Lloyd Fosdick
Guest Editor

Faster Methods for Random Sampling

JEFFREY SCOTT VITTER

ABSTRACT: Several new methods are presented for selecting n records at random without replacement from a file containing N records. Each algorithm selects the records for the sample in a sequential manner—in the same order the records appear in the file. The algorithms are online in that the records for the sample are selected iteratively with no preprocessing. The algorithms require a constant amount of space and are short and easy to implement. The main result of this paper is the design and analysis of Algorithm D, which does the sampling in O(n) time, on the average; roughly n uniform random variates are generated, and approximately n exponentiation operations (of the form a^b, for real numbers a and b) are performed during the sampling. This solves an open problem in the literature. CPU timings on a large mainframe computer indicate that Algorithm D is significantly faster than the sampling algorithms in use today.

1. INTRODUCTION

Many computer science and statistics applications call for a sample of n records selected randomly without replacement from a file containing N records or for a random sample of n integers from the set $\{1, 2, 3, \ldots, N\}$. Both types of random sampling are essentially equivalent; for convenience, in this paper we refer to the former type of sampling, in which records are selected. Some important uses of sampling include market surveys, quality control in manufacturing, and probabilistic algorithms. Interest in this subject stems from work on a new external sorting method called BucketSort that uses random sampling for preprocessing [7].

One way to select the n records is to generate an independent random integer k between 1 and N and to select the kth record if it has not already been selected; this process is repeated until n records have been selected. (If $n > N/2$, it is faster to select the $N - n$ records *not* in the sample.) This is an example of a *nonsequential algorithm* because the records in the sample might not be selected in linear order. For example, the 84th record in the file may be selected before the 16th record in the file is selected. The algorithm requires the generation of $O(n)$ uniform random variates, and it runs in $O(n)$ time if there is enough extra space to check in constant time whether the kth record has already been selected. The checking can be done in $O(N)$ space using a bit array or with $O(n)$ pointers using hashing techniques (e.g., [2, 6]). In either case, the space required may be prohibitive.

Often, we want the n records in the sample to be in the same order that they appear in the file so that they can be accessed sequentially, for example, if they reside on disk or tape. In order to accomplish this using a nonsequential algorithm, we must sort the records by their indices *after* the sampling is done. This requires $O(n \log n)$ time using a comparison-based sorting algorithm like quicksort or heapsort; address-calculation sorting can reduce the sorting time to $O(n)$, on the average, but it requires space for $O(n)$ pointers. Nonsequential algorithms thus take nonlinear time, or their space requirements are very large and the algorithm is somewhat complicated.

Some of this research was done while the author was consulting for the IBM Palo Alto Scientific Center. Support was also provided in part by NSF Research Grant MCS-81-05324, by an IBM research contract, and by ONR and DARPA under Contract N00014-83-K-0146 and ARPA Order No. 4786. An extended abstract of this research appears in [10].

More importantly, the n records cannot be output in sequential order *online*: It takes $O(n)$ time to output the first element since the sorting can begin only after all n records have been selected.

The alternative we take in this paper is to investigate *sequential* random sampling algorithms, which select the records in the same order that they appear in the file. The sequential sampling algorithms in this paper are ideally suited to online use since they iteratively select the next record for the sample in an efficient way. They also have the advantage of being extremely short and simple to implement.

The measure of performance we use for the algorithms is CPU time, not I/O time. This is reasonable for records stored on random-access devices like RAM or disk since all the algorithms take $O(n)$ I/O time in this case. It is reasonable for tape storage as well since many tape drives have a fast-forward speed that can quickly skip over unwanted records. In terms of sampling n integers out of N, the I/O time is insignificant because there is no file of records being read.

The main result of this paper is the design and analysis of a fast new algorithm, called Algorithm D, which does the sequential sampling in $O(n)$ time, on the average. This yields the optimum running time up to a constant factor, and it solves the open problem listed in exercise 3.4.2–8 in [6]. Approximately n uniform random variates are generated during the algorithm, and roughly n exponentiation operations (of the form $a^b = \exp(b \ln a)$, for real numbers a and b) are performed. The method is much faster than the previously fastest-known sequential algorithm, and it is faster and simpler than the nonsequential algorithms mentioned above.

In the next section, we discuss Algorithm S, which up until now was the method of choice for sequential random sampling. In Section 3, we state and analyze three new methods (Algorithms A, B, and C); the main result, Algorithm D, is presented in Section 4. The naive implementation of Algorithm D requires the generation of approximately $2n$ uniform random variates and the computation of roughly $2n$ exponentiation operations. One of the optimizations given in Section 5 reduces both counts from $2n$ to n. The analysis in Section 6 shows that the running time of Algorithm D is linear in n. The performance of Algorithm S and the four new methods is summarized in Table I.

Section 7 gives CPU timings for FORTRAN 77 implementations of Algorithms S, A, C, and D on a large mainframe IBM 3081 computer system; the running times of these four algorithms (in microseconds) are approximately $16N$ (Algorithm S), $4N$ (Algorithm A), $8n^2$ (Algorithm C), and $55n$ (Algorithm D). In Section 8, we draw conclusions and discuss related work. The Appendix gives the Pascal-like versions of the FORTRAN programs used in the CPU timings. A summary of this work appears in [10].

TABLE I: Performance of Algorithms

Algorithm	Average Uniform Random Variates	Average Running Time
S	$\frac{(N+1)n}{n+1}$	$O(N)$
A	n	$O(N)$
B	n	$O\left(n^2 \log\log\left(\frac{N}{n}\right)\right)$
C	$\frac{n(n+1)}{2}$	$O(n^2)$
D	$\approx n$	$O(n)$

2. ALGORITHM S

In this section, the sequential random sampling method introduced in [3, 4] is discussed. The algorithm sequentially processes the records of the file and determines whether each record should be included in the sample. When n records have been selected, the algorithm terminates. If m records have already been selected from among the first t records in the file, the $(t + 1)$st record is selected with probability

$$\binom{N-t-1}{n-m-1} \bigg/ \binom{N-t}{n-m} = \frac{n-m}{N-t}. \qquad (2\text{-}1)$$

In the implementation below, the values of n and N decrease during the course of execution. All of the algorithms in this paper follow the convention that *n is the number of records remaining to be selected and N is the number of records that have not yet been processed.* (This is different from the implementations of Algorithm S in [3, 4, 6] in which n and N remain constant, and auxiliary variables like m and t in (2-1) are used.) With this convention, the probability of selecting the next record for the sample is simply n/N. This can be proved directly by the following short but subtle argument: If at any given time we must select n more records at random from a pool of N remaining records, then the next record should be chosen with probability n/N.

The algorithms in this paper are written in an English-like style used by the majority of papers on random sampling in the literature. In addition, Pascal-like implementations are given in the Appendix.

ALGORITHM S. This method sequentially selects n records at random from a file containing N records, where $0 \le n \le N$. The uniform random variates generated in Step S1 must be independent of one another.

S1. [Generate U.] Generate a random variate U that is uniformly distributed between 0 and 1.

S2. [Test.] If $NU > n$, go to Step S4.

S3. [Select.] Select the next record in the file for the sample, and set $n := n - 1$ and $N := N - 1$. If $n > 0$, then return to Step S1; otherwise, the sample is complete and the algorithm terminates.

S4. [Don't select.] Skip over the next record (do not include it in the sample), set $N := N - 1$, and return to Step S1. ∎

Before the algorithm is run, each record in the file has the same chance of being selected for the sample. Furthermore, the algorithm never runs off the end of the file before n records have been chosen: If at some point in the algorithm we have $n = N$, then each of the remaining n records in the file will be selected for the sample with probability one. The average number of uniform random variates generated by Algorithm S is $(N + 1)n/(n + 1)$, and the average running time is $O(N)$. Algorithm S is studied further in [6].

3. THREE NEW SEQUENTIAL ALGORITHMS

We define $S(n, N)$ to be the random variable that counts the number of records to *skip over* before selecting the next record for the sample. The parameter n is the number of records remaining to be selected, and N is the total number of records left in the file. In other words, the $(S(n, N) + 1)$st record is the next one selected. Often we will abbreviate $S(n, N)$ by S in which case the parameters n and N will be implicit.

In this section, three new methods (Algorithms A, B, and C) for sequential random sampling are presented and analyzed. A fourth new method (Algorithm D), which is the main result of this paper, is described and analyzed in Sections 4 and 5. Each method decides which record to sample next by generating S and by skipping that many records. The general form of all four algorithms is as follows:

Step 1. Generate a random variate $S(n, N)$.

Step 2. Skip over the next $S(n, N)$ records in the file and select the following one for the sample. Set $N := N - S(n, N) - 1$ and $n := n - 1$. Return to Step 1 if $n > 0$.

