13.7 Random Number Generation

Input description: Nothing, or perhaps a seed.

Problem description: Generate a sequence of random integers.

Discussion: Random numbers have an enormous variety of interesting and important applications. They form the foundation of simulated annealing and related heuristic optimization techniques. Discrete event simulations run on streams of random numbers, and are used to model everything from transportation systems to casino poker. Passwords and cryptographic keys are typically generated randomly. Randomized algorithms for graph and geometric problems are revolutionizing these fields and establishing randomization as one of the fundamental ideas of computer science.

Unfortunately, generating random numbers looks a lot easier than it really is. Indeed, it is fundamentally impossible to produce truly random numbers on any deterministic device. Von Neumann [Neu63] said it best: "Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin." The best we can hope for are *pseudorandom* numbers, a stream of numbers that appear as if they were generated randomly.

There can be serious consequences to using a bad random-number generator. In one famous case, a Web browser's encryption scheme was broken with the discovery that the seeds of its random-number generator employed too few random bits [GW96]. Simulation accuracy is regularly compromised or invalidated by poor random number generation. This is an area where people shouldn't mess around, but they do. Issues to think about include:

- *Should my program use the same "random" numbers each time it runs?* – A poker game that deals you the exact same hand each time you play quickly loses interest. One common solution is to use the lower-order bits of the machine clock as the *seed* or starting point for a stream of random numbers, so that each time the program runs it does something different.

Such methods are adequate for games, but not for serious simulations. There are liable to be periodicities in the distribution of random numbers whenever calls are made in a loop. Also, debugging is seriously complicated when program results are not repeatable. Should your program crash, you cannot go back and discover why. A possible compromise is to use a deterministic pseudorandom-number generator, but write the current seed to a file between runs. During debugging, this file can be overwritten with a fixed initial value of the seed.

- *How good is my compiler's built-in random number generator?* – If you need uniformly-generated random numbers, and won't be betting the farm on the accuracy of your simulation, my recommendation is simply to use what your compiler provides. Your best opportunity to mess things up is with a bad choice of starting seed, so read the manual for its recommendations.

  If you *are* going to bet the farm on the results of your simulation, you had better test your random number generator. Be aware that it is very difficult to eyeball the results and decide whether the output is really random. This is because people have very skewed ideas of how random sources should behave and often see patterns that don't really exist. Several different tests should be used to evaluate a random number generator, and the statistical significance of the results established. The National Institute of Standards and Technology (NIST) has developed a test suite for evaluating random number generators, discussed below.

- *What if I must implement my own random-number generator?* – The standard algorithm of choice is the *linear congruential generator*. It is fast, simple, and (if instantiated with the right constants) gives reasonable pseudorandom numbers. The nth random number R_n is a function of the $(n-1)$st random number:

  $$R_n = (aR_{n-1} + c) \bmod m$$

  In theory, linear congruential generators work the same way roulette wheels do. The long path of the ball around and around the wheel (captured by $aR_{n-1} + c$) ends in one of a relatively small number of bins, the choice of which is extremely sensitive to the length of the path (captured by the mod m-truncation).

  A substantial theory has been developed to select the constants a, c, m, and R_0. The period length is largely a function of the modulus m, which is typically constrained by the word length of the machine.

  Note that the stream of numbers produced by a linear congruential generator repeats the instant the first number repeats. Further, computers are fast enough to make 2^{32} calls to a random-number generator in a few minutes. Thus, any 32-bit linear congruential generator is in danger of cycling, motivating generators with significantly longer periods.

- *What if I don't want such large random numbers?* – The linear congruential generator produces a uniformly-distributed sequence of large integers that can be easily scaled to produce other uniform distributions. For uniformly distributed real numbers between 0 and 1, use R_i/m. Note that 1 cannot be realized this way, although 0 can. If you want uniformly distributed integers between l and h, use $\lfloor l + (h - l + 1)R_i/m \rfloor$.

- *What if I need nonuniformly distributed random numbers?* – Generating random numbers according to a given nonuniform distribution can be a tricky business. The most reliable way to do this correctly is the acceptance-rejection method. We can bound the desired geometric region to sample from by a box and then select a random point p from the box. This point can be generated by selecting the x and y coordinates independently at random. If it lies within the region of interest, we can return p as being selected at random. Otherwise we throw it away and repeat with another random point. Essentially, we throw darts at random and report those that hit the target.

