Zero Tolerance for Bias

TERENCE KELLY

Evenhanded random selection is fiendishly tricky. The combinatorics of fairness is especially vexing, and the consequences exceptionally weighty, in *shuffling*. Shuffling a set means randomly choosing an ordered sequence of its elements. For example, shuffling {A,B,C} means choosing with equal probability one of 3! = 3 × 2 × 1 = 6 permutations: ABC, ACB, BAC, BCA, CAB, or CBA. If that sounds easy, read on.

HIGH STAKES

Shuffling is crucial to a wide range of important applications. Shuffling amino acid sequences is a major and essential use of randomization in bioinformatics.[12] Shuffling default web-browser options can resolve an antitrust crackdown.[32] Online poker sites shuffle virtual decks of playing cards, hauling in $60 billion annually.[9] The stakes are highest when American draft lotteries shuffle birthdays to muster citizens for military duty: 16 million U.S. conscripts served in 20th-century wars that killed more than 600,000 Americans.

Where shirts and lives can be lost, fairness is paramount. After lackadaisical shuffling awarded too many tropical vacations to December birthdays in the 1969 draft lottery, public outrage compelled reform.[26,29] The resulting tandem-shuffle protocol draws pairs of labeled marbles from separate mixing machines, one for birthdays

"We would like to have a drawing this year that appears impartial, both to those professionally curious and to those whose lives are involved."

— Curtis W. Tarr, Director of Selective Service, June 1970 [25]

and another for draft-sequence numbers[25,28] Nowadays shuffling is often computerized, but that's no excuse for bias. Gambling regulations, for example, require that mathematical probabilities in virtual games match those in their physical counterparts.[21] Fairness isn't just a good idea—it's the law.

Despite the conceptual simplicity of shuffling and the importance of fairness, bad advice and buggy code abound. Seemingly respectable authorities have botched shuffling for decades at a stretch. The nasty thing about broken shuffle code is that it usually appears to work correctly in conventional tests. Truly getting it right requires reasoning carefully about the combinatorics of random selection.

So that's what we'll learn to do. We'll start by dissecting a meticulously curated, museum-quality compendium of ways to bungle a shuffle—from a venerable programming textbook. Then we'll develop a correct shuffling program that scales far beyond toy-sized problems while avoiding bias. Finally we'll consider how to obtain the random numbers required by all random-selection algorithms. The mental checklist that we accumulate along the way will enable us to audit off-the-shelf solutions. Ultimately we'll see that unbiased shuffling is so easy that there's little excuse to do it any other way.

SEE HOW NOT TO PROGRAM

The terse C program in figure 1 summarizes widespread shuffling errors. It contains at least three distinct biases and several other dubious features. Seldom will you encounter code with higher defect density.

FIGURE 1: ERROR-RIDDEN SHUFFLE

```
1  int main(void) {
2    enum { N = 47 };  int i, A[N];
3    for (i = 0; i < N; i++) A[i] = i;  // fill A[] with 0..N-1
4    srand(time(NULL));                 // seed PRNG (seed bias)
5    for (i = 0; i < N; i++) {          // shuffle A[] (shuffle bias)
6      int r = rand() % N;              // pick random entry (modulo bias)
7      int t = A[r]; A[r] = A[i]; A[i] = t;  // swap A[r] <-> A[i]
8    }
9    for (i = 0; i < N; i++) printf(" %d", A[i]);
10   printf("\n");
11 }
```

Sucker approaching poker table: "Is this a game of chance?"

Card shark: "Not the way I play it."

—W.C. Fields, My Little Chickadee

Line 6 of figure 1 is prone to *modulo bias*. The programmer wants a random number uniformly distributed between zero and *N*-1, but `rand() % N` does not guarantee uniformity. Figure 2 demonstrates the spectacular unfairness of modulo bias. The programmer naïvely expects `rand() % N < N/2` on line 4 of figure 2 to evaluate true as often as false, but thanks to modulo bias it evaluates true *twice* as often as false. Odds so slanted would spark a gunfight in an Old West casino and a lawsuit in a modern one.

