: logarithmic time or a `clz` operation that might be even more expensive than a division on some systems. Hence, Lemire proposed an unbiased algorithm that uses the approach of the Multiply-and-Shift algorithm coupled with rejection sampling to remove bias. The algorithm (see Algorithm 8) uses one divison operation at most and may also use no division operation at all. It is the main result of the underlying paper of this report [21].

Algorithm 7 The Lemire algorithm.

Require: source of uniformly-distributed random integers in $[0, 2^W)$ given by `rand()`
Require: size of target interval given by n
1 $x \leftarrow$ `rand()`
2 $m \leftarrow x \cdot n$
3 $l \leftarrow m \bmod 2^W$
4 **if** $l < n$ **then**
5 $\quad t \leftarrow (2^W - n) \bmod n$ `/*` $(2^W - n) \bmod n \equiv 2^W \bmod n$ `*/`
6 $\quad$ **while** $l < t$ **do**
7 $\quad\quad x \leftarrow$ `rand()`
8 $\quad\quad m \leftarrow x \cdot n$
9 $\quad\quad l \leftarrow m \bmod 2^W$
10 **return** $m \div 2^W$

Note that the remainder operation in line 3 and 9 can be computed by a logical And operation and the division operation in line 10 by a Right-BitShift as mentioned in Section 1 Thus, the only *real* division operation is the remainder operation in line 5.

Since we use the same mapping as in Multiply-and-Shift, we remind the reader of the $(i+1)^{\text{th}}$ 2^W-interval introduced earlier: $[i \cdot 2^W, (i+1) \cdot 2^W)$. By Lemma 1, there exist exactly

$\lfloor 2^W/n \rfloor$ multiples of n in the restricted $(i+1)^{\text{th}}$ 2^W-interval $[i \cdot 2^W + (2^W \bmod n), (i+1) \cdot 2^W)$ as n perfectly divides $(i+1) \cdot 2^W - (i \cdot 2^W + (2^W \bmod n)) = 2^W - (2^W \bmod n)$. Thus, we can remove bias if we reject all multiples of n in the interval $[i \cdot 2^W, i \cdot 2^W + (2^W \bmod n))$ for every i.

The above algorithm does just that in lines 5 and 6. But it also makes use of the following observation: since $t = 2^W \bmod n$, we have $t < n$ and thus if we have $l \geq n$, we also know that $l \geq t$. In this case we know that we do not have to reject the current value without computing t in the first place. Thus, we can completely skip the division operation with probability $p = \frac{2^W - n}{2^W} = 1 - \frac{n}{2^W}$.

4 Conclusion

We have seen several different algorithms that can be used to sample a random integer in an interval $[0, n)$. The results are summarized in Table 1. As we introduce rejection sampling for unbiased algorithms, runtimes for those are taken as expected constant or logarithmic time, while runtimes for biased algorithms are constant overall.

	expected number of integer division operations	maximal number of integer division operations	Unbiased?	(Expected) Runtime
Modulo Reduction	1	1	✗	$\Theta(1)$
Floating-Point Conversion	0	0	✗	$\Theta(1)$
Multiply-and-Shift	0	0	✗	$\Theta(1)$
OpenBSD	2	2	✓	$\Theta(1)$
Java	$\frac{2^W}{2^W - (2^W \bmod n)}$	∞	✓	$\Theta(1)$
Fast-Dice-Roller	0	0	✓	$\Theta(\log n)$
Bitmask	0	0	✓	$\Theta(1)$
Lemire	$\frac{n}{2^W}$	1	✓	$\Theta(1)$

Table 1 Overview over different sampling algorithms: expected/maximal number of division operations, unbiased or biased, (expected) runtime

While the bias is usually insignificant and negligible in most use-cases, if we sample millions of times, it can easily become apparent and strongly distort the final dataset depending on the algorithm. Hence, it is important to generate unbiased random integers at the cost of a little execution time. Common software libraries seem to rely on algorithms that always require at least one division, if not two or more which is inefficient on most processors. While Bitmask has in theory zero division operations, the required `clz` operation can be as inefficient as the `div` instruction. Hence, the algorithm introduced by Lemire is a perfect fit for modern x64 processors as for small n, we require no division (or equally bad) operation at all and also bound the maximum number of divisions to at most one.

For different technologies however, support for the computation of the full multiplication might be lacking such as in Graphics Processing Units (GPUs) [18] or DRAM Processing Units (DPUs) used for In-Memory-Processing in the UPMEM architecture [7, 8]. In the

UPMEM-PIM toolchain, for example, an `clz` operation seems to greatly outshine a `div` operation [16].

References

1 Apple. Arc4 random number generator for openbsd. https://opensource.apple.com/source/Libc/Libc-1158.50.2/gen/FreeBSD/arc4random.c, 2008. Last accessed: 2023-12-03.

2 The Go authors. Package rand implements pseudo-random number generators. https://github.com/golang/go/blob/master/src/math/rand/rand.go, 2017. Last accessed: 2023-12-03.