The four methods differ from one another in how they perform Step 1. Generating S involves generating one or more random variates that are uniformly distributed between 0 and 1. As in Algorithm S in the last section, *all uniform variates are assumed to be independent of one another.*

The range of $S(n, N)$ is the set of integers in the interval $0 \le s \le N - n$. The distribution function $F(s) = \text{Prob}\{S \le s\}$, for $0 \le s \le N - n$, can be expressed in two ways:

$$F(s) = 1 - \frac{(N-s-1)^{\underline{n}}}{N^{\underline{n}}} = 1 - \frac{(N-n)^{\underline{s+1}}}{N^{\underline{s+1}}}. \qquad (3\text{-}1)$$

(We use the notation $a^{\underline{b}}$ to denote the "falling power" $a(a-1)\ldots(a-b+1) = a!/(a-b)!$.) We have $F(s) = 0$ for $s < 0$ and $F(s) = 1$ for $s \ge N - n$. The two formulas in (3-1) follow by induction from the relation

$$\begin{aligned} 1 - F(s) &= \text{Prob}\{S > s\} \\ &= \text{Prob}\{S > s - 1\}\left(1 - \frac{n}{N-s}\right) \\ &= (1 - F(s-1))\left(1 - \frac{n}{N-s}\right). \qquad (3\text{-}2) \end{aligned}$$

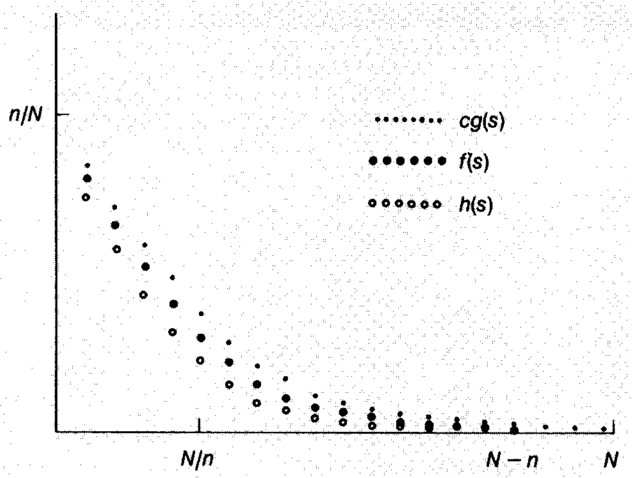

FIGURE 1. The probability function $f(s) = \text{Prob}\{S = s\}$ is graphed as a function of s. The mean and standard deviation of S are both approximately N/n. The quantities $cg(s)$ and $h(s)$ that are used in Algorithm D are also graphed for the case in which the random variable X is integer-valued.

The expression $n/(N - s)$ is the probability that the $(s + 1)$st record is selected for the sample, given that the first s records are not selected. The probability function $f(s) = \text{Prob}\{S = s\}$, for $0 \le s \le N - n$, is equal to $F(s) - F(s - 1)$. Substituting (3-1), we get the following two expressions for $f(s)$, $0 \le s \le N - n$:

$$f(s) = \frac{n}{N}\,\frac{(N-s-1)^{\underline{n-1}}}{(N-1)^{\underline{n-1}}} = \frac{n}{N}\,\frac{(N-n)^{\underline{s}}}{(N-1)^{\underline{s}}}. \qquad (3\text{-}3)$$

When $s < 0$ or $s > N - n$, we define $f(s) = 0$. An alternate derivation of (3-3) follows from the combinatorial identity

$$f(s) = \binom{N-s-1}{n-1} \Big/ \binom{N}{n}.$$

The expected value $\mathscr{E}(S)$ is equal to

$$\mathscr{E}(S) = \sum sf(s) = \frac{N-n}{n+1}, \qquad (3\text{-}4)$$

and the variance $\text{var}(S)$ is equal to

$$\text{var}(S) = \sum s^2 f(s) - \mathscr{E}(S)^2 = \frac{(N+1)(N-n)n}{(n+2)(n+1)^2}. \qquad (3\text{-}5)$$

Both the expected value and the standard deviation of S are $\approx N/n$. The probability function $f(s)$ is graphed in Figure 1.

3.1 Algorithm A

This is, by far, the simplest of the four methods. It is based on the observation that $f(s)$ is equal to the difference $F(s) - F(s - 1)$. We can generate S by setting it equal to the minimum value s such that $U \le F(s)$, where U is uniformly distributed on the unit interval.

By (3-1), we have

$$U \le 1 - \frac{(N-n)^{\underline{s+1}}}{N^{\underline{s+1}}};$$

$$\frac{(N-n)^{\underline{s+1}}}{N^{\underline{s+1}}} \le 1 - U.$$

The random variable $V = 1 - U$ is uniformly distributed as U is, so we can generate V directly, as in the following algorithm.

ALGORITHM A. This method sequentially selects n records at random from a file containing N records, where $0 \le n \le N$. The uniform random variates generated in Step A1 must be independent of one another.

- **A1.** [Generate V.] Generate a random variate V that is uniformly distributed between 0 and 1.
- **A2.** [Find minimum s.] Search sequentially for the minimum value $s \ge 0$ so that $(N-n)^{\underline{s+1}} \le N^{\underline{s+1}}V$. Set $S := s$.
- **A3.** [Select the $(S+1)$st record.] Skip over the next $S(n, N)$ records in the file and select the following one for the sample. Set $N := N - S(n, N) - 1$ and $n := n - 1$. Return to Step A1 if $n > 0$. ▮

Steps A1–A3 are iterated n times, once for each selected record in the sample. In order to generate S, the inner loop implicit in Step A2 is executed $O(S+1)$ times; each loop takes constant time. The total time spent executing Step A2 is $O(\sum_{1\le j\le n}(S+1)) = O(N)$. The total time is thus $O(N)$.

Algorithms S and A both require $O(N)$ time, but the number n of uniform variates generated by Algorithm A is much less than $(N+1)n/(n+1)$, which is the average number of variates generated by Algorithm S. Depending on the implemetation, Algorithm A can be four to eight times faster.

Algorithm A is similar to the one proposed in [3], except that in the latter method, the minimum $s > 0$ satisfying $U \le F(s)$ is found by recomputing $F(s)$ from scratch for each successive value of s. The resulting algorithm takes $O(nN)$ time. As the authors in [3] noted, that algorithm was definitely slower than their implementation of Algorithm S.

3.2 Algorithm B (Newton's Method)

In Step A2 of Algorithm A, the minimum value s satisfying $U \le F(s)$ is found by means of a sequential search. Another way to do that is to find the "approximate root" s of the equation

$$F(s) \approx U, \tag{3-6}$$

by using a variant of Newton's interpolation method. This resultant method is called Algorithm B.

Since $F(s)$ does not have a continuous derivative, we use in its place the *difference function*

$$\Delta F(s) = F(s+1) - F(s) = f(s+1).$$

Newton's method can be shown to converge for this situation. The method requires $O(\log\log S)$ iterations to find the approximate root s of (3-6) when s is large. Each iteration involves the computation of $F(s)$ and $\Delta F(s)$, for some value s. The evaluation of $F(s)$ requires $O(n)$ time, and $\Delta F(s) = f(s+1)$ can be computed from $F(s)$ in constant time using (3-1) and (3-3). Thus, the time per selected record is $O(n \log\log S)$ for large S. The total sampling time is bounded by $O(\sum_{1\le t\le n} t \log\log S)$. Using the constraint that $\sum_{1\le t\le n} S \le N - n$, it is easy to show that $\sum_{1\le t\le n} t \log\log S$ is maximized when each S is approximately N/n. Hence, the total running time for Algorithm B is $O(n^2(1 + \log\log(N/n)))$ in the worst case. It can be shown that the average running time is not better than the worst-case time by more than a constant factor.

We can obtain a higher order convergence in the search for the minimum s by replacing Newton's method with an interpolation scheme that uses higher order differences $\Delta^k F(s) = \Delta^{k-1}F(s+1) - \Delta^{k-1}F(s)$. Each difference $\Delta^k F(s)$, for $k > 1$, can be computed in constant time from $\Delta^{k-1}F(s)$ using the formula

$$\Delta^k F(s) = -\left(\frac{n-k+1}{N-s-k}\right)\Delta^{k-1}F(s). \tag{3-7}$$

Algorithm B does not seem to be of practical interest, especially when compared to Algorithms A and D, so further details are omitted.

3.3 Algorithm C (Independence Method)

Let $U_1, U_2, \ldots, U_n$ be independent and uniformly distributed random variables on the unit interval. The distribution function $F(s) = \text{Prob}\{S \le s\}$ can be expressed algebraically as

$$\begin{aligned} F(s) &= 1 - \prod_{1\le k\le n} \frac{N-s-k}{N-k+1} \\ &= 1 - \prod_{1\le k\le n} (1 - F_k(s+1)), \end{aligned} \tag{3-8}$$

where we let $F_k(x) = \text{Prob}\{(N-k+1)U_k \le x\} = x/(N-k+1)$ be the distribution function of the random variable $(N-k+1)U_k$. By independence, we have

$$\begin{aligned} &\prod_{1\le k\le n} (1 - F_k(s+1)) \\ &\quad = \prod_{1\le k\le n} \text{Prob}\{(N-k+1)U_k > s+1\} \\ &\quad = \text{Prob}\left\{\min_{1\le k\le n} \{(N-k+1)U_k\} > s+1\right\}. \end{aligned}$$

Substituting this back into (3-8), we get

$$\begin{aligned} F(s) &= 1 - \text{Prob}\left\{\min_{1\le k\le n} \{(N-k+1)U_k\} > s+1\right\} \\ &= \text{Prob}\left\{\min_{1\le k\le n} \{(N-k+1)U_k\} \le s+1\right\} \\ &= \text{Prob}\left\{\left\lfloor \min_{1\le k\le n} \{(N-k+1)U_k\}\right\rfloor \le s\right\}. \end{aligned} \tag{3-9}$$

(The notation $\lfloor x \rfloor$, which is read "floor of x," denotes the largest integer $\le x$.) This shows that S has the same distribution as the floor of the minimum of the n independent random variables $NU_1, (N-1)U_2, \ldots, (N-n+1)U_n$. The following algorithm makes use of this fact to generate S.

