Fast batched dice rolls

Nevin Brackett-Rozinsky

March 2024

Abstract

We describe, and prove the correctness of, an algorithm to generate multiple independent uniformly random bounded integers from a single uniformly random binary word. In the common case, our method uses one multiplication and zero division operations per value produced. As a result, we can efficiently roll several dice with only one call to a random number generator. This has many potential applications, and we focus in particular on shuffling arrays.

1 Introduction

Random numbers are useful in many computer programs. Programming languages generally provide methods to generate uniformly random integers in the range $[0, 2^L)$ for some L, commonly 64 or 32. We refer to such numbers as L-bit random words, and the functions which produce them as random number generators (RNGs).

Many applications require uniformly random integers from other bounded ranges, such as $[0, b)$ for some b, for example when rolling dice or choosing random elements from an array. This is a discrete uniform distribution, and we refer to these numbers as bounded random integers, or dice rolls. We will focus on the case where $0 < b \leq 2^L$, and on applications which require multiple dice rolls, such as shuffling arrays.

Our main result is in section 4, where we present and prove the correctness of an algorithm to generate multiple independent bounded random integers from a single RNG call, using in the common case only one multiplication and zero division operations per die roll. The method is based on an existing algorithm due to Lemire that we summarize in section 3, and our proof uses mixed-radix notation as described in section 2.

After establishing our main result we then develop further optimizations, including a vectorized SIMD (single instruction, multiple data) implementation that shuffles up to 17 elements with exactly two multiplications, zero divisions, and an average of 1.0054 calls to a 64-bit RNG.

1.1 Mathematical notation

We use "$\div$" to denote integer division: $a \div b = \lfloor a/b \rfloor$ is the greatest integer less than or equal to a/b. We use "mod" to denote the Euclidean remainder: $(a \bmod b) = a - b \cdot (a \div b)$. We use only non-negative integers, which implies $0 \leq (a \bmod b) < b$. Division is much slower than multiplication on modern computers, unless the divisor is a power of 2, so these operations should generally be avoided when possible.

We use "$\otimes$" to denote full-width multiplication: $a \otimes b = (x, y)$ means x and y are integers such that $2^L x + y = ab$ and $0 \leq y < 2^L$. The high part of the product is $x = ab \div 2^L$ and the low part is $y = (ab \bmod 2^L)$. On many computer systems full-width multiplication is a fast intrinsic operation.

We use pi notation "$\prod$" to denote products, and we often omit the bounds when they can be inferred from context. Thus if b_i is defined for each i from 1 through k, then $\prod b_i = b_1 b_2 \cdots b_k$. We also use sigma notation "$\sum$" to denote sums, and an underlined superscript to denote the falling factorial: $n^{\underline{k}} = n!/(n-k)! = n(n-1)\cdots(n-(k-1))$.

2 Mixed-radix numbers

We use mixed-radix notation in the proof of our main result. Mixed-radix notation is a positional number system in which each digit position has its own base. This contrasts with decimal notation where every digit is in base 10, or binary where every digit is in base 2. Mixed-radix numbers are allowed, but not required, to have a different base for each digit. Each base is a positive integer, and each digit is a non-negative integer less than its base.

We denote both the bases and digits of a mixed-radix number with ordered tuples beginning with the most-significant digit, and use context to distinguish them. For example, a 2-digit mixed-radix number written in base (b_1, b_2) whose digits are (a_1, a_2) represents the value $a = a_1 b_2 + a_2$. A k-digit mixed-radix number in base $(b_1,\ b_2,\ \ldots,\ b_k)$ can represent all the integers from 0 through $b-1$, where $b = b_1 b_2 \cdots b_k$, and this representation is unique. If its digits are $(a_1,\ a_2,\ \ldots,\ a_k)$, then it represents the value:

$$a = \sum_{i=1}^{k} \left(a_i \prod_{j>i}^{k} b_j \right) = a_1(b_2 b_3 \cdots b_k) + a_2(b_3 b_4 \cdots b_k) + \cdots + a_{k-1} b_k + a_k$$

Given an integer a in the range $[0, b)$, its mixed-radix representation can be obtained by taking the remainders of successive quotients when dividing by the b_i in reverse order.

There is a direct correspondence between mixed-radix notation and multi-dimensional arrays, which may help with intuition. A k-dimensional array of size $b_1 \times b_2 \times \cdots \times b_k$ can be indexed either by coordinates $(a_1,\ a_2,\ \ldots,\ a_k)$ or by a linear position in the underlying storage. That position is the value of the coordinate vector interpreted as a mixed-radix number in base $(b_1,\ b_2,\ \ldots,\ b_k)$. For example, a 2-digit number in base (b_1, b_2) corresponds to a 2-dimensional array with b_1 rows and b_2 columns, where the first digit specifies the row and the second digit the column.

2.1 Independence of digits

A finite collection of discrete random variables $\vec{a} = (a_1,\ a_2,\ \ldots,\ a_k)$ is called **independent** if, for every possible outcome $\vec{x} = (x_1,\ x_2,\ \ldots,\ x_k)$ of all their values, the probability of those values all occurring is equal to the product of the probabilities for each value to occur individually. In other words, if $\Pr(\vec{a} = \vec{x}) = \prod \Pr(a_i = x_i)$ for every $\vec{x}$, then the a_i are independent.

Lemma 1. If a uniformly random integer a in $[0, b)$ is written as a mixed-radix number in base $(b_1,\ b_2,\ \ldots,\ b_k)$, where $b = \prod b_i$, then its digits $(a_1,\ a_2,\ \ldots,\ a_k)$ are independent and uniformly random integers in the ranges $0 \le a_i < b_i$.

Proof. The digits are clearly integers in those ranges, by the definition of mixed-radix notation. They are also uniform, because every possible sequence of digits is produced by exactly one integer in $[0, b)$, all of which are equally likely. This implies $\Pr(a_i = x_i) = 1/b_i$ for each i and every integer x_i in $[0, b_i)$.

