Random Sampling with a Reservoir

JEFFREY SCOTT VITTER
Brown University

We introduce fast algorithms for selecting a random sample of n records without replacement from a pool of N records, where the value of N is unknown beforehand. The main result of the paper is the design and analysis of Algorithm Z; it does the sampling in one pass using constant space and in $O(n(1 + \log(N/n)))$ expected time, which is optimum, up to a constant factor. Several optimizations are studied that collectively improve the speed of the naive version of the algorithm by an order of magnitude. We give an efficient Pascal-like implementation that incorporates these modifications and that is suitable for general use. Theoretical and empirical results indicate that Algorithm Z outperforms current methods by a significant margin.

CR Categories and Subject Descriptors: G.3 [**Mathematics of Computing**]: Probability and Statistics—*probabilistic 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: Analysis of algorithms, optimization, random sampling, rejection method, reservoir

1. INTRODUCTION

Random sampling is basic to many computer applications in computer science, statistics, and engineering. The problem is to select without replacement a random sample of size n from a set of size N. For example, we might want a random sample of n records out of a pool of N records, or perhaps we might need a random sample of n integers from the set $\{1, 2, \ldots, N\}$.

Many algorithms have been developed for this problem when the value of N is known beforehand [1–3, 6–8, 10]. In this paper we study a very different problem: sampling when N is *unknown* and cannot be determined efficiently. The problem arises, for example, in sampling records that reside on a magnetic tape of indeterminate length. One solution is to determine the value of N in one pass and then to use one of the methods given in [10] during a second pass. However, predetermining N may not always be practical or possible. In the tape analogy, the extra pass though the tape could be very expensive.

Support for this research was provided in part by *NSF* research grants *MCS*-81-05324 and DCR-84-03613, by an IBM research contract, by an IBM Faculty Development Award, and by ONR and DARPA under Contract *N*00014-83-*K*-0146 and ARPA Order No. 4786. An extended abstract of this research appeared in [11].

Author's address: Department of Computer Science, Brown University, Providence, RI 02912.

Table I. Performance of Algorithms R, X, Y, and Z

Algorithm	Average number of uniform random variates	Average CPU time
R	$N - n$	$O(N)$
X	$\approx 2n \ln \frac{N}{n}$	$O(N)$
Y	$\approx 2n \ln \frac{N}{n}$	$O\left(n^2\left(1 + \log \frac{N}{n} \log \log \frac{N}{n}\right)\right)$
Z	$\approx 3n \ln \frac{N}{n}$	$O\left(n\left(1 + \log \frac{N}{n}\right)\right)$

For that reason, *we will restrict ourselves to processing the file of records in one sequential pass and without knowledge of N.* The powerful techniques developed in [10] can be applied to yield several fast new algorithms for this problem. The main result of this paper is the design and analysis of Algorithm Z, which does the sampling in *optimum* time and using a constant amount of space. Algorithm Z is significantly faster than the sampling methods in use today.

The measure of performance we will use in this paper to compare algorithms is central processing unit (CPU) time. Input/output (I/O) time will be ignored for the following reason: Any algorithm for this problem can be implemented using the framework that we introduce in Section 3. This reduces the I/O time dramatically by taking advantage of the random access of disks and the fast-forwarding capabilities of modern tape drives; the resulting I/O time is the same regardless of the algorithm. The remaining bottleneck is often the CPU time. The algorithms we introduce in this paper succeed in reducing the CPU time significantly, so that it is no longer a bottleneck.

It turns out that all sampling methods that process the file in one pass can be characterized as *reservoir algorithms.* In the next section, we define what we mean by a reservoir algorithm and discuss Algorithm R, which was previously the method of choice for this problem. In Section 3, we present a new framework for describing reservoir algorithms and derive a lower bound on the CPU time required. Algorithms X and Y are introduced and analyzed in Section 4. The main result, Algorithm Z, is presented in Section 5. We give several optimizations in Section 6 that reduce the CPU time of the naïve version of Algorithm Z by a factor of roughly 8. The theoretical analysis of the algorithm appears in Section 7. An efficient implementation of Algorithm Z is given in Section 8 that incorporates the optimizations discussed in Section 6. The performance of these algorithms is summarized in Table I.

The empirical timings given in Section 9 support the theoretical results and show that Algorithm Z outperforms the other methods by substantial margins. The CPU times (in microseconds) of optimized FORTRAN 77 implementations of Algorithms R, X, and Z on a VAX 11/780 computer are roughly $160N$, $40N$, and $950n \ln (N/n) - 1250n$, respectively. Our results are summarized in Section 10.

2. RESERVOIR ALGORITHMS AND ALGORITHM R

All the algorithms we study in this paper are examples of *reservoir algorithms.* We shall see in the next section that every algorithm for this sampling problem must be a type of reservoir algorithm. The basic idea behind reservoir algorithms is to select a sample of size $\geq n$, from which a random sample of size n can be generated. A reservoir algorithm is defined as follows:

Definition 1. The first step of any reservoir algorithm is to put the first n records of the file into a "reservoir." The rest of the records are processed sequentially; records can be selected for the reservoir only as they are processed. An algorithm is a reservoir algorithm if it maintains the invariant that after each record is processed a true random sample of size n can be extracted from the current state of the reservoir.

At the end of the sequential pass through the file, the final random sample must be extracted from the reservoir. The reservoir might be rather large, and so this process could be expensive. The most efficient reservoir algorithms (including the ones we discuss in this paper) avoid this step by always maintaining a set of n designated *candidates* in the reservoir, which form a true random sample of the records processed so far. When a record is chosen for the reservoir, it becomes a candidate and replaces one of the former candidates; at the end of the sequential pass through the file, the current set of n candidates is output as the final sample.

Algorithm R (which is is a reservoir algorithm due to Alan Waterman) works as follows: When the $(t + 1)$st record in the file is being processed, for $t \geq n$, the n candidates form a random sample of the first t records. The $(t + 1)$st record has a $n/(t + 1)$ chance of being in a random sample of size n of the first $t + 1$ records, and so it is made a candidate with probability $n/(t + 1)$. The candidate it replaces is chosen randomly from the n candidates. It is easy to see that the resulting set of n candidates forms a random sample of the first $t + 1$ records.

The complete algorithm is given below. The current set of n candidates is stored in the array C, in locations $C[0], C[1], \ldots, C[n - 1]$. Built-in Boolean function *eof* returns **true** if the end of file has been reached. The random number generator *RANDOM* returns a *real* number in the unit interval. The procedure call *READ_NEXT_RECORD*(R) reads the next record from the file and stores it in the record R. The procedure call *SKIP_RECORDS*(k) reads past (i.e., skips over) the next k records in the file.

```
{Make the first n records candidates for the sample}
for j := 0 to n − 1 do READ_NEXT_RECORD(C[j]);
t := n;                                   {t is the number of records processed so far}
while not eof do                                  {Process the rest of the records}
  begin
  t := t + 1;
  M := TRUNC (t × RANDOM( ));          {M is random in the range 0 ≤ M ≤ t − 1}
  if M < n then          {Make the next record a candidate, replacing one at random}
    READ_NEXT_RECORD(C[M])
  else                                                {Skip over the next record}
    SKIP_RECORDS(1)
  end;
```

When the end of file has been reached, the n candidates stored in the array C form a true random sample of the N records in the file.

