OPTIMUM ALGORITHMS FOR TWO RANDOM SAMPLING PROBLEMS*

(extended abstract)

Jeffrey Scott Vitter

Department of Computer Science
Brown University
Box 1910
Providence, RI 02912

Abstract

Several fast new algorithms are presented for sampling n records at random from a file containing N records. The first problem we solve deals with sampling when N is known, and the the second problem considers the case when N is unknown. The two main results in this paper are Algorithms D and Z. Algorithm D solves the first problem by doing the sampling with a small constant amount of space and in $O(n)$ time, on the average; roughly n uniform random variates are generated, and approximately n exponentiation operations are performed during the sampling The sample is selected sequentially and online; it answers an open problem in [Knuth 81]. Algorithm Z solves the second problem by doing the sampling using $O(n)$ space, roughly $n\ln(N/n)$ uniform random variates and $O(n(1+\log(N/n)))$ time, on the average. Both algorithms are time- and space-optimum and are short and easy to implement.

1. INTRODUCTION

Many computer science and statistics applications call for a sample of n records selected randomly 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, so in this paper we will 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. The author's interest in the subject stems from work on a fast new external sorting method called BucketSort that uses random sampling for preprocessing ([Lindstrom and Vitter 81, 82]).

*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 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.

In this paper we find and analyze efficient algorithms for the two random sampling problems described below. Access to the records is always done in a sequential manner. For each case we measure only the CPU time and not the I/O time. This is reasonable for records stored on random-access devices like RAM or disk, and it's even reasonable for tape storage, since tapes have a fast-forward speed that can quickly skip over unwanted records. In terms of random sampling of n integers out of N, the I/O time is insignificant, since there is no file of records being read.

Problem 1: When N is known. If the value of N is known, one way to select the n records is to repeatedly generate a random integer k between 1 and N and select the kth record if it has not already been selected. The algorithm terminates when n records have been selected. (If $n > N/2$, it is faster select the $N-n$ records *not* in the sample.) This is an example of a *non-sequential algorithm*, because the records in the sample may 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, but requires excessive space.

Often we want the n records in the sample to be in the same order that they appear in the file so that, for example, they can be accessed sequentially 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 takes nonlinear time or else the space requirement is very large and the algorithm is complicated.

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 for this problem 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 we present (Algorithms A, B, C, and D) are suited ideally to online use, since they iteratively select the

0272-5428/83/0000/0065$01.00 © 1983 IEEE

next record for the sample in an efficient way. They also have the advantage that they require a small constant amount of space and are simple to implement.

The main result for this problem is the design and analysis of Algorithm D, which does the sequential sampling with about n uniform random variates and in $O(n)$ time, on the average. This yields the optimum running time, and it solves the open problem listed in exercise 3.4.2–8 in [Knuth 81]. The method is much faster than the previously fastest-known sequential algorithm (Algorithm S), and it is much faster and simpler than the nonsequential algorithms mentioned above. The algorithms are covered in Section 2; their performance is summarized in the table below.

Alg.	ave. # unif. random vars.	ave. exec. time
S	$\frac{(N+1)n}{n+1}$	$O(N)$
A	n	$O(N)$
B	n	$O(n^2 \log\log N)$
C	$\frac{n(n+1)}{2}$	$O(n^2)$
D	$\approx n$	$O(n)$

Section 2.6 gives CPU timings for FORTRAN implementations of Algorithms S, A, and D on an IBM 3081 mainframe; the constants of proportionality in the above big-oh terms for these three algorithms (in μseconds) are approximately 30 (Algorithm S), 4 (Algorithm A), and 50 (Algorithm D).

Problem 2: When N is unknown. This problem arises, for example, if the records to be sampled from reside on a tape of indeterminate length. One way to get a random sample when N is unknown is first to determine the value of N, which reduces this problem to the previous one, and then to use Algorithm D. However, predetermining N may not always be practical or possible. For example, in the tape analogy, rewinding may not be allowed.

In this section we will look at methods for sampling when N cannot be predetermined that access the records sequentially. In this case the best we can do is use a type of "reservoir" algorithm, which cannot be online, since the entire sample must be selected before the first record is output. A reservoir algorithm starts out with the first n records as candidates for the sample. It maintains the invariant that the candidates form a true random sample of all the records processed so far. When the last record is processed, the candidates are sorted internally by index and returned as the final sample. We can show that every algorithm that processes the records in a sequential manner is a type of reservoir algorithm.

The method of choice up until now was Algorithm R in [Knuth 81], which was developed by Alan Waterman. We present three new reservoir methods (Algorithms X, Y, and Z) that use the same amount of space as Waterman's method, but are much faster. They are discussed in Section 3; their performance is summarized in the table below. The time for the final internal sort is not included. Algorithm Z, which is the main result for this problem, achieves the optimum running time.