Another problem with figure 1 is the use of the standard C pseudorandom number generator (PRNG), which is a bug unto itself because `rand()` has so often been implemented so badly.[23] Nothing prevents this tradition from persisting;

FIGURE 2: MODULO BIAS DELIVERS EXTREME NON-UNIFORMITY

```
1  int main(void) {
2    int i, N = 2*(RAND_MAX/3), L = 0, H = 0;
3    for (i = 0; i < 1000; i++)     // Programmer expects rand() % N to be
4      if (rand() % N < N/2) L++;   // uniformly distributed on [0..N-1] and
5      else                  H++;   // so expects L ~= H.  But modulo bias
6    printf("L=%d H=%d\n", L, H);   // "wraps" 1/3 of rand() return values
7  }                                // to low end of interval so L ~= 2*H.
```

the latest C standard explicitly condones an obsolete implementation.[14] To make matters worse, the classic defects of `rand()` interact badly with the `rand() % N` anti-pattern: If `N` is a small power of two such as 2, 4, or 8, `rand() % N` yields the low-order bits of `rand()`'s return value. Unfortunately, many bad implementations offer especially poor randomness in the low bits.[23] On some commercial Unix systems of yesteryear, for example, consecutive calls to `rand() % 2` would return 0,1,0,1,0,1,... ad infinitum. Poor low-bit randomness would bite several iterations of line 6 in figure 1.

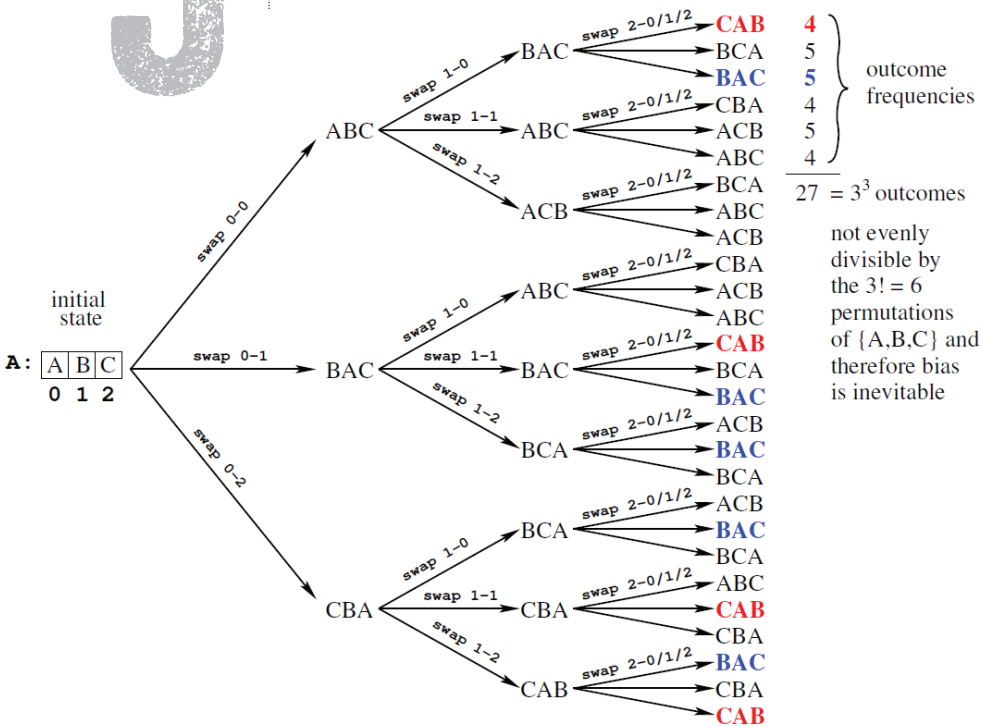
In addition to generic bugs involving PRNG misuse, the program of figure 1 contains two further severe biases specific to shuffling.