ALGORITHM C (*Independence Method*). This method sequentially selects n records at random from a file containing N records where $0 \le n \le N$. The uniform random variates generated in Step C1 must be independent of one another.

```
C1. [Generate $U_k$, $1 \le k \le n$.] Generate $n$ independent random variates $U_1, U_2, \ldots, U_n$, each uniformly distributed between 0 and 1.
C2. [Find minimum.] Set $S := \lfloor \min_{1 \le k \le n} \{(N - k + 1)U_k\} \rfloor$. If $S = N - n + 1$, set $S$ to an arbitrary value between 0 and $N - n$. (We can ignore this test if the random number generator used in Step C1 can only produce numbers less than 1.)
C3. [Select the $(S + 1)$st record.] Skip over the next $S(n, N)$ records in the file and select the following one for the sample. Set $N := N - S(n, N) - 1$ and $n := n - 1$. Return to Step C1 if $n > 0$. ∎
```

Steps C1–C3 are iterated n times, once for each selected record. The selection of the jth record in the sample, where $1 \le j \le n$, requires the generation of $n - j + 1$ independent uniform random variates and takes $O(n - j + 1)$ time. Hence, Algorithm C requires $n + (n - 1) + \cdots + 1 = n(n + 1)/2$ uniform variates, and it runs in $O(n^2)$ time.

4. ALGORITHM D (REJECTION METHOD)

It is interesting to note that if the term $(N - k + 1)U_k$ in (3-9) is replaced by NU_k, the resulting expression would be the distribution function for the minimum of n real numbers in the range from 0 to N. That distribution is the continuous counterpart of S, and it approximates S well. One of the key ideas in this section is that we can generate S in constant time by generating its continuous counterpart and then "correcting" it so that it has exactly the desired distribution function $F(s)$.

Algorithm D has the general form described at the beginning of Section 3. The random variable S is generated by an application of von Neumann's *rejection-acceptance method* to the discrete case. We use a random variable X that is easy to generate and that has a distribution which approximates $F(s)$ well. For simplicity, we assume that X is either a continuous or an integer-valued random variable. Let $g(x)$ denote the density function of X if X is continuous or else the probability function of X if X is integer-valued. We choose a constant $c \ge 1$ so that

$$f(\lfloor x \rfloor) \le cg(x), \qquad (4\text{-}1)$$

for all x in the domain of $g(x)$.

In order to generate S, we generate X and a random variate U that is uniformly distributed on the unit interval. If $U > f(\lfloor X \rfloor)/cg(X)$ (which occurs with low probability), we *reject* $\lfloor X \rfloor$ and start all over by generating a new X and U. When the condition $U \le f(\lfloor X \rfloor)/cg(X)$ is finally satisfied, then we *accept* $\lfloor X \rfloor$ and make the assignment $S := \lfloor X \rfloor$. A modification of the following lemma is proven in [6].

LEMMA 1
The random variate S generated by the above procedure has distribution (3-1).

The comparison $U > f(\lfloor X \rfloor)/cg(X)$ that is made in order to decide whether $\lfloor X \rfloor$ should be rejected involves the computation of $f(\lfloor X \rfloor)$, which by (3.3) requires $O(\min\{n, \lfloor X \rfloor + 1\})$ time. Since the probability of rejection is very small, we can avoid this expense most of the time by substituting for $f(s)$ a more quickly computed function $h(s)$ such that

$$h(s) \le f(s). \qquad (4\text{-}2)$$

With high probability, we will have $U \le h(\lfloor X \rfloor)/cg(X)$. When this occurs, it follows that $U \le f(\lfloor X \rfloor)/cg(X)$, so we can accept $\lfloor X \rfloor$ and set $S := \lfloor X \rfloor$. The value of $f(\lfloor X \rfloor)$ must be computed only when $U > h(\lfloor X \rfloor)/cg(X)$, which happens rarely. This technique is sometimes called a *squeeze method* since we have $h(\lfloor x \rfloor) \le f(\lfloor x \rfloor) \le cg(x)$. Typical values of the functions $f(s)$, $cg(s)$, and $h(s)$ are graphed in Figure 1 for the case in which X is an integer-valued random variable.

When n is large with respect to N, the rejection technique may be slower than the previous algorithms in this section due to the overhead involved in generating X, $h(\lfloor X \rfloor)$, and $g(X)$. For large n, Algorithm A is the fastest sampling method. The following algorithm utilizes a constant α that specifies where the tradeoff is: If $n < \alpha N$, the rejection technique is used to do the sampling; otherwise, if $n \ge \alpha N$, the sampling is done by Algorithm A. The value of α depends on the particular computer implementation. Typical values of α can be expected to be in the range 0.05–0.15. For the implementation described in Section 7, we have $\alpha \approx 0.07$.

ALGORITHM D (*Rejection Method*). This method sequentially selects n records at random from a file containing N records, where $0 \le n \le N$. At any given point in the algorithm, the variable n stores the number of records that remain to be selected for the sample, and N stores the number of (unprocessed) records left in the file. The uniform random variates generated in Step D2 must be independent of one another. The functions $g(x)$ and $h(s)$ and the constant $c \ge 1$ depend on the current values of n and N, and they must satisfy (4-1) and (4-2). The constant α is in the range $0 \le \alpha \le 1$.

```
D1. [Is $n \ge \alpha N$?] If $n \ge \alpha N$, use Algorithm A to do the sampling and then terminate the algorithm. (Otherwise, we use the rejection technique of Steps D2–D5.)
D2. [Generate $U$ and $X$.] Generate a random variate $U$ that is uniformly distributed between 0 and 1 and a random variate $X$ that has density function or probability function $g(x)$.
D3. [Accept?] If $U \le h(\lfloor X \rfloor)/cg(X)$, then set $S := \lfloor X \rfloor$ and go to Step D5.
D4. [Accept?] If $U \le f(\lfloor X \rfloor)/cg(X)$, then set $S := \lfloor X \rfloor$. Otherwise, return to Step D2.
D5. [Select the $(S + 1)$st record.] Skip over the next $S(n, N)$ records in the file and select the following one for the sample. Set $N := N - S(n, N) - 1$ and $n := n - 1$. Return to Step D2 if $n > 0$. ∎
```

Choosing the Parameters

Two good ways (namely, (4-3) and (4-5)) for choosing the parameters X, c, $g(x)$, and $h(s)$ are presented below. Discovering these two ways is the hard part of this section; once they are determined, it is easy to prove Lemmas 2 and 3, which show that (4-3) and (4-5) satisfy conditions (4-1) and (4-2).

The first way works better when n^2/N is small, and the second way is better when n^2/N is large. An easy rule for deciding which to use is as follows: If $n^2/N \le \beta$, then we use X_1, c_1, $g_1(x)$, and $h_1(s)$; else if $n^2/N > \beta$, then X_2, c_2, $g_2(s)$, and $h_2(s)$ are used. The value of the constant β is implementation-dependent. We show in Section 5 that in order to minimize the average number of uniform variates generated by Algorithm D, we should set $\beta \approx 1$.The running times of the FORTRAN implementations discussed in Section 7 are minimized by $\beta \approx 50$.

Our first choice of the parameters is

$$g_1(x) = \begin{cases} \dfrac{n}{N}\left(1-\dfrac{x}{N}\right)^{n-1}, & \text{if } 0 \le x \le N; \\ 0, & \text{otherwise;} \end{cases}$$

$$c_1 = \frac{N}{N-n+1}; \tag{4-3}$$

$$h_1(s) = \begin{cases} \dfrac{n}{N}\left(1-\dfrac{s}{N-n+1}\right)^{n-1}, & \text{if } 0 \le s \le N-n; \\ 0, & \text{otherwise.} \end{cases}$$

The random variable X_1 with density $g_1(x)$ has the *beta distribution* scaled to the interval $[0, N]$ and with parameters $a = 1$ and $b = n$. It is the continuous counterpart of S, as mentioned in the beginning of this section: The value of X_1 can be thought of as the smallest of n real numbers chosen independently and uniformly from the interval $[0, N]$.

