Chapter Twelve
RANDOM SAMPLING

1. INTRODUCTION.

In this chapter we consider the problem of the selection of a random sample of size k from a set of n objects. This is also called sampling without replacement since duplicates are not allowed. There are several issues here which should be clarified in this, the introductory section.

1. Some users may wish to generate an ordered random sample. Not unexpectedly, it is easier to generate unordered random samples. Thus, algorithms that produce ordered random samples should not be compared on an equal basis with other algorithms.
2. Sometimes, n is not known, and we are asked to grab each object in turn and make an instantaneous decision whether to include it in our random sample or not. This can best be visualized by considering the objects as being given in a linked list and not an array.
3. In nearly all cases, we worry about the expected time complexity as a function of k and n. In typical situations, n is much larger than k, and we would like to have expected time complexities which are bounded by a constant times k, uniformly over n.
4. The space required by an algorithm is defined as the space required outside the original array of n records or objects and outside the array of k records to be returned. Some of the algorithms in this chapter are bounded workspace algorithms, i.e. the space requirements are $O(1)$.

The strategies for sampling can be partitioned as follows: (i) classical sampling: generate random objects and include them in the sample if they have not already been picked; (ii) sequential sampling: generate the sample by traversing the collection of objects once and making instantaneous decisions during that one pass; (iii) oversampling: by means of a simple technique, obtain a random sample (usually of incorrect size), and in a second phase, adjust the sample so that it has the right size. Each of these strategies has some very competitive algorithms, so that no strategy should a priori be excluded from contention.

We assume that the set of objects is $\{1,2, \ldots, n\}$. If the objects are different, then these integers should be considered as pointers (indices) to the objects in an array.

2. CLASSICAL SAMPLING.

2.1. The swapping method.

Assume that the objects are given in array form: $A[1], \ldots, A[n]$. Then, if we are allowed to permute the objects, random sampling is extremely simple. We can choose an object uniformly and at random, and swap it with the last object. If we need another object, we choose one uniformly from among the first $n-1$ objects, and swap with the $n-1$st object, and so forth. This algorithm takes time proportional to k, and $O(1)$ extra space is needed. The disadvantage is that the sample is not ordered. Also, record swapping is sometimes not allowed. We are allowed to swap pointers though, but this would then require $\Theta(n)$ extra space for pointers. If there are no records to begin with, then the space requirement is $\Omega(n)$. Formally we have:

Swapping method

FOR $i := n$ DOWNTO $n-k+1$ DO
    Generate a uniform [0,1] random variate U.
    $X \leftarrow \lceil iU \rceil$
    Swap $(A[X], A[i])$
RETURN $A[n-k+1], \ldots, A[n]$

The swapping method is very convenient. If we set $k = n$, then the returned array is a random permutation. Thus, the swapping method is based upon the principle that generating a random subset of size k is equivalent to generating the first k entries in a random permutation.

2.2. Classical sampling with membership checking.

If we are not allowed to swap information, then we are forced to check whether a certain element is not already picked. The checking can be achieved in a number of ways via different data structures. Regardless of the data structure, we can formulate the algorithm:

Classical sampling with membership checking

$S \leftarrow \emptyset$ (S will be the set of random integers to be returned)
FOR $i := 1$ TO k DO
    REPEAT
        Generate a random integer Z in $\{1, \ldots, n\}$.
    UNTIL NOT Member (Z) (Member returns true if an integer is already picked, and false otherwise.)
    $S \leftarrow S \cup \{Z\}$
RETURN S

The data structure used for S should support the following operations: initialize empty set, insert, member. Among the tens of possible data structures, the following are perhaps most representative:

A. The bit-vector implementation. Define an array of n bits, which are initially set to false, and which are switched to true upon insertion of an element.

B. An unordered array of chosen elements. Elements are added at the end of the array.

C. A binary search tree of chosen elements. The expected depth of the k-th element added to the tree is $\sim 2\log(k)$. The worst-case depth can be as large as k.

D. A height-balanced binary tree or 2-3 tree of chosen elements. The worst-case depth of the tree with k elements is $O(\log(k))$.

E. A bucket structure (open hashing with chaining). Partition $\{1, \ldots, n\}$ into k about equal intervals, and keep for each interval (or: bucket) a linked list of all elements chosen until now.

F. Closed hashing into a table of size a bit larger than k.

It is perhaps useful to give a list of expected complexities of the various operations needed on these data structures. We also include the space requirements, with the convention that the array of k integers to be returned is in any case

included in the space requirements.

DATA STRUCTURE	Initialize	Insert	Member	Space requirements
Bit vector	n	1	1	n
Unordered array	1	1	k	k
Binary search tree	1	$\log(k)$	$\log(k)$	k
Height-balanced tree	1	$\log(k)$	$\log(k)$	k
Buckets	k	1	1	k
Closed hashing	k	1	1	k

Timewise, none of the suggested data structures is better than the bit-vector data structure. The problem with the bit-vector implementation is not so much the extra storage proportional to n, because we can often use the already existing records and use common programming tricks (such as changing signs etcetera) to store the extra bits. The problem is the re-initialization necessary after a sample has been generated. At the very least, this will force us to consider the selected set S, and turn the k bits off again for all elements in S. Of course, at the very beginning, we need to set all n bits to false.