We also have $\Pr(a = x) = 1/b$ for every integer x in $[0, b)$ by the uniformity of a. Since $b = \prod b_i$, it follows that $\Pr(a = x) = \prod \Pr(a_i = x_i)$, where the x_i are the mixed-radix digits of x. By the

uniqueness of mixed-radix representation, $a = x$ if and only if every $a_i = x_i$, hence the a_i are independent. □

In terms of multidimensional arrays, lemma 1 says that when a linear index is chosen uniformly at random, the coordinates of the location it represents are themselves independent and uniformly random in their respective ranges. In the simple case of 2D arrays, choosing a random square on a rectangular grid is equivalent to choosing a random row and column independently.

3 Existing algorithms

There are many ways to generate bounded random integers from random bits, and it is non-trivial to do so efficiently. Several widely-used algorithms are described in Lemire (2019), including a then-novel strategy to avoid expensive division operations that we will call **Lemire's method**, which has now been adopted by several major programming languages. It works as follows:

Let r be a uniformly random integer in $[0, 2^L)$, and $[0, b)$ the desired target range with $0 < b \leq 2^L$. Perform the full-width multiplication $b \otimes r = (x, y)$. If $y \geq (2^L \bmod b)$, which we call **Lemire's criterion**, then x is uniformly random in $[0, b)$. Otherwise, try again with a new r.

In the common case, Lemire's method uses one L-bit random word and one multiplication per die roll. Note that $(2^L \bmod b) < b$, so if $y \geq b$ then also $y \geq (2^L \bmod b)$. This means $(2^L \bmod b)$ is only needed when $y < b$, which is rare for b much smaller than 2^L, so the division is usually avoided. It can be computed when needed, at most once per die roll, using $((2^L - b) \bmod b)$.

Another approach, which uses random bits one at a time rather than entire L-bit random words, is described in Lumbroso (2013) and given the name "**fast dice roller**". It works by setting $x = 0$ and $y = 1$, then repeatedly doubling y and appending a random bit to x. Whenever $y \geq b$, if $x < b$ then return x, otherwise subtract b from both x and y and continue.

The fast dice roller algorithm uses on average fewer random bits per die roll than other methods. Although it does not use division, it does involve hard-to-predict branches which can be slow. By generating numbers in batches, the fast dice roller algorithm can approach the theoretical minimum number of random bits consumed per die roll, however this involves slow division operations.

3.1 Batched dice rolls

The traditional idea behind generating bounded random numbers in batches is that, to simulate rolling dice with b_1 and b_2 sides, a single die with $b_1 b_2$ sides is rolled instead. The resulting number is uniformly random in $[0, b_1 b_2)$, and the desired dice rolls can be obtained by taking its remainder and quotient upon dividing by b_1.

These values are independent and uniformly random integers in $[0, b_1)$ and $[0, b_2)$ respectively, and the method generalizes to more dice by taking successive remainders of quotients upon dividing by each b_i. Note that the effect is to compute the mixed-radix digits of the single larger die roll, with bases given by the b_i in reverse order. This uses fewer random bits than rolling each die separately, however it is only of theoretical interest because division operations are slow in practice.

Our approach works differently. Rather than extract the mixed-radix digits via division, we instead build them up with multiplication. Specifically, we extend Lemire's method to generate multiple bounded random integers from a single random word, with zero division operations in the common case.

4 Main result

Theorem 1. Let r_0 be an L-bit random word, meaning r_0 is a uniformly random integer in $[0, 2^L)$. Let $(b_1, b_2, \ldots, b_k)$ be positive integers with $b = \prod b_i \leq 2^L$. For each i from 1 to k, perform the full-width multiplication $b_i \otimes r_{i-1} = (a_i, r_i)$. If $r_k \geq (2^L \bmod b)$, then the a_i are independent and uniformly random integers in the ranges $0 \leq a_i < b_i$.

Proof. We will first show that when the conditions of the theorem are met, if the resulting values $(a_1, a_2, \ldots, a_k)$ are interpreted as the digits of a mixed-radix number a in base $(b_1, b_2, \ldots, b_k)$, then a is a uniformly random integer in $[0, b)$. Once this is established, we will invoke lemma 1.

For each i from 1 to k, the full-width product $b_i \otimes r_{i-1} = (a_i, r_i)$ means $b_i r_{i-1} = 2^L a_i + r_i$. Since both r_{i-1} and r_i are in $[0, 2^L)$, this implies $0 \leq a_i < b_i$. So each a_i is a valid mixed-radix digit for base b_i, and a is well-defined.

Let c_i be the value obtained by truncating a to its first i digits. In other words, c_i is the value represented by the mixed-radix number $(a_1, a_2, \ldots, a_i)$ in base $(b_1, b_2, \ldots, b_i)$. Thus $c_1 = a_1$, and $c_i = b_i c_{i-1} + a_i$ for $1 < i \leq k$.

We claim that $(b_1 b_2 \cdots b_i) \otimes r_0 = (c_i, r_i)$, and we prove it by finite induction on i. The claim is equivalent to $r_0(b_1 b_2 \cdots b_i) = 2^L c_i + r_i$, which is true for $i = 1$. If $1 < i \leq k$ we use the inductive hypothesis that the claim is true for $i-1$, in order to prove it for i. Thus:

$$\begin{aligned}
r_0(b_1 b_2 \cdots b_i) &= r_0(b_1 b_2 \cdots b_{i-1}) b_i \\
&= (2^L c_{i-1} + r_{i-1}) b_i && \text{(inductive hypothesis)} \\
&= 2^L (b_i c_{i-1} + a_i) + r_i && \text{(definition of } a_i \text{ and } r_i\text{)} \\
&= 2^L c_i + r_i && \text{(formula for } c_i\text{)}
\end{aligned}$$

This completes the induction and proves the claim for each i from 1 to k. We know that $c_k = a$ and $\prod b_i = b$, hence substituting $i \to k$ in the claim gives $b \otimes r_0 = (a, r_k)$. But this is just the full-width product of b with a random word, so we can apply Lemire's criterion. If $r_k \geq (2^L \bmod b)$ then, by Lemire, a is a uniformly random integer in $[0, b)$.