The first is *shuffle bias*. Lines 6 and 7 visit every element in `A[]` and swap it with an element chosen at random *from the entire array*. Figure 3 explains shuffle bias by working through the tree of consequences for shuffling {A,B,C} in this way. Branches represent fair random swaps and leaves on the right represent outcomes, all of which are equally likely. Note that some permutations result more often than others. For example, BAC appears five times but CAB only four. Eight outcomes have C in the first position, nine begin with A, and ten start with B. The fundamental problem is a mismatch between shuffle outcomes and permutations. Applied to N items, the biased shuffle of figure 1 yields N^N outcomes, each of which is one of $N!$ permutations. But $N!$ does not evenly divide N^N for any N greater than 2, so some permutations must occur more often than others.

The second shuffle-specific bias in figure 1, *seed bias* on line 4, has a similar flavor but more dire consequences.

3

FIGURE 3: TREE OF OUTCOMES FROM BIASED SHUFFLE OF $N = 3$ ITEMS



Tip: The "bc" utility offers convenient big-int arithmetic.

The number of PRNG seeds, 2^B for a B-bit seed, doesn't match the number of permutations, $N!$ for shuffling N items. If $N!$ doesn't evenly divide 2^B, bias is inevitable. Worse yet, $N!$ is usually far larger than 2^B, so the vast majority of permutations cannot possibly be generated by any seed. For example, the 32-bit `srand()` seed on many of today's Unix-like systems can generate fewer than 13! outcomes. It's hopeless for poker because $52! > 2^{225}$. Even a 128-bit seed would be inadequate for poker: The fraction

of permutations that could be generated is less than $2^{128}/2^{225} = 1/2^{97}$, which is 1 in 158,456,325,028,528,675,187,087,900,672. Unbiased shuffling means that all permutations are equally likely until the choice of one permutation is determined by exogenous true-random entropy alone (i.e., the seed, if we're using a PRNG-based shuffler). A shuffler that uses a PRNG with a small seed, however, arbitrarily renders nearly all permutations *impossible* and does so *before* seed entropy is supplied.

For applications such as poker where security is a concern, figure 1 contains at least two vulnerabilities beyond the use of a non-cryptographically-secure PRNG. First, the Unix epoch time is a rather predictable PRNG seed (line 4), which may allow a cheater to guess the seed and compute the resulting shuffled deck. Second, the small 32-bit seed space enables anyone to precompute a lookup table of all possible decks, which in turn allows a cheater to infer much about the state of the deck in a game from his own hand. Similar vulnerabilities plagued a popular backgammon program[24] and allowed successful attacks on commercial poker software.[31]

Remarkably, all of the ills of figure 1 afflict the shuffling code in a prominent textbook series. With a reputable publisher, proudly Ivy-educated authors, and nine pricey editions spanning three decades, this series shows no obvious red flags. The second edition from the mid-1990s, however, recommends `rand() % N` for scaling random numbers to a desired range and presents shuffling code with modulo bias, shuffle bias, seed bias, and the other problems noted previously. The recent ninth edition partly addresses some but not all of these defects. Its shuffling

code is similar to that of the second edition, but it points out the shuffle bias and invites readers to investigate unbiased shuffles on their own. It also mentions the poor quality of standard C `rand()` and recommends better PRNGs for serious uses. However the latest edition does not mention seed bias, it continues to recommend `rand() % N` with no mention of modulo bias, and it doesn't mention the predictability of `srand(time(NULL))`.

Sadly, nobody has a monopoly on bad code or bad advice. Confusion is widespread, even among well-heeled players facing high stakes. Microsoft botched the shuffle that was supposed to be its ticket out of antitrust trouble.[32] The official Java documentation conflates the period of a PRNG—the number of calls before the output stream repeats—with the cause or remedy of seed bias:

> For applications that generate large permutations, consider a generator whose period is much larger than the total number of possible permutations; otherwise, it will be impossible to generate some of the intended permutations. For example, if the goal is to shuffle a deck of 52 cards, the number of possible permutations is 52! (52 factorial), which is approximately $2^{225.58}$, so it may be best to use a generator whose period is roughly 2^{256} or larger....[4]