The first important quantity is the expected number of iterations in the sampling algorithm.

Theorem 2.1.

The expected number of iterations in classical sampling with membership checking is

$$\sum_{i=1}^{k} \frac{n}{n-i+1}.$$

For $k=n$, this is $n\sum_{i=1}^{n}\frac{1}{i}\sim n\ \log(n)$. When $k \leq \left\lceil \frac{n}{2} \right\rceil$, this number is $\leq 2k$.

Proof of Theorem 2.1.

Observe that to generate the i-th random integer, we carry out a series of independent experiments, each having probability of success $\frac{n-i+1}{n}$. This yields the given expected value. The asymptotic result when $k=n$ is trivially true. The general upper bound is obtained by a standard integral argument: bound the sum from above by

$$n \sum_{i=n-k+1}^{n} \frac{1}{i}$$

$$\leq n\left(\frac{1}{n-\left\lceil \frac{n}{2} \right\rceil+1} + \int_{n-k+1}^{n} \frac{1}{x}\, dx \right)$$

$$= n\left(\frac{1}{n-\left\lceil\frac{n}{2}\right\rceil+1}+\log\left(\frac{n}{n-k+1}\right)\right)$$

$$\leq 2+n\log\left(1+\frac{k-1}{n-k+1}\right)$$

$$\leq 2+\frac{n}{n-k+1}(k-1)$$

$$\leq 2+2(k-1)=2k\ .\ \blacksquare$$

What matters here is that the expected time increases no faster than $O(k)$ when k is at most half of the sample. Of course, when k is larger than $\frac{n}{2}$, one should really sample the complement set. In particular, the expected time for the bit-vector implementation is $O(k)$. For the tree methods, we obtain $O(k\log(k))$. If we work with ordered or unordered lists, then the generation procedure takes expected time $O(k^2)$. Finally, with the hash structure we have expected time $O(k)$ provided that we can show that the expected time of an insert or a delete is $O(1)$ (apply Wald's equation). Assume that we have a bucket structure with mk equal-sized intervals, where $m \geq 1$ is a design integer usually equal to 1. The interval number is an integer between 1 and mk, and integer $x \in \{1, \ldots, n\}$ is hashed to interval $\left\lceil\frac{x}{n}mk\right\rceil$. Thus, if the hash table has k elements, then every interval has about $\frac{1}{m}$ elements. The expected number of comparisons needed to check the membership of a random integer in a hash table containing i elements is bounded from above by $E(1+n_Z)$ where n_Z is equal to the number of elements in the interval Z, and Z is a random interval index, chosen with probability proportional to the cardinality of the interval. The "1" accounts for the comparison spent checking the endmarker in the chain. Thus, the expected number of comparisons is not greater than

$$1+\sum_{j=1}^{mk}\frac{\left(\frac{n}{mk}+1\right)}{n}n_j$$

$$\leq 1+i\frac{\left(\frac{n}{mk}+1\right)}{n}$$

$$= 1+\frac{i}{mk}+\frac{i}{n}\ .$$

In the worst case ($i=k$), this upper bound is $1+\frac{1}{m}+\frac{k}{n}\leq 2+\frac{1}{m}$. The upper bound is very loose. Nevertheless, we have an upper bound which is clearly $O(1)$. Also, if we can afford the space, it pays to take m as large as possible. One

possible hashing algorithm is given below:

Classical sampling with membership checking based on a hash table

This algorithm uses three arrays of integers of size k. An array of headers Head [1], . . . , Head [k] is initially set to 0. An array pointers to successor elements Next[1], . . . , Next[k] is also set to 0. The array A [1], . . . , A [k] will be returned.

```
FOR i:=1 TO k DO
    Accept ← False
    REPEAT
        Generate a random integer Z uniformly distributed on {1, . . . , n}.
        Bucket ←1+⌊k(Z−1)/n⌋
        Top ← Head [ Bucket ]
        IF Top=0
            THEN
                Head [ Bucket ] ←i
                A [k]←Z
                Accept ← True
            ELSE
                WHILE A [Top]≠Z AND Top≠0 DO
                    (Top, Top* ) ← (Next [Top], Top)
                IF Top=0 THEN
                    A [i]←Z
                    Next [ Top* ] ←i
                    Accept ← True
    UNTIL Accept
RETURN A [1], . . . , A [k]
```