By lemma 1, since a is uniformly random in $[0, b)$, its mixed-radix digits in base $(b_1, b_2, \ldots, b_k)$ are independent and uniformly random in the ranges $[0, b_i)$. But those digits are $(a_1, a_2, \ldots, a_k)$, so the theorem is proved. Each a_i produced this way is a uniformly random integer in $[0, b_i)$, and the a_i are independent, provided that $r_k \geq (2^L \bmod b)$. □

5 Implementation

In some applications the values of b_i are known ahead of time, possibly even at compile time. In that case the value of $t = (2^L \bmod b)$ can be precomputed, and theorem 1 can be implemented succinctly as shown in algorithm 1.

Algorithm 1 — Batched dice rolls (known threshold)

Require: Source of uniformly random integers in $[0, 2^L)$
Require: Target intervals $[0, b_i)$ for i in $1 \dots k$, with $1 \leq \prod b_i \leq 2^L$
Require: The value $t = (2^L \bmod \prod b_i)$
Ensure: The a_i are independent and uniformly random in $[0, b_i)$

```
repeat
    r ← random integer in [0, 2^L)
    for i in 1...k do
        (a_i, r) ← b_i ⊗ r                    ▷ Full-width multiply
    end for
until r ≥ t
return (a_1, a_2, ..., a_k)
```

In other applications the values of b_i are not known ahead of time. In that case the threshold t must be computed when needed, which involves a division operation. It can be avoided when $r \geq b$, however computing b still requires some extra multiplications. This approach is shown in algorithm 2.

Algorithm 2 — Batched dice rolls (unknown threshold)

Require: Source of uniformly random integers in $[0, 2^L)$
Require: Target intervals $[0, b_i)$ for i in $1 \dots k$, with $1 \leq \prod b_i \leq 2^L$
Ensure: The a_i are independent and uniformly random in $[0, b_i)$

```
r ← random integer in [0, 2^L)
for i in 1...k do
    (a_i, r) ← b_i ⊗ r                        ▷ Full-width multiply
end for

b ← ∏ b_i
if r < b then
    t ← (2^L mod b)
    while r < t do
        r ← random integer in [0, 2^L)
        for i in 1...k do
            (a_i, r) ← b_i ⊗ r
        end for
    end while
end if

return (a_1, a_2, ..., a_k)
```

We must have $b \leq 2^L$ in order to use theorem 1, and for algorithm 2 we would prefer to have b at least an order of magnitude smaller than 2^L. If b is too close to 2^L then there is a high probability

of taking the slow path that needs to calculate t, and possibly having to reroll the whole batch of dice.

In many applications it is possible to bound the b_i in such a way that a value of u satisfying $b \leq u \ll 2^L$ is known ahead of time. This allows for a faster implementation that avoids computing b most of the time, by enclosing lines 6–15 of algorithm 2 within an "if $r < u$" block.

5.1 Loop dependencies

Whichever version of the algorithm is used, at its core is a loop containing a single full-width multiplication "$b_i \otimes r$". The value of b_i is known, but each pass through the loop computes the value of r that will be used for the next iteration. This constitutes a loop-carried dependency, and it means that each iteration must complete before the next can begin.

Modern processors generally have the ability to carry out more than one operation at a time. This is called instruction-level parallelism, and it allows several calculations to be "in flight" simultaneously. A given CPU might take four cycles to complete a multiplication, yet be able to initiate a new multiplication every cycle and work on up to four at once.

However, it is only possible to start a new multiplication if its input values are available. In our case, the value of r is not ready until the previous multiplication finishes. This means the example CPU must wait four cycles per iteration of our loop, instead of only one, before beginning the next.

We can alleviate this issue by interleaving the calculations for multiple batches of dice. That way the multiplications for each batch can be in flight together. The next iteration of the loop still must wait for the first multiplication to complete, but the processor is not idle during that time. It can work on the multiplications for the other batches.

A version with two interleaved batches is shown in algorithm 3. For brevity we illustrate this with precomputed thresholds, but the same concept can be applied regardless. It is also possible to interleave more than two batches.

Algorithm 3 — Batched dice rolls (interleaved)

Require: Source of uniformly random integers in $[0, 2^L)$
Require: Target intervals $[0, b_i)$ for i in $1 \ldots k$, with $1 \leq \prod b_i \leq 2^L$
Require: Target intervals $[0, \beta_i)$ for i in $1 \ldots k$, with $1 \leq \prod \beta_i \leq 2^L$
Require: The values $t_1 = (2^L \bmod \prod b_i)$ and $t_2 = (2^L \bmod \prod \beta_i)$
Ensure: The a_i are uniformly random in $[0, b_i)$
Ensure: The α_i are uniformly random in $[0, \beta_i)$
Ensure: The a_i and α_i are all independent

1: **repeat**
2: $r_1 \leftarrow$ random integer in $[0, 2^L)$
3: $r_2 \leftarrow$ random integer in $[0, 2^L)$
4: **for** i in $1 \ldots k$ **do**
5: $(a_i, r_1) \leftarrow b_i \otimes r_1$ ▷ Full-width multiply
6: $(\alpha_i, r_2) \leftarrow \beta_i \otimes r_2$
7: **end for**
8: **until** $r_1 \geq t_1$ and $r_2 \geq t_2$
9: **return** $(a_1, a_2, \ldots, a_k)$ and $(\alpha_1, \alpha_2, \ldots, \alpha_k)$

As written, if either batch fails the threshold test, both batches will be recomputed. This is easy to improve in a more optimized implementation. On the other hand, if the thresholds are small relative to 2^L, both batches will succeed with high probability and the point may be moot.