This is misleading and irrelevant. A long period does not preclude bias, and shuffling more than once without re-seeding increases the size of the required seed. Some variants of the "Mersenne Twister" PRNG, for example, have periods far larger than 2^{256}, but they accept 64-bit

seeds. Therefore, shufflers that employ these PRNGs yield one of at most 2^{64} different outcomes in the first shuffle after seeding, which is less than 21!. For problems beyond toy size, severe bias is inevitable despite the long period. Some of today's off-the-shelf PRNGs can accept seeds roughly 20,000 bits long, but 2^{20000} is less than 2087!. A single unbiased shuffle of one million items involves more than 18 million bits of entropy. Furthermore, repeated shuffling increases the entropy requirements: A series of J independent unbiased shuffles of N items can have $(N!)^J$ possible outcomes, all of which must be equiprobable. The series therefore requires at least $\log_2((N!)^J)$ bits of input entropy, which is J times more than required for a single shuffle.

THE RIGHT SHUFF

Shuffle bias became obsolete in 1964—thirty years before the buggy textbook series discussed earlier—when Durstenfeld published an unbiased algorithm to shuffle an array in place.[8] The basic idea is simple and intuitive, mirroring the definition of factorial: Repeatedly append to the output sequence an item chosen with uniform probability from the dwindling pool of unselected items. Draw the tree of outcomes for Durstenfeld's algorithm, as for the bogus shuffle in figure 3, and you'll get exactly $N!$ distinct leaves.

To be equiprobable, the item choices in Durstenfeld's algorithm must avoid modulo bias. The key to banishing modulo bias is *rejection*: discarding random numbers that would lead to bias if mapped onto a desired range. For example, to choose fairly among four alternatives using

a six-sided die, roll until the outcome is in the range [1..4]. Discard 5 or 6 because deterministically mapping these bad values into the acceptable range would introduce bias. Rejecting bad values ensures fairness at the cost of discarding random numbers. The simple rejection methods used below may at most double the expected number of random numbers consumed; in practice the cost is usually far lower.

Figure 4 shows a complete shuffle program built around a variant of Durstenfeld's algorithm. The shuffle loop of lines 16–18 fills array A[] from left to right by swapping into each position a random *unselected* entry. The loop

FIGURE 4: UNBIASED SHUFFLE PROGRAM

```
1  typedef uint32_t u32;  typedef uint64_t u64;  // for brevity
2
3  // return uniform random number in range [0..K-1] where 0 < K <= 2^32
4  static u32 U(u64 K) {
5    u32 R;  u64 M = 1 + (u64)UINT32_MAX;
6    assert(0 < K && K <= M);
7    do { size_t nr = fread(&R, sizeof R, (size_t)1, stdin);
8         assert(1 == nr);
9    } while ((u64)R >= K * (M / K));  // reject R that would cause bias
10   return (u32)((u64)R % K);  // unbiased thanks to rejection
11 }  // type casts above make explicit C's automatic conversions
12
13 int main(void) {
14   enum { N = 47 };  u32 i, A[N];
15   for (i = 0; i < N; i++) A[i] = i;  // fill A[]
16   for (i = 0; i < N-1; i++) {        // left->right Durstenfeld shuffle
17     u32 r = i + U(N-i);              // randomly pick unselected entry
18     u32 t = A[i]; A[i] = A[r]; A[r] = t;  // swap
19   }
20   for (i = 0; i < N; i++) printf(" %" PRIu32, A[i]);
21   printf("\n");
22 }
```

invariant is that array entries less than `i` contain an equiprobably selected and equiprobably permuted subset of `A[]`. An unselected entry is chosen at random on line 17 by function `U()`, a replacement for `random_number % range` that uses rejection to avoid bias.

The body of `U()` on lines 4 through 11 merits close study. `U()` takes a 64-bit argument `K` that must lie in $[1..2^{32}]$, a restriction enforced on line 6. The long argument allows callers to freely express the full range of reasonable ways to scale a 32-bit random number: `U(1)` returns zero; $U(2^{32})$ returns a raw (unscaled) random number; and any other value of the `K` argument returns a random number uniformly distributed on $[0..K-1]$. The endpoints of the scaling range require no special treatment. If the `K` argument were a 32-bit number, requesting a raw random number would be impossible.