The hashing algorithm requires $2k$ extra storage space. The array returned is not sorted, but sorting can be done in linear expected time. We give a short formal proof of this fact. It is only necessary to travel from bucket to bucket and sort the elements within the buckets (because an order-preserving hash function was used). If this is done by a simple quadratic method such as bubble sort or selection sort, then the overall expected time complexity is $O(k)$ (for the overhead costs) plus a constant times

$$E\left(\sum_{i=1}^{mk} n_i{}^2\right).$$

But n_i is hypergeometric with parameters n, l, k where l is the number of integers in the i-th bucket (this is about $\frac{n}{mk}$), i.e. for each j,

$$P(n_i=j) = \frac{\binom{l}{j}\binom{n-l}{k-j}}{\binom{n}{k}} .$$

We know that $E(n_i)=\frac{kl}{n}$, and this tends to $\frac{1}{m}$ as $k,n\to\infty$, and it does not exceed $\frac{1}{m}+\frac{k}{n}$ in any case. Simple computations show that

$$Var(n_i) = \frac{n-k}{n-1}\frac{n-l}{n}\frac{kl}{n}$$

which in turn tends to $\frac{1}{m}$ as $k,n\to\infty$, without exceeding $\frac{1}{m}+\frac{k}{n}$ for any value of k,m,n. Combining this, we see that the the expected time complexity is a constant times

$$\sim k(1+\frac{1}{m}) .$$

It is not greater than a constant times

$$mk\left((\frac{1}{m}+\frac{k}{n})^2+(\frac{1}{m}+\frac{k}{n})\right) = k(1+\frac{km}{n})(1+\frac{1}{m}+\frac{k}{n}) .$$

These expressions show that it is important to take m large. One should not fall into the trap of letting m increase with k,n because the set-up time is proportional to mk, the number of buckets. The hashing method with chaining, as given here, was implicitly given by Muller (1958) and studied by Ernvall and Nevalainen (1982). Its space inefficiency is probably its greatest drawback. Closed hashing with a table of size k has been suggested by Nijenhuis and Wilf (1975). Ahrens and Dieter (1985) consider closed hashing tables of size mk where now m is a number, not necessarily integer, greater than 1. See also Teuhola and Nevalainen (1982). It is perhaps instructive to give a brief description of the algorithm of Nijenhuis and Wilf (1975). An unordered sample $A[1], \ldots, A[k]$ will be generated, and an auxiliary vector Next[1], . . . , Next[k] of links is needed in the process. A pointer p points to the largest index i for which $A[i]$ is not yet specified.

Algorithm of Nijenhuis and Wilf

```
[SET-UP]
p ←k +1
FOR i :=1 TO k DO A [i ]←0
[GENERATOR]
REPEAT
    Generate a random integer X uniformly distributed on 1, . . . , n . Set
    Bucket←X mod k +1.
    IF A [Bucket]=0
        THEN A [Bucket]←X ,Next[Bucket]←0
        ELSE
            WHILE A [Bucket]≠X DO
                IF Next[Bucket]=0
                    THEN
                        REPEAT p ←p −1 UNTIL p =0 OR A [p ]=0
                        Next[Bucket]←p
                        Bucket←p
                    ELSE Bucket←Next[Bucket]
UNTIL p =0
RETURN A [1], . . . , A [k ]
```

The algorithm of Nijenhuis and Wilf differs slightly from standard closed hashing schemes because of the vector of links. The links actually create small linked lists within the table of size k. When we look at the cost associated with the algorithm, we note first that the expected number of uniform random variates needed is at the same as for all other classical sampling schemes (see Theorem 2.1). The search for an empty space ($p \leftarrow p-1$) takes time $O(k)$. The search for the end of the linked list (inner WHILE loop) takes on the average fewer than 2.5 link accesses per random variate X, independent of when X is generated and how large k and n are (Knuth, 1969, pp. 513-518). Thus, both expected time and space are $O(k)$.

2.3. Exercises.

1. The number of elements n_1 that end up in a bucket of capacity l in the bucket method is hypergeometrically distributed with parameters n, k, l. That is,

$$P(n_1=i) = \frac{\binom{l}{i}\binom{n-l}{k-i}}{\binom{n}{k}}, \quad 0 \le i \le \min(k,l).$$

In the text, we needed the expected value and variance of n_1. Derive these quantities.

2. Prove that the expected time in the algorithm of Nijenhuis and Wilf is $O(k)$.

3. **Weighted sampling without replacement.** Assume that we wish to generate a random sample of size k from $\{1, \ldots, n\}$, where the integers $1, \ldots, n$ have weights w_i. Drawing an integer from a set of integers is to be done with probability proportional to the weight of the integer. Using classical sampling, this involves dynamically updating a selection probability vector. Wong and Easton (1980) suggest setting up a binary tree of height $O(\log(n))$ in time $O(n)$ in a preprocessing step, and using this tree in the inversion method. Generating a random integer takes time $O(\log(n))$, while updating the tree has a similar cost. This leads to a method with worst-case time $O(k \log(n)+n)$. The space requirement is proportional to n (space is less critical because the vector of weights must be stored anyway). Develop a dynamic structure based upon the alias method or the method of guide tables, which has a better expected time performance for all vectors of weights.