Another way to achieve instruction-level parallelism is by breaking apart a single batch into smaller groups. This is only effective when k is large relative to the number of multiplications that can be in flight at once. To illustrate, we will choose specific values for L, k, and the b_i.

Suppose an application needs to roll 6-sided dice in batches of 20, on a 64-bit machine. Thus $L = 64$, $k = 20$, and every $b_i = 6$. This is viable because $6^{20} \leq 2^{64}$, and the value $t = (2^{64} \bmod 6^{20}) \approx 1.4 \times 10^{15}$ can be found ahead of time. It may seem large, but t is about 13,000 times smaller than 2^{64}, so the odds of succeeding on the first try are better than 99.99%.

We can split the 20 dice rolls into four batches of five. To do so, we simply "jump ahead" the values of r to what they will be for the 5^{th}, 10^{th}, and 15^{th} iterations. Per the induction from our proof of theorem 1, we know that each $r_i = (r_0(b_1 b_2 \cdots b_i) \mod 2^L)$. All the b_i are 6 here, so we just need to multiply the initial r_0 by 6^5, 6^{10}, and 6^{15}, and take the 64 low-order bits. The resulting process is shown in algorithm 4.

Algorithm 4 — Batched dice rolls (64-bit 6-sided dice in 4 groups of 5)

Require: Source of uniformly random integers in $[0, 2^{64})$

Ensure: The a_i are independent and uniformly random in $[0, 6)$

repeat
  $r_0 \leftarrow$ random integer in $[0, 2^{64})$
  $r_5 \leftarrow (6^5 r_0 \bmod 2^{64})$ ▷ Wrapping multiply
  $r_{10} \leftarrow (6^{10} r_0 \bmod 2^{64})$
  $r_{15} \leftarrow (6^{15} r_0 \bmod 2^{64})$
  for i in $1 \ldots 5$ **do**
    $(a_i, r_0) \leftarrow 6 \otimes r_0$ ▷ Full-width multiply
    $(a_{5+i}, r_5) \leftarrow 6 \otimes r_5$
    $(a_{10+i}, r_{10}) \leftarrow 6 \otimes r_{10}$
    $(a_{15+i}, r_{15}) \leftarrow 6 \otimes r_{15}$
  end for
until $r_{15} \geq (2^{64} \bmod 6^{20})$ ▷ Pre-computed threshold
return $(a_1, a_2, \ldots, a_{20})$

Whereas the direct approach would take $20 \cdot 4 = 80$ cycles for the multiplications on our example CPU, this interleaved version takes only $6 \cdot 4 + 3 = 27$ cycles, a savings of 66%. (There are effectively 6 iterations rather than 5 because we must wait for the last multiplication to finish, and an extra 3 cycles at the start to initiate jumping ahead the r values.) Of course, an optimizing compiler might replace multiplication by 6 with a combination of bit-shifts and additions, throwing off our estimates. Benchmarking is necessary for optimizing any real implementation.

When the b_i are known ahead of time, it is also possible to parallelize the algorithm using SIMD vector operations, which we discuss in section 7.

6 Shuffling arrays

One algorithm that involves many dice rolls is the **Fisher-Yates shuffle** for randomly permuting an array, which proceeds as follows. Beginning with n elements to be shuffled, roll an n-sided die and swap the element at the resulting position to the end of the array. There are now $n-1$ elements to shuffle, so roll an $(n-1)$-sided die and swap the element at that position to the end of the remaining unshuffled portion. Continue in this manner with an $(n-2)$-sided die, then $n-3$, and so on. At the end, the array is shuffled.

This situation is well suited for batched dice rolls. With n elements to shuffle, we can roll k dice of sizes n, $n-1$, ..., $n-(k-1)$ in a batch, and perform the corresponding swaps in the array. This leaves $n-k$ elements to shuffle, and we can repeat the process until the entire array is shuffled.

In order to use this method, we need a way to decide what batch size k to use for a given n. We discuss some possibilities in section 6.1. For now, we denote by n_k the largest value of n for which we will use batches of size k.

Additionally, we would like to maintain an upper bound for the product of the dice sizes in a batch, so that we do not need to calculate that product every time. In other words, we want a value $u \geq n^{\underline{k}}$, ideally with $u \ll 2^L$ so the fast-path will succeed with high probability. We can start by precalculating the largest product that we will ever see for a batch of size k, namely $u_k = n_k{}^{\underline{k}}$.

We could simply use u_k as the upper bound for all batches of size k. However, to improve efficiency we would like to lower the value of u as n decreases. There are many ways to do so. Perhaps the most convenient is to say that whenever the current upper bound fails and we need to compute the actual product $n^{\underline{k}}$, we will assign that product to u and use it as the upper bound for subsequent batches of size k in this shuffle.

We illustrate this approach in algorithm 5, which carries out the dice rolls and swaps for a single batch of size k, when there are n elements to shuffle. It takes an upper bound u as an input, and at the end returns an upper bound for the next iteration. Usually the return value will equal the input u, however if this batch needed to calculate its true product, then the return value will equal that product.

This example implementation performs each swap immediately after rolling the corresponding die, before the threshold for correctness has been checked. This is valid because, if the threshold fails, the dice will be rerolled and the swaps redone. When the threshold succeeds, the dice rolls were fair so the swaps are correct, and the prior order of the elements does not matter.

It would also be possible to roll all the dice first, then perform the swaps. Doing so avoids the possibility of unnecessary extra swaps when rerolling. However, if both a swap and a multiplication can be in flight at the same time on a given computer system, then the immediate swaps may help to alleviate the loop-carried dependency issue described in section 5.1, because the next multiplication can begin during the current swap. Another possibility would be to interleave multiple batches of dice rolls.