We can generate X_1 very quickly with only one uniform or exponential random variate. Let $Z_1, Z_2, \ldots, Z_n$ denote n independent and uniformly chosen real numbers from the interval $[0, N]$. We define $G_1(x)$ to be the distribution function of X_1. By independence, we have

$$\begin{aligned} G_1(x) &= \text{Prob}\{X_1 \le x\} \\ &= 1 - \text{Prob}\{X_1 > x\} \\ &= 1 - \prod_{1 \le k \le n} \text{Prob}\{Z_k > x\} \\ &= 1 - \left(1 - \frac{x}{N}\right)^n. \end{aligned}$$

It is well known that we can generate a random variate X_1 with distribution $G_1(x)$ by setting

$$X_1 := G_1^{-1}(U) \quad \text{or} \quad X_1 := G_1^{-1}(e^{-Y}),$$

where U is uniformly distributed on the unit interval and Y is exponentially distributed. By algebraic manipulation, we get

$$G_1^{-1}(y) = N(1 - (1 - y)^{1/n}).$$

Since $1 - U$ is uniformly distributed when U is, we can generate X_1 by setting

$$X_1 := N(1 - U^{1/n}) \quad \text{or} \quad X_1 := N(1 - e^{-Y/n}). \tag{4-4}$$

The following lemma shows that the definitions in (4-3) satisfy requirements (4-1) and (4-2).

LEMMA 2.

The choices of $g_1(x)$, c_1, and $h_1(s)$ *in* (4-3) *satisfy the relation*

$$h_1(s) \le f(s) \le c_1 g_1(s + 1).$$

Note that since $g_1(x)$ is a nonincreasing function, this immediately implies (4-1).

PROOF

The proof is straightforward. First we prove the second inequality. We have

$$\begin{aligned} f(s) &= \frac{n}{N} \frac{(N-s-1)^{\underline{n-1}}}{(N-1)^{\underline{n-1}}} \\ &= \frac{n}{N-n+1} \frac{(N-s-1)^{\underline{n-1}}}{N^{\underline{n-1}}} \\ &\le \frac{n}{N-n+1} \left(\frac{N-s-1}{N}\right)^{n-1} \\ &= \frac{N}{N-n+1} \cdot \frac{n}{N}\left(1 - \frac{s+1}{N}\right)^{n-1} = c_1 g_1(s+1). \end{aligned}$$

The "≤" term in the above derivation follows because $\dfrac{N-s-1-k}{N-k} \le \dfrac{N-s-1}{N}$, for $0 \le k \le n-2$. The first inequality can be proved in the same way:

$$\begin{aligned} h_1(s) &= \frac{n}{N}\left(\frac{N-s-n+1}{N-n+1}\right)^{n-1} \\ &\le \frac{n}{N} \frac{(N-s-1)^{\underline{n-1}}}{(N-1)^{\underline{n-1}}} = f(s). \end{aligned}$$

The "≤" term follows since $\dfrac{N-s-n+1}{N-n+1} \le \dfrac{N-s-1-k}{N-1-k}$, for $0 \le k \le n-2$. ∎

The second choice for the parameters is

$$g_2(s) = \frac{n-1}{N-1}\left(1 - \frac{n-1}{N-1}\right)^s, \quad s \ge 0;$$

$$c_2 = \frac{n}{n-1} \frac{N-1}{N}; \tag{4-5}$$

$$h_2(s) = \begin{cases} \dfrac{n}{N}\left(1 - \dfrac{n-1}{N-s}\right)^s, & \text{if } 0 \le s \le N-n; \\ 0, & \text{otherwise.} \end{cases}$$

The random variable X_2 with probability function $g_2(s)$ has the *geometric distribution.* Its range of values is the set of nonnegative integers.

We can generate X_2 quickly with a single uniform or exponential random variate by setting

$$X_2 := \left\lfloor (\ln U) \Big/ \ln\left(1 - \frac{n-1}{N-1}\right) \right\rfloor$$

or (4-6)

$$X_2 := \left\lfloor -Y \Big/ \ln\left(1 - \frac{n-1}{N-1}\right) \right\rfloor,$$

where U is uniformly distributed on the unit interval and Y is exponentially distributed. This is easy to prove: Let p denote the fraction $(n-1)/(N-1)$. We have $X_2 = s$ if and only if $s \leq (\ln U)/\ln(1-p) < s+1$, which is equivalent to the condition $(1-p)^s \geq U > (1-p)^{s+1}$, and this occurs with probability $g_2(s) = p(1-p)^s$.

The following lemma shows that (4-5) satisfies requirements (4-1) and (4-2).

Lemma 3.
The choices $g_2(s)$, c_2, and $h_2(s)$ in (4-5) satisfy the relation

$$h_2(s) \leq f(s) \leq c_2 g_2(s)$$

Proof
This proof is along the lines of the proof of Lemma 2. To prove the first inequality, we note that

$$f(s) = \frac{n}{N}\frac{(N-n)^{\underline{s}}}{(N-1)^{\underline{s}}} \leq \frac{n}{N}\left(\frac{N-n}{N-1}\right)^s$$
$$= \frac{n}{n-1} \frac{N-1}{N} \cdot \frac{n-1}{N-1}\left(1 - \frac{n-1}{N-1}\right)^s = c_2 g_2(s).$$

The first inequality can be proved in the same way:

$$h_2(s) = \frac{n}{N}\left(\frac{N-s-n+1}{N-s}\right)^s \leq \frac{n}{N}\frac{(N-n)^{\underline{s}}}{(N-1)^{\underline{s}}} = f(s). \blacksquare$$

5. OPTIMIZING ALGORITHM D

In this section, four modifications of the naive implementation of Algorithm D are given that can improve the running time significantly. In particular, the last two modifications cut the number of uniform random variates generated and the number of exponentiation operations performed by half, which makes the algorithm run twice as fast. Two detailed implementations utilizing these modifications are given in the Appendix.

5.1 When to Test $n \geq \alpha N$

The values of n and N decrease each time S is generated in Step D5. If initially we have $n/N \approx \alpha$, then during the course of execution, the value of n/N will probably be sometimes $< \alpha$ and sometimes $\geq \alpha$. When that is the case, it might be advantageous to modify Algorithm D and do the "Is $n \geq \alpha N$?" test each of the n times S must be generated. If $n < \alpha N$, then we generate S by doing Steps D2–D4; otherwise, steps A1 and A2 are executed. This can be implemented by changing the "go to" in Step D5 so that it returns to Step D1 instead of to D2, and by the following substitution for Step D1:

D1. [Is $n \geq \alpha N$?] If $n \geq \alpha N$, then generate S by executing Steps A1 and A2 of Algorithm A, and go to Step D5. (Otherwise, S will be generated by Steps D2–D4.)

When $n/N \approx \alpha$, the time required to do the $n-1$ extra "Is $n \geq \alpha N$?" tests will be compensated for by the decreased time for generating S; if $n/N \not\approx \alpha$, this modification will cause a slight increase in the running time (approximately 1–2 percent). An important advantage of this modification when X_1 is used for X in the rejection technique is to guard against "worst case" behavior, which happens when the value of N decreases drastically and becomes roughly equal to n as a result of a very large value of S being generated; in such cases, the running time of the remainder of the algorithm will be quite large, on the average unless the modification is used.

The implementations of Algorithm D in the Appendix use a slightly different modification, in which a "Is $n \geq \alpha N$?" test is done at the start of each loop until the test is true, after which Algorithm A is called to finish the sampling. The resulting program is simpler than the first modification, and it still protects against worst-case behavior.

5.2 The Special Case $n = 1$

The second modification speeds up the generation of S when only one record remains to be selected. The random varible $S(1, N)$ is uniformly distributed among the integers $0 \leq s \leq N-1$; thus when $n = 1$ we can generate S directly by setting $S := \lfloor NU \rfloor$, where U is uniformly distributed on the unit interval. (The case $U = 1$ happens with zero probability so when we have $U = 1$, we can assign S arbitrarily.) This modification can be applied to all the sampling algorithms discussed in this paper.

5.3 Reducing the Number of Uniform Random Variates Generated

The third modification allows us to reduce the number of uniform random variates used in Algorithm D by half. Each generation of X as described in (4-4) and (4-6) requires the generation of an independent uniform random variate, which we denote by V. (In the case in which an exponential variate is used to generate X, we assume that the exponential variate is generated by first generating a uniform variate, which is typically the case.) Except for the first time X is generated, the variate V (and hence X) can be computed in an independent way using the values of U and X from the previous loop, as follows: During Steps D3 and possibly D4 of the previous inner loop, it was determined that either $U \leq y_1$, $y_1 < U \leq y_2$, or $y_2 < U$, where $y_1 = h(\lfloor X \rfloor)/cg(X)$ and $y_2 = f(\lfloor X \rfloor)/cg(X)$. We compute V for the next loop by setting

$$V := \begin{cases} \dfrac{U}{y_1}, & \text{if } U \leq y_1; \\ \dfrac{U - y_1}{y_2 - y_1}, & \text{if } y_1 < U \leq y_2; \\ \dfrac{U - y_2}{1 - y_2}, & \text{if } y_2 < U. \end{cases} \qquad (5\text{-}1)$$

The following lemma can be proven using the definitions of independence and of V:

LEMMA 4.
The value V computed via (5-1) *is a uniform random variate that is independent of all previous values of X and of whether or not each X was accepted.*

5.4 Reducing the Number of Exponentiation Operations

An exponentiation operation is the computation of the form $a^b = \exp(b \ln a)$, for real numbers a and b. It can be done in constant time using the library functions *EXP* and *LOG*. For simplicity, each computation of exp or ln is regarded as "half" an exponentiation operation.