3. SEQUENTIAL SAMPLING.

3.1. Standard sequential sampling.

In sequential sampling, we want an ordered sample of size k drawn from $1, \ldots, n$. An unordered sample can always be obtained by one of the methods described in the previous section, and in many cases (e.g. the hashing methods), sorting can be done extremely efficiently in expected time $O(k)$. What we will do in this chapter is different. The methods described here are fundamentally one pass methods in which the random sample is constructed in order. There are two possible strategies: first, we could grab each integer in $1, \ldots, n$ in turn, and decide whether to take it or leave it. It turns out, as we will see below, that for each decision, we need only compare a new uniform random variate with a certain threshold. Unfortunately, this standard sequential sampling algorithm takes

time proportional to n: it becomes particularly inefficient when k is much smaller than n. The second strategy circumvents this problem by generating the spacings between successive integers. Assume for a moment that each spacing can be generated in expected time $O(1)$ uniformly over all parameter values. Then the spacings method takes expected time $O(k)$. The problem here is that the distribution of the spacings is rather complicated; it also depends upon the partially generated sample.

In the standard sequential sampling algorithm of Jones (1962) and Fan, Muller and Rezucha (1962), the probability of selection of an integer depends upon only two quantities: the number of integers remaining to be selected, and the number of integers not yet processed. Initially, these quantities are k and n. To keep the notation simple, we will let k decrease during execution of the algorithm.

Standard sequential sampling

FOR $i:=1$ TO n DO
    Generate a uniform [0,1] random variate U.
    IF $U \leq \frac{k}{n-i+1}$ THEN select i, $k \leftarrow k-1$

Integer 1 is selected with probability $\frac{k}{n}$ as can easily be seen from the following argument: there are

$$\binom{n}{k}$$

ways of choosing a subset of size k from $1, \ldots, n$. Furthermore, of these,

$$\binom{n-1}{k-1}$$

include integer 1. The probability of inclusion of 1 should therefore be the ratio of these two numbers, or k/n. Note that this argument uses only k, the number of remaining integers to be selected, and n, the number of integers not yet processed. It can be used inductively to prove that the algorithm is correct. Note for example that if at any time in the algorithm $k=n$, then each of the remaining n integers in the file is selected with probability one. If at some point $k=0$, no more integers are selected. The time taken by the algorithm is proportional to n, but no extra space is needed. For small values of n, the standard sequential algorithm has little competition.

3.2. The spacings method for sequential sampling.

We say that a random variable X has the distribution $D(k,n)$ when X is distributed as the minimal integer in a random subset of size k drawn from $\{1, \ldots, n\}$. The spacings method for sequential sampling is defined as follows:

The spacings method for sequential sampling

$Y \leftarrow 0$ (Y is a running pointer)
REPEAT
    Generate a random integer X with distribution $D(k,n)$.
    $k \leftarrow k-1$, $n \leftarrow n-X$ (update parameters).
    Select $Y+X$, set $Y \leftarrow Y+X$
UNTIL $k=0$

In the algorithm, the original values of k and n are destroyed - this saves us the trouble of having to introduce two new symbols. If we can generate $D(k,n)$ random variates in expected time $O(1)$ uniformly over k and n, then the spacings method takes expected time $O(k)$. The space requirements depend of course on what is needed for the generation of $D(k,n)$. There are many possible algorithms for generating a $D(k,n)$ random variable. We discuss the following approaches:

1. The inversion method (Devroye and Yuen, 1981; Vitter, 1984).
2. The ghost sample method (Devroye and Yuen, 1981).
3. The rejection method (Vitter, 1983, 1984).

The three methodologies will be discussed in different subsections. All techniques require a considerable programming effort when implemented. In cases 1 and 3, most of the energy is spent on numerical problems such as the evaluation of ratios of factorials. Case 2 avoids the numerical problems at the expense of some additional storage (not exceeding $O(k)$). We will first state some properties of $D(k,n)$.

Theorem 3.1.

Let X have distribution $D(k,n)$. Then

$$P(X>i) = \frac{\binom{n-i}{k}}{\binom{n}{k}} \quad ,0\leq i\leq n-k \ ,$$

$$P(X=i) = \frac{\binom{n-i}{k-1}}{\binom{n}{k}} \quad ,1\leq i\leq n-k+1 \ .$$

Proof of Theorem 3.1.

Argue by counting the number of subsets of k out of n, the number of subsets of k out of $n-i$, and the number of subsets of $k-1$ out of $n-i$. ■

Theorem 3.2.

The random variable $X=\min(X_1, \ldots, X_k)$ is $D(k,n)$ distributed whenever $X_1, \ldots, X_k$ are independent random variables and each X_i is uniformly distributed on $\{1, \ldots, n-k+i\}$.

Proof of Theorem 3.2.

For $0\leq i\leq n-k$, notice that

$$P(Y>i) = \prod_{j=1}^{k} \frac{n-k+i-j}{n-k+i} = \prod_{j=0}^{k-1} \frac{n-i-j}{n-j} = \frac{\binom{n-i}{k}}{\binom{n}{k}} \ ,$$

which was to be shown. ■

From Theorem 3.2, we deduce without further work:

Theorem 3.3.

Let X be $D(k,n)$ distributed, and let Y be the minimum of k iid uniform $\{1, \ldots, n-k+1\}$ random variables. Then X is stochastically greater than Y, that is,

$$P(X>i) \geq P(Y>i) \quad \text{,all } i \text{ .}$$

Furthermore, related to the closeness of X and Y is the following collection of inequalities.

