CMU-CS-79-113

Generating Sorted Lists of Random Numbers

Jon Louis Bentley[1]
James B. Saxe
Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

Abstract

The empirical testing of a program often calls for generating a set of random numbers and then immediately sorting them. In this paper we consider the problem of accomplishing that process in a single step: generating a sorted list of random numbers (specifically, reals chosen uniformly from [0,1]). The method we describe generates the randoms in linear time, is perfectly random (if it can call a perfectly random generator for a single uniform), and can be described in just three lines of Algol or Pascal code. If the numbers are not required to be generated all at once (but are rather to be used one-at-a-time), then the method can be implemented as a subroutine to produce the "next" number and requires only constant storage.

Key Words and Phrases: random number generation, sorting, probabilistic methods in algorithm design, linear-time algorithms.

CR Categories: 5.25, 5.31, 5.5

---

1. Also with the Department of Mathematics.

This research was supported in part by the Office of Naval Research under Contract N0014-76-C-0370.

1. Introduction

The first step of many computer algorithms is to sort the input data. When testing these programs to determine runtimes empirically, one usually generates N random numbers and then sorts them. The efficiency of sorting algorithms is well-known (see Knuth [1973]), however, so it is often not necessary to test the sorting procedure empirically in a particular program. In this application (as well as many others), it is desirable to generate a sorted list of random numbers as quickly as possible. In this paper we will study the problem of generating a sorted list of N reals drawn uniformly from [0,1].

The most obvious method for generating a sorted list of randoms is to first generate N randoms (see Knuth [1969, Chapter 3]) and then sort them. This method requires time proportional to N lg N in the worst case, but this can be reduced to linear expected time if a "bucket" sort is used (see Knuth [1973, Section 5.2.1]). This linear expected time algorithm is rather complicated to code, and requires extra space proportional to N. A knowledge of elementary probability theory, however, allows one to use more sophisticated approaches.

In this paper we will investigate a new method for generating sorted lists of randoms that has significant advantages over all previous approaches. We will begin by discussing previous work in Section 2. In Section 3 we will study some important probabilistic lemmas, and then show in Section 4 how these can be used to make efficient programs. A discussion of this approach is offered in Section 5.

2. Previous Work

Before presenting our new algorithms for generating sorted lists of random numbers, we will mention, for purposes of completeness and of comparison, the best previously known method for generating sorted lists of random numbers. Although the method seems to be well-known among statisticians, the present authors are unable to find a description of its computational aspects in the statistical literature. The algorithm is based on the following lemma.

Lemma 1:

If $X_1, X_2, \ldots, X_{n+1}$ are independent random variables with exponential distribution of any fixed mean, then the values

$$Y_j = [\sum_{1\le i\le j} X_i] \;/\; [\sum_{1\le i\le n+1} X_i]$$

for $j = 1, \ldots, n$ are distributed as the order statistics of a random sample of size n from U[0,1].

Proof:

We omit the proof of this lemma as it is well-known (see, for example, Johnson and Kotz [1970, Chapter 18]) and is not essential to the main thrust of this paper. □

An algorithm derived from Lemma 1 is described by the following pseudo-Pascal code. It assumes that RAND is a function that on each call returns an independent random number from the uniform distribution on [0,1]; a random exponential is then achieved by negating the natural logarithm of RAND. The effect of the algorithm is to fill elements 1..N of the array X with sorted random numbers independently drawn from U[0,1].

```
Sum ← 0;
for I ← 1 to N do
    X[I] ← Sum ← Sum - ln(RAND);
Sum ← Sum - ln(RAND);
for I ← 1 to N do
    X[I] ← X[I]/Sum;
```

Program 1. Filling an array with sorted randoms.

It is obvious that this method is a very efficient way of generating sorted lists of numbers chosen uniformly on [0,1]. Its one computational disadvantage, however, is that it is inherently a two-pass algorithm--the first to place the numbers into the array and the second to normalize them. We will now turn our attention to a new, single-pass algorithm.