Alg.	ave. # unif. random vars.	ave. exec. time
R	$N-n$	$O(N)$
X	$\approx n\ln(N/n)$	$O(N)$
Y	$\approx n\ln(N/n)$	$O\big(n^2(1+\log(N/n)\log\log N)\big)$
Z	$\approx n\ln(N/n)$	$O\big(n(1+\log N/n)\big)$

Typical values of the constant Γ are in the range 5–15. Much of the analysis and many of the derivations are not included in this version of the paper for purposes of brevity. For the same reason, Algorithms D and Z are the only new algorithms that are covered in detail.

2. PROBLEM 1: WHEN N IS KNOWN

This section reviews Algorithm S, which up until now was the method of choice for sequential random sampling, and then covers the four new sequential methods. The main result of this section is Algorithm D, which is faster than all other random sampling algorithms, both sequential and nonsequential.

2.1. ALGORITHM S

In this section we discuss the sequential random sampling method introduced in [Fan, Muller, and Rezucha 62] and [Jones 62]. The algorithm in sequence processes the next record in the file and determines whether it should be included in the sample. When n records have been selected, the algorithm terminates. The algorithm never runs past the last record in the file, and the n selected records form a true random sample of size n.

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

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

In the implementation below, the values of n and N decrease during the course of execution. All the algorithms in Section 2 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 [Fan, Muller, and Rezucha 62], [Jones 62], and [Knuth 81], 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.

```
while $n > 0$
    begin
    Generate an independent uniform r. v. $U$;
    if $NU < n$ then
        begin
        Select the next record for the sample;
        $n := n - 1$
        end
    else Skip over the next record
            (do not include it in the sample);
    $N := N - 1$;
    end;
```

2.2 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 we present three new methods (Algorithms A, B, and C) for sequential random sampling that outperform Algorithm S. A fourth new method (Algorithm D) is described in Sections 2.3–2.5. Much of the analyses of the sampling methods are omitted for lack of space. Each method decides which record to sample next by generating S and skipping that many records. The general form of all four algorithms is as follows:

```
while $n > 0$
    begin
    Generate an independent random variate $S(n, N)$;
    Skip over the next $S(n, N)$ records and
      select the following one for the sample;
    $N := N - S(n, N) - 1$;
    $n := n - 1$;
    end;
```

Algorithms A, B, C, and D differ from one another in how they generate $S(n, N)$. 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:

$$F(s) = 1 - \frac{(N-s-1)^{\underline{n}}}{N^{\underline{n}}} = 1 - \frac{(N-n)^{\underline{s+1}}}{N^{\underline{s+1}}}. \qquad (2\text{–}2)$$

(We use the notation $a^{\underline{b}}$ to denote the "falling power" $a(a-1)\ldots(a-b+1) = a!/(a-b)!$.) Similarly, we have two equivalent expressions for the probability function $f(s) = \text{Prob}\{S = s\}$, for $0 \le s \le N - n$:

$$f(s) = \frac{n}{N}\frac{(N-s-1)^{\underline{n-1}}}{(N-1)^{\underline{n-1}}} = \frac{n}{N}\frac{(N-n)^{\underline{s}}}{(N-1)^{\underline{s}}}. \qquad (2\text{–}3)$$

The expected value of S is $(N-n)/(n+1)$, and the standard deviation of S is approximately the same. The probability function $f(s)$ is graphed in Figure 1.

Algorithm A. We can generate S by setting it equal to the minimum value $s \ge 0$ such that $U \le F(s)$, where U is uniformly distributed on the unit interval. By (2–2), 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.

```
while $n > 0$
    begin
    Generate an independent uniform r. v. $V$;
    Search sequentially for the minimum $s \geq 0$
      such that $(N - n)^{\underline{s+1}} \leq N^{\underline{s+1}} V$;
    $S := s$;
    Skip over the next $S$ records and select
      the following one for the sample;
    $N := N - S(n, N) - 1$;
    $n := n - 1$;
    end;
```

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. Algorithm A is several times faster.

Algorithm A is similar to the one proposed in [Fan, Muller, and Rezucha 62], except that in the latter method, the minimum $s > 0$ satisfying $U \leq F(s)$ is found by recomputing $F(s)$ from scratch for each successive value of s. The resulting algorithm takes $O(nN)$ time. As the authors noted, that algorithm was definitely slower than their implementation of Algorithm S.

Algorithm B. In Algorithm A, the minimum value s satisfying $U \leq F(s)$ is found by means of a sequential search. Another way to do that is to find the "approximate root" s of the equation $F(s) \approx U$ by using a variant of Newton's interpolation method. We call the resulting method 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. 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 the higher order differences $\Delta^k F(s) = \Delta^{k-1} F(s + 1) - \Delta^{k-1} F(s)$. Each difference $\Delta^k F(s)$ can be computed in constant time from $\Delta^{k-1} F(s)$ by one multiplication and one division. Algorithm B does not seem to be of practical interest, compared to Algorithms A and D, so we will omit the details.