Theorem 3.4.

Let X and Y be as in Theorem 3.3. Then

$$\frac{n+1}{k+1} = E(X) \geq E(Y) \geq \frac{n-k+1}{k+1} .$$

In particular,

$$0 \leq E(X)-E(Y) \leq 1 .$$

Proof of Theorem 3.4.

In the proof, we let $U_1, \ldots, U_k$ be iid uniform [0,1] random variables. Note that

$$E(X) = \frac{1}{\binom{n}{k}} \sum_{i=1}^{n-k+1} i\binom{n-i}{k-1} = \frac{\binom{n+1}{k+1}}{\binom{n}{k}} = \frac{n+1}{k+1} .$$

Also,

$$E(Y) \geq (n-k+1)E(\min(U_1, \ldots, U_k)) = \frac{n-k+1}{k+1} .$$

Clearly,

$$E(X)-E(Y) \leq \frac{k}{k+1} . \blacksquare$$

3.3. The inversion method for sequential sampling.

The distribution function F for a $D(k,n)$ random variable X is

$$F(i) = P(X \leq i) = 1 - \frac{\binom{n-i}{k}}{\binom{n}{k}} \qquad 0 \leq i \leq n-k \ .$$

Thus, if U is a uniform [0,1] random variable, the unique integer X with the property that

$$F(X-1) < U \leq F(X)$$

has distribution function F, and is thus $D(k,n)$ distributed. The solution can be obtained sequentially by computing $F(1)$, $F(2)$,... until for the first time U is exceeded. The expected number of iterations is $E(X) = \frac{n+1}{k+1}$. The expected time complexity depends upon how F is computed. If $F(i)$ is computed from scratch (Fan, Muller and Rezucha, 1962), then time proportional to $k+1$ is needed, and X is generated in expected time proportional to n. This is unacceptable as it would lead to an $O(nk)$ sampling algorithm. Luckily, we can compute F recursively by noting that

$$\frac{1-F(i+1)}{1-F(i)} = \frac{\binom{n-i-1}{k}}{\binom{n-i}{k}} = \frac{n-i-k}{n-i} \ .$$

Using this, plus the fact that $1-F(0)=1$, we see that X can be generated in expected time proportional to $\frac{n+1}{k+1}$, and that a random sample can thus be generated in expected time proportional to n. This is still rather inefficient. Moreover, the recursive computation of F leads to unacceptable round-off errors for even moderate values of k and n. If F is recomputed from scratch, one must be careful in the handling of ratios of factorials so as not to introduce large cancelation errors in the computations. Thus, help can only come if we take care of the two key stumbling blocks:

1. The efficient computation of F.
2. The reduction of the number of iterations in the solution of $F(X-1) < U \leq F(X)$.

These issues are dealt with in the next section, where an algorithm of Devroye and Yuen (1981) is given.

3.4. Inversion-with-correction.

A reduction in the number of iterations for solving the inversion inequalities is only possible if we can guess the solution pretty accurately. This is possible thanks to the closeness of X to Y as defined in Theorems 3.3 and 3.4. The random variable Y introduced there has distribution function G where

$$G(i) = P(Y \leq i) = 1 - \left(\frac{n-k+1-i}{n-k+1} \right)^k , \quad 0 \leq i \leq n-k .$$

Recall that $F \leq G$ and that $0 \leq E(X-Y) \leq 1$. By inversion of G, Y can be generated quite simply as

$$Y \leftarrow \left\lfloor (1-(1-U)^{\frac{1}{k}})(n-k+1)+1 \right\rfloor$$

where U is the same uniform [0,1] random variate that will be used in the inversion inequalities for X. Because X is at least equal to Y, it suffices to start looking for a solution by trying $Y, Y+1, Y+2, \ldots$. This, of course, is the principle of inversion-with-correction explained in more detail in section III.2.5. The algorithm can be summarized as follows:

Inversion-with-correction (Devroye and Yuen, 1981)

IF $n=k$
THEN RETURN $X \leftarrow 1$
ELSE
Generate a uniform [0,1] random variate U.
$X \leftarrow \left\lfloor (1-(1-U)^{\frac{1}{k}})(n-k+1)+1 \right\rfloor$
$T \leftarrow 1-F(X)$
WHILE $1-U \leq T$ DO
$T \leftarrow T\frac{n-k-X}{n-X}$
$X \leftarrow X+1$
RETURN X

The point here is that the expected number of iterations in the WHILE loop is $E(X-Y)$, which is less than or equal to 1. Therefore, the expected time taken by the algorithm is a constant plus the expected time needed to compute F at one point. In the worst possible scenario, F is computed as a ratio of products of integers since

$$1-F(i) = \prod_{j=0}^{k-1} \frac{n-i-j}{n-j} .$$

This takes time proportional to k. The random sampling algorithm would therefore take expected time proportional to k^2. Interestingly, if F can be computed in time $O(1)$, then X can be generated in expected time $O(1)$, and the random sampling algorithm takes expected time $O(k)$. Furthermore, the algorithm requires bounded workspace.