If the internal memory is not large enough to store the n candidates, the algorithm can be modified as follows: The reservoir is stored sequentially on secondary storage; pointers to the current candidates are stored in internal memory in an array, which we call I. (We assume that there is enough space to store n pointers.) Suppose during the algorithm that record R' is chosen as a candidate to replace record R, which is pointed to by $I[k]$. Record R is written sequentially onto secondary storage, and $I[k]$ is set to point to R. The above code can be modified by replacing the initial **for** loop with the following code:

```
for j := 0 to n − 1 do
  begin
  Copy the jth record onto secondary storage;
  Set I[j] to point to the jth record;
  end;
```

Program statement "*READ_NEXT_RECORD*($C[\mathscr{M}]$)" should be replaced by

```
begin
Copy the next record onto secondary storage;
Set I[M] to point to that record
end
```

Retrieval of the final sample from secondary storage can be sped up by retrieving the records in sequential order. This can be done by sorting the pointers $I[1]$, $I[2], \ldots, I[n]$. The sort should be very fast because it can be done in internal memory.

The average number of records in the reservoir at the end of the algorithm is

$$n + \sum_{n \le t < N} \frac{n}{t+1} = n(1 + H_N - H_n) \approx n\left(1 + \ln \frac{N}{n}\right). \tag{2.1}$$

The notation H_k denotes the "kth harmonic number," defined by $\sum_{1 \le i \le k} 1/i$. An easy modification of the derivation shows that the average number of records chosen for the reservoir after t records have been processed so far is

$$n(H_N - H_t) \approx n \ln \frac{N}{t}. \tag{2.2}$$

It is clear that Algorithm R runs in time $O(N)$ because the entire file must be scanned and since each record can be processed in constant time. This algorithm can be reformulated easily by using the framework that we develop in the next section, so that the I/O time is reduced from $O(N)$ to $O(n(1 + \log(N/n)))$.

3. OUR FRAMEWORK FOR RESERVOIR ALGORITHMS

The limiting restrictions on algorithms for this sampling problem are that the records must be read sequentially and at most once. This means that any algorithm for this problem must maintain a reservoir that contains a random sample of size n of the records processed so far. This gives us the following generalization.

THEOREM 1. *Every algorithm for this sampling problem is a type of reservoir algorithm.*

Let us denote the number of records processed so far by t. If the file contains $t + 1$ records, each of the $t + 1$ records has probability $n/(t + 1)$ of being in a

true random sample of size n. This means that the $(t + 1)$st record should be chosen for the reservoir with probability $\geq n/(t + 1)$. Thus the average size of the reservoir must be at least as big as that in Algorithm R, which is $n(1 + H_N - H_n) \approx n(1 + \ln(N/n))$. This gives a lower bound on the time required to do the sampling.

The framework we use in this paper to develop reservoir algorithms faster than Algorithm R revolves around the following random variate:

Definition 2. The random variable $\mathcal{S}(n, t)$ is defined to be the number of records in the file that are *skipped over* before the next record is chosen for the reservoir, where n is the size of the sample and where t is the number of records processed so far. For notational simplicity, we shall often write $\mathcal{S}$ for $\mathcal{S}(n, t)$, in which case the parameters n and t will be implicit.

The basic idea for our reservoir algorithms is to repeatedly generate $\mathcal{S}$, skip that number of records, and select the next record for the reservoir. As in Algorithm R, we maintain a set of n candidates in the reservoir. Initially, the first n records are chosen as candidates, and we set $t := n$. Our reservoir algorithms have the following basic outline:

```
while not eof do                                   {Process the rest of the records}
  begin
  Generate an independent random variate S(n, t);
  SKIP_RECORDS(S);                                 {Skip over the next S records}
  if not eof then
    begin                {Make the next record a candidate, replacing one at random}
    M := TRUNC (n × RANDOM( ));        {M is uniform in the range 0 ≤ M ≤ n − 1}
    READ_NEXT_RECORD(C[M])
    end
  t := t + S + 1;
  end;
```

Our sampling algorithms differ from one another in the way that $\mathcal{S}$ is generated. Algorithm R can be cast in this framework: it generates $\mathcal{S}$ in $O(\mathcal{S})$ time, using $O(\mathcal{S})$ calls to *RANDOM*. The three new algorithms we introduce in this paper (Algorithms X, Y, and Z) generate $\mathcal{S}$ faster than does Algorithm R. Algorithm Z, which is covered in Sections 5–9, generates $\mathcal{S}$ in constant time, on the average, and thus by (2.1) it runs in average time $O(n(1 + \log(N/n)))$. By the lower bound we derived above, this is optimum, up to a constant factor.

The range of $\mathcal{S}(n, t)$ is the set of nonnegative integers. The distribution function $F(s) = \text{Prob}\{\mathcal{S} \leq s\}$, 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 $a^{\underline{b}}$ denotes the "falling power" $a(a - 1) \cdots (a - b + 1) = a!/(a - b)!$, and the corresponding notation $a^{\overline{b}}$ denotes the "rising power" $a(a + 1) \cdots (a + b - 1) = (a + b - 1)!/(a - 1)!$.) The two corresponding expressions for the probability function $f(s) = \text{Prob}\{\mathcal{S} = s\}$, 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 $\mathscr{S}$ is equal to

$$\text{expected}(\mathscr{S}) = nt^{\underline{n}} \sum_{0 \le s} \frac{s}{(t+s+1)^{\underline{n+1}}} = \frac{t-n+1}{n-1}. \tag{3.3}$$

This can be derived using *summation by parts:*

$$\sum_{a \le s < b} u(s)\ \Delta v(s) = u(s)v(s)\,|_a^b - \sum_{a \le s < b} v(s+1)\ \Delta u(s),$$

where $\Delta v(s) = v(s+1) - v(s)$. We use $u = s$, $\Delta v = 1/(t+s+1)^{\underline{n+1}}$. The standard deviation of $\mathscr{S}$ is approximately equal to expected ($\mathscr{S}$), but slightly greater.

A random variate $\mathscr{S}$ is generated for each candidate chosen. If the last candidate chosen is not the Nth record in the file, then $\mathscr{S}$ is generated one extra time in order to move past the end of file; this happens with probability $1 - n/N$. Combining this with (2.2), we find that the average number of times $\mathscr{S}$ is generated, after t records have already been processed, is $n(H_N - H_t) + 1 - n/N$.

4. ALGORITHMS X AND Y

In this section we develop two new algorithms using the framework developed in the last section. We can generate an independent random variate $\mathscr{S}$ with distribution $F(s)$ by first generating an independent uniform random variate $\mathscr{U}$ and then setting $\mathscr{S}$ to the smallest value $s \ge 0$ such that $\mathscr{U} \le F(s)$. Substituting (3.1) for $F(s)$, the stopping condition is equivalent to

$$\frac{(t+1-n)^{\overline{s+1}}}{(t+1)^{\overline{s+1}}} \le 1 - \mathscr{U}.$$

The quantity $1 - \mathscr{U}$ is independent and uniformly distributed because $\mathscr{U}$ is. Hence we can replace $1 - \mathscr{U}$ in this inequality by an independent uniform random variate, which we call $\mathscr{V}$. The new stopping condition is

$$\frac{(t+1-n)^{\overline{s+1}}}{(t+1)^{\overline{s+1}}} \le \mathscr{V}. \tag{4.1}$$

Algorithm X. Algorithm X finds the minimum value $s \ge 0$ satisfying (4.1) by simple sequential search, as follows:

$\mathscr{V} :=$ *RANDOM*( ); {$\mathscr{V}$ is uniform on the unit interval}
Search sequentially for the minimum $s \ge 0$ *such that* $(t+1-n)^{s+1}/(t+1)^{s+1} \le \mathscr{V}$;
$\mathscr{S} := s$;

Let us denote the left-hand side of (4.1) by $H(s)$. The speed of the method is due to the fact that $H(s+1)$ can be computed in constant time from $H(s)$; the sequential search takes $O(\mathscr{S} + 1)$ time. When $n = 1$, (3.3) shows that expected($\mathscr{S}$) is unbounded, and so Algorithm X should not be used. If $n > 1$, the total running time of Algorithm X is $O(\sum \mathscr{S}) = O(N)$, on the average. The execution times for Algorithms R and X are both $O(N)$, but Algorithm X is several times faster, because it calls *RANDOM* only once per generation of $\mathscr{S}$, rather than $O(\mathscr{S})$ times. The Pascal-like code for Algorithm X appears in the first part of the implementation of Algorithm Z in Section 8.