Line 7 slurps raw 32-bit random numbers from `stdin`. Sometimes performance or other considerations recommend hard-wiring a particular random-number source or a fixed menu of sources. Postponing the choice until runtime, however, is more flexible and future-proof. That's why the GNU command-line shuffle utility "`shuf`" offers the option of reading random numbers from a file or pipe. Users may supply whatever random bits they please.

The rejection threshold on line 9 of figure 4 accepts as many random numbers as possible while ensuring that the modulus operation on line 10 yields an unbiased return value. A random number `R` is acceptable if it is drawn with uniform probability from a range that `K` divides evenly (without remainder), because then `R % K` is unbiased. Variable `M` equals 2^{32} (line 5). Because integer division truncates, `M / K`

on line 9 is the number of complete *K*-long segments that fit on the integer number line between zero and $2^{32}-1$ inclusive. The range of integers less than `K * (M / K)` divides evenly by `K`; any `R` outside this range must be rejected.

The approach of figure 4 isn't the only way to shuffle without bias. We can implement a one-to-one correspondence between integers in the range [0..*N*!–1] and permutations of *N* items. Then we simply draw from this range a *single* random number with uniform probability and map the random number to a random permutation. This alternate approach can be more frugal with entropy than the code in figure 4; it is implemented in the "unpack" program in the example code tarball. Finally, don't overlook the option of physical shuffling in situations where computerization is not a hard requirement. Casinos and lotteries continue to use physical random selection for good reasons.

WHENCE RANDOMNESS?

Random selection algorithms such as the shuffler of figure 4 require random numbers. How can we obtain suitable random numbers?

Pseudo-random number generators are useful for many purposes, but unbiased shuffling isn't one of them. The purpose of a PRNG is to "stretch" a small amount of true-random entropy into a larger quantity of ersatz random bits, which is reasonable if genuine entropy is expensive and the stretching does no harm. But as we've seen, PRNG-based shufflers are prone to bias caused by mismatches between the number of PRNG seeds and the number of permutations. These and other fundamental

problems with PRNG-based shuffling are reviewed at greater length in a note included with this column's example code. If we don't use a PRNG, then we need not worry about the dizzying combinatorics of seed bias and the checkered history of PRNGs.[15,20,23]

An unbiased shuffler can consume true-random bits directly, and the required quantity of random numbers is easy to calculate. For example, the program of figure 4 consumes, in expectation, at most 64 random bits per item when shuffling up to 2^{32} items. The alternative approach of directly mapping a single large random number to a permutation requires, in expectation, $2 \times \lceil \log_2(N!) \rceil$ bits of entropy to shuffle N items. Is it difficult to obtain this much genuine entropy nowadays?

True-random numbers are cheap and plentiful on many computers today. Many CPUs support the unprivileged `RDSEED` instruction, which extracts entropy from an on-silicon, crypto-grade, NIST-compliant, *non*-deterministic thermal noise source.[11] On Linux, check `/proc/cpuinfo` to see if your CPU supports it. `RDSEED` delivers tens of millions of random bits per second on middle-aged Intel servers. At that rate, a future draft lottery that shuffles individual citizens instead of birthdays could randomly permute the entire U.S. population without bias in minutes; parents of twins and triplets might prefer this to shuffling birthdays. True-random number generators are no longer high-end CPU features. Popular, cheap, and rugged Raspberry Pi single-board computers have featured true RNGs for at least a decade.[33]

Caveat: The history of true-random number generators is as troubled as that of pseudorandom generators. RAND

Entropy Alchemy

Real-world entropy sources seldom obey the idealized rules of probability theory. Bias is pervasive. Fortunately, von Neumann conjured a trick to transmute a dull leaden biased coin into a brilliant golden fair coin: Flip twice. If the outcomes are heads-heads or tails-tails, emit no output and start over. Otherwise emit the second outcome of the pair: for H-T emit T and for T-H emit H. If flips are independent, then the output stream is that of a fair coin, regardless of the actual coin's bias.