We also note that in the implementation of algorithm 5 it would be possible to avoid computing t some fraction of the time by wrapping lines 9–16 in another "if $r < u$ block". We have omitted this for simplicity.

Algorithm 5 — Batch partial shuffle (immediate swap)

Require: Source of uniformly random integers in $[0, 2^L)$
Require: Array z whose first n elements need to be shuffled
Require: Batch size $k \leq n$ for which $n^{\underline{k}} \leq 2^L$
Require: Upper bound $u \geq n^{\underline{k}}$
Ensure: Only the first $(n - k)$ elements of z remain to be shuffled

$r \leftarrow$ random integer in $[0, 2^L)$
for i in $1 \ldots k$ **do**
    $(a, r) \leftarrow (n + 1 - i) \otimes r$ ▷ Full-width multiply
    swap $z[a]$ and $z[n-i]$ ▷ Zero-based indexing
end for

if $r < u$ **then**
    $u \leftarrow n^{\underline{k}}$ ▷ Falling factorial
    $t \leftarrow (2^L \bmod u)$
    while $r < t$ **do**
        $r \leftarrow$ random integer in $[0, 2^L)$
        for i in $1 \ldots k$ **do**
            $(a, r) \leftarrow (n + 1 - i) \otimes r$
            swap $z[a]$ and $z[n-i]$
        end for
    end while
end if

return u ▷ For the next batch

To shuffle a full array, we first select the largest k such that $n_k \geq n$, and set $u = u_k$. Then we shuffle in batches of k, updating u along the way, until $n \leq n_{k+1}$. At that point we set $u = u_{k+1}$ and shuffle in batches of $k+1$, and so forth up to some predetermined maximum batch size. Finally, when the number of remaining elements becomes smaller than the previous batch size, we finish the shuffle with one last batch.

6.1 Batch sizes

The batch size k presents a tradeoff. On one hand, we want to roll as many dice as we can with each random word. On the other hand, we want each batch of dice to succeed on the first try with high probability, so we won't have to reroll. These goals are in opposition, and we seek a balance between them.

The question becomes, at what array length n should we start rolling dice in batches of k. In other words, what values of n_k should be used. This of course depends on L, and any real implementation should use benchmarks on the target hardware to determine the answer. However, a preliminary analysis can help to identify the right ballpark.

It might at first seem that we could simply maximize the expected number of dice which succeed on the first try. That would suggest choosing n_k so the product $n_k{}^{\underline{k}}$ is about $2^L/k$, because then

the expected number of successful dice rolls per batch would be about $k-1$ for either a batch of size k or $k-1$, hence it is the natural crossover point between those batch sizes. However this is not optimal, because the second roll is much more computationally expensive than the first. The second roll incurs the cost of calculating both $n^{\underline{k}}$ and $(2^L \bmod n^{\underline{k}})$, so it should be avoided even more strongly.

Instead, let us estimate the computational cost of each die roll. We will use the time taken per multiplication as our unit of cost. Denote by c_{div} the cost of a division, and by c_{rng} the cost of a call to the random number generator, both in terms of the cost of a multiplication. These operations are generally slower than multiplication, though the exact amounts can vary substantially.

We will assume that division is about 16 times slower than multiplication, and an RNG call is about 2 times slower than multiplication. In other words, we take $c_{div} = 16$ and $c_{rng} = 2$. Many processors have faster division than this, and many random number generators are slower than this, so we expect to obtain conservative estimates for n_k.

That is because larger batches are more likely to require a division than smaller batches, but also larger batches mean fewer overall batches and thus fewer RNG calls than smaller batches. Thus larger batches are more efficient when division is fast and the RNG is slow, whereas smaller batches are more efficient when the opposite is true. By assuming a relatively slow division and fast RNG, we err on the side of smaller batches and thus lower values of n_k.

With n elements to shuffle, a batch of size k has probability $p_1 = u/2^L$ of failing the first threshold test. We will assume that $u \approx n^{\underline{k}}$. When this threshold fails, it incurs the cost of computing the product $b = n^{\underline{k}}$ and the remainder $(2^L \bmod b)$. We treat this cost as $k-1$ multiplications and 1 division, though for $k \geq 4$ it is possible to compute the product in fewer multiplications.

The probability of needing to reroll the batch is $p_2 = (2^L \bmod b)/2^L$, so the expected number of rolls is $1/(1-p_2)$. Every roll including the first incurs the cost of 1 RNG call and k multiplications, and produces k values. So we find the expected cost per element to be:

$$c_{per} \approx \frac{1}{k}\left(\frac{c_{rng}+k}{1-p_2} + (c_{div}+k-1)p_1\right)$$

Using this, a given value of L, and our estimated costs for division and RNG calls, we can compute the cost per element for batches of size k at each n. This enables us to identify when it is beneficial to increase k, and thus what the values of n_k should be. In particular, we will choose n_k so that the estimated cost per element with a group of k is cheaper than with a group of $k-1$ for all n up to n_k, but not $n_k + 1$. Doing so for $L = 64$ with k up to 8, and for $L = 32$ with k up to 4, yields the values for n_k shown in table 1.

k	$L = 64$	$L = 32$
2	1,358,187,913	20,724
3	929,104	581
4	26,573	109
5	3,225	
6	815	
7	305	
8	146	

Table 1: Estimated values for n_k

These are likely to be overestimates for the optimal n_k because we have omitted the cost of mispredicted branches. After all, we want every batch to succeed with high probability, which the processor should be able to predict. In the rare event where the first threshold fails, that prediction will be incorrect and an additional cost is incurred that we did not account for. As a rough attempt to adjust for this, we may reduce the values of n_k from table 1 somewhat arbitrarily down to the next power of 2.

Whichever values of n_k are chosen, the values $u_k = {n_k}^{\underline{k}}$ can be precomputed and stored in a table. Since the batch size k will be fairly small, perhaps at most 8, this table will not take up much space.