If we accept the logarithm of the gamma function as a function that can be computed in constant time, then F can be computed in time $O(1)$ via:

$$\log(1-F(i)) = \log(\Gamma(n-i+1))+\log(\Gamma(n-k+1)) \\ -\log(\Gamma(n-i-k+1))+\log(\Gamma(n+1)) .$$

Of course, here too we are faced with some cancelation error. In practice, if one wants a certain fixed number of significant digits, there is no problem computing $\log(\Gamma)$ in constant time. From Lemma X.1.3, one can easily check that for $n \geq 8$, the series truncated at $k=3$ gives 7 significant digits. For $n<8$, the logarithm of n can be computed directly. There are other ways for obtaining a certain accuracy. See for example Hart et al. (1968) for the computation of $\log(\Gamma)$ as a ratio of two polynomials. See also section X.1.3 on the computation of factorials in general.

A final point about cancelation errors in the computation of $1-(1-U)^{1/k}$ when k is large. When E is an exponential random variable, the following two random variables are both distributed as $1-(1-U)^{1/k}$:

$$1-e^{-\frac{E}{k}}$$

$$\frac{\tanh(\frac{E}{2k})}{1+\tanh(\frac{E}{2k})} .$$

The second random variable is to be preferred because it is less susceptible to cancelation error.

3.5. The ghost point method.

Random variables with distribution $D(k,n)$ can also be generated by exploiting special properties such as Theorem 3.2. Recall that X is distributed as

$$1+\left\lfloor \min((n-k+1)U_1,(n-k+2)U_2, \ldots, (n-k+k)U_k) \right\rfloor$$

where $U_1, \ldots, U_k$ are independent uniform [0,1] random variables. Direct use of this property leads of course to an algorithm taking time $\Theta(k)$. Therefore, the random sampling algorithm corresponding to it would take time proportional to k^2. What distinguishes the algorithm from the inversion algorithms is that no heavy computations are involved. In the ghost point (or ghost sample) method, developed in Devroye and Yuen (1981), the fact that X is almost distributed as

the minimum of k iid random variables is exploited. The expected time per random variate is bounded from above uniformly over all $k \leq \rho n$ for some constant $\rho \in (0,1)$. Unfortunately, extra storage proportional to k is needed.

We coined the term "ghost point" because of the following embedding argument, in which X is written as the minimum of k independent random variables, which are linked to k iid random variables provided that we treat some of the iid random variables as non-existent. The iid random variables are $X_1, \ldots, X_k$, each uniformly distributed on $\{1, \ldots, n-k+1\}$. If we were to define X as the minimum of the X_i's, we would obtain an incorrect result. We can correct however by treating some of the X_i's as ghost points: define independent Bernoulli random variables $Z_1, \ldots, Z_k$ where $P(Z_i=1)=\frac{i-1}{n-k+i}$. The X_i's for which $Z_i=1$ are to be deleted. Thus, we can define an updated collection of random variables, $X_1', \ldots, X_k'$, where

$$X_i' = \begin{cases} X_i & \text{if } Z_i=0 \\ n-k+1 & \text{if } Z_i=1 \end{cases}.$$

Theorem 3.5.

For the construction given above,

$$X = \min(X_1', \ldots, X_k')$$

is $D(k,n)$ distributed.

Proof of Theorem 3.5.

Fix $0 \leq i \leq n-k$. Then,

$$\begin{aligned} P(X>i) &= \prod_{j=1}^{k} P(X_j'>i) \\ &= \prod_{j=1}^{k} (P(Z_i=1)+P(Z_i=0)P(X_i>k)) \\ &= \prod_{j=1}^{k} \left(\frac{j-1}{n-k+i}+\frac{n-k+1}{n-k+j}\frac{n-k+1-i}{n-k+1}\right) \\ &= \prod_{j=1}^{k} \frac{n-k+j-i}{n-k+j} \\ &= \frac{\binom{n-i}{k}}{\binom{n}{k}}. \blacksquare \end{aligned}$$

Every X_i has an equal probability of being the smallest. Thus, we can keep generating uniformly random integers from $1, \ldots, k$, without replacement of course, until we find one for which $Z_i = 0$, i.e. until we find an index for which the X_i is not a ghost point. Assume that we have skipped over m ghost points in the process. Then the X_i in question is distributed as the $m+1$-st smallest of the original sequence $X_1, \ldots, X_k$. The point is that such a random variable can be generated in expected time $O(1)$ because beta random variates can be generated in $O(1)$ expected time. Before proceeding with the expected time analysis, we give the algorithm:

The ghost point method

[SET-UP]
An auxiliary linked list L is needed, which is initially empty. The maximum list size is k. The stack size is Size.
Size $\leftarrow 0$.
[GENERATION]
REPEAT
    REPEAT
        Generate an integer W uniformly distributed on $\{1, \ldots, k\}$.
    UNTIL W is not in L
    Add W to L, Size $\leftarrow$ Size $+1$.
    Generate a uniform [0,1] random variate U.
UNTIL $U \geq \dfrac{W-1}{n-k+W}$
Generate a beta (Size, k − Size + 1) random variable B (note that B is distributed as the "Size" smallest of k iid uniform [0,1] random variables.)
RETURN $X \leftarrow \lfloor 1 + B(n-k+1) \rfloor$

We refer to the section on beta random variate generation for uniformly fast generators. If a beta variate generator is not locally available, one can always generate B as $\dfrac{G}{G+G'}$ where G, G' are independent gamma (W) and gamma $(k-W+1)$ random variables respectively.