Corporation's landmark 1940s true-RNG was a balky contraption.[3] In the 1990s three commercial true-RNG gizmos all failed Marsaglia's statistical randomness tests.[19] Recently the AMD implementation of RDSEED's partner RDRAND was utterly broken.[27] And whenever security is a concern we must ponder Ken Thompson's timeless advice about trusting anything designed and built by strangers: "Don't."[30]

Fortunately, we can mitigate the risks of dubious entropy sources by combining sources, in the spirit of the tandem-shuffle draft-lottery protocol noted earlier. Sometimes it's safe and helpful to simply bitwise-XOR independent random bit streams together; for the fine print, and for more sophisticated approaches, read about *randomness mergers*. The sidebar presents another technique for coping with imperfect entropy sources.

Zero tolerance for bias is a hard requirement in the most important practical applications of random selection. Zero tolerance is both feasible and prudent in many other applications. Of course, as a programmer you're sometimes free to adopt a different policy. The important thing is to make conscious and well-informed decisions about bias; quantify and characterize all biases present in your software; and clearly explain to all stakeholders your rationale for permitting bias.

DRILLING DEEPER

O'Connor reviews the history of shuffling algorithms.[22] Knuth and Yao[16] and Lumbroso[18] develop frugal rejection methods for unbiased selection in situations where random numbers are expensive. Press et al. describe scientific-grade PRNGs,[23] and Boneh and Shoup describe cryptographically secure PRNGs.[2] Diaconis and Fulman mathematically analyze physical card shuffling.[7]

BITS

The example code tarball at https://queue.acm.org/downloads/2024/Drill_Bits_12_example_code.tar.gz contains the shufflers of figures 1 and 4; a shuffler that maps a single large random number to a permutation; a program that generates true-random numbers with `RDSEED`; a quick-and-dirty little program to harvest entropy from keystroke timings; a script to estimate the entropy needs of large shuffles; and a note on PRNG-based shuffling.

DRILLS

1. Is the expression "`random_number % N`" ever *not* biased? Quantify its bias as a function of `N`, assuming that the random number is uniform on [0..2^{32}–1]. Define bias for each particular *N* value as the ratio of most/least likely outcome probabilities.
2. Audit the purportedly bias-free replacements for "`random_number % N`" provided in some programming and scripting languages.
3. Modify the program in figure 4 into a poker-dealing program with a "stop when done"[13] shuffle (i.e., shuffle

only the top of the deck as needed for the current game). Bentley considers a related sampling problem.[1]

4. Compare function `U()` in figure 4 with algorithm 3 and appendix A of Lemire.[17] Consider behavior at the endpoints of the *s* argument. Does algorithm 3 handle `s = 0`? Can the caller express `s = 2`^{L}?
5. Modify `U()` to consume fewer random bits when possible. For example, `U(8)` requires only three random bits.
6. Wikipedia sketches two proofs that $N!$ does not evenly divide N^N for $N > 2$, but (to my mind) both proofs contain gaps.[10] Fill in the gaps.
7. Do physicists think true randomness exists in nature?
8. Compare the performance of the example program in figure 4 when reading random numbers from my `RDSEED` program versus `/dev/urandom` or a hardwired PRNG.
9. "`ps -AF | sha512sum`" is a quick trick for harvesting entropy. How many bits of entropy does it yield?
10. Are keystroke, storage, or network timings reliable sources of entropy inside a virtual machine running under a hypervisor? Check out VirtIO-RNG and `/dev/hwrng`.
11. Is seed bias really a problem? Consider shuffling a standard 52-card deck of playing cards. If all 52! permutations are equally likely, what is the probability that the top five cards make a royal flush? Now compute that probability for a PRNG-based shuffle with a *B*-bit seed for several reasonable values of *B*. More generally, *characterize* the bias introduced by a small seed in terms of changes to the probabilities that players face. Does a *large* seed ever introduce bias?