When only a small number of elements are being shuffled, the batches can be selected ahead of time and their thresholds precomputed. For example, on a 64-bit system we might use batches of $k = 8$ dice when n is small. In that case, we can make a table of thresholds, say for the first sixteen 8-element batches (2–9), (10–17), ..., (122–129). Then, whenever we need to shuffle up to 129 elements, including at the end of a larger shuffle, we simply roll dice using those batches and perform the required swaps accordingly. This strategy reduces the expected cost per element because we already know the thresholds for success, so we never need to perform additional multiplications or divisions to find them.

7 Optimization and vectorization

Many modern processors support vectorized instructions to perform the same operation on multiple values. These "single instruction, multiple data" (SIMD) instructions can be substantially faster and more efficient than separately computing that same operation on each value individually. There are a variety of different sizes of SIMD vectors that may be supported. For this exposition we will assume that 256-bit vectors are available, which may be treated as vectors of four 64-bit integers, eight 32-bit integers, or sixteen 16-bit integers. Other possibilities are also valid, and may be treated similarly.

There are several ways to apply SIMD instructions to batch dice rolls. One option is to interleave several batches, along the lines of algorithm 3, and make a vector of the sizes of the first die from each batch. Then an elementwise full-width multiplication with a vector of random words will both roll those dice, and produce a vector containing the next value of r for each batch. This can be multiplied by a vector of the sizes of the second die from each batch, and so forth.

At the end, the final values of r need to be checked against the threshold for each batch. When using this approach as part of a shuffle, an upper bound u can be maintained as in algorithm 5, to avoid the extra work of computing the product and threshold for each group most of the time.

Alternatively, rather than interleave multiple batches, we can instead vectorize a single batch of dice rolls when the sizes of the dice are known ahead of time. Let L be the bit-width of the vector elements. Let $\vec{\boldsymbol{b}} = (b_1, b_2, \ldots, b_k)$ be a vector of the sizes of the dice to be rolled, with $\prod b_i \leq 2^L$. Since the b_i are known ahead of time, we can precompute their partial products $d_i = \prod_{j<i} b_j$. This means $d_1 = 1$, and each other $d_i = b_1 b_2 \cdots b_{i-1}$. Let $\vec{\boldsymbol{d}} = (d_1, d_2, \ldots, d_k)$ be a vector of those partial products.

Given an L-bit random word r_0, we multiply each element of $\vec{\boldsymbol{d}}$ by r_0, and keep the low-order L bits of the result. In other words, we calculate the elementwise wrapping product $\vec{\boldsymbol{p}} = (r_0 \vec{\boldsymbol{d}} \bmod 2^L)$. From the proof of theorem 1, we see that each $p_i = r_{i-1}$. Then we calculate the elementwise full-width product $\vec{\boldsymbol{b}} \otimes \vec{\boldsymbol{p}} = (\vec{\boldsymbol{a}}, \vec{\boldsymbol{r}})$. Now if $r_k \geq (2^L \bmod \prod b_i)$, the a_i are independent and uniformly random in $[0, b_i)$.

This process effectively carries out our batch dice rolling algorithm with only two vectorized

multiplications, provided that wide enough SIMD registers are available. In particular, we must have L equal to a valid SIMD element bit-width such as 8, 16, 32, or 64, and k at most the number of such elements that fit in a SIMD vector. That restriction on k is easily lifted, however, by using more than one SIMD vector.

7.1 Prechecked thresholds

In some particular cases, it is possible to check the threshold for a batch of dice before rolling them. By inspecting the initial random word r_0, it may be possible to determine whether the entire batch of dice will need to be rerolled. If so, a new random word can be generated immediately, without needing to roll the unsuccessful batch of dice. This is possible when the threshold for the batch is a power of 2, which means the product b of the sizes of the dice is a factor of $2^L - 2^M$ for some natural number $M < L$.

Let us first consider the case where $b = 2^L - 2^M$, which implies b is a multiple of 2^M. By theorem 1, the low part of the final product, $r_k = (r_0 b \bmod 2^L)$, must be at least the threshold $t = (2^L \bmod b) = 2^M$ for the results to be fair. Since b is a multiple of 2^M, so too is r_k, which means the only way to fail the threshold with $r_k < t$ is when $r_k = 0$. But if $r_k = 0$ then r_0 must be a multiple of 2^{L-M}, because that is the only way to make $r_0 b = r_0(2^L - 2^M)$ a multiple of 2^L.

This means, when $b = 2^L - 2^M$, we can check the threshold without ever calculating r_k, and thus before rolling the dice, simply by observing whether the initial random word r_0 is a multiple of 2^{L-M}. If it is, the threshold fails and we need a new r_0. Otherwise, the dice are fair.

Now suppose b is a factor of $2^L - 2^M$. Then there is some integer b_{k+1} such that $b \cdot b_{k+1} = 2^L - 2^M$. We can create a new batch of dice by appending a die of size b_{k+1} onto our existing batch. This extended batch has a product equal to $2^L - 2^M$, so we can precheck its threshold by testing whether r_0 is a multiple of 2^{L-M}. If it passes the precheck, then we can roll the dice in the extended batch and they will all be fair.

Since we do not need the result of the b_{k+1}-sided die, we can ignore it. Moreover, since it is the last die in the batch, we do not have to roll it at all, because doing so has no effect on the previous die rolls. Thus we can precheck the threshold for a batch whenever its product b is a factor of $2^L - 2^M$. In particular, when the batch threshold is a power of 2, if r_0 is a multiple of 2^{L-M} the threshold fails and we need a new random word, otherwise it succeeds and the dice will be fair.

7.2 Shuffling 2–17

The very smallest shuffles can be highly optimized. There are many ways to do this, and we will demonstrate an approach using 16-element SIMD vectors of 16-bit integers, for a total vector width of 256 bits. The last 16 steps in a Fisher-Yates shuffle involve element counts running from 17 down through 2, so we focus on rolling dice with sizes 2–17. We will split them into four batches in a way that allows the thresholds for fairness to be prechecked at the start.