First, the case in which X_1 is used for X is considered. By the last modification, only one uniform random variate must be generated during each loop, but each loop still requires *two* exponentiation operations: one to compute X_1 from V using (4-4) and the other to compute

$$\frac{h_1(\lfloor X_1 \rfloor)}{c_1 g_1(X_1)} = \frac{N-n+1}{N}\left(\frac{N-n-\lfloor X_1 \rfloor+1}{N-n+1}\,\frac{N}{N-X_1}\right)^{n-1}.$$

We can cut down the number of exponentiations to roughly one per loop in the following way: Instead of doing the test $U \le h_1(\lfloor X_1 \rfloor)/c_1 g_1(X_1)$, we use the equivalent test

$$\left(\frac{N}{N-n+1}\,U\right)^{1/(n-1)} \le \frac{N-n-\lfloor X_1 \rfloor+1}{N-n+1}\,\frac{N}{N-X_1}. \qquad (5\text{-}2)$$

If the test is true, which is almost always the case, we set V' to the quotient of the LHS divided by the RHS; the resulting V' has the same distribution as the $(n-1)$st root of a uniform random variate. Since n decreases by 1 before the start of the next loop, we can generate the next value of X_1 without doing an exponentiation by setting

$$X_1 := N(1 - V') \qquad (5\text{-}3)$$

(cf., (4-4)). Thus, in almost all cases, only one exponentiation operation is required per loop. Another important advantage of using the test (5-2) instead of the test $U \le h_1(\lfloor X_1 \rfloor)/c_1 g_1(X_1)$ is that the possibility of floating point underflow is eliminated.

When X_2 is used for X, we have a similar situation, though not quite as favorable. The computation of X_2 from V requires two ln operations (which counts as one exponentiation operation, as explained above). Another exponentiation operation in each loop is required to compute

$$\frac{h_2(X_2)}{c_2 g_2(X_2)} = \left(\frac{N-n-X_2+1}{N-X_2}\,\frac{N-1}{N-n}\right)^{X_2}.$$

The number of exponentiations can be cut down to about 1.5 per loop, as follows: Instead of doing the test $U \le h_2(X_2)/c_2 g_2(X_2)$, we use the equivalent test

$$\ln U \le X_2 \times \left(\ln\left(1 - \frac{n-1}{N-X_2}\right) - \ln\left(1 - \frac{n-1}{N-1}\right)\right). \qquad (5\text{-}4)$$

If the test is true, which is almost always the case, we set V' to the difference of the LHS minus the RHS; the resulting V' has the same distribution as the natural logarithm of a uniform random variate. We can generate the next value of X_2 by setting

$$X_2 := \left\lfloor V' \Big/ \ln\left(1 - \frac{n-1}{N-1}\right)\right\rfloor \qquad (5\text{-}5)$$

(cf., (4-6)). The common term $\ln(1 - (n-1)/(N-1))$ in (5-4) and (5-5) need only be computed once per loop, so the total number of ln operations is roughly three per loop (which counts as 1.5 exponentiation operations per loop).

The modification discussed in this section is an efficient alternative to using (5-1) for computing V, for the case $U \le y_1$. When we have $U > y_1$, which happens with very low probability, it is quicker to generate V by calling a random number generator than it is to use a technique similar to (5-1).

6. ANALYSIS OF ALGORITHM D

In this section, we prove that the average number of uniform random variates generated by Algorithm D and the average running time are both $O(n)$. We also discuss how the correct choice of X_1 or X_2 during each iteration of the algorithm can further improve performance.

6.1 Average Number $V(n, N)$ of Uniform Random Variates

The average number of uniform random variates generated during Algorithm D is denoted by $V(n, N)$. We use $V'(n, N)$ to denote the average number of uniform variates for the modified version of the algorithm in which each variate X is computed from the previous values of U and X. Theorems 1 and 2 show that $V(n, N)$ and $V'(n, N)$ are approximately equal to $2n$ and n, respectively.

THEOREM 1
The average number V(n, N) of uniform random variates used by the unmodified Algorithm D is bounded by

$$V(n, N) \le \begin{cases} \dfrac{2nN}{N-n+1}, & \text{if } n < \alpha N; \\ n, & \text{if } n \ge \alpha N. \end{cases} \qquad (6\text{-}1)$$

When $n < \alpha N$, we have $V(n, N) \approx 2n(1 + n/N)$. The basic idea of the proof is that U and X must be generated roughly $1 + n/N$ times, on the average, in order to generate each of the n values of S. Thus, approximately $2n(1 + n/N)$ variates are needed for the sampling. The difficult part of the following proof is accounting for the fact that the values of n and N change during the course of execution.

PROOF
It is assumed that X_1, $g_1(x)$, c_1, and $h_1(s)$, which are defined in (4-3), are used for X, $g(x)$, c, and $h(s)$ in Algorithm D. If $n \ge \alpha N$, then Algorithm A is used, and exactly n uniform random variates are generated.

For the $n < \alpha N$ case, (6-1) is derived by induction on n. In order to generate S, the average number of times Steps D2–D4 are executed is $1/(1 - r)$, where r is the probability of rejection. The probability of rejection is $r = \int_0^N g(t)(1 - f(t)/(cg(t)))\,dt = 1 - 1/c$. By substitution, each generation of S requires an average of $1/(1 - r) = c$ iterations, which corresponds to $2c$ uniform random variates. If $n = 1$, we have $c = 1$, so $\lfloor X \rfloor$ is accepted immediately and we have $V(1, N) = 2$. Now let us assume that (6-1) is true for samples of size $n - 1$; we will show that it remains true for n sampled records. By (6-1) and (3-3), we have

$$\begin{aligned} V(n, N) &\le \frac{2N}{N - n + 1} + \sum_{0 \le s \le N-n} f(s) V(n - 1, N - s - 1) \\ &\le \frac{2N}{N - n + 1} \\ &\quad + \sum_{0 \le s \le N-n} \frac{n}{N} \frac{(N - s - 1)^{\underline{n-1}}}{(N - 1)^{\underline{n-1}}} \frac{2(n - 1)(N - s - 1)}{N - s - n + 1} \\ &= \frac{2N}{N - n + 1} + \frac{2n(n - 1)}{N^{\underline{n}}} \\ &\quad \sum_{0 \le s \le N-n} (N - s - 1)(N - s - 1)^{\underline{n-2}} \\ &= \frac{2N}{N - n + 1} + \frac{2n(n - 1)}{N^{\underline{n}}} \\ &\quad \left(\sum_{0 \le s \le N-n} (N - s)^{\underline{n-1}} - \sum_{0 \le s \le N-n} (N - s - 1)^{\underline{n-2}} \right) \\ &= \frac{2N}{N - n + 1} + \frac{2n(n - 1)}{N^{\underline{n}}} \\ &\quad \left(\frac{(N + 1)^{\underline{n}}}{n} - (n - 1)! - \frac{N^{\underline{n-1}}}{n - 1} + (n - 2)! \right) \\ &\le \frac{2nN}{N - n}. \end{aligned}$$

This completes the proof of Theorem 1. ∎

THEOREM 2
The average number $V'(n, N)$ of uniform random variates used by Algorithm D with the second and third modifications described in the previous section is bounded by

$$V'(n, N) \le \begin{cases} \dfrac{nN}{N - n + 1}, & \text{if } n < \alpha N; \\ n, & \text{if } n \ge \alpha N. \end{cases} \qquad (6\text{-}2)$$

PROOF
We need only consider the case $n < \alpha N$. By the last modification, the variate X is generated using a uniform random variate V computed from the previous values of U and X. Thus, the first iteration requires the generation of the two uniform random variates U and V, but each successive loop requires only the generation of U. By the second modification, the last iteration does not require any random variates to be generated. Thus, the number of uniform random variates that must be generated is exactly ½ $V(n, N)$. The theorem follows from (6-1). ∎

In our derivation of (6-1) and (6-2), we assumed that X_1 was used for X throughout Algorithm D. We can do better if we sometimes use X_2 for X. We showed above that we need an average of c iterations to generate each successive S. For X_1, we have $c_1 = N/(N - n + 1) \approx 1 + n/N$; for X_2, we have $c_2 = (n/(n - 1))((N - 1)/N) \approx 1 + 1/n$. Thus, we could use X_1 when $n^2/N \le \beta$, and we could use X_2 when $n^2/N > \beta$, where $\beta \approx 1$.