For the analysis, we assume that $k \leq \rho n$ where $\rho \in (0,1)$ is a constant. Let N denote the number of W random variates generated in the inner REPEAT loop. It will appropriately measure the complexity of the algorithm provided that we can check membership in list L in constant time.

Theorem 3.6.

For the ghost point algorithm, we have

$$E(N) \leq c\frac{1+\rho}{(1-\rho)^2}$$

where $c>0$ is a universal constant and $k \leq \rho n$ where $\rho \in (0,1)$. Furthermore, the expected length of the list L, i.e. the expected value of Size, does not exceed $\frac{1}{1-\rho}$.

Proof of Theorem 3.6.

If T is the eventual value of Size, then

$$E(N \mid T) = \sum_{i=1}^{T} \frac{k}{k-i+1}.$$

Therefore, for constant $a \in (0,1)$,

$$E(N) = E(\sum_{i=1}^{T} \frac{k}{k-i+1}) = \sum_{i=1}^{k} \frac{k}{k-i+1} P(T \geq i) \quad \text{(by a change of } \textstyle\int\text{)}$$

$$\leq E(T^2) \sum_{i=1}^{k} \frac{k}{i^2(k-i+1)}$$

$$\leq E(T^2)\left(\frac{k}{k-\lfloor ak \rfloor+1} \sum_{i=1}^{\infty} \frac{1}{i^2} + k \sum_{i>\lfloor ak \rfloor} \frac{1}{i^2}\right)$$

$$\leq E(T^2)\left(\frac{\pi^2}{6(1-a)} + k\left(\frac{1}{(ak)^2} + \int_{ak}^{\infty} \frac{1}{x^2}\, dx\right)\right)$$

$$= E(T^2)\left(\frac{\pi^2}{6(1-a)} + \frac{1}{ka^2} + \frac{1}{a}\right)$$

which is approximately minimal when

$$a = \frac{\sqrt{6}}{\pi+\sqrt{6}}.$$

The upper bound is thus not greater than a constant times $E(T^2)$. But T is stochastically smaller than a geometric random variable with probability of success $\frac{n-k+1}{n} \geq 1-\rho$. Thus, $E(T) \leq 1/(1-\rho)$ and

$$E(T^2) \leq \left(\frac{1}{1-\rho}\right)^2 + \frac{\rho}{(1-\rho)^2} = \frac{1+\rho}{(1-\rho)^2}. \quad \blacksquare$$

The value of the constant c can be deduced from the proof. However, no attempt was made to obtain the best possible constant there. The assumption that membership checking in L can be done in constant time requires that a bit vector of k flags be used, indicating for each integer whether it is included in L or not. Setting up the bit vector takes time proportional to k. However, this cost is to be born just once, for after one variate X is generated, the flags can be reset by emptying the list L. The expected time taken by the reset operation is thus equal to a constant plus the expected length of the list, which, as we have shown in Theorem 6, is bounded by $1/(1-\rho)$. For the global random sampling algorithm, the total expected cost of setting and resetting the bit vector does not exceed a constant times k.

Fortunately, we can avoid the bit vector of flags altogether. Membership checking in list L can always be done in time not exceeding the length of the list. Even with this grotesquely inefficient implementation, one can show (see exercises) that the expected time for generating X is bounded uniformly over all $k \leq \rho n$.

The issue of membership checking can be sidestepped if we generate integers without replacement by the swapping method. This would require an additional vector initially set to $1, \ldots, k$. After X is generated, this vector is slightly permuted - its first "Size" members for example constitute our list L. This does not matter, as long as we keep track of where integer k is. To get ready for generating a $D(k-1,n)$ random variate, we need only swap k with the last element of the vector, so that the first $k-1$ components form a permutation of $1, \ldots, k-1$. Thus, fixing the vector between random variates takes a constant time. Note also that to generate X, the expected time is now bounded by a constant times the expected length of the list, which we know is not greater than $1/(1-\rho)$. This is due to the fact that the inner loop of the algorithm is now replaced by one loopless section of code.

When $k > \rho n$, one should use another algorithm, such as the following piece taken from the standard sequential sampling algorithm:

$X \leftarrow 0$
REPEAT
    Generate a uniform random variate U.
    $X \leftarrow X+1$
UNTIL $U \leq \dfrac{k}{n-X+1}$
RETURN X

The expected number of uniform [0,1] random variates needed by this algorithm is $E(X) = \dfrac{n+1}{k+1} \leq \dfrac{n}{k} \leq \dfrac{1}{\rho}$. The combination of the two algorithms depending

upon the relative sizes of k and n yields an $O(1)$ expected time algorithm for generating X. The optimal value of the threshold ρ will vary from implementation to implementation. Note that if a membership swap vector is used, it is best to reset the vector after each X is generated by traversing the list in LIFO order.