3. Probabilistic Arguments

The correctness of the algorithms to be presented in Section 4 rests on the following two lemmas. Lemma 2 will allow us to generate, in constant time, the largest of n independent uniformly distributed random numbers. Lemma 3 shows that once we have generated the k largest of n independent uniform randoms, the problem of generating the $k+1^{st}$ largest reduces to the problem of generating the largest of n-k independent uniform randoms.

Lemma 2:

The probability distribution of the maximum of n independent random numbers from the distribution U[0,1] is the same as that of the n^{th} root of a single number from U[0,1].

Proof:

Note that the both distributions mentioned range over the interval [0,1]. Let $q \in [0,1]$. It suffices to show that numbers from either distribution have equal probability of being in [0,q].

If X is drawn from U[0,1], then $P(X^{1/n}<q) = P(X<q^n) = q^n$. On the other hand, the largest of a set of n numbers in [0,1] will lie in [0,q] iff all n lie in [0,q]. Since the probability of a single number drawn from U[0,1] will lie in [0,q] is q, it follows that the probability of n numbers drawn independently from U[0,1] all being less than q is also q^n. □

Lemma 3:

Let n and k be positive integers, $n < k$. Let $y_1, \ldots, y_k$ be elements of [0,1] such that $y_1 \geq y_2 \geq \ldots \geq y_k$. Then, for n random numbers $X_1, \ldots, X_n$ chosen independently from U[0,1] the distribution, conditional on the largest k being $y_1, \ldots, y_k$, of the $k+1^{st}$ largest is the same as the distribution of the largest of n-k numbers uniformly selected from $[0,y_k]$.

Proof[1]:

We note first that the probability of the k+1st largest of the X_i being equal to y_k is zero, as is the probability of the largest of n-k independent draws from $U[0,y_k]$ being equal to y_k. It remains to consider the case where the k+1st largest of the X_i lies in $[0,y_k)$.

Consider the event space of all sets of n independent draws from $U[0,1]$. The subspace containing all events wherein the largest k numbers drawn are $y_1, \ldots, y_k$ may be partitioned into a number of equivalence classes.[2] Each such equivalence class may be obtained by assigning the y_i to k of the X_i, picking all events from the full space which satisfy these assignments, and throwing away all events in which any of the n-k "unspecified" X_i happen to be larger than or equal to y_k. Thus, the distribution of the smallest n-k entries, within each equivalence class, is precisely the distribution of n-k independent draws from $U[0,y_k)$. Since there are finitely many equivalence classes, it follows that the distribution of the n-k smallest entries, within the union of all equivalence classes (*i.e.*, contingent only on the k largest draws being $y_1, \ldots, y_k$ and on the k+1st being less than y_k) is identical to the distribution of n-k independent draws from $U[0,y_k)$. This completes consideration of the case in which the k+1st largest of the X_i is in $[0,y_k)$, so we are done. □

4. Programs

In this section we will see how the basic probabilistic facts discussed in the last section can be used to make programs for generating sorted lists of randoms. In all these programs we will assume that we have a subroutine RAND that returns a random number drawn uniformly from $[0,1]$. All the programs that we will describe produce correct output in the sense that if RAND satisfies the probabilistic definition of $U[0,1]$, then the output of our program will satisfy the probabilistic definitions of a

[1] Since this paper is intended primarily for non-statisticians, we have attempted to minimize statistical notation in the presentation of this lemma, at the expense of conciseness. A more general form of this well-known result is more formally presented as Theorem 2.7 of David [1970].

[2] This number (the number of event classes) may range from 1 to $n!/(n-k)!$, depending on the number and pattern of equalities among the y_k.

sorted list of N such randoms.

Lemma 2 of Section 3 allows us to generate the maximum of N uniforms in [0,1] by evaluating $\text{RAND}^{1/N}$, which we will call CurMax (for reasons soon to become obvious). After we have done that, Lemma 3 allows us to solve the remainder of the problem by generating N-1 randoms uniform on [0,CurMax]. We can accomplish this by taking as the maximum the value of $\text{CurMax}\cdot\text{RAND}^{1/(N-1)}$, and so forth. This process can be described precisely by the following program in pseudo-Pascal, which places the random numbers into the array X in decreasing order.

```
CurMax ← 1.0;
for I ← N downto 1 do
    X[I] ← CurMax ← CurMax * $\text{RAND}^{1/I}$;
```

Program 2. Straightforward implementation.