We use $L = 16$ bits for each of the four batches, so we need 64 random bits in the form of four random 16-bit words. These may be generated via a single 64-bit RNG call, or two 32-bit RNG calls, or any other appropriate manner. Then we test the random words to see if they will pass the threshold, and replace them if needed. Once we have four suitable 16-bit random words, we arrange them in a 16-element SIMD vector according to our partition of 2–17.

Next, we multiply vector of random words by a vector of partial products of the batches, to generate the values of r_{i-1}. Finally, we multiply those values by a vector of dice sizes to obtain the resulting dice rolls a_i. Because the thresholds have been prechecked, we do not need the low

part of the final product, so it can be a 16-bit high multiply rather than a full-width multiply that would produce 32-bit output. This keeps all the SIMD operations at 256 bits wide.

There are many suitable ways to partition the numbers 2–17. For prechecks to work as explained in section 7.1, we want the threshold for each batch to be a power of 2. We choose the following partition, and label the batches A through D:

$$\begin{aligned} A &= (\ 2,\ \ 3,\ \ 4,\ 11) \\ B &= (\ 5,\ \ 6,\ 16,\ 17) \\ C &= (\ 7,\ \ 8,\ \ 9,\ 10) \\ D &= (12,\ 13,\ 14,\ 15) \end{aligned}$$

The thresholds for these batches are 2^6, 2^8, 2^4, and 2^4, respectively:

$$\prod A = (2^{16} - 2^6) \,/\, 248 \qquad \prod C = (2^{16} - 2^4) \,/\, 13$$
$$\prod B = (2^{16} - 2^8) \,/\, 8 \qquad \prod D = (2^{16} - 2^4) \,/\, 2$$

Thus, given a 16-bit random word r_A, if it is not a multiple of 2^{10} then we can use r_A to roll a batch of dice with sizes from group A, and the results will be fair. The same applies for the groups B, C, and D, with their respective prechecks for r_B, r_C, and r_D to be multiples of 2^8, 2^{12}, and 2^{12}. The probability that all four batches succeed on the first try is:

$$\begin{aligned} \Pr(\text{success}) &= \left(1 - 2^{-10}\right)\left(1 - 2^{-8}\right)\left(1 - 2^{-12}\right)\left(1 - 2^{-12}\right) \\ &\approx 99.46\% \end{aligned}$$

The odds of at least one group failing the precheck are about 1 in 186. If a 64-bit RNG is used to generate all four 16-bit words at once, it will be called an average of 1.0054 times. An implementation of this approach is shown in algorithm 6. The vector of partial products $\vec{\boldsymbol{d}}$ there is defined such that each entry d_i equals the product of the elements prior to $i{+}1$ in its same batch. For example, batch B starts (5, 6, 16), so $d_{16-1} = 5 \cdot 6$.

Algorithm 6 — SIMD dice rolls (2–17 with precheck)

Require: Source of uniformly random integers in $[0, 2^{16})$

Ensure: The a_i are independent and uniformly random in $[0, 2)$ through $[0, 17)$

repeat
  $(r_A, r_B, r_C, r_D) \leftarrow$ random integers in $[0, 2^{16})$
until $(r_A \bmod 2^{10})$ and $(r_B \bmod 2^8)$ and $(r_C \bmod 2^{12})$ and $(r_D \bmod 2^{12})$ are all nonzero
$\vec{\boldsymbol{r}} \leftarrow [r_A, r_A, r_A, r_B, r_B, r_C, r_C, r_C, r_C, r_A, r_D, r_D, r_D, r_D, r_B, r_B]$
$\vec{\boldsymbol{d}} \leftarrow [1, 2, 6, 1, 5, 1, 7, 56, 504, 24, 1, 12, 156, 2184, 30, 480]$ ▷ Partial products
$\vec{\boldsymbol{n}} \leftarrow [2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17]$
$\vec{\boldsymbol{p}} \leftarrow \vec{\boldsymbol{r}}\vec{\boldsymbol{d}} \bmod 2^{16}$ ▷ Elementwise wrapping multiply
$\vec{\boldsymbol{a}} \leftarrow \vec{\boldsymbol{p}}\vec{\boldsymbol{n}} \div 2^{16}$ ▷ Elementwise high multiply
return $\vec{\boldsymbol{a}}$

The benefit of using these batches is that the thresholds can be prechecked. However, by using different batches which cannot be prechecked, it is possible to attain a higher probability of success.

For example, the following 16-bit batches have thresholds of 0, 88, 86, and 16, and thus a 99.71% chance that they all succeed on the first try, or about 1 in 345 odds of needing to reroll:

Batch	Thr.
2, 4, 8, 16	0
3, 6, 9	88
5, 7, 10, 11, 17	86
12, 13, 14, 15	16

Table 2: A partition of 2–17 into 4 batches with optimal 16-bit thresholds

Those are the highest possible odds of success for 16-bit batches. Better odds are possible with 32-bit or 64-bit batches, at the cost of increasing the bit-width of the vector multiplications from 256 for 16-bit, to 512 or 1024. The maximum probability of success for 32-bit batches is about 99.96%, or 1 in 2563 odds of rerolling, which is attained by the following batches:

Batch	Thr.
2, 3, 6, 7, 8, 9, 14, 15	826816
4, 5, 10, 11, 12, 13, 16, 17	848896

Table 3: A partition of 2–17 into 2 batches with optimal 32-bit thresholds

Using 64-bit values, all the numbers from 2 through 17 fit in a single batch with threshold 82677794799616, and thus probability of success 99.9995%, or 1 in 223116 odds of rerolling.

7.3 Card shuffling

There are 52 playing cards in a standard deck, so shuffling 52-element arrays is quite common. It may thus be of interest to formulate a highly optimized algorithm for that purpose. We observe that $\log_2(52!) \approx 225.6$, so it seems reasonable to use 256 random bits for this task.