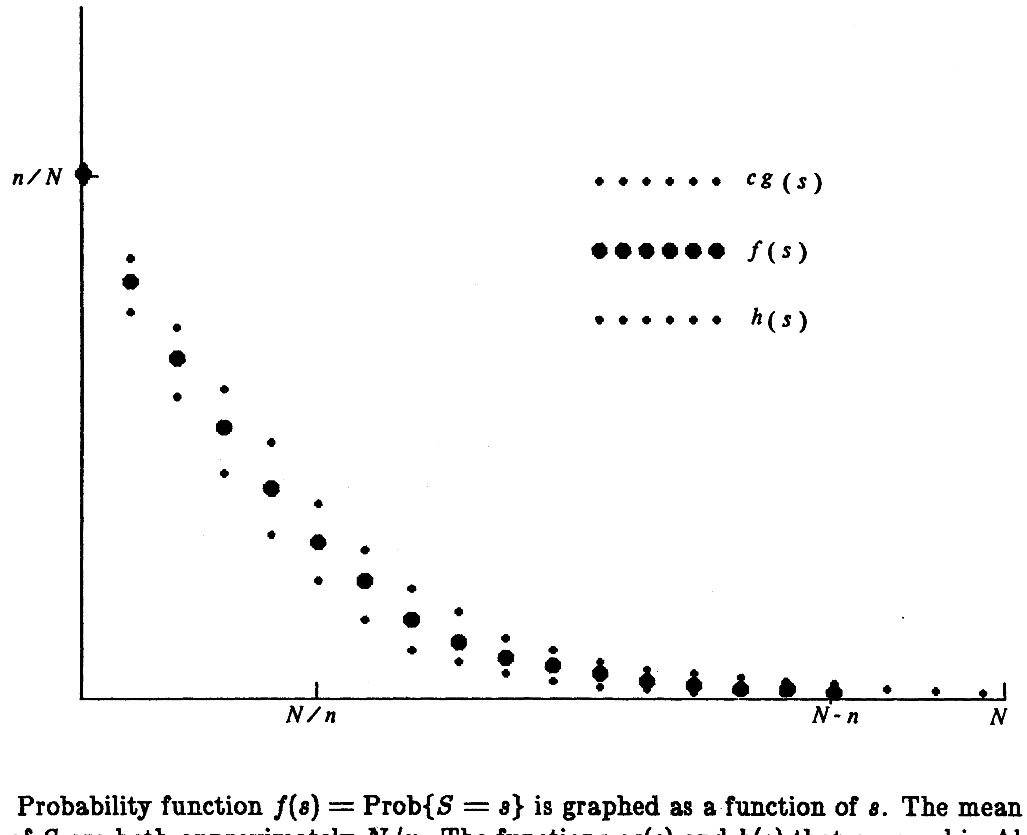

Figure 1. Probability function $f(s) = \text{Prob}\{S = s\}$ is graphed as a function of s. The mean and standard deviation of S are both approximately N/n. The functions $cg(s)$ and $h(s)$ that are used in Algorithm D are also graphed; the random variable X is assumed to be integer-valued.

Algorithm C. The distribution function $F(s) = \text{Prob}\{S \le s\}$ can be expressed algebraically as

$$F(s) = 1 - \prod_{1 \le k \le n} \frac{N - s - k}{N - k + 1}$$
$$= 1 - \prod_{1 \le k \le n} (1 - F_k(s+1)), \qquad (3\text{–}8)$$

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 $(N-k+1)U_k$. The random variables U_1, $U_2, \ldots, U_n$ are independent and uniformly distributed on the unit interval. By independence, we have

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

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

$$F(s) = 1 - \text{Prob}\left\{\min_{1 \le k \le n} \{(N-k+1)U_k\} > s+1\right\}$$
$$= \text{Prob}\left\{\left\lfloor \min_{1 \le k \le n} \{(N-k+1)U_k\} \right\rfloor \le s\right\}.$$

(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, $(N-1)U_2, \ldots, (N-n+1)U_n$. Algorithm C is based on this idea: it iteratively generates S by setting $S := \lfloor \min_{1 \le k \le n} \{(N-k+1)U_k\} \rfloor$.

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.

2.3. ALGORITHM D

It is important to note that if the term $(N-k+1)U_k$ in the above expression for $F(s)$ were instead 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 2.2. The random variable S is generated by an application of a discrete version of von Neumann's *rejection-acceptance method*. 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

$$f(\lfloor x \rfloor) \le cg(x), \qquad (2\text{–}4)$$

for all real numbers 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 $S := \lfloor X \rfloor$. The following lemma shows that S has the desired distribution.