Algorithm Y. An alternate way to find the minimum s satisfying (4.1) is to use binary search, or better yet, a variant of Newton's interpolation method. The latter approach is the basis for Algorithm Y.

The value of s we want to find is the "approximate root" of the equation

$$H(s) \approx \mathscr{V}.$$

Here we apply the discrete version of Newton's method. In place of the derivative of $H(s)$, we use the difference function

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

We can show that Newton's method converges to give the value of $\mathscr{S}$ in $O(1 + \log \log \mathscr{S})$ iterations. (By convention, we define $\log \log \mathscr{S} = 0$ when $\mathscr{S} \leq b$, where b is the base of the logarithm.) Each interation requires the computation of $H(s)$ and $\Delta H(s)$, which takes $O(n)$ time. The total running time is given in the following theorem:

THEOREM 2. *The average running time of Algorithm* Y *used to do the sampling is*

$$O\left(n^2\left(1 + \log \frac{N}{n} \log \log \frac{N}{n}\right)\right).$$

PROOF. By the above remarks, the running time of Algorithm Y is

$$O\left(n \sum_{1 \leq i < \mathscr{T}} (1 + \log \log \mathscr{S}_i)\right), \tag{4.2}$$

where $\mathscr{S}_i$ denotes the ith value of $\mathscr{S}$ generated, and where $\mathscr{T}$ is the number of times $\mathscr{S}$ is generated before the algorithm terminates. In probability language, $\mathscr{T}$ is called a *bounded stopping time*, since it must be true that $\mathscr{T} \leq N - n$. Let us denote the quantity $1 + \log \log \mathscr{S}_i$ by $\mathscr{L}_i$. The variates $\mathscr{L}_1, \mathscr{L}_2, \mathscr{L}_3, \ldots$ are not independent of one another, because the distribution function $F(s)$ of $\mathscr{S}$ depends upon the current value of t. However, we can "bound" each random variate $\mathscr{L}_i$ by a random variate $\mathscr{L}_i' = 1 + \log \log \mathscr{S}(n, N - 1)$, such that $\mathscr{L}_1'$, $\mathscr{L}_2', \mathscr{L}_3', \ldots$ are independent of one another. By "bound", we mean that the distribution function for $\mathscr{L}_i'$ is everywhere greater than the distribution function for $\mathscr{L}_i$. If we imagine that $\mathscr{L}_i$ and $\mathscr{L}_i'$ are both generated by first generating $\mathscr{S}_i$ and $\mathscr{S}(n, N - 1)$ via Algorithm X or Y using the same value of $\mathscr{V}$, then this condition guarantees that $\mathscr{L}_i \leq \mathscr{L}_i'$. Hence we can bound (4.2) by

$$O\left(n \sum_{1 \leq i \leq \mathscr{T}} (1 + \mathscr{L}_i')\right). \tag{4.3}$$

The random variates $\mathscr{L}_1', \mathscr{L}_2', \mathscr{L}_3', \ldots$ are independent and identically distributed. We can apply the classical form of Wald's Lemma (see [5]) and bound the average value of (4.3) by

$$O(\text{expected}(\mathscr{T}) \cdot (1 + \text{expected}(\log \log \mathscr{S}(n, N - 1)))). \tag{4.4}$$

We showed at the end of the last section that expected($\mathscr{T}$) = $n(H_N - H_n) + 1 - n/N = O(n(1 + \log(N/n)))$. By Jensen's inequality (see [4]), when $n > 1$, we have

$$\begin{aligned}\text{expected}(\log\log \mathscr{S}(n, N-1)) &\le \log\log(\text{expected}(\mathscr{S}(n, N-1))) \\ &= \log\log\frac{N-n}{n-1} \\ &= O\left(\log\log\frac{N}{n}\right).\end{aligned}$$

For the case $n = 1$, we can show that expected(log log $\mathscr{S}(1, N - 1)) = O(1)$. Putting this together, we can bound (4.4) by $O(n^2(1 + \log(N/n)\log\log(N/n)))$. □

It is possible to obtain a higher-order convergence than with Newton's method by using higher-order differences, defined by $\Delta^k H(s) = \Delta^{k-1}H(s + 1) - \Delta^{k-1}H(s)$. For $k > 1$, each difference $\Delta^k H(s)$ can be computed in constant time from a lower order difference using the formula

$$\Delta^k H(s) = -\left(\frac{n + k - 1}{t + s + k + 1)}\right)\Delta^{k-1}H(s).$$

It may be possible to choose k so large that $\mathscr{S}$ can be generated in a constant number of iterations. The total running time would then be bounded by $O((k + n)n(1 + \log(N/n)))$. The value of k would be at least $\log^*(N/n)$, where $\log^* x$ is defined to be the number of times the log function must be applied, initially to the argument x, until the result is ≤ 1. However, the overhead involved would probably be too large to make this practical. A much faster algorithm is given in the next section.

5. ALGORITHM Z

Algorithm Z follows our general format described in Section 3. The skip random variate $\mathscr{S}(n, t)$ is generated in constant time, on the average, by a modification of von Neumann's *rejection–acceptance* method. The main idea is that we can generate $\mathscr{S}$ quickly by generating a fast approximation and "correcting" it so that its resulting distribution is the desired distribution $F(s)$.

We assume that we have a continuous random variable $\mathscr{X}$ that can be generated quickly and whose distribution approximates $F(s)$ well. We denote the probability density function of $\mathscr{X}$ by $g(x)$. We choose a constant $c \ge 1$ such that

$$f(\lfloor x \rfloor) \le cg(x), \tag{5.1}$$

for all $x \ge 0$.

We generate $\mathscr{S}$ by generating an independent $\mathscr{X}$ and an independent uniform random variate $\mathscr{U}$ on the unit interval. If $\mathscr{U} > f(\lfloor\mathscr{X}\rfloor)/cg(\mathscr{X})$ (which occurs with low probability), we *reject* $\lfloor\mathscr{X}\rfloor$ and start over by generating a new $\mathscr{X}$ and $\mathscr{U}$. When the condition $\mathscr{U} \le f(\lfloor\mathscr{X}\rfloor)/cg(\mathscr{X})$ is finally true, then we *accept* $\lfloor\mathscr{X}\rfloor$ and set $\mathscr{S} := \lfloor\mathscr{X}\rfloor$. The following lemma, which is easy to prove and is left to the reader, shows that this procedure is valid.

LEMMA 1. *The random variate $\mathscr{S}$ generated in the above manner has the distribution given by* (3.1).