We may take those bits in the form of sixteen 16-bit words, or eight 32-bit words, or four 64-bit words. The tradeoff among these options is that wider words produce batches which are more likely to succeed, at the cost of increasing the bit-width required for the vector multiplications.

Our strategy will be to partition the integers 2–52 into batches whose products fit in the chosen bit-width L. Then we can precompute the partial products of those batches, and arrange them in a list according to the size of the corresponding die from 2 through 52. When it is time to shuffle, we generate an L-bit random word for each batch, and arrange them in a matching list so that each position belonging to a member of a given batch in the first list contains the random word for that batch in the second list.

We multiply those two lists elementwise, the partial products times the random words, and keep the low part of each result. We then multiply those values elementwise by the numbers 2–52, and keep the high part of each result. This is the vectorized version of our algorithm, so if each batch passes its threshold test then the results are fair dice rolls. Those thresholds can be checked at the beginning, before rolling the dice, by multiplying each random word by the full product of the batch for which it will be used. If any result is less than the threshold for its batch, then a new random word is needed.

We have found, by exhaustive computer search, the maximum probabilities of success for bit-widths $L = 16$, 32, and 64. With 16-bit words, the best possible odds are 98.0% to succeed, or 1 in 50 to reroll. An example of an optimal partition of 2–52 into sixteen 16-bit batches, alongside their respective thresholds, is shown in table 4.

Batch	Thr.	Batch	Thr.
2, 4, 8, 16, 32	0	12, 51	52
10, 13, 18, 28	16	25, 27	61
5, 14, 24, 39	16	31, 44, 48	64
20, 26, 42	16	7, 11, 17, 50	86
35, 36, 52	16	29, 46, 49	170
34, 41, 47	18	38, 40, 43	176
9, 19	43	6, 15, 22, 33	196
30, 37	46	3, 21, 23, 45	331

Table 4: A partition of 2–52 into 16 batches with optimal 16-bit thresholds

With 32-bit words, the best possible odds are 99.98% to succeed, or 1 in 5793 to reroll. An example of an optimal partition of 2–52 into eight 32-bit batches, alongside their respective thresholds, is shown in table 5.

Batch	Thr.
2, 4, 5, 22, 26, 29	2336
3, 12, 16, 18, 36, 37	2560
8, 10, 32, 33, 40, 41	4096
9, 14, 17, 31, 35, 42, 44	85936
6, 19, 23, 25, 28, 45, 52	131296
7, 15, 20, 30, 38, 39, 46	131296
11, 13, 21, 27, 47, 49	187807
24, 34, 43, 48, 50, 51	196096

Table 5: A partition of 2–52 into 8 batches with optimal 32-bit thresholds

With 64-bit words, the best possible odds are 99.99997% to succeed, or 1 in 3344007 to reroll. An example of an optimal partition of 2–52 into four 64-bit batches, alongside their respective thresholds, is shown in table 6.

Batch	Threshold
6, 7, 8, 9, 23, 24, 26, 30, 36, 39, 43, 52	625134247936
2, 3, 4, 5, 20, 25, 31, 35, 40, 41, 46, 47, 51	1006453551616
13, 14, 15, 16, 21, 28, 29, 32, 33, 37, 42, 44, 49	1683350388736
10, 11, 12, 17, 18, 19, 22, 27, 34, 38, 45, 48, 50	2201420271616

Table 6: A partition of 2–52 into 4 batches with optimal 64-bit thresholds

These optimal thresholds for $L = 16$, 32, and 64 are unique, though there are many partitions which achieve them. Using a larger value of L increases the probability that the batches all succeed on the first try. However, it also increases the total bit-width of the vector multiplications, and thus the running time of the algorithm. This presents a tradeoff, and the best choice may depend on the specific application or processor architecture.

With 256-bit SIMD vectors, when $L = 16$ the 51 integers from 2–52 fit in 4 vectors, for a total width of 1024 bits. It may be possible to reduce that to 3 vectors, and thus 768 bits, by separately handling the first 16-bit batch, which consists entirely of powers of 2. When $L = 32$, the 51 numbers fit in 7 vectors, for a total width of 1792 bits. And when $L = 64$ they fit in 13 vectors, for a total width of 3328 bits.

Using this approach to shuffle a 52-card deck, if 256 new random bits are generated whenever any reroll is needed, the expected number of random bits required with 16-bit words is just over 261, while with 32-bit or 64-bit words it is just over 256. For comparison, to perform the same shuffle using the fast dice roller algorithm of Lumbroso, the expected number of random bits is just under 278, and using Lemire's method with 16-bit words it is just over 816.

8 Conclusion

We have shown that Lemire's nearly-divisionless method of generating bounded random integers can be extended to generate multiple such numbers from a single random word. This can greatly reduce the number of random bits used, without substantially increasing the amount of computation required, when rolling several dice or shuffling an array. The benefit of this batched approach is due to making fewer calls to a random number generator. If the cost of an RNG call is very low, there might not be an appreciable benefit from doing so.

Our core algorithm can be modified in a variety of ways for many different situations. We have demonstrated several methods of optimizing the implementation, including fast-paths, precomputed thresholds, and SIMD vectorization. We expect that with proper tuning, the use of batched dice rolls has the potential to make small shuffles faster and more efficient, including as a base case for larger shuffles.

References

Lemire, Daniel (Jan. 2019). "Fast Random Integer Generation in an Interval". In: *ACM Transactions on Modeling and Computer Simulation* 29.1, pp. 1–12. ISSN: 1558-1195. DOI: 10.1145/3230636. arXiv: 1805.10941. URL: http://dx.doi.org/10.1145/3230636.

Lumbroso, Jérémie (2013). *Optimal Discrete Uniform Generation from Coin Flips, and Applications*. arXiv: 1304.1916. URL: https://arxiv.org/abs/1304.1916.