Lemma 1. *The random variate S generated by the above procedure has distribution* (2–2).

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 requires $O(\min\{n, \lfloor X \rfloor + 1\})$ time using (2–3). 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 approximation $h(s)$ such that

$$h(s) \le f(s). \qquad (2\text{–}5)$$

With high probability, we 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. 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, $h(\lfloor X \rfloor)$, $g(X)$, and $f(\lfloor X \rfloor)$, For large n, Algorithm A is probably the fastest sampling method. The following algorithm utilizes a constant $\alpha < 1$ that specifies where the tradeoff is: if $n < \alpha N$, the rejection technique is used to do the sampling; otherwise, if $n \ge$

αN, 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 0.05–0.20.

In the code below, n is the number of records that remain to be selected, and N is the number of unprocessed records in the pool. The functions $g(x)$ and $h(s)$ and the constant $c \geq 1$ depend on the current values of n and N, and they must satisfy (2–4) and (2–5). The constant α is in the range $0 \leq \alpha < 1$.

```
while ($n > 0$) and ($n \leq \alpha N$)
    begin
    repeat
        Generate an independent r.v. $X$
          with density or probability fn. $g(x)$;
        Generate an independent uniform r.v. $U$;
        if $U \leq h(\lfloor X \rfloor)/cg(X)$ then break-loop
    until $U \leq f(\lfloor X \rfloor)/cg(X)$;
    $S := \lfloor X \rfloor$;
    Skip over the next $S$ records and select
      the following one for the sample;
    $N := N - S - 1$;
    $n := n - 1$;
    end;
if $n > 0$ then
    Call Algorithm A to finish the sampling
```

Choosing the Parameters. We will now present two good ways for choosing the parameters X, c, $g(x)$, and $h(s)$. The first way works better when n^2/N is small, and the second way is better when n^2/N is large. The rule for deciding which to use is as follows: if $n^2/N \leq \beta$, 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. In order to minimize the average number of uniform variates generated by Algorithm D, we should set $\beta \approx 1$; the details are omitted.

Our first choice of the parameters is

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

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

$$h_1(s) = \begin{cases} \dfrac{n}{N}\left(1 - \dfrac{s}{N - n + 1}\right)^{n-1}, & 0 \leq s \leq N - n; \\ 0, & \text{otherwise.} \end{cases} \tag{2–6}$$

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: the value of X_1 can be thought of as the smallest of n numbers chosen independently and uniformly from the real numbers $0 \leq x \leq N$. We can generate X_1 in constant time by setting

$$X_1 := N\left(1 - V^{1/n}\right) \quad \text{or} \quad X_1 := N\left(1 - e^{-Y/n}\right), \tag{2–7}$$

where V is uniform random variate and Y is an exponential variate.

Lemma 2. *The choices of $g_1(x)$, c_1, and $h_1(s)$ in* (2–6) *satisfy the relation*

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

Note that since $g_1(x)$ is a nonincreasing function, this immediately implies (2–4).

The second choice for the parameters is

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

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

$$h_2(s) = \begin{cases} \dfrac{n}{N}\left(1 - \dfrac{n-1}{N-s}\right)^s, & 0 \leq s \leq N - n; \\ 0, & \text{otherwise.} \end{cases} \tag{2–8}$$

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. It can be generated very quickly with a single uniform or exponential random variate.

Lemma 3. *The choices $g_2(s)$, c_2, and $h_2(s)$ in* (2–8) *satisfy relations* (2–4) *and* (2–5):

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

2.4. OPTIMIZATION TECHNIQUES

This section gives a nice example of the interplay of theory and practice by studying how Algorithm D can be optimized. We'll show that by using some statistical and programming techniques, the running time of the naive implementation of the algorithm can be cut by a factor of 2.

For simplicity, let's assume that X_1 is used for X. The random variable X_1 can be generated in constant time by first generating an independent uniform random variate V, as in (2–7). Hence the naive way of generating X and U requires two uniform random variates per loop. We can reduce that number from two per loop to just one per loop by generating V based on the previous values of U and X in a way that guarantees independence, as follows: During the previous **repeat**-loop, it was determined that either $U \le y_1$, $y_1 < U \le y_2$, or $y_2 < U$, where $y_1 = h(\lfloor X \rfloor)/cg(X)$ and $y_2 = f(\lfloor X \rfloor)/cg(X)$. We can compute V for the next loop by setting

$$V := \begin{cases} \dfrac{U}{y_1}, & \text{if } U \le y_1; \\ \dfrac{U - y_1}{y_2 - y_1}, & \text{if } y_1 < U \le y_2; \\ \dfrac{U - y_2}{1 - y_2}, & \text{if } y_2 < U. \end{cases}$$