In the above program the variable CurMax represents the current maximum of the range in which I randoms are to be generated. (A program essentially equivalent to Program 2 was described by Friedman [1971] for use in random event generation in a physics context. He did not, however, observe the generality of his method.)

In Program 2 we exponentiate to a fractional power. Since most programming languages do not directly support such a statement, this step is usually implemented as

```
X[I] ← CurMax ← CurMax * exp(ln(RAND)/I).
```

The multiplication in that statement might be a source of numerical error, so it can be replaced by an addition as in the following program to fill X with sorted randoms.

```
LnCurMax ← 0.0;
for I ← N downto 1 do
    begin
    LnCurMax ← LnCurMax + ln(RAND)/I;
    X[I] ← exp(LnCurMax)
    end;
```

Program 3. Multiplication replaced by addition.

Note that with perfect arithmetic this procedure will produce exactly the same output as Program 2 (assuming the use of the same procedure RAND); this program, however, is numerically more robust than its predecessor.

In many applications the variables are not all needed at one time, but rather can be used "on the fly". If this is indeed the case, then using the N array words of X is very wasteful of storage. We would prefer to have an algorithm that can generate the "next" value. We will now describe such an algorithm as two Pascal subroutines. Procedure InitSorted is passed an integer N and initializes the global variables I (an integer) and LnCurMax (a real); NextSorted is a parameterless function that returns the next value in decreasing order (unless N values have already been returned).

```
procedure InitSorted(N: integer);
     begin
     I ← N;
     LnCurMax ← 0.0
     end;

function NextSorted: real;
     begin
     if I <= 0 then Abort;
     LnCurMax ← LnCurMax + ln(RAND)/I;
     I ← I-1;
     NextSorted ← exp(LnCurMax)
     end;
```

Program 4. On-line generation of sorted randoms.

Making N successive calls on NextSorted after executing InitSorted(N) will produce exactly the same output as executing either Program 2 or Program 3 (although not in the array X). If an N+1st call is made on NextSorted then abnormal termination will be effected by calling procedure Abort. As this algorithm is stated it returns the values in decreasing order; if increasing order is preferred then this can be accomplished by subtracting the result from one.

5. Discussion

Programs 3 and 4 of the previous section have been implemented as Pascal programs; these programs are described by Bentley and Saxe [1979]. Both implementations required approximately 250 microseconds to generate a single random number when executed on a Digital Equipment Corporation PDP-10 KL processor.[1] To compare these programs to more straightforward methods of solving this problem we wrote a program that generates an array of N random uniforms and then uses Quicksort to sort the array. The implementation of Program 3 was somewhat slower than the sorting methods for values up to N = 250; after that point Program 3 is faster. A significant advantage of our programs over the naive methods, however, is that while the sorting algorithm was described by some 80 lines of Pascal code, our method requires only a dozen lines. To ensure that the randomness properties of our algorithms were not adversely affected by roundoff errors or by using a linear-congruential psuedo-random number generator, we ran a number of statistical tests to determine the randomness of the resulting numbers--all tests were passed with flying colors.

Throughout this paper our programs have taken logarithms of real numbers uniformly distributed on [0,1]. Notice that this leads to an undefined result if the value of the random number is zero. Although this does not affect the theory underlying the paper (since we only took such logarithms to "simulate" fractional exponentiation or generate exponentially distributed randoms), this will affect programs implementing these methods. Such programs should take logarithms of randoms uniform on (0,1]. Since most RAND subroutines return values uniform on [0,1), this can be accomplished by using 1-RAND as the desired random number.

Although it is clear that the method of Program 3 is superior to a

[1]The Pascal compiler used in these tests does not produce very efficient code; the authors suspect that the speed of the programs could be substantially increased by careful hand-coding. This is unnecessary in most applications, however, since the use of this method is usually enough to remove the process of generating sorted randoms from the time bottleneck of the program.

generate-and-sort solution in almost all applications, it is more difficult to compare Program 3 with Program 1. Program 1 is faster than Program 3 (Program 1 uses an addition, a logarithm, a multiplication, and three array accesses for each random; Program 3 uses an additional exponentiation, but only one array access), but Program 3 is shorter to code. The primary advantage of the method of Section 4 over Program 1 is that this method can be implemented on-line; the method of Program 1 has no on-line version corresponding to Program 4.