The bottleneck in this process is computing $f(\lfloor \mathscr{X} \rfloor)/cg(\mathscr{X})$, which takes $O(\min\{n, \lfloor \mathscr{X} \rfloor + 1\})$ time using (3.2). We can avoid this computation most of the time by approximating $f(s)$ by a more quickly computed function $h(s)$, such that for all $s \geq 0$, we have

$$h(s) \leq f(s). \tag{5.2}$$

If $\mathscr{U} \leq h(\lfloor \mathscr{X} \rfloor)/cg(\mathscr{X})$, then by transitivity we have $\mathscr{U} \leq f(\lfloor \mathscr{X} \rfloor)/cg(\mathscr{X})$, and so we can accept $\lfloor \mathscr{X} \rfloor$. The value of $f(\lfloor \mathscr{X} \rfloor)$ must be computed only when $\mathscr{U} > h(\lfloor \mathscr{X} \rfloor)/cg(\mathscr{X})$, which happens with small probability. This technique is sometimes called the *squeeze method*, since we have $h(\lfloor x \rfloor) \leq f(\lfloor x \rfloor) \leq cg(x)$, for all x.

Owing to the overhead in computing these functions, the rejection technique generates $\mathscr{S}$ more slowly than does the method used in Algorithm X when $t \leq Tn$, for some constant $T > 1$. Typical values of T can be expected to be in the range 10–40. For example, in the implementation described in Sections 8 and 9, we have $T \approx 22$. The outline for how Algorithm Z generates $\mathscr{S}(n, t)$ appears below:

```
if t ≤ T × n then Use the inner loop of Algorithm X to generate S
else begin
  repeat
    Generate an independent random variable X with density function g(x);
    U := RANDOM( );                          {U is uniform on the unit interval}
    if U ≤ h(⌊X⌋)/cg(X) then break loop
  until U ≤ f(⌊X⌋)/cg(X);
  S := ⌊X⌋
  end;
```

The tricky part in the development of Algorithm Z was finding appropriate choices for the parameters $g(x)$ (the density function of $\mathscr{X}$), c, and $h(s)$ that gave fast running times. The following choices seem to work best:

$$g(x) = \frac{n}{t+x}\left(\frac{t}{t+x}\right)^n, \qquad x \geq 0;$$

$$c = \frac{t+1}{t-n+1}; \tag{5.3}$$

$$h(s) = \frac{n}{t+1}\left(\frac{t-n+1}{t+s-n+1}\right)^{n+1}, \qquad x \geq 0.$$

The random variable $\mathscr{X}$ is not a commonly encountered distribution, but it can be generated very quickly, in constant time. Let us denote the distribution function of $\mathscr{X}$ by $G(x)$. We have

$$G(x) = \text{Prob}\{\mathscr{X} \leq x\} = \int_0^x g(x)\,dx = 1 - \left(\frac{t}{t+x}\right)^n. \tag{5.4}$$

It is easy to show that we can generate a random variable $\mathscr{X}$ with distribution $G(x)$ by setting $\mathscr{X} = G^{-1}(\mathscr{Y})$ or $\mathscr{X} = G^{-1}(e^{-\mathscr{Y}})$, where $\mathscr{Y}$ is uniformly distributed on [0, 1] and $\mathscr{Y}$ is exponentially distributed. Substituting (5.4), we get

$$G^{-1}(y) = t((1-y)^{-1/n} - t).$$

Since $1 - \mathscr{V}$ is uniformly distributed when $\mathscr{V}$ is, we can generate $\mathscr{X}$ by setting

$$\mathscr{X} := t(\mathscr{V}^{-1/n} - 1) \qquad \text{or} \qquad \mathscr{X} := t(e^{-\mathscr{Y}/n} - 1). \tag{5.5}$$

The following lemma shows that the choice of parameters in (5.3) satisfies requirements (5.1) and (5.2).

LEMMA 2. *The choices of* $g(x)$, c, *and* $h(s)$ *in* (5.3) *satisfy the relation*

$$h(s) \le f(s) \le cg(s+1).$$

This satisfies condition (5.2). Since $g(x)$ is a monotone decreasing function, this also implies (5.1).

PROOF. First let us prove the first inequality. We have

$$h(s) = \frac{n}{t+1}\left(\frac{t-n+1}{t+s-n+1}\right)^{n+1} = \frac{n(t-n+1)}{(t+1)(t+s-n+1)}\left(\frac{t-n+1}{t+s-n+1}\right)^{n}.$$

The first quotient can be bounded by $n/(t+s+1)$, since it is easy to show that $(t+s-n+1)(t+1) \ge (t+s+1)(t-n+1)$. Similarly, we can bound the second quotient by $t^n/(t+s)^n$, because we have $(t-n+1)/(t+s-n+1) \le (t-k)/(t+s-k)$, for $0 \le k \le n-1$. By (3.2), this proves the first inequality. The second inequality in Lemma 2 can be proved in the same way:

$$f(s) = \frac{n}{t+s+1}\frac{t^n}{(t+s)^n} = \frac{n}{t+s-n+1}\frac{t^n}{(t+s+1)^n}.$$

The first quotient can be bounded by $n(t+1)/((t+s+1)(t-n+1))$, since $(t+s-n+1)(t+1) \ge (t+s+1)(t-n+1)$. The second quotient is bounded by $(t/(t+s+1))^n$, since $(t-k)/(t+s+1-k) \le t/(t+s+1)$, for $0 \le k \le n-1$. By (5.3), this proves the second inequality. □

6. OPTIMIZING ALGORITHM Z

In this section we give three optimizations that dramatically improve the running time of the naïve version of Algorithm Z. These modifications are incorporated into the implementation of Algorithm Z that appears in Section 8.

Threshold Optimization. We have already seen this important optimization technique in the last section, where it was included as part of the basic algorithm. We used a constant parameter T to determine how $\mathscr{S}$ should be generated: If $t \le Tn$, then the inner loop of Algorithm X is used to generate $\mathscr{S}$; otherwise, the rejection–acceptance technique is used. The value of T is typically set in the range 10–40. The choice $T \approx 22$ worked best in the implementation described in Sections 8 and 9; the running time was cut by a factor of roughly 4. Some theoretical analysis is given in Section 7.3. However, there is an even more basic reason why this optimization is important (which is why we made it part of the basic algorithm): It guards against floating-point overflow! Numerical considerations are discussed further in Section 8.

RANDOM Optimization. The rest of this section is devoted to more sophisticated techniques to eliminate costly calls to mathematical subroutines. The first optimization we shall consider allows us to cut the number of calls to *RANDOM* by more than one-third. Each generation of $\mathscr{X}$ via (5.5) requires the generation

of an independent uniform random variate (or else the generation of an exponential random variate, which usually involves the prior generation of a uniform variate). Let us call this random variate $\mathscr{V}$. Except for the first time $\mathscr{X}$ is generated, we can can avoid a call to *RANDOM* and compute $\mathscr{V}$ (and thus $\mathscr{X}$) in an independent way by making use of the values of $\mathscr{U}$ and $\mathscr{X}$ from the previous loop. The previous loop ended for one of three reasons: $\mathscr{U} \le q_1$, $q_1 < \mathscr{U} \le q_2$, or $q_2 < \mathscr{U}$, where $q_1 = h(\lfloor \mathscr{X} \rfloor)/cg(\mathscr{X})$ and $q_2 = f(\lfloor \mathscr{X} \rfloor)/cg(\mathscr{X})$. We can compute $\mathscr{V}$ for the next loop, as follows:

$$\mathscr{V} := \begin{cases} \dfrac{\mathscr{U}}{q_1}, & \text{if } \mathscr{U} \le q_1; \\ \dfrac{\mathscr{U} - q_1}{q_2 - q_1}, & \text{if } q_1 < \mathscr{U} \le q_2; \\ \dfrac{\mathscr{U} - q_2}{1 - q_2}, & \text{if } q_2 < \mathscr{U}. \end{cases} \tag{6.1}$$

We leave it as an exercise for the reader to prove the following lemma from the definitions of independence and of $\mathscr{V}$:

LEMMA 3. *The value $\mathscr{V}$ computed via* (6.1) *is a uniform random variate that is independent of all previous values of $\mathscr{X}$ and of whether or not each $\mathscr{X}$ was accepted.*

Similar tricks can be employed in order to compute the value of $\mathscr{M}$ without having to call *RANDOM*, but it does not seem to be worth the added effort. The big savings comes about in the next optimization.