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

Lemma 4. *The value V computed above is a uniform random variate that is independent of all previous values of X and of whether or not each X was accepted.*

At this point, only one uniform random variate must be generated during each loop, but each loop still requires *two* exponentiation operations: one to compute X from V and the other to compute $h(\lfloor X \rfloor)/cg(X)$. When X_1 is used for X, we have

$$\frac{h(\lfloor X \rfloor)}{cg(X)} = \frac{N - n + 1}{N}\left(\frac{N - n - \lfloor X \rfloor + 1}{N - n + 1}\right)^{n-1}.$$

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

$$\left(\frac{N}{N - n + 1}U\right)^{1/(n-1)} \le \frac{N - n - \lfloor X \rfloor + 1}{N - n + 1}.$$

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 without doing an exponentiation by setting

$$X := N(1 - V').$$

(Cf. (2–7).) Thus, in almost all cases, only one exponentiation operation is required per loop. Similar optimizations can be used when X_2 is used for X.

2.5. ANALYSIS OF THE REJECTION METHOD

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 improve performance further. The derivations are omitted for purposes of brevity.

Average Number $V(n, N)$ of Uniform Random Variates. We will denote the average number of uniform random variates generated during Algoritm D by the symbol $V(n, N)$. The following theorem shows that $V(n, N)$ is bounded by a linear function of n.

Theorem 1. *The average number $V(n, N)$ of uniform random variates used by Algorithm D is bounded by*

$$V(n, N) \le \begin{cases} \dfrac{nN}{N - n + 1}, & \text{if } n < \alpha N; \\ n, & \text{if } n \ge \alpha N. \end{cases} \tag{2–9}$$

When $n < \alpha N$, we have $V(n, N) \approx n(1 + n/N)$. The proof is by induction on n. The basic ideas of the proof are that U and X must be generated roughly $1 + n/N$ times, on the average, in order to generate each S, and that the ratio n/N does not change much each of the n times S is generated.

In our derivation of (2–9), we assume that X_1 is used for X throughout Algorithm D. We can do better if we sometimes use X_2 for X. The details will be included in the full version of the paper.

Average Execution Time $T(n, N)$. We will use $T(n, N)$ to represent the total average running time of Algorithm D.

Theorem 2. *The average running time $T(n, N)$ of Algorithm D is bounded by a linear function of n. The constant implicit in the big-oh notation is very small.*

Outline of Proof. If $n \ge \alpha N$, then Algorithm A is used, and the running time is $O(n)$. All we need to show is that the case $n < \alpha N$ also requires linear time. 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. The **repeat**-loop is executed c times, on the average, when S is generated; each execution of the **repeat**-loop (not counting the test $U \le f_1(\lfloor X \rfloor)/c_1 g_1(X)$) takes d_1 time units, for some very small constant d_1. The proof of Theorem 1 shows that the total contribution to $T(n, N)$ from the statements in the **repeat**-loop is bounded by

$$d_1 \frac{nN}{N - n + 1}.$$

The tricky part of the proof is to consider the contribution to $T(n,N)$ from the test $U \leq f_1(\lfloor X \rfloor)/c_1 g_1(X)$ at the end of the **repeat**-loop. The time for each execution of the test is bounded by $d_2 \cdot \min\{n, \lfloor X_1 \rfloor + 1\} \leq d_2(\lfloor X_1 \rfloor + 1)$, for some very small constant d_2. The **repeat**-loop is executed an average of c_1 times per generation of S. The probability that $U \geq h_1(\lfloor X \rfloor)/c_1 g_1(X)$ (which is the probability that the test $U \leq f_1(\lfloor X \rfloor)/c_1 g_1(X)$ will be executed next) is $\leq 1 - h_1(\lfloor X \rfloor)/c_1 g_1(X)$. Hence, the time spent executing the test $U \leq f_1(\lfloor X \rfloor)/c_1 g_1(X)$ in order to generate S is bounded by

$$c_1 \int_0^N d_2(x+1) g_1(x) \left(1 - \frac{h_1(x)}{c g_1(x)}\right) dx$$
$$= c_1 d_2 \int_0^N (x+1) g_1(x)\, dx - d_2 \int_0^N (x+1) h_1(x)\, dx.$$

The first integral is

$$c_1 d_2(\mathcal{E}(X_1) + 1) = \frac{N}{N-n+1} d_2 \frac{N+n+1}{n+1}.$$

The second integral equals

$$\frac{d_2}{c_1} \frac{N+2}{n+1} = \frac{N-n+1}{N} d_2 \frac{N+2}{n+1}.$$

The difference of the two integrals is bounded by

$$3d_2 \frac{N}{N-n+1}.$$

The proof of Theorem 1 shows that the total contribution of the test $U \leq f_1(\lfloor X \rfloor)/c_1 g_1(X)$to $T(n,N)$ is at most

$$3d_2 \frac{nN}{N-n+1}.$$

This completes the proof of Theorem 2. ∎

As before we can do better when $n^2/N > \beta$ (for some constant $\beta \approx 1$) by using X_2 for X instead of X_1. The details are omitted for lack of space.