The following intuitive argument indicates that this might reduce $V'(n, N)$ to

$$V'(n, N) \approx \begin{cases} n\left(1 + \dfrac{n}{N}\right), & \text{if } n^2/N \le 1, \quad n < \alpha N; \\ n\left(1 + \dfrac{1 + \ln(n^2/N)}{n}\right), & \text{if } n^2/N > 1, \quad n < \alpha N; \\ n, & \text{if } n \ge \alpha N. \end{cases} \qquad (6\text{-}3)$$

The informal justification of (6-3) is based on the observation that the ratio n/N usually does not change much during the execution of Algorithm D. At the expense of mathematical rigor, we will make the simplifying assumption that n/N remains constant during execution. The value of $n^2/N = n(n/N)$ decreases linearly to 0 as n decreases to 0. If initially we have $n^2/N \le 1$ and $n < \alpha N$, then n^2/N will always be ≤ 1 during execution, so X_1 will be used throughout Algorithm D; thus, $V'(n, N) \approx n(1 + n/N)$, as in Theorem 2. If instead we have $n^2/N > 1$ and $n < \alpha N$ initially, then X_2 will be used for X the first $n - N/n$ times S is generated, after which we will have $n^2/N \approx 1$. The random variable X_1 will be used for X the last N/n times S is generated. Hence, the total number of uniform random variates is approximately

$$\begin{aligned} &\left(1 + \frac{1}{n}\right) + \left(1 + \frac{1}{n - 1}\right) \\ &\qquad + \cdots + \left(1 + \frac{n}{N + n}\right) + \frac{N}{n}\left(1 + \frac{n}{N}\right) \\ &= n - \frac{N}{n} + H_n - H_{N/n} + \frac{N}{n} + 1 \\ &\approx n + 1 + \ln \frac{n^2}{N}. \end{aligned}$$

(The symbol H_n denotes the nth harmonic number $1 + 1/2 + \cdots + 1/n$.) This completes the argument.

6.2 Average Execution Time $T(n, N)$

$T(n, N)$ is used to represent the average total running time of Algorithm D. As shown in Table II, we can

TABLE II: Times per Step for Algorithm D.

Step	Time per Step
D1	d_1
D2	d_2
D3	d_3
D4	$d_4 \cdot \min\{n, \lfloor X \rfloor + 1\}$
D5	d_5

bound the time it takes to execute each step of Algorithm D exactly once by the quantities d_1, d_2, d_3, $d_4 \cdot \min\{n, \lfloor X \rfloor + 1\}$, and d_5, where each d_i is a positive real-valued constant.

If initially we have $n/N \geq \alpha$, then Algorithm A is used to do the sampling, and the average running time $T(n, N)$ can be bounded closely by $d_1 + d_a'N + d_a''n$, for some constants d_a' and d_a''. The following theorem shows that $T(n, N)$ is at most linear in n.

THEOREM 3.
The average running time $T(n, N)$ of Algorithm D is bounded by

$$T(n, N) \leq \begin{cases} d_1 + \dfrac{nN}{N - n + 1}(d_2 + d_3 + 3d_4) + d_5 n, & \\ & \text{if} \quad n < \alpha N; \\ d_1 + d_a'N + d_a''n, & \text{if} \quad n \geq \alpha N. \end{cases} \tag{6-4}$$

PROOF
All that is needed is the $n < \alpha N$ case of (6-4), in which the rejection technique is used. We assume that X_1, $g_1(x)$, c_1, and $h_1(s)$ are used in place of X, $g(x)$, c, and $h(s)$ throughout Algorithm D. Steps D2 and D3 are each executed c times, on the average, when S is generated. The proof of Theorem 1 shows that the total contribution to $T(n, N)$ from Steps D1, D2, D3, and D5 is bounded by

$$d_1 + \frac{nN}{N - n + 1}(d_2 + d_3) + d_5 n.$$

The tricky part in this proof is to consider the contribution to $T(n, N)$ from Step D4. The time for each execution of Step D4 is bounded by $d_4 \cdot \min\{n, \lfloor X_1 \rfloor + 1\} \leq d_4(\lfloor X_1 \rfloor + 1)$. Step D3 is executed an average of c_1 times per generation of S. The probability that $U \geq h_1(\lfloor X \rfloor)/c_1 g_1(X)$ in Step D3 (which is the probability that Step D4 is executed next) is $1 - h_1(\lfloor X \rfloor)/c_1 g_1(X)$. Hence, the time spent executing Step D4 in order to generate S is bounded by

$$c_1 \int_0^N d_4(x + 1) g_1(x) \left(1 - \frac{h_1(x)}{c g_1(x)}\right) dx$$

$$= c_1 d_4 \int_0^N (x + 1) g_1(x)\, dx - d_4 \int_0^N (x + 1) h_1(x)\, dx.$$

The first integral is

$$c_1 d_4(\mathscr{E}(X_1) + 1) = c_1 d_4 \frac{N + n + 1}{n + 1}.$$

The second integral equals

$$\frac{d_4}{c_1} \frac{N + 2}{n + 1}$$

The difference of the two integrals is bounded by

$$3d_4c_1$$

The proof of Theorem 1 shows that the total contribution of Step D4 to $T(n, N)$ is at most

$$3nd_4c_1 = 3d_4 \frac{nN}{N - n + 1}.$$

This completes the proof of Theorem 3. ∎

We proved the time bound (6-4) using X_1 for X throughout the algorithm. We can do better if we instead use X_1 for X when $n^2/N \leq \beta$ and X_2 for X when $n^2/N > \beta$. We showed in Section 6.1 that the value $\beta \approx 1$ minimizes the average number of uniform variates generated. The value of β that optimizes the average running time of Algorithm D depends on the computer implementation. For the FORTRAN implementation described in Section 7, we have $\beta \approx 50$.

The constants d_i, for $2 \leq i \leq 5$, have different values when X_2 is used for X than when X_1 is used. In order to get an intuitive idea of how much faster Algorithm D is when we use X_1 *and* X_2, let us assume that the values of the constants d_i are the same for X_2 as they are for X_1. If we bound the time for Step D4 by $d_4 n$ rather than by $d_4(\lfloor X_1 \rfloor + 1)$ as we did in the proof of Theorem 3, we can show that when $n^2/N \leq \beta$ the time required to generate S using X_1 for X is at most

$$\frac{N}{N - n + 1}\left(d_2 + d_3 + 2d_4 \frac{n^2}{N}\right) + d_5. \tag{6-5}$$

Similarly, we can prove that the time required to generate S when $n^2/N > \beta$ using X_2 for X is bounded by roughly

$$\frac{n}{n - 1} \frac{N - 1}{N}\left(d_2 + d_3 + 6d_4 \frac{N}{n^2}\right) + d_5. \tag{6-6}$$

(The proof that Step D4 takes $\leq 6d_4(N - 1)/(n(n - 1))$ time to generate each S requires intricate approximations.) The bounds (6-5) and (6-6) are equal when $n^2/N \approx \beta$, for some constant $1 \leq \beta \leq \sqrt{3}$. For simplicity, let us assume that $\beta \approx 1$ (which means that $d_4 \ll d_2 + d_3 + d_5$). By an informal argument similar to the one at the end of the last section, we can show that the running time of Algorithm D is reduced to

$$T(n, N) \approx \begin{cases} d_1 + n\left(1 + \dfrac{n}{N}\right)\left(d_2 + d_3 + d_4 \dfrac{n^2}{N}\right) + d_5 n, & \\ & \text{if} \quad n^2/N \leq \beta, \quad n < \alpha N; \\ d_1 + n\left(1 + \dfrac{1 + \ln(n^2/N)}{n}\right)(d_2 + d_3) + d_5 n & \\ \quad + d_4\left(\dfrac{N}{n} + 1 + 6\left(\ln \dfrac{n^2}{N} + \dfrac{n}{N} - \dfrac{1}{n}\right)\right), & \\ & \text{if} \quad n^2/N > \beta, \quad n < \alpha N; \\ d_1 + d_a'N + d_a''n, & \text{if} \quad n \geq \alpha N. \end{cases} \tag{6-7}$$

7. EMPIRICAL COMPARISONS

Algorithms S, A, C, and D have been implemented in FORTRAN 77 on an IBM 3081 mainframe computer in

TABLE III: Average CPU Times (IBM 3081)

Algorithm	Average Execution Time (microseconds)
S	$\approx 17N$
A	$\approx 4N$
C	$\approx 8n^2$
D	$\approx 55n$

order to get a good idea of the limit of their performance. The FORTRAN implementations are direct translations of the Pascal-like versions given in the Appendix. The average CPU times are listed in Table III.

For example, for the case $n = 10^3$, $N = 10^8$, the CPU times were 0.5 hours for Algorithm S, 6.3 minutes for Algorithm A, 8.3 seconds for Algorithm C, and 0.052 seconds for Algorithm D. The implementation of Algorithm D that uses X_1 for X is usually faster than the version that uses X_2 for X, since the last modification in Section 5 causes the number of exponentiation operations to be reduced to roughly n when X_1 is used, but to only about $1.5n$ when X_2 is used. When X_1 is used for X, the modifications discussed in Section 5 cut the CPU time for Algorithm D to roughly half of what it would be otherwise.

These timings give a good lower bound on how fast these algorithms run in practice and show the relative speeds of the algorithms. On a smaller computer, the running times can be expected to be much longer.

8. CONCLUSIONS AND FUTURE WORK

We have presented several new algorithms for sequential random sampling of n records from a file containing N records. Each algorithm does the sampling with a small constant amount of space. Their performance is summarized in Table I, and empirical timings are shown in Table III. Pascal-like implementations of several of the algorithms are given in the Appendix.