Subroutine Optimization. We can speed up Algorithm Z by a factor of almost 2 by cutting in half the number of operations of the form x^y, where x and y are either real (floating-point) numbers or large integers. The calculation of $x^y = \exp(y \ln x)$ involves an implicit call to the two mathematical library subroutines *EXP* (exponential) and *LOG* (logarithm). The calculation takes constant time, but the constant is much larger than the time for a multiplication or division.

Each loop in the naive implementation of Algorithm Z requires two operations of the form x^y: one to compute $\mathscr{X}$ from $\mathscr{V}$ using (5.5) and the other to compute

$$\frac{h(\lfloor \mathscr{X} \rfloor)}{cg(\mathscr{X})} = \frac{(t-n+1)^2(t+\mathscr{X})}{(t+1)^2(t+\lfloor \mathscr{X} \rfloor - n+1)} \left(\frac{(t-n+1)(t+\mathscr{X})}{(t+\lfloor \mathscr{X} \rfloor - n+1)t} \right)^n. \tag{6.2}$$

(The computation of $f(\lfloor \mathscr{X} \rfloor)/cg(\mathscr{X})$ also requires an operation of the form x^y, but as we shall see in the next section, the computation is seldom done.) Instead of doing the test "Is $\mathscr{U} \le h(\lfloor \mathscr{X} \rfloor)/cg(\mathscr{X})$?", we use the equivalent test of finding whether

$$\left(\frac{\mathscr{U}(t+1)^2(t+\lfloor \mathscr{X} \rfloor - n+1)}{(t-n+1)^2(t+\mathscr{X})} \right)^{1/n} \le \frac{(t-n+1)(t+\mathscr{X})}{(t+\lfloor \mathscr{X} \rfloor - n+1)t}. \tag{6.3}$$

If this test is true, which happens with high probability, we set $\mathscr{W}$ to the quotient of the right-hand side divided by the left-hand side. The resulting random variate $\mathscr{W}$ has the same distribution as $\mathscr{V}^{-1/n}$. Thus we can compute $\mathscr{X}$ without an operation of the form x^y by setting

$$\mathscr{X} := t(\mathscr{W} - 1). \tag{6.4}$$

(The reader should compare this with (5.5).) Hence we usually need only one operation of the form x^y per loop rather than two.

An important side effect of using the test (6.3) instead of the naive "Is $\mathscr{V} \leq h(\lfloor \mathscr{X} \rfloor)/cg(\mathscr{X})$?" test is that it eliminates floating-point overflow, which can occur in (6.2). As a practical matter, the full *RANDOM* optimization should not be used when subroutine optimization is used. The $\mathscr{V} \leq q_1$ case of the *RANDOM* optimization is subsumed by subroutine optimization; the remaining two cases of (6.1) happen rarely, and would be expensive to implement when the subroutine optimization is in effect.

7. THEORETICAL ANALYSIS OF ALGORITHM Z

In this section we show analytically that the average number of random variates generated and the expected running time for Algorithm Z are optimum, up to a constant factor. We conclude the section with a theoretical basis for why the parameter T should be set to a constant in the threshold optimization discussed in the last section.

7.1 Average Number of Calls to *Random*

We shall prove that the expected number of calls to the random number generator made by Algorithm Z is bounded by approximately $3n(\ln(N/n))$ and $2n(\ln(N/n))$, depending on whether the *RANDOM* optimization is used. This shows that the *RANDOM* optimization reduces the number of calls to *RANDOM* by a factor of one-third.

We shall use the term "naïve version of Algorithm Z" to refer to the *pure* rejection technique, in which not even the threshold optimization is used. We define $RAND(n, t, N)$ to be the expected number of calls to *RANDOM* made by the naïve version of Algorithm Z in order to finish the sampling when $t \leq N$ records have already been processed. Similarly, we define $OPT(n, t, N)$ to be the average number of times *RANDOM* is called when the *RANDOM* optimization is used, starting from the point at which $t \leq N$ records have been processed.

THEOREM 3. *The expected number of calls to RANDOM during the naive version of Algorithm Z, after* $n + 2 \leq t \leq N$ *records have already been processed, is*

$$RAND(n, t, N) \leq n\left(\frac{2(n+1)}{t-n-1} + 3(H_N - H_t)\right). \tag{7.1}$$

PROOF. The basic idea of the proof is that $\mathscr{S}$ can be generated using $\approx 3 + 2n/t$ calls to *RANDOM*, on the average. At the end of Section 3, we showed that the average number of times $\mathscr{S}$ must be generated is $n(H_N - H_t) + 1 - n/N$. Intuitively, the total number of calls to *RANDOM* is thus $\approx 3n(H_N - H_t)$. The major hurdle in the proof is to take into account the fact that t is increasing during the course of the algorithm.

The naive version of Algorithm Z calls *RANDOM* once each time $\mathscr{U}$, $\mathscr{X}$, or $\mathscr{M}$ must be generated. The average number of times $\mathscr{M}$ is generated is $n(H_N - H_t)$, as shown at the end of Section 3. Let us define $UV(n, t, N)$ to be the average

number of times $\mathscr{U}$ or $\mathscr{V}$ is generated, where $\mathscr{V}$ is the implicit uniform random variate used to generate $\mathscr{X}$. To prove the theorem it suffices to show that

$$UV(n, t, N) \le 2n\left(\frac{n+1}{t-n-1} + H_N - H_t\right) + 2. \tag{7.2}$$

We use induction on t. For $t = N$, we have $UV(n, N, N) = 0$, and (7.2) is trivially true. Now let us suppose that (7.2) is true for $t + 1, t + 2, \ldots, N$; we will show that it remains true for t.