2.6. COMPARISON OF RUNNING TIMES

Algorithms S, A, and D were implemented in FORTRAN on a large mainframe IBM 3081 computer, in order to get an idea of the limit of their performance. Their average CPU times are given by the following approximations:

Algorithm	average execution time (seconds)
S	$\approx 3 \times 10^{-5} N$
A	$\approx 4 \times 10^{-6} N$
D	$\approx 5 \times 10^{-5} n$

For example, for the case $n = 10^2$, $N = 10^8$, the CPU times were 0.75 hours (Algorithm S), 6.3 minutes (Algorithm A), and 0.0048 seconds (Algorithm D). The optimizations discussed in Section 2.4 cut the CPU time by roughly half of what it was previously.

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 should be several times longer.

3. PROBLEM 2: WHEN N IS UNKNOWN

The best solution to the problem when N cannot be predetermined is a "reservoir" algorithm, which operates as follows: Initially, the first n records are put in the reservoir as "candidates" for the final sample. From time to time, more records are put in the reservoir as candidates, taking away candidate status from randomly-chosen former candidates. The algorithm maintains the invariant that the current set of n candidates in the reservoir forms a true random sample of the records processed so far. When the end of the file is reached, the candidates are sorted by index and returned as the final sample. We can show that any algorithm that solves this problem without predetermining N and by processing the records in sequential order must be a type of reservoir algorithm.

Many more records are put in the reservoir than end up as part of the final sample. It is shown in [Knuth 81] that the resulting size of the reservoir is $n(1 + H_N - H_n) \approx n(1 + \ln(N/n))$. We can show that on the average, this size of reservoir is required for any algorithm, which implies a lower bound of $\Omega\big(n(1 + \log(N/n))\big)$ time for the sampling.

This section begins with a description of Algorithm R, due to Alan Waterman, which appears in [Knuth 81]. Three new more efficient reservoir methods (Algorithms X, Y, and Z) are then presented. Algorithm Z, which is the main result for this problem, realizes the above optimum time bound.

All the reservoir methods use an n-slot array I of pointers to keep track of which records in the reservoir are currently candidates. The pointer values $I[k]$, for $1 \leq k \leq n$ are the addresses of the actual candidates. The array I is small enough to fit in internal memory, so the final sorting phase is very fast.

3.1. ALGORITHM R

After initializing the reservoir to the first n records, Algorithm R repeatedly decides whether the next record in the file should be moved into the reservoir and made a candidate. When the $(t+1)$st record is being considered, the record has a $n/(t+1)$ chance of being in a random sample of size n out of a pool of the first $t+1$ records, so Algorithm R makes it a candidate with probability $n/(t+1)$. The candidate it replaces is chosen randomly. The complete algorithm is below. The builtin boolean function *eof* returns **true** iff the end of the file has been reached.

```
{ Initialize the reservoir to store the first $n$ records }
Copy the first $n$ records into the reservoir;
for $j := 1$ to $n$ do Set $I[j]$ to point to the $j$th record;
$t := n$;
while not eof do   { Process the rest of the records }
    begin
    $t := t + 1$;
    Generate an independent random integer $M$,
      $1 \leq M \leq t$;
    if $M \leq n$ then
        begin
        Copy the next record into the reservoir;
        Set $I[M]$ to point to that record
        end
    else Skip over the next record
            (do not include it in the reservoir);
    end;
{ Final internal sorting phase }
Sort the array $I$ so that $I[1] < \cdots < I[n]$;
for $j := 1$ to $n$ do Output the record at address $I[j]$;
```

3.2. TWO NEW RESERVOIR METHODS

Using the same idea as in Section 2.2, we will define $S(n,t)$ to be the random variable that counts the number of records to *skip over* before selecting the next record for the reservoir. The parameter n is the number of records in the sample, t is the number of records processed so far, and N is the (unknown) number of records in the file.