12. PRNGs make it easy to reproduce runs of a randomized program by reusing saved seeds. Is reproduction possible for programs such as figure 4 that read random numbers from `stdin`? See the documentation for GNU `shuf`.
13. Study the Linux `/dev/random` and `/dev/urandom` PRNGs and `/proc/sys/kernel/random/`. How have these evolved since they were first introduced?
14. Durstenfeld's algorithm is still going strong after 60 years. What's the life expectancy of a PRNG? Are any PRNGs from the 1960s still used today? Are today's PRNGs expected to be usable until 2084?
15. If PRNG output "passes statistical tests of randomness," should we instead conclude that the tests have failed (i.e., failed to detect data that aren't truly random)?

Acknowledgments

I thank Jon Bentley and John Dilley for helpful conversations and for reviewing an outline, an early draft, and the example code; Kevin O'Malley for reviewing the example code; Skaff Elias for insights and pointers related to online gambling; and Suyash Mahar and Haris Volos for measuring the speed of the `RDSEED` instruction.

References

1. Bentley, J. 2000. *Programming Pearls*, 2nd edition. Chapter 12: A Sample Problem. ACM Press.
2. Boneh, D., Shoup, V. 2023. *A Graduate Course in Applied Cryptography*; https://toc.cryptobook.us/book.pdf.
3. Brown, G. W. 1949. History of RAND's random digits — summary. Technical Report P-113, RAND Corporation.

https://www.rand.org/pubs/papers/2008/P113.pdf.
4. Oracle Corporation. 2024. Choosing a PRNG algorithm; https://docs.oracle.com/en/java/javase/21/core/choosing-prng-algorithm.html.
5. Deitel, H. M., Deitel, P. J. 1994. *C: How to Program*, second edition. Prentice Hall. Modulo bias: pp. 160-163, 183, 215, 289, and 403; shuffle bias: pp. 402-404; seed bias: pp. 289 and 403.
6. Deitel, H. M., Deitel, P. J. 2023. *C: How to Program*, ninth edition (Global Edition). Pearson. Modulo bias: pp. 250-254, 396, and 545; shuffle bias: pp. 545-546 and 574; seed bias: p. 544; secure PRNGs: p. 275.
7. Diaconis, P. Fulman, J. 2023. *The Mathematics of Shuffling Cards*. American Mathematical Society.
8. Durstenfeld, R. 1964. Algorithm 235: random permutation. *Communications of the ACM 7(7)*. The entire article occupies the lower-right corner of page 420. https://dl.acm.org/doi/pdf/10.1145/364520.364540.
9. Yahoo! Finance. 2024. Online poker industry review and forecast to 2030; https://finance.yahoo.com/news/online-poker-industry-review-forecast-130200589.html.
10. Fisher-Yates shuffle. 2024. Wikipedia; https://en.wikipedia.org/wiki/Fisher–Yates_shuffle.
11. Intel digital random number generator: software implementation guide. 2018. https://www.intel.com/content/www/us/en/developer/articles/guide/intel-digital-random-number-generator-drng-software-implementation-guide.html.
12. Jones, D. 2010. Good practice in (pseudo) random number generation for bioinformatics