Both $\mathscr{U}$ and $\mathscr{V}$ are generated once each time the body of the **repeat** loop is executed. The average number of iterations of the **repeat** loop required to generate $\mathscr{S}(n, t)$ is $1/(1 - r)$, where r is the probability of rejection. We have

$$r = \int_0^\infty g(x)\left(1 - \frac{f(\lfloor x \rfloor)}{cg(x)}\right) dx = 1 - \frac{1}{c}.$$

Combining the two formulas, we find that each generation of $\mathscr{S}(n, t)$ requires an average of $1/(1 - r) = c$ iterations of the **repeat** loop, thus accounting for $2c$ calls to *RANDOM*. By use of the induction hypothesis, we have

$$\begin{aligned}
UV(n, t, N) &\le 2c + \sum_{0 \le s < N-t} f(s) UV(n, t + s + 1, N) \\
&\le \frac{2(t+1)}{t-n+1} - \sum_{0 \le s < N-t} \frac{nt^{\underline{n}}}{(t+s+1)^{\underline{n+1}}}\left(2n\left(\frac{n+1}{t+s-n} + H_N - H_{t+s+1}\right) + 2\right) \\
&\le \frac{2(t+1)}{t-n+1} + 2n^2(n+1)t^{\underline{n}}\left(\sum_{0 \le s < N-t} \frac{1}{(t+s+1)^{\underline{n+2}}}\right) \\
&\quad + 2nt^{\underline{n}}(nH_N + 1)\left(\sum_{0 \le s < N-t} \frac{1}{(t+s+1)^{\underline{n+1}}}\right) + 2n^2 t^{\underline{n}}\left(\sum_{0 \le s < N-t} \frac{H_{t+s+1}}{(t+s+1)^{\underline{n+1}}}\right).
\end{aligned}$$

All three summations can be computed in closed form by using summation by parts, as discussed at the end of Section 3. For the first summation, we use $u = 1$, $\Delta v = 1/(t + s + 1)^{\underline{n+2}}$; for the second, we use $u = 1$, $\Delta v = 1/(t + s + 1)^{\underline{n+1}}$; and for the third, we use $u = H_{t+s+1}$, $\Delta v = 1/(t + s + 1)^{\underline{n+1}}$. Plugging in the values for the summations and bounding the sum of the smaller terms by 0, we get

$$\begin{aligned}
UV(n, t, N) &\le \frac{2(t+1)}{t-n+1} + \frac{2n^2}{t-n} + 2nH_N + 2 - 2nH_{t+1} - \frac{2(t-n+1)}{t+1} \\
&= \frac{2n}{t-n+1} + \frac{2n^2}{t-n} + 2n(H_N - H_t) + 2 \\
&\le 2n\left(\frac{n+1}{t-n} + H_N - H_t\right) + 2.
\end{aligned}$$

This proves (7.2) and thus completes the proof of Theorem 3. □

COROLLARY 1. *The average number of calls to RANDOM made by Algorithm Z using the threshold optimization is bounded by*

$$\begin{cases} 2n(H_N - H_n) + 1 - \dfrac{n}{N}, & \text{if } Tn \geq N; \\ n\left(\dfrac{2(n+1)}{Tn - n - 1} + 3H_N - H_{Tn} - 2H_n\right) + 3 - \dfrac{1}{T}, & \text{if } Tn \leq N. \end{cases} \tag{7.3}$$

PROOF. In the $Tn \geq N$ case, the variate $\mathscr{S}$ is always generated using the method of Algorithm X. The number of times $\mathscr{S}$ is generated is $n(H_N - H_n) + 1 - n/N$. Each generation of $\mathscr{S}$ requires two calls to *RANDOM* to generate $\mathscr{U}$ and $\mathscr{M}$, except possibly the last time $\mathscr{S}$ is generated. If the Nth record is the last record chosen for the reservoir (which happens with probability n/N), then $\mathscr{M}$ must be generated; otherwise, the last time $\mathscr{S}$ is generated causes the remaining records in the file to be skipped over, and $\mathscr{M}$ does not have to be generated. The formula for the $Tn \geq N$ case follows immediately.

For the $Tn \leq N$ case, the method of Algorithm X is used until the first Tn records have been processed. In the remainder of the algorithm, the rejection technique makes at most $RAND(n, Tn, N)$ calls to *RANDOM*. We assume that T is large enough so that $Tn \geq n + 2$, which is always the case in actual implementations. Combining the $Tn \geq N$ case with Theorem 3 gives the result. □

THEOREM 4. *Let us consider the modified version of the naive Algorithm Z in which only the RANDOM optimization is used. The expected number of times RANDOM is called during the sampling, after t records have already been processed, for* $t \geq n + 2$, *is*

$$OPT(n, t, N) \leq n\left(\frac{n+1}{t - n - 1} + 2(H_N - H_t)\right) + 2. \tag{7.4}$$

PROOF. With this modification, *RANDOM* is not called to generate $\mathscr{X}$, except for the first time $\mathscr{X}$ is generated. This reduces $UV(n, t, N)$ to half its former value, plus 1. The result follows immediately, from the proof of Theorem 3. □

The following corollary follows from Theorem 4 in the same manner that Corollary 1 follows from Theorem 3:

COROLLARY 2. *The average number of calls to RANDOM made by Algorithm Z using the threshold and RANDOM optimizations is bounded by*

$$\begin{cases} 2n(H_N - H_n) + 1 - \dfrac{n}{N}, & \text{if } Tn \geq N; \\ n\left(\dfrac{n+1}{Tn - n - 1} + 2(H_N - H_n)\right) + 3 - \dfrac{1}{T}, & \text{if } Tn \leq N. \end{cases} \tag{7.5}$$

7.2 Average Running Time

We shall show that the average execution time of Algorithm Z is $O(n(1 + \log(N/n)))$, which by the remarks at the beginning of Section 3 is optimum, up to a constant factor. Let use define $TIME(n, t, N)$ to be the expected execution time

of the naive version of Algorithm Z to finish the sampling when t records have been processed so far.

THEOREM 5. *The expected running time of the naive version of Algorithm Z after $t \le N$ records have already been processed is*

$$TIME(n, t, N) = O\left(n\left(1 + \log \frac{N}{t}\right)\right).$$

PROOF. Let us assume that $n \ge 2$. The $n = 1$ case is discussed later. The only statement in Algorithm Z that takes more than constant time to execute once is the test "Is $\mathscr{U} \le f(\lfloor \mathscr{X} \rfloor)/cg(\mathscr{X})$?" in the **repeat** loop, because it requires the evaluation of $f(\lfloor \mathscr{X} \rfloor)$, which takes $O(\min\{n, \lfloor \mathscr{X} \rfloor + 1\})$ time, using (5.1). Every statement in Algorithm X is executed at most $c = (t + 1)/(t - n + 1)$ times, on the average, each time $\mathscr{S}$ is generated. By the proof of Theorem 3, the total running time of Algorithm Z minus the time spent executing the *repeat* loop test is bounded by

$$O\left(n\left(\frac{n+1}{t-n-1} + H_N - H_t\right)\right) = O\left(n\left(1 + \log \frac{N}{t}\right)\right).$$

The difficult part of the proof is bounding the execution time spent evaluating $f(\lfloor \mathscr{X} \rfloor)$ throughout the course of the algorithm, as t increases. The time per evaluation is bounded by $d \cdot \min\{n, \lfloor \mathscr{X} \rfloor + 1\} \le d(\lfloor \mathscr{X} \rfloor + 1)$, for some small constant $d \ge 0$. The probability that the "Is $\mathscr{U} \le h(\lfloor \mathscr{X} \rfloor)/cg(\mathscr{X})$?" test is false (which is the probability that $f(\lfloor \mathscr{X} \rfloor)$ must be evaluated next) is $1 - h(\lfloor \mathscr{X} \rfloor)/cg(\mathscr{X})$. Thus for a single evaluation of $\mathscr{S}$, we can bound the time spent evaluating $f(\lfloor \mathscr{X} \rfloor)$ by

$$c\int_0^\infty d(x+1)g(x)\left(1-\frac{h(x)}{cg(x)}\right)dx = cd\int_0^\infty (x+1)g(x)\,dx - d\int_0^\infty (x+1)h(x)\,dx.$$

The first integral is equal to

$$cd(\text{expected}(\mathscr{X}) + 1) = \frac{t+1}{t-n+1}\,d\,\frac{t+n-1}{n-1};$$

the second integral equals

$$d\,\frac{t(t-n+1)}{(n-1)(t+1)}.$$

By some algebraic simplification, the difference of the first integral minus the second can be bounded by

$$3cd + \frac{2cd}{n-1}.$$

By the derivation of (7.2), it follows that the total amount of time spent evaluating $f(\lfloor \mathscr{X} \rfloor)$ during the course of the algorithm is at most

$$\left(3 + \frac{2}{n-1}\right)d\left(n\left(\frac{n+1}{t-n-1} + H_N - H_t\right) + 1\right) = O\left(n\left(1 + \log \frac{N}{t}\right)\right).$$

This completes the proof for the $n \geq 2$ case.