  This method is correct, but it can be slow. If the volume of the region of interest is small relative to that of the box, most of our darts will miss the target. Efficient generators for Gaussian and other special distributions are described in the references and implementations below.

  Be cautious about inventing your own technique, however, since it can be tricky to obtain the right probability distribution. For example, an *incorrect* way to select points uniformly from a circle of radius r would be to generate polar coordinates by selecting an angle from 0 to 2π and a displacement between 0 and r—both uniformly at random. In such a scheme, half the generated points will lie within $r/2$ of the center, when only one-fourth of them should be! This is different enough to seriously skew the results, while being sufficiently subtle that it can easily escape detection.

- *How long should I run my Monte Carlo simulation to get the best results?* – The longer you run a simulation, the more accurately the results should approximate the limiting distribution, thus increasing accuracy. However, this is true only until you exceed the *period*, or cycle length, of your random-number generator. At that point, your sequence of random numbers repeats itself, and further runs generate no additional information.

  Instead of jacking up the length of a simulation run to the max, it is usually more informative to do many shorter runs (say 10 to 100) with different seeds and then consider the range of results you see. The variance provides a healthy measure of the degree to which your results are repeatable. This exercise corrects the natural tendency to see a simulation as giving "the" correct answer.

Implementations: See *http://random.mat.sbg.ac.at* for an excellent website on random-number generation and stochastic simulation. It includes pointers to papers and literally dozens of implementations of random-number generators.

Parallel simulations make special demands on random-number generators. How can we ensure that random streams are independent on each machine? L'Ecuyer et.al. [LSCK02] provide object-oriented generators with a period length of approximately 2^{191}. Implementations in C, C++, and Java are available at *http://www.iro.umontreal.ca/~lecuyer/myftp/streams00/*. Independent streams of random numbers are supported for parallel applications. Another possibility is the *Scalable Parallel Random Number Generators Library (SPRNG)* [MS00], available at *http://sprng.cs.fsu.edu/*.

Algorithms 488 [Bre74], 599 [AKD83], and 712 [Lev92] of the *Collected Algorithms of the ACM* are Fortran codes for generating non-uniform random numbers according to several probability distributions, including the normal, exponential, and Poisson distributions. They are available from Netlib (see Section 19.1.5 (page 659)).

The National Institute of Standards [RSN+01] has prepared an extensive statistical test suite to validate random number generators. Both the software and the report describing it are available at *http://csrc.nist.gov/rng/*.

True random-number generators extract random bits by observing physical processes. The website *http://www.random.org* makes available random numbers derived from atmospheric noise that passes the NIST statistical tests. This is an amusing solution if you need a small quantity of random numbers (say, to run a lottery) instead a random-number generator.

Notes: Knuth [Knu97b] provides a thorough treatment of random-number generation, which I heartily recommend. He presents the theory behind several methods, including the middle-square and shift-register methods we have not described here, as well as a detailed discussion of statistical tests for validating random-number generators.

That said, see [Gen04] for more recent developments in random number generation. The Mersenne twister [MN98] is a fast random number generator of period $2^{19937} - 1$. Other modern methods include [Den05, PLM06]. Methods for generating nonuniform random variates are surveyed in [HLD04]. Comparisons of different random-number generators in practice include [PM88].

Tables of random numbers appear in most mathematical handbooks as relics from the days before there was ready access to computers. Most notable is [RC55], which provides one million random digits.

The deep relationship between randomness, information, and compressibility is explored within the theory of Kolmogorov complexity, which measures the complexity of a string by its compressibility. Truly random strings are incompressible. The string of seemingly random digits of π cannot be random under this definition, since the entire sequence is defined by any program implementing a series expansion for π. Li and Vitáni [LV97] provide a thorough introduction to the theory of Kolmogorov complexity.

Related Problems: Constrained and unconstrained optimization (see page 407), generating permutations (see page 448), generating subsets (see page 452), generating partitions (see page 456).