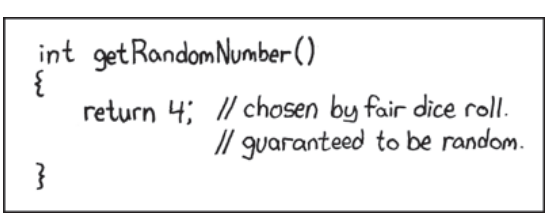
applications; http://www0.cs.ucl.ac.uk/staff/D.Jones/GoodPracticeRNG.pdf.
13. Kelly, T. 2020. Efficient graph search: stop when done. *acmqueue 18(4)*; https://queue.acm.org/detail.cfm?id=3424304.
14. Kelly, T. 2023. Catch-23: the new C standard sets the world on fire. *acmqueue 21(1)*, 12-29; https://dl.acm.org/doi/pdf/10.1145/3588242.
15. Kleptographically insecure PRNGs. 2024. Wikipedia; https://en.wikipedia.org/wiki/Cryptographically_secure_pseudorandom_number_generator#NSA_kleptographic_backdoor_in_the_Dual_EC_DRBG_PRNG.
16. Knuth, D. E., Yao, A. C. 2000. *Selected Papers on Analysis of Algorithms*, chapter 34: The Complexity of Nonuniform Random Number Generation, 545–603. Stanford Center for the Study of Language and Information. Updated version of 1976 paper.
17. Lemire, D. 2018. Fast random integer generation in an interval. https://arxiv.org/abs/1805.10941.
18. Lumbroso, J. 2013. Optimal discrete uniform generation from coin flips, and applications. arXiv 1304-1916; https://arxiv.org/abs/1304.1916.
19. Marsaglia, G. 1996. The Marsaglia random number CDROM. https://web.archive.org/web/20100612204426/http://stat.fsu.edu/pub/diehard/cdrom/pscript/cdmake.ps.
20. Menn, J. 2013. Secret contract tied NSA and security industry pioneer. Reuters. A tale of bribery, backdoors, and a "cryptographically secure" PRNG that wasn't; https://www.reuters.com/article/uk-usa-security-rsa-idUKBRE9BJ1CJ20131220/.

21. Minimum standards for gaming devices. 2023. Regulation 14, Section 14.040, paragraph 5. Nevada Gaming Commission and the Nevada Gaming Control Board; https://gaming.nv.gov/regs/statutes-regs/.
22. O'Connor, D. 2014. A historical note on shuffle algorithms. Academia; https://www.academia.edu/12056201/OConnor_A_Historical_Note_on_Shuffle_Algorithms.
23. Press, W. H., Teukolsky, S. A., Vetterling, W. T., Flannery, B. P. 2007. *Numerical Recipes: The Art of Scientific Computing*, third edition. Cambridge University Press. Chapter 7 covers PRNGs; pages 341–342 offer baseline advice. Not all of this book's advice is good; page 343 recommends `PRNG() % N`. PDF available at https://numerical.recipes/book.html.
24. Rigged backgammon [USENET news thread]. Google Groups; https://groups.google.com/g/rec.games.backgammon/c/Dh33pOhN-dE/m/OTHUPU49AAAJ.
25. Rosenbaum, D. E. 1970. Draft officials redesign lottery procedures to make the system more random. *The New York Times* (June 25), 17; https://timesmachine.nytimes.com/timesmachine/1970/06/25/issue.html.
26. Rosenbaum, D. E. 1970. Statisticians charge draft lottery was not random. *The New York Times* (January 4), 66. A classic case study on how to botch a *physical* shuffle. Second column contains typesetting bug (vertical rotation); https://timesmachine.nytimes.com/timesmachine/1970/01/04/issue.html.
27. Salter, J. 2010. How a months-old AMD microcode bug destroyed my weekend. arsTechnica;

https://arstechnica.com/gadgets/2019/10/how-a-months-old-amd-microcode-bug-destroyed-my-weekend/.

28. Selective service system lottery. https://www.sss.gov/about/return-to-draft/lottery/.
29. Starr, N. 1997. Nonrandom risk: the 1970 draft lottery. *Journal of Statistics Education 5(2)*; https://jse.amstat.org/v5n2/datasets.starr.html.
30. Thompson, K. 1984. Reflections on trusting trust. Turing Award lecture. *Communications of the ACM 27(8)*; https://dl.acm.org/doi/pdf/10.1145/358198.358210.
31. Viega, J., McGraw, G. 2002. *Building Secure Software*. Addison-Wesley. See pages 238–241 for attacks on poker.
32. Weir, R. 2010. Doing the Microsoft shuffle: algorithm fail in browser ballot; https://www.robweir.com/blog/2010/02/microsoft-random-browser-ballot.html. Noteworthy for the innovative use of an intransitive comparison function.
33. True Random Number Generator. MagPi magazine, issue 40 (December 2015). https://magpi.raspberrypi.com/issues/40

Terence Kelly *(tpkelly@acm.org) never leaves randomness to chance.*



https://xkcd.com/221/