Although we have described our method for generating sorted lists of uniform random numbers, the same method can be extended to generate sorted numbers from other distributions. To generate numbers from distribution F for which the inverse F^{-1} is known, it is only necessary to apply the monotone function F^{-1} to each of the outputs of Programs 3 or 4, and the resulting sorted list will satisfy all the desired properties.

Acknowledgements

The helpful comments of Bill Eddy, Don Knuth, and Larry Rafsky are gratefully acknowledged.

References

Bentley, J. L. and J. B. Saxe [1979]. "Algorithm. Generating sorted lists of randoms," attached.

David, H. A. [1970]. *Order Statistics*, John Wiley and Sons, New York, New York.

Friedman, J. H. [1971]. "Random event generation with preferred frequency distributions," *Journal of Computational Physics* 7, 2, (April 1971), pp. 201-218.

Johnson, N. L. and S. Kotz [1970]. *Continuous Univariate Distributions-2*, Houghton-Mifflin, Boston, Massachussetts.

Knuth, D. E. [1969]. *The Art of Computer Programming, volume 2: Seminumerical*

Algorithms, Addison-Wesley, Reading, Massachusetts.

Knuth, D. E. [1973]. *The Art of Computer Programming, volume 3: Sorting and Searching*, Addison-Wesley, Reading, Massachusetss.

ALGORITHM. Generating Sorted Lists of Randoms

DESCRIPTION

This Pascal program implements two algorithms described by Bentley and Saxe [1979] for generating sorted lists of random numbers. The theory underlying these algorithms can be found in that paper.

REFERENCE

Bentley, J. L. and J. B. Saxe [1979]. "Generating sorted lists of random numbers," attached.