3 Albert-László Barabási. *Network Science Book*. Network Science, 2014.

4 Albert-László Barabási and Réka Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, 1999. URL: https://www.science.org/doi/abs/10.1126/science.286.5439.509, arXiv:https://www.science.org/doi/pdf/10.1126/science.286.5439.509, doi:10.1126/science.286.5439.509.

5 Daniel J. Bernstein. Chacha, a variant of salsa20. Technical report, The University of Illinois at Chicago, 2008.

6 Stephen Cleary. Implementing gcc's builtin functions. https://blog.stephencleary.com/2010/10/implementing-gccs-builtin-functions.html, 2010. Last accessed: 2023-12-03.

7 The UPMEM SAS company. The upmem architecture. https://www.upmem.com/. Last accessed: 2023-12-04.

8 The UPMEM SAS company. User manual — upmem dpu sdk 2023.2.0 documentation. https://sdk.upmem.com/2023.2.0/#, 2023. Last accessed: 2023-12-04.

9 The Arm Documentation. Arm a-profile a64 instruction set architecture. https://developer.arm.com/documentation/ddi0602/2023-09/Base-Instructions/CLZ--Count-Leading-Zeros-?lang=en, 2023. Last accessed: 2023-12-03.

10 Paul L. Erdos and Alfréd Rényi. On random graphs. i. *Publicationes Mathematicae Debrecen*, 2022. URL: https://api.semanticscholar.org/CorpusID:253789267.

11 Agner Fog. Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for intel, amd and via cpus. Technical Report. Copenhagen University College of Engineering, Copenhagen, Denmark., 2016. Pages 278+; Last accessed: 2023-12-03. URL: https://www.agner.org/optimize/instruction_tables.pdf.

12 Free Software Foundation. The gnu c++ library api reference. https://gcc.gnu.org/onlinedocs/libstdc++/api.html, 2017. Last accessed: 2023-12-03.

13 Free Software Foundation. The gnu c++ library api reference. https://gcc.gnu.org/onlinedocs/gcc/Other-Builtins.html, 2017. Last accessed: 2023-12-03.

14 The Rust Foundation. The rust standard library. https://doc.rust-lang.org/std/primitive.u32.html#method.leading_zeros, 2017. Last accessed: 2023-12-03.

15 Walter Gautschi, editor. *Mathematics of Computation 1943–1993: a half-century of computational mathematics*, pages 504, "Perhaps Pocklington also deserves credit as the inventor of the randomized algorithm". American Mathematical Society, Providence, 1994.

16 Lukas Geis. Random number generation in the pim-architecture. https://github.com/lukasgeis/upmem-rng/tree/main, 2023. Last accessed: 2023-12-04.

17 Donald E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley, Boston, third edition, 1997.

18 W. B. Langdon. A fast high quality pseudo random number generator for nvidia cuda. In *Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference: Late Breaking Papers*, GECCO '09, pages 2511–2514, New York, NY, USA, 2009. Association for Computing Machinery. doi:10.1145/1570256.1570353.

19 Pierre L'Ecuyer. Tables of linear congruential generators of different sizes and good lattice structure. *Math. Comput.*, 68:249–260, 01 1999. doi:10.1090/S0025-5718-99-00996-5.

20 Daniel Lemire. A fast alternative to the modulo reduction. Online Blog, 2016. Last accessed: 2023-12-03. URL: https://lemire.me/blog/2016/06/27/a-fast-alternative-to-the-modulo-reduction/.

21 Daniel Lemire. Fast random integer generation in an interval. *ACM Transactions on Modeling and Computer Simulation*, 29(1):1–12, January 2019. URL: http://dx.doi.org/10.1145/3230636, doi:10.1145/3230636.

22 Jérémie Lumbroso. Optimal discrete uniform generation from coin flips, and applications, 2013. arXiv:1304.1916.

23 George Marsaglia. Xorshift rngs. *Journal of Statistical Software*, 8(14):1–6, 2003. URL: https://www.jstatsoft.org/index.php/jss/article/view/v008i14, doi:10.18637/jss.v008.i14.

24 Makoto Matsumoto and Takuji Nishimura. Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. Model. Comput. Simul.*, 8(1):3–30, jan 1998. doi:10.1145/272991.272995.

25 Melissa E. O'Neill. Pcg: A family of simple fast space-efficient statistically good algorithms for random number generation. Technical Report HMC-CS-2014-0905, Harvey Mudd College, Claremont, CA, September 2014.

26 S. K. Park and K. W. Miller. Random number generators: Good ones are hard to find. *Commun. ACM*, 31(10):1192–1201, oct 1988. doi:10.1145/63039.63042.

27 W. H. Payne, J. R. Rabung, and T. P. Bogyo. Coding the lehmer pseudo-random number generator. *Commun. ACM*, 12(2):85–86, feb 1969. doi:10.1145/362848.362860.

28 Jeffrey S. Vitter. Random sampling with a reservoir. *ACM Trans. Math. Softw.*, 11(1):37–57, mar 1985. doi:10.1145/3147.3165.

29 Alastair J. Walker. An efficient method for generating discrete random variables with general distributions. *ACM Trans. Math. Softw.*, 3(3):253–256, sep 1977. doi:10.1145/355744.355749.