The main result of this paper is the design and analysis of Algorithm D, which runs in $O(n)$ time, on the average; it requires the generation of approximately n uniform random variates and the computation of roughly n exponentiation operations. The inner loop of Algorithm D that generates S gives an optimum average-time solution to the open problem listed in Exercise 3.4.2–8 of [6]. Algorithm D is very efficient and simple to implement, so it is ideally suited for computer implementation.

There are a couple other interesting methods that have been developed independently. The online sequential algorithms in [5] use a complicated version of the rejection–acceptance method, which does not run in $O(n)$ time. Preliminary analysis indicates that the algorithms run in $O(n + N/n)$ time; they are linear in n only when n is not too small, but not too large. For small or large n, Algorithm D should be much faster.

J. L. Bentley (personal communication, 1983) has proposed a clever two-pass method that is not online, but does run in $O(n)$ time, on the average. In the first pass, a random sample of integers is generated by truncating each element in a random sample of cn uniform real numbers in the range $[0, N + 1)$, for some constant $c > 1$; the real numbers can be generated sequentially by the algorithm in [1]. If the resulting sample of truncated real numbers contains $m \geq n$ distinct integers, then Algorithm S (or better yet, Algorithm A) is applied to the sample of size m to produce the final sample of size n; if $m < n$, then the first pass is repeated. The parameter $c > 1$ is chosen to be as small as possible, but large enough to make it very unlikely that the first pass must be repeated; the optimum value of c can be determined for any given implementation. During the first pass, the m distinct integers are stored in an array or linked list, which requires space for $O(m)$ pointers; however, this storage requirement can be avoided if the random number generator can be re-seeded for the second pass, so that the program can regenerate the integers on the fly. When re-seeding is done, assuming that the first pass does not have to be repeated, the program requires $m + cn$ random number generations and the equivalent of about $2cn$ exponentiation operations. For maximum efficiency, two different random number generators are required in the second pass: one for regenerating the real numbers and the other for Algorithm S or A. The second pass can be done with only one random number generator, if during the first pass $2cn - 1$ random variates are generated instead of cn, with only every other random variate used and the other half ignored. FORTRAN 77 implementations of Bentley's method (using Algorithm A and two random number generators for the second pass) on an IBM 3081 mainframe run in approximately $105n$ microseconds. The amount of code is comparable to the implementations of Algorithm D in the Appendix.

Empirical study indicates that round-off error is insignificant in the algorithms in this paper. The random variates S generated by Algorithm D pass the standard statistical tests. It is shown in [1] that the rule (4-4) for generating X_1 works well numerically. Since one of the ways Algorithm D generates S is by first generating X_1, it is not surprising that the generated S values are also valid statistically.

The ideas in this paper have other applications as well. Research is currently underway to see if the rejection technique used in Algorithm D can be extended to generate the kth record of random sample of size n from a pool of N records in constant time, on the average. The generation of $S(n, N)$ in Algorithm D handles the special case $k = 1$; iterating the process as in Algorithm D generates the index of the kth record in $O(k)$ time. The distribution of the index of the kth record is an example of the negative hypergeometric distribution. One possible approach to generating the index in constant time is to approximate the negative hypergeometric distribution by the beta distribution with parameters $a = k$ and $b = n - k + 1$ and normalized to the interval $[0, N]$. An alternate approximation is the negative binomial distribution. Possibly the rejection technique combined with a partitioning approach can give the desired result.

When the number N of records in the file is not known *a priori* and when reading the file more than once is not allowed or desired, none of the algorithms mentioned in this paper can be used. One way to sample when N is unknown beforehand is the Reservoir Sampling Method, due to A. G. Waterman, which is listed as Algorithm R in [6]. It requires N uniform random variates and runs in $O(N)$ time. In [9, 10], the rejection technique is applied to yield a much faster algorithm that requires an average of only $O(n + n \ln(N/n))$ uniform random variates and $O(n + n \ln(N/n))$ time.

```
while $n > 0$ do
    begin
    if $N \times RANDOM(\,) \le n$ then
        begin
        Select the next record in the file for the sample;
        $n := n - 1$
        end
    else Skip over the next record (do not include it in the sample);
    $N := N - 1$
    end;
```

ALGORITHM S: All variables have type *integer*.

```
$top := N - orig\_n$;
for $n := orig\_n$ downto 2 do
    begin
    { Step A1 }
    $V := RANDOM(\,)$;
    { Step A2 }
    $S := 0$;
    $quot := top/N$;
    while $quot > V$ do
        begin
        $S := S + 1$;
        $top := top - 1$;
        $N := N - 1$;
        $quot := quot \times top/N$
        end;
    { Step A3 }
    Skip over the next $S$ records and select the following one for the sample;
    $N := N - 1$
    end;
{ Special case $n = 1$ }
$S := TRUNC(N \times RANDOM(\,))$;
Skip over the next $S$ records and select the following one for the sample;
```

ALGORITHM A: The variables *V* and *quot* have type *real*. All other variables have type *integer*.

APPENDIX

This section gives Pascal-like implementations of Algorithms S, A, C, and D. The FORTRAN programs used in Section 7 for the CPU timings are direct translations of the programs in this section.

Two implementations of Algorithm D are given: the first uses X_1 for X, and the second uses X_2 for X. The first implementation is recommended for general use. These two programs use a non-standard Pascal construct for looping. The statements within the loop ap-

```
limit := N − orig_n + 1;
for n := orig_n downto 2 do
    begin
    { Steps C1 and C2 }
    min_X := limit;
    for mult := N downto limit do
        begin
        X := mult × RANDOM( );
        if X < min_X then min_X := X
        end;
    S := TRUNC(min_X);
    { Step C3 }
    Skip over the next S records and select the following one for the sample;
    N := N − S − 1;
    limit := limit − S
    end;
{ Special case n = 1 }
S := TRUNC(N × RANDOM( ));
Skip over the next S records and select the following one for the sample;
```

ALGORITHM C: The variables *X* and *min_X* have type *real*. All other variables have type *integer*.

pear between the reserved words **loop** and **end loop**; the execution of the statement **break loop** causes the flow of control to exit the current innermost loop.

Liberties have been taken with the syntax of identifier names, for the sake of readability. The × symbol is used for multiplication. Parentheses are used to enclose null arguments in calls to functions (like *RANDOM*) that have no parameters.

Variables of type *real* should be double precision so that round-off error will be insignificant, even when N is very large. Roughly $\log_{10}N$ digits of precision will suffice. Care should be taken to assure that intermediate calculations are done in full precision. Variables of type *integer* should be able to store numbers up to value N.

The code for the random number generator *RANDOM* is not included. For the CPU timings in Section 7, we used a machine-independent version of the linear congruential method, similar to the one given in [8]. The function *RANDOM* takes no arguments and returns a double-precision uniform random variate in the interval [0, 1). Both implementations of Algorithm D assume that the range of *RANDOM* is restricted to the open interval (0, 1). This restriction can be lifted for the first implementation of Algorithm D with a couple simple modifications, which will be described later.

Algorithm D

Two implementations are given for Algorithm D below: X_1 is used for X in the first, and X_2 is used for X in the second. The optimizations discussed in Section 5 are used. The first implementation given below is preferred and is recommended for all ranges of n and N; the second implementation will work well also, but is slightly slower for the reasons given in Section 7, especially when n is small. The range for the random number function *RANDOM* is assumed to be the open interval (0, 1).

As explained in Sections 4 and 5, there is a constant α that determines which of Algorithms D and A should be used for the sampling: If $n < \alpha N$, then the rejection technique is faster; otherwise, Algorithm A should be used. This optimization guards against "worst-case" behavior that occurs when $n \approx N$ and when X_1 is used for X, as explained in Section 5. The value of α is typically in the range 0.05–0.15. For the IBM 3081 implementation discussed in Section 7, we have $\alpha \approx 0.07$. Both implementations of Algorithm D use an *integer* constant *alpha_inverse* > 1 (which is initialized to $\approx 1/\alpha$) and an *integer* variable *threshold* (which is always equal to *alpha_inverse* × n).