We present in this section two new methods for reservoir sampling, called Algorithms X and Y. These two methods along with Algorithm Z, which is covered in the next three sections, follow the same basic approach. After the reservoir is initialized, as in Algorithm R, these three algorithms repeatedly determine which record to put in the reservoir next by generating S and skipping over that many records. When the end of file is reached, the selecting terminates. The general format is below. The initialization of the reservoir and the final internal sorting phase are not included, since they are the same as before.

```
while not eof do   { Process the rest of the records }
    begin
    $t := t + 1$;
    Generate an independent r. v. $S(n,t)$;
    Skip over the next $S(n,t)$ records;
    if not eof then
        begin
        Copy the next record into the reservoir;
        Generate an indep. random integer $M$,
          $1 \leq M \leq n$;
        Set $I[M]$ to point to that record
        end
    end;
```

Algorithms X, Y, and Z differ in how they generate S. The range of $S(n,t)$ is the set of non-negative integers. The distribution function $F(s) = \text{Prob}\{S \leq s\}$, for $s \geq 0$, can be expressed in two ways:

$$F(s) = 1 - \frac{t^{\underline{n}}}{(t+s+1)^{\underline{n}}} = 1 - \frac{(t+1-n)^{\overline{s+1}}}{(t+1)^{\overline{s+1}}}. \tag{3-1}$$

(The notation $a^{\overline{b}}$ denotes the "rising power" $a(a+1)\ldots(a+b-1) = (a+b-1)!/(a-1)!$.) The two corresponding expressions for the probability function $f(s) = \text{Prob}\{S = s\}$, for $s \geq 0$ are

$$f(s) = \frac{n}{t+s+1}\frac{t^{\underline{n}}}{(t+s)^{\underline{n}}} = \frac{n}{t-n} - \frac{(t-n)^{\overline{s+1}}}{(t+1)^{\overline{s+1}}}. \tag{3-2}$$

The expected value of S is $(t-n+1)/(n-1)$, and the standard deviation of S is slightly more.

For reasons of brevity, Algorithms X and Y are not included in this extended abstract. They are based on the ideas of Algorithms A and B. Algorithm X sets S to the minimum $s \geq 0$ such that $U \leq F(s)$. Algorithm Y uses Newton's method to find the minimum s.

3.3. ALGORITHM Z

This method follows the basic form described at the beginning of Section 3.2. It uses the rejection technique developed in Section 2.3. There is some value of $\Gamma > 1$ such that when $t \le \Gamma n$, it is faster to generate S à la Algorithm X. Typical values of Γ are in the range 5–15. The subprogram below is the portion of Algorithm Z that generates S.

```
if $t \le \Gamma n$ then Use Algorithm X to generate S
else begin
    repeat
        Generate an independent r.v. X
          with density or probability fn. $g(x)$;
        Generate an independent uniform r.v. U;
        if $U \le h(\lfloor X \rfloor)/cg(X)$ then break-loop
    until $U \le f(\lfloor X \rfloor)/cg(X)$;
    $S := \lfloor X \rfloor$
    end;
```

Our choice of parameters is

$$\begin{aligned} g(x) &= \frac{n}{t+x}\left(\frac{t}{t+x}\right)^n, \qquad x \ge 0; \\ c &= \frac{t+1}{t-n+1}; \qquad\qquad (3\text{–}3) \\ h(s) &= \frac{n}{t+1}\left(\frac{t-n+1}{t+s-n+1}\right)^{n+1} \qquad x \ge 0; \end{aligned}$$

Lemma 5. *The choices of $g_1(x)$, c_1, and $h_1(s)$ in* (3–3) *satisfy the relation*

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

This satisfies condition (2–5). Since $g(x)$ is a monotone decreasing function, this also implies (2–4). The distribution for X is not a common distribution, as are the distributions for X_1 and X_2 in Problem 1, but it can also be generated quickly in constant time.

3.4. OPTIMIZATION TECHNIQUES

As in Section 2.4, we can reduce the number of uniform random variates required to generate S to roughly one per loop. In addition similar techniques can be applied to Algorithms X, Y, and Z so that the random variable V need not be generated in order to decide which record to replace in the reservoir. Similarly, the number of exponentiation operations can be reduced from two per loop to one per loop using techniques similar to those in Section 2.4. The details are omitted for lack of space.

3.5. ANALYSIS OF ALGORITHM Z

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

Theorem 3. *The average number $V(n,N)$ of uniform random variates is bounded by*

$$V(n,N) \le \begin{cases} n(H_N - H_n) + \dfrac{n(n+1)}{\Gamma n - n - 1}, & \text{if } \Gamma n < N; \\ n(H_N - H_n), & \text{if } \Gamma n \ge N. \end{cases} \qquad (3\text{–}4)$$

Typical values of Γ are in the range 5–15. The notation H_k denotes the kth harmonic number, defined by $H_k = \sum_{1 \le j \le k} 1/j$.

Average Execution Time $T(n,N)$. We let $T(n,N)$ denote the total average running time of Algorithm Z.

Theorem 4. *The average running time $T(n,N)$ of Algorithm Z is bounded by $O\big(n(1+\log(N/n))\big)$.*

The main task in the proof is showing that the evaluation of $f(\lfloor X \rfloor)$, which requires $d \cdot \min\{n, \lfloor X \rfloor + 1\}$ time, does not contribute much to $T(n,N)$. The fact that t changes during the course of the algorithm adds extra difficulty.