ALGORITHM

```
(*          ROUTINES FOR GENERATION OF SORTED RANDOM NUMBERS          *)

(* The algorithms  used in this program  are taken from "Generating
   sorted lists  of  random numbers",  hereinafter referred  to  as
   "Bentley and Saxe."  The reader should refer to that article for
   a discussion and justification of the algorithms.  The procedure
   GenSorted implements  Program 3 of Bentley  and Saxe for filling
   an  array with  sorted random  numbers uniformly  drawn from the
   interval  [0,1].   The procedure  InitSorted  and  the  function
   NextSorted together implement Program  4 of Bentley and Saxe for
   generating sorted random numbers on-line.  The main program is a
   test driver which exercises these routines.                     *)


const
    MaxRands = 100;        (* Maximum number of random numbers
                              generated by Gensorted *)
    TestSize = 25;         (* Number of sorted randoms to generate--
                              used by test driver.  (Note: TestSize
                              must be <= MaxRands) *)

type
    RandArray =  array [1..MaxRands] of real;
```

```
var
    (* Variables for  on-line generation  of sorted randoms.  These
       variable  names are the same  as those used in  Program 4 of
       Bentley  and Saxe,  except that  they have  been preceded by
       "OLG" (for On-Line  Generation) to lessen the probability of
       name  conflicts with other global  variables which may occur
       in programs using the on-line generation routines.            *)
    OLGI: integer;        (* The next random number generated by
                             NextSorted will be the OLGI-th
                             smallest. *)
    OLGLnCurMax: real;    (* The natural logarithm of the previous
                             number generated by NextSorted.  Before
                             the first call of a sequence, LnCurMax
                             is set to 0, i.e., ln(1). *)

    (* Variables used by driver *)
    J: integer;
    Y: RandArray;

    (* Storage used by underlying random number generator *)
    RandHold: integer;


(* Procedures for generation of uniform random numbers                 *)
(* The  built-in function, Random, takes  a single integer argument
   and returns a pseudo-random real number in the range [0,1).  The
   argument (here  named RandHold) is a VAR  parameter used to hold
   the current random seed, and  is altered by each call to Random.  *)
(* Note:  The function  Random is  not a  Standard Pascal  built-in
          function.  At your site the random number function may go
          by a different name, or  it may even be necessary for you
          to write your own.   (See CALGO Algorithms 266 and 294 or
          Section  3.2 of Knuth's The  Art of Computer Programming,
          Volume  2:   Semi-Numerical  Algorithms,  Addison-Wesley,
          1969.)   Also,  the method  of  initializing  the  random
          number generator  may vary from site  to site.  In short,
          the programmer should be prepared to rewrite the routines
          Rand  and  InitRand  to  conform  to  the  local  runtime
          environment.                                                 *)

procedure InitRand;
    begin
    RandHold := 0
    end;

function Rand: real;
    begin
    Rand := 1-Random(RandHold)  (* return a number in (0,1]. *)
    end;
```

```
(* Routine  to place  N random  numbers uniformly  drawn from [0,1)
   into X[1..N] in ascending order.                                  *)
(* The  algorithm used  here is  that of  Program 3  of Bentley and
   Saxe.                                                             *)

procedure GenSorted(var X: RandArray; N: integer);
    var
        I: integer;
        LnCurMax: real;
    begin
    LnCurMax := 0.0;
    for I := N downto 1 do
        begin
        LnCurMax := LnCurMax + ln(Rand)/I;
        X[I] := exp(LnCurMax)
        end
    end;


(* Routines to generate sorted randoms on-line                       *)
(* To  generate N random  numbers from [0,1],  sorted in descending
   order:  First  call InitSorted(N).  The next  N evaluations  of
   NextSorted will return  the random numbers, in descending order.
   If  InitSort is called  again before  N calls have  been made to
   NextSorted, the  current sequence of randoms will  be lost and a
   new sequence will begin with the next call to NextSorted.         *)
(* Note: If an ascending sequence of random numbers is desired, the
   final  assignment statement  of Nextsorted should  be altered to
   read "NextSorted := 1 - exp(OLGLnCurMax)".                        *)
(* The algorithms used here are from Program 4 of Bentley and Saxe.
   The  global variable  names, I  and LnCurMax,  occurring in that
   program have here been  preceded by "OLG" to guard against their
   accidental use by other pieces of code.                           *)

procedure InitSorted(N: integer);
    begin
    OLGI := N;
    OLGLnCurMax := 0.0
    end;

function NextSorted: real;
    begin
    if OLGI <= 0 then
        begin
        writeln(tty, 'Too many calls on NextSorted, Aborted');
        halt
        end;
    OLGLnCurMax := OLGLnCurMax + ln(Rand)/OLGI;
    OLGI := OLGI-1;
    NextSorted := exp(OLGLnCurMax)
    end;
```

```
(* Driver to test both single-shot and on-line generation of sorted
   randoms                                                           *)

begin (* main program *)
InitRand;

(* Test  single-shot  generation  of  sorted  randoms  by  filling
   Y[1..TestSize] with ascending  sorted random numbers and dumping
   results to output file.                                           *)
GenSorted(Y, TestSize);
writeln('Dump of array Y after execution of GenSorted');
for J := 1 to TestSize do
    writeln(J, Y[J]:10:5);

(* Test on-line generation of sorted randoms by generating TestSize
   random numbers  in descending order and  writing them to output.  *)
writeln;
writeln('Commencing test of on-line generation');
InitSorted(TestSize);
for J := 1 to TestSize do
    writeln(J, NextSorted:10:5)
end.
```

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER CMU-CS-79-113	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) GENERATING SORTED LISTS OF RANDOM NUMBERS		5. TYPE OF REPORT & PERIOD COVERED Interim
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) J.L. Bentley and J. B. Saxe		8. CONTRACT OR GRANT NUMBER(s) N00014-76-C-0370
9. PERFORMING ORGANIZATION NAME AND ADDRESS Carnegie-Mellon University Computer Science Department Pittsburgh, PA 15213		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Office of Naval Research Arlington, VA 22217		12. REPORT DATE March 1979
		13. NUMBER OF PAGES 16
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Same as above		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT (of this Report) Approved for public release; distribution unlimited		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)		
18. SUPPLEMENTARY NOTES		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number)		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number)		

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)