We shall see in Section 8 that Algorithm Z should not be used as is when $n = 1$, primarily for numerical reasons. When $n = 1$, we instead run Algorithm Z for a larger (but still small) value of n, say, $n_0 = 5$, to get a random sample of size n_0. Once the sample of size n_0 is generated, we can recover the random sample of size n from it with a simple $O(n_0)$-time procedure given in the next section. Thus we have $TIME(1, t, N) = TIME(n_0, t, N) + O(n_0)$. This completes the proof of Theorem 5. □

The following corollary shows that Algorithm Z achieves optimum time, up to a constant factor.

COROLLARY 3. *The average running time of Algorithm Z using the threshold optimization is*

$$O\left(n\left(1 + \log \frac{N}{n}\right)\right).$$

The use of the *RANDOM* or subroutine optimizations would reduce the running time further by some constant factor.

7.3 The Threshold Value

In the threshold optimization discussed in Section 6, we generate $\mathscr{S}$ using the method of Algorithm X until t gets large enough, after which we use the rejection technique to generate $\mathscr{S}$. It was suggested that a constant be used, which we call T: The threshold point for switching from Algorithm X to the rejection technique is when $t \geq Tn$. In this section we shed some light on why the threshold point is approximately a linear function of n.

The average time required to generate $\mathscr{S}$ using the method of Algorithm X is roughly

$$d_1 \cdot \text{expected}(\mathscr{S}) = d_1 \frac{t - n + 1}{n - 1} \approx \frac{d_1 t}{n}, \tag{7.6}$$

for some small constant $d_1 > 0$. (The formula for expected($\mathscr{S}$) appears as (3.3).) We can bound the average time required to generate $\mathscr{S}$ using the rejection technique by applying the techniques of the proof of Theorem 5 as follows: The body of the **repeat** loop is generated $c \approx 1 + n/t$ times, on the average, in order to generate $\mathscr{S}$. With probability $1 - h(\lfloor \mathscr{X} \rfloor)/cg(\mathscr{X}) \approx n/t$, we must evaluate $f(\lfloor \mathscr{X} \rfloor)$, which takes at most $O(\text{expected}(\mathscr{S})) = O(t/n)$ time, on the average. Putting this all together, the average time to generate $\mathscr{S}$ using the rejection technique is roughly

$$d_2\left(1 + \frac{n}{t}\right)\left(d_3\left(1 - \frac{n}{t}\right) + d_4 \frac{n}{t} \frac{t}{n}\right) = \Theta(1), \tag{7.7}$$

for suitable small constants d_2, d_3, $d_4 > 0$. Thus, the value of t at which (7.6) and (7.7) are equal is $\Theta(n)$.

In the implementation discussed in the next two sections, the initialization $T = 22$ seemed to work best. Typical values of T can be expected to be in the range 10–40.

8. IMPLEMENTATION OF ALGORITHM Z

In this section we give an efficient Pascal-like implementation of Algorithm Z. This program incorporates the threshold, *RANDOM*, and subroutine optimizations discussed in Section 6. The method of Algorithm X is used until $t > thresh$, where *thresh* is equal to $T \times n$. The constant T should be initialized to some integer in the range 10–40; in the timings described in the next section, performance is best when T is initialized to 22.

A few nonstandard notations are used: Statements between the reserved words **loop** and **end loop** form the body of a loop; execution of the statement **break loop** causes flow of control to exit the loop. The × symbol is used for multiplication, $x \uparrow y$ stands for x^y, and parentheses enclosing a null argument are used for function calls with no parameters.

Operations of the form x^y, where y is either a real number or a large integer, are computed as $EXP(y \times LOG(x))$ using the mathematical library subroutines *EXP* and *LOG*. The variables $\mathscr{W}$, $\mathscr{X}$, $\mathscr{U}$, *lhs*, *rhs*, *y*, and *quot* have type *real*; all other variables have type *integer*.

The random number generator *RANDOM* takes no arguments and returns a uniform random variate in the range (0, 1). We will let u denote the smallest number returned by *RANDOM*, *max_int* the largest number that can be stored in a *integer* variable, and *max_real* the largest number that can be stored in a *real* variable. In order to prevent integer and floating point overflow, we must have

$$Nx^{-1/n} < max\_int \qquad \text{and} \qquad Nx^{-1/n} < max\_real.$$

This guarantees, for example, that the random variates $\mathscr{S}$ and $\mathscr{X}$ will be able to be stored in their appropriate variables and that the computation of $f(\lfloor \mathscr{X} \rfloor)$ will not cause overflow in pathological situations.

These conditions can be satisfied easily on most computers, in the following way: Some bound on N can be obtained, even if it is only a weak bound. The next step is to determine the minimum value of n that satisfies the two conditions, substituting for N the upper bound just obtained. Let us call that value n_0; typically, n_0 should be a fairly small number. If the desired sample size n is less than n_0, the solution is to apply the basic philosophy of reservoir sampling recursively. First we run Algorithm Z to find a random sample of size n_0. Then we generate a random sample of size n from the random sample of size n_0 using the simple procedure below:

```
for j := 0 to n0 − 1 do already_selected := false;
num_selected = 0;
while num_selected < n do
  begin
  M := TRUNC(n0 × RANDOM( ));
  if not already_selected[M] then
        begin
        OUTPUT(C[M]);
        already_selected[M] := true;
        num_selected := num_selected + 1
        end
  end
```

An alternative to this procedure is to use one of the sequential sampling methods given in [10].