4. CONCLUSIONS AND RELATED WORK

This paper discusses the problem of how to select n records for a sample out of a pool of N records, or equivalently, how to select n integers out of the first N whole numbers. We have presented several algorithms that solve two problems of random sampling: one in which N is known, and the other in which N cannot be predetermined efficiently. All the algorithms for the first problem are ideally suited to online use, since the records are selected iteratively in the same order as they appear in the file. The performances of all the algorithms discussed in this paper are summarized in the table at the end of Section 1. Timings of three algorithms for the first problem are given in Section 2.6. The detailed proofs and algorithms are given in [Vitter 83a, 83b].

The main results of this paper are the design and analysis of Algorithms D and Z, which use a rejection technique to do the sampling in optimum time. Algorithms D and Z are also short and easy to implement, since the complexity is in the choice of $g(x)$, c, and $h(x)$, not in the program. Algorithm D, which solves the case when N is known, runs in $O(n)$ time, on the average. The constant factor associated with the big-oh term is very small, due to the statistical and programming techniques used in Section 2.4 to further optimize the algorithm; it requires the generation of approximately n uniform random variates and roughly n exponentiation operations, on the average. The mechanism for generating $S(n, N)$ gives an optimum solution to the open problem listed in exercise 3.4.2–8 of [Knuth 81]. The analysis is outlined in Section 2.5. Algorithm Z, which handles the case when N cannot be predetermined, uses approximately $n \ln(N/n)$ uniform random variates and it runs in $O(n(1 + \log(N/n)))$ time, on the average.

There are a couple other interesting methods that have been developed independently. The online sequential algorithms in [Kawarasaki and Sibuya 82] use a rather complicated version of the rejection technique that does not run in $O(n)$ time; preliminary analysis indicates that the algorithms run in $O(N/n^2)$ time. They are fast only when n is not too small, but not too large; Algorithm A may well be faster in practical situations.

Bentley [Bentley 82] has proposed the following two-pass method that is not online, but does run in $O(n)$ time. In the first pass, a random sample of size greater than n is generated by truncating 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 algorithms in [Bentley and Saxe 80]. If the resulting sample has size $m \geq n$, then Algorithm S (or A) is applied to the sample of size m to produce the final sample of n records; 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 easily for any given implementation. During the first pass, the indices of the records chosen for the sample are stored in an array or linked list, which requires $O(cn)$ space; however, this storage requirement can be avoided if the random number generator can be reseeded for the second pass, so that the program can regenerate the indices 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 cn exponentiation operations.

Preliminary study indicates that roundoff error is insignificant in the algorithms in this paper. The random variates S generated by Algorithm D pass the standard statistical tests.

The ideas in this paper have other applications as well. Research is currently underway to see if the rejection technique can be extended to generate the kth record of a random sample of size n in constant time, on the average. One possible approach is to approximate the distribution of the index of the kth record 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.

ACKNOWLEDGMENTS

The author would like to thank Phil Heidelberger for discussions on ways to reduce the number of random variates generated per loop in Algorithm D from two to one.

REFERENCES

1. J. L. Bentley. Private Comm. (December 1982).
2. J. L. Bentley and J. B. Saxe. Generating Sorted Lists of Random Numbers. *ACM Trans. on Math. Software*, 6, 3 (Sept. 1980), 359–364.
3. C. T. Fan, M. E. Muller, and I. Rezucha. Development of Sampling Plans by Using Sequential (Item by Item) Selection Techniques and Digital Computers. *American Statistical Assn. Journal*, 57 (June 1962), 387–402.
4. T. G. Jones. A Note on Sampling a Tape File. *Communications of the ACM*, 5, 6 (June 1962), 343.
5. J. Kawarasaki and M. Sibuya. Random Numbers for Simple Random Sampling Without Replacement. *Keio Math. Sem. Rep*, No. 7 (1982), 1–9.
6. D. E. Knuth. *The Art of Computer Programming.* Volume 2: *Seminumerical Algorithms.* Addison-Wesley, Reading, MA (second edition 1981).
7. E. E. Lindstrom and J. S. Vitter. The Design and Analysis of BucketSort: A Fast External Sorting Method for Use with Associative Secondary Storage. Technical Report ZZ20-6455, IBM Palo Alto Scientific Center (December 1981).
8. E. E. Lindstrom and J. S. Vitter. Analysis of BucketSort for Bubble Memory Secondary Storage. Technical Report ZZ20-6458, IBM Palo Alto Scientific Center (October 1982). Patent pending.
9. J. S. Vitter. Faster Methods for Random Sampling. Technical Report CS-82-21, Brown University (August 1982).
10. J. S. Vitter. Random Sampling with a Reservoir. Technical Report CS-83-17, Brown University, (July 1983).