Sections 4 and 6 mention that there is a constant β such that if $n^2/N \le \beta$, then it is better to use X_1, c_1, $g_1(x)$, and $h_1(s)$ in Algorithm D; otherwise, X_2, c_2, $g_2(s)$, and $h_2(s)$ should be used. The value of β for the IBM 3081 implementation discussed in Section 7 in $\beta \approx 50$. If maximum efficiency is absolutely necessary, it is recommended that the two programs be combined: X_2 should be used for X until the condition $n^2/N \le \beta$ becomes true, after which X_1 should be used for X. There should be no need to continue testing the condition once it becomes true.

```
V_prime := EXP(LOG(RANDOM( ))/n);
quant1 := N - n + 1; quant2 := quant1/N;
threshold := alpha_inverse × n;
while (n > 1) and (threshold < N) do
     begin
     loop
          { Step D2: Generate U and X }
          loop
               X := N × (1.0 - V_prime);
               S := TRUNC(X);
               if S < quant1 then break loop;
               V_prime := EXP(LOG(RANDOM( ))/n)
          end loop;
          y := RANDOM( )/quant2;       { U is the value returned by RANDOM }
          { Step D3: Accept? }
          LHS := EXP(LOG(y)/(n - 1));
          RHS := ((quant1 - S)/quant1) × (N/(N - X));
          if LHS ≤ RHS then
               begin      { Accept S, since $U \le h(\lfloor X \rfloor)/cg(X)$ }
               V_prime := LHS/RHS;
               break loop
               end;
          { Step D4: Accept? }
          if n - 1 > S then
               begin bottom := N - n; limit := N - S end
          else begin bottom := N - S - 1; limit := quant1 end;
          for top := N - 1 downto limit do
               begin y := y × top/bottom; bottom := bottom - 1 end;
          if EXP(LOG(y)/(n - 1)) ≤ N/(N - X) then
               begin      { Accept S, since $U \le f(\lfloor X \rfloor)/cg(X)$ }
               V_prime := EXP(LOG(RANDOM( ))/(n - 1));
               break loop
               end;
          V_prime := EXP(LOG(RANDOM( ))/n)
     end loop;
     { Step D5: Select the (S + 1)st record }
     Skip over the next S records and select the following one for the sample;
     N := N - S - 1; n := n - 1;
     quant1 := quant1 - S; quant2 := quant1/N;
     threshold := threshold - alpha_inverse
     end;
if n > 1 then Call Algorithm A to finish the sampling
else begin      { Special case n = 1 }
     S := TRUNC(N × V_prime);
     Skip over the next S records and select the following one for the sample
     end;
```

ALGORITHM D: Using X_1 for X.

```
V_prime := LOG(RANDOM( ));
quant1 := N - n + 1;
threshold := alpha_inverse × n;
while (n > 1) and (threshold < N) do
     begin
     quant2 := (quant1 - 1)/(N - 1); quant3 := LOG(quant2);
     loop
          { Step D2: Generate U and X }
          loop
               S := TRUNC(V_prime/quant3);        { X is equal to S }
               if S < quant1 then break loop;
               V_prime := LOG(RANDOM( ))
          end loop;
          LHS := LOG(RANDOM( ));        { U is the value returned by RANDOM }
          { Step D3: Accept? }
          RHS := S × (LOG((quant1 - S)/(N - S)) - quant3);
          if LHS $\le$ RHS then
               begin        { Accept S, since $U \le h(\lfloor X \rfloor)/cg(X)$ }
               V_prime := LHS - RHS;
               break loop
               end;
          { Step D4: Accept? }
          y := 1.0;
          if n - 1 > S then
               begin bottom := N - n; limit := N - S end
          else begin bottom := N - S - 1; limit := quant1 end;
          for top := N - 1 downto limit do
               begin y := y × top/bottom; bottom := bottom - 1 end;
          V_prime := LOG(RANDOM( ));
          if quant3 $\le$ -(LOG(y) + LHS)/S then
               break loop        { Accept S, since $U \le f(\lfloor X \rfloor)/cg(X)$ }
     end loop;
     { Step D5: Select the (S + 1)st record }
     Skip over the next S records and select the following one for the sample;
     N := N - S - 1; n := n - 1;
     quant1 := quant1 - S;
     threshold := threshold - alpha_inverse
     end;
if n > 1 then Call Algorithm A to finish the sampling
else begin        { Special case n = 1 }
     S := TRUNC(N × RANDOM( ));
     Skip over the next S records and select the following one for the sample
     end;
```

ALGORITHM D: Using X_2 for X.

Using X_1 for X

The variables *U*, *X*, *V_prime*, *LHS*, *RHS*, *y*, and *quant2* have type *real*. The other variables have type *integer*. The program above can be modified to allow *RANDOM* to return the value 0.0 by replacing all expressions of the form *EXP*(*LOG*(*a*)/*b*) by $a^{1/b}$.

The variable *V_prime* (which is used to generate X) is always set to the *n*th root of a uniform random variate, for the current value of *n*. The variables *quant1*, *quant2* , and *threshold* equal $N - n + 1$, $(N - n + 1)/N$, and *alpha_inverse* × *N*, respectively, for the current values of *N* and *n*.

Using X_2 for X

The variables *U*, *V_prime*, *LHS*, *RHS*, *y*, *quant2*, and *quant3* have type *real*. The other variables have type *integer*. Let $x > 0$ be the smallest possible number returned by *RANDOM*. The *integer* variable *S* must be large enough to store $-(\log_{10}x)N$.

The variable *V_prime* (which is used to generate X) is always set to the natural logarithm of a uniform random variate. The variables *quant1*, *quant2*, *quant3*, and *threshold* equal $N - n + 1$, $(N - n)/(N - 1)$, $\ln((N - n)/(N - 1))$, and *alpha_inverse* × *n*, for the current values of *N* and *n*.

Acknowledgments The author would like to thank Phil Heidelberger for interesting discussions on ways to reduce the number of random variates generated in Algorithm D from two per loop to one per loop. Thanks also go to the two anonymous referees for their helpful comments.

REFERENCES

1. Bentley, J.L. and Saxe, J.B. Generating sorted lists of random numbers. *ACM Trans. Math. Softw. 6*, 3 (Sept. 1980), 359–364.
2. Ernvall, J. and Nevalainen, O. An algorithm for unbiased random sampling. *Comput. J. 25*, 1 (January 1982), 45–47.
3. Fan, C.T., Muller, M.E., and Rezucha, I. Development of sampling plans by using sequential (item-by-item) selection techniques and digital computers. *Am. Stat. Assn. J.* 57 (June 1962), 387–402.
4. Jones, T.G. A note on sampling a tape file. *Commun. ACM*, 5, 6 (June 1962), 343.
5. Kawarasaki, J. and Sibuya, M. Random numbers for simple random sampling without replacement. *Keio Math. Sem. Rep* No. 7 (1982), 1–9.
6. Knuth, D.E. *The Art of Computer Programming*, Vol. 2, *Seminumerical Algorithms*. Addison-Wesley, Reading, MA (second edition, 1981).
7. Lindstrom, E.E. and Vitter, J.S. The design and analysis of BucketSort for bubble memory secondary storage. Tech. Rep. CS-83-23, Brown University, Providence, RI, (September 1983). See also *U.S. Patent Application Provisional Serial No.* 500741 (filed June 3, 1983).
8. Sedgewick, R. *Algorithms*. Addison-Wesley, Reading, MA (1983).
9. Vitter, J.S. Random sampling with a reservoir. Tech. Rep. CS-83-17, Brown University, Providence, RI, (July 1983).
10. Vitter, J.S. Optimum algorithms for two random sampling problems. In *Proceedings of the 24th IEEE Symposium on Foundations of Computer Science*, Tucson, AZ (November 1983), 65–75.

CR Categories and Subject Descriptors: G.3 **[Mathematics of Computing]**: Probability and Statistics—*probabalistic algorithms, random number generation, statistical software*; G.4 **[Mathematics of Computing]**: Mathematical Software—*algorithm analysis*

General Terms: Algorithms, Design, Performance, Theory

Additional Key Words and Phrases: random sampling, analysis of algorithms, rejection method, optimization

Received 8/82; revised 12/83; accepted 2/84

Author's Present Address: Jeffrey S. Vitter, Assistant Professor of Computer Science, Department of Computer Science, Box 1910, Brown University, Providence, RI 02912; jsv.brown @ CSNet-Relay

CORRIGENDUM: *Human Aspects of Computing*

Izak Benbasat and Yair Wand. Command abbreviation behavior in human-computer interaction. *Commun. ACM 27*, 4 (Apr. 1984), 376–383. Page 380: Table II should read:

TABLE II. Data on Abbreviation Behavior*

No. of Characters in Command	Command Name	No. of Times Used	Percent Distribution of Characters Used								Average No. of Characters Used	Weighted Average for Group
			1	2	3	4	5	6	7	8		
4	VARY	86	5	7	3	85	—	—	—	—	3.69	
4	RUSH	280	5	20	4	71	—	—	—	—	3.41	
4	SORT	5	0	0	0	100	—	—	—	—	4.00	
4	HELP	25	0	0	0	100	—	—	—	—	4.00	
4	EXIT	12	0	25	0	75	—	—	—	—	3.50	
4	STOP	3	0	0	0	100	—	—	—	—	4.00	3.52
5	POINT	442	27	4	17	1	51	—	—	—	3.46	
5	ORDER	27	7	0	7	0	85	—	—	—	4.56	
5	NAMES	28	0	0	7	0	93	—	—	—	4.86	3.60
6	SELECT	87	0	0	15	0	0	85	—	—	5.55	
6	REPORT	596	0	0	62	0	0	37	—	—	4.09	
6	CANCEL	35	0	0	14	0	0	86	—	—	5.57	4.34
7	COLUMNS	10	0	0	40	0	20	0	40	—	5.00	5.00
8	QUANTITY	404	40	1	14	17	12	0	0	17	3.45	
8	SIMULATE	520	1	0	88	0	0	0	0	11	3.51	3.48

* Excludes users who did not use abbreviations.