The optimized code for Algorithm Z appears below. The first part of the program is essentially the code for Algorithm X, which does the sampling until t gets large enough.

```
{Make the first n records candidates for the sample}
for j := 0 to n − 1 do READ_NEXT_RECORD(C[j]);
t := n;                                   {t is the number of records processed so far}
{Process records using the method of Algorithm X until t is large enough}
thresh := T × n;
num := 0;                                 {num is equal to t − n}
while not eof and (t ≤ thresh) do
    begin
    V := RANDOM( );                       {Generate V}
    S := 0;
    t := t + 1; num := num + 1;
    quot := num/t;
    while quot > V do                     {Find min S satisfying (4.1)}
        begin
        S := S + 1;
        t := t + 1; num := num + 1;
        quot := (quot × num)/t
        end;
    SKIP_RECORDS(S);                      {Skip over the next S records}
    if not eof then
        begin          {Make the next record a candidate, replacing one at random}
        M := TRUNC(n × RANDOM( ));  {M is uniform in the range 0 ≤ M ≤ n − 1}
        READ_NEXT_RECORD(C[M])
        end
    end;

{Process the rest of the records using the rejection technique}
W := EXP(− LOG(RANDOM( ))/n);             {Generate W}
term := t − n + 1;                        {term is always equal to t − n + 1}
while not eof do
    begin
    loop
        {Generate U and X}
        U := RANDOM( );
        X := t × (W − 1.0);
        S := TRUNC(X);                    {S is tentatively set to ⌊X⌋}
        {Test if U ≤ h(S)/cg(X) in the manner of (6.3)}
        lhs := EXP(LOG(((U × (((t + 1)/term) ↑ 2)) × (term + S))/(t + X))/n);
        rhs := (((t + X)/(term + S)) × term)/t;
        if lhs ≤ rhs then
            begin W := rhs/lhs; break loop end;
        {Test if U ≤ f(S)/cg(X)}
        y := (((U × (t + 1))/term) × (t + S + 1))/(t + X);
        if n < S then begin denom := t; numer_lim := term + S end
        else begin denom := t − n + S; numer_lim := t + 1 end;
        for numer := t + S downto numer_lim do
            begin y := (y × numer)/denom; denom := denom − 1 end;
        W := EXP(− LOG(RANDOM( ))/n);     {Generate W in advance}
        if EXP(LOG(y)/n) ≤ (t + X)/t then break loop
    end loop;
    SKIP_RECORDS(S);                      {Skip over the next S records}
```

Table II. Timings for Algorithms R, X, and Z (in seconds)

	Algorithm R	Algorithm X	Algorithm Z
$N = 10^6$			
$n = 10^1$	171	41	0.1
$n = 10^2$	153	35	0.8
$n = 10^3$	155	37	5
$n = 10^4$	158	47	31
$n = 10^6$	176	95	96
$N = 10^7$			
$n = 10^1$	1595	430	0.1
$n = 10^2$	1590	356	1
$n = 10^3$	1553	371	8
$n = 10^4$	1560	387	55
$n = 10^5$	1604	500	303

```
if not eof then
    begin          {Make the next record a candidate, replacing one at random}
    M := TRUNC (n × RANDOM( )); {M is uniform in the range 0 ≤ M ≤ n − 1}
    READ_NEXT_RECORD(C[M])
    end;
t := t + S + 1;
term := term + S + 1
end;
```

9. EMPIRICAL COMPARISONS

This section presents the results of empirical timings for Algorithms R, X, and Z run on a VAX 11/780 computer. The programs were written in FORTRAN 77, in order to get high speed. The random number generator used was a machine-independent version of the linear congruential method, similar to the one described in [9]. The programs for Algorithms X and Z were direct translations of those given in the last section. For Algorithm R we optimized the basic code by putting it into the framework discussed in Section 3.

The running times of the algorithms (in microseconds) are as follows:

$$\approx 160N \qquad \text{(Algorithm R)},$$

$$\approx 40N \qquad \text{(Algorithm X)},$$

$$\approx 950n \ln \frac{N}{n} - 1250n \qquad \text{(Algorithm Z)}.$$

The actual timings for Algorithm R and X were much higher for very small and very large values of n. The formula for Algorithm Z was obtained using least squares curve fitting.

The timings were done for a large range of values of n and N. The values given in Table II are representative of the results. All times are given in seconds. When $N = 10^7$, Algorithm R requires roughly 26 minutes of CPU time, Algorithm X requires about 6 minutes, but Algorithm Z uses only a few seconds.

To put these CPU timings in perspective, let us consider the corresponding I/O time required for the $N = 10^7$, $n = 10^2$ example, assuming an average record

size of 15–20 bytes. The total file size would be roughly 150–200 megabytes. IBM 3380 disk drives can read/write information at the rate of 6 megabytes per second, and IBM 3420 model 8 tape drives can read/write at the rate of 1.25 megabytes per second. Roughly 800 megabytes can be stored in a 3380, whereas tapes for a 3420 can store roughly 180 megabytes.

In the disk case, the sequential read time for the file is roughly 30 seconds. However, the algorithms we have developed for this paper do not need to read all the records; several records can be skipped at once. If the record length is fixed, then I/O time is reduced to roughly $n \ln(N/n)$ short seeks and reads; the corresponding I/O time is reduced to less than 2 seconds. Even if record lengths are varying, the number of records in each track can be stored in the track header; skipping is done by reading consecutive track headers until the desired track is found. The resulting I/O time is substantially less than 30 seconds. The corresponding CPU times for this example are 1590 seconds (Algorithm R), 356 seconds (Algorithm X), and 1 second (Algorithm R).

For the tape example, the time for a sequential read is about 150 seconds. However, tape drives can make use of their fast-forward capabilities to skip over unwanted records at a higher speed, and so the I/O time can be reduced, especially if the records are of fixed length. The CPU times are the same as above.

In smaller examples, the CPU time used by Algorithm X might be roughly as small as the I/O time required; however, the internal processing and the I/O might not be able to be overlapped, and so the total running time could be double the I/O time. The advantage of Algorithm Z is that the CPU time is insignificant. The algorithm is fast enough so that it is useful even when the value of N is known.

10. SUMMARY OF THE RESULTS

In this paper we have considered algorithms for selecting a random sample of size n from a file containing N records, in which the value of N is not known to the algorithm. We have shown that any sampling algorithm for this problem that processes the file sequentially can be characterized as a *reservoir* algorithm. The distinguishing feature of reservoir algorithms is that a sample of size $\geq n$ is taken during a sequential pass through the file; the final sample is obtained by selecting n records at random from this larger sample. All reservoir algorithms require $\Omega(n(1 + \log(N/t)))$ time.

The main result of the paper is the design and analysis of Algorithm Z, which uses a rejection–acceptance technique to do the sampling in optimum time, up to a constant factor. Several optimizations are given that improve the running time of the naive version by a factor of roughly 8. Central processing unit timings done on a VAX 11/780 computer indicate that Algorithm Z is significantly faster than the reservoir-sampling algorithms in use today. An efficient Pascal-like implementation of Algorithm Z suitable for general use appears in Section 8.

REFERENCES

1. BENTLEY, J. L. Personal communication, Apr. 1983; see [11].
2. ERNVALL, J. AND NEVALAINEN, O. An algorithm for unbiased random sampling. *Comput. J. 25*, 1 (Jan. 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. Assoc. J. 57* (June 1962), 387–402.
4. FELLER, W. *An Introduction to Probability Theory and Its Applications*, vol. I, 3rd ed. Wiley, 1968.
5. FELLER, W. *An Introduction to Probability Theory and Its Applications*, vol. II, 2nd ed. Wiley, 1971.
6. JONES, T. G. A note on sampling a tape file. *Commun. ACM 5*, 6 (June 1962), 343.
7. KAWARASAKI, J., AND SIBUYA, M. Random numbers for simple random sampling without replacement. *Keio Math. Sem. Rep.* No. 7 (1982), 1–9.
8. KNUTH, D. E. *The Art of Computer Programming*. vol. 2: *Seminumerical Algorithms*, 2nd ed. Addison-Wesley, Reading, Mass., 1981.
9. SEDGEWICK, R. *Algorithms*. Addison-Wesley, Reading, Mass., 1981.
10. VITTER, J. S. Faster methods for random sampling. *Commun. ACM 27*, 7 (July 1984). 703–718.
11. VITTER, J. S. *Optimum algorithms for two random sampling problems*. In *Proceedings of the 24th Annual IEEE Symposium on Foundations of Computer Science* (Tucson, Az., Nov. 7–9), IEEE, New York, 1983 pp. 65–75.

Received February 1984 accepted October 1984.