3.6. The rejection method.

The generation of $D(k,n)$ random variates by the rejection method creates special problems, because the probabilities p_i contain ratios of factorials. Whenever we evaluate p_i, we can use one of two approaches: p_i is evaluated in constant time (this, in fact, assumes that the logarithm of the Γ function is available in constant time, and that we do give up our infinite accuracy because a Stirling series approximation is used), and p_i is computed in time proportional to $k+1$ (i.e. the factorials are evaluated explicitly). With the latter model, called the explicit factorial model, it does not suffice to find a dominating probability vector q_i which satisfies

$$p_i \leq cq_i$$

for some constant c independent of k,n. We could indeed still end up with an expected time complexity that is not uniformly bounded over k,n. Thus, in the explicit factorial model, we have to find good dominating and squeeze curves which will allow us to effectively avoid computing p_i except perhaps about $O(\frac{1}{k})$ percent of the time. Because $D(k,n)$ is a two-parameter family, the design is quite a challenge. We will not be concerned with all the details here, just with the flavor of the problem. The detailed development can be found in Vitter (1984). Nearly all of this section is an adaptation of Vitter's results. Gehrke (1984) and Kawarasaki and Sibuya (1982) have also developed rejection algorithms, similar to the ones discussed in this section.

At the very heart of the design is once again a collection of inequalities. Recall that for a $D(k,n)$ random variable X,

$$p_i = P(X=i) = \frac{\binom{n-i}{k-1}}{\binom{n}{k}} \qquad (1 \leq i \leq n-k+1).$$

Theorem 3.7.

We have

$$h_1(i) \le p_i \le c_1 g_1(i+1)$$

where

$$h_1(i) = \frac{k}{n}\left(1-\frac{i-1}{n-k+1}\right)^{k-1} \qquad (1 \le i \le n-k+1),$$

$$c_1 = \frac{n}{n-k+1},$$

$$g_1(x) = \frac{k}{n}\left(1-\frac{x-1}{n}\right)^{k-1} \qquad (1 \le x \le n+1).$$

Also,

$$h_2(i) \le p_i \le c_2 g_2(i+1)$$

where

$$h_2(i) = \frac{k}{n}\left(1-\frac{k-1}{n-i+1}\right)^{i-1} \qquad (1 \le i \le n-k+1),$$

$$c_2 = \frac{k}{k-1}\frac{n-1}{n},$$

$$g_2(i) = \frac{k-1}{n-1}\left(1-\frac{k-1}{n-1}\right)^{i-1} \qquad (i \ge 1).$$

Note that g_1 is a density in x, and that g_2 is a probability vector in i.

Proof of Theorem 3.7.

Note that

$$p_i = \frac{k}{n-k+1}\prod_{j=0}^{k-2}\frac{n-i-j}{n-j}$$

$$\le \frac{k}{n-k+1}\left(\frac{n-i}{n}\right)^{k-1}$$

$$= \frac{k}{n-k+1}\left(1-\frac{i}{n}\right)^{k-1}$$

$$= c_1 g_1(i+1).$$

Furthermore,

$$h_1(i) = \frac{k}{n}\left(1-\frac{i-1}{n-k+1}\right)^{k-1}$$

$$\le \frac{k}{n}\prod_{j=0}^{k-2}\frac{n-k-i+2+j}{n-k+1+j}$$

$$= \frac{k}{n} \prod_{j=0}^{k-2} \frac{n-i-j}{n-1-j}$$

$$= p_i \ .$$

This concludes the first half of the proof. For the second half, we argue similarly. Indeed, for $i \geq 1$,

$$p_i = \frac{k}{n} \prod_{j=0}^{i-2} \frac{n-k-j}{n-1-j}$$

$$\leq \frac{k}{n} \left(\frac{n-k}{n-1} \right)^{i-1}$$

$$= \frac{k}{k-1} \frac{n-1}{n} \frac{k-1}{n-1} \left(1 - \frac{k-1}{n-1} \right)^{i-1}$$

$$= c_2 g_2(i) \ .$$

Furthermore,

$$h_2(i) = \frac{k}{n} \left(\frac{n-k-i+2}{n-i+1} \right)^{i-1} \leq \frac{k}{n} \prod_{j=0}^{i-2} \frac{n-k-j}{n-1-j} = p_i \ . \blacksquare$$

Random variate generators based upon both groups of inequalities are now easy to find, because g_1 is basically a transformed beta density, and g_2 is a geometric probability vector. In the case of g_1, we need to use rejection from a continuous density of course. The expected number of iterations in case 1 is $c_1 = n/(n-k+1)$ (which is uniformly bounded over all k, n with $k \leq \rho n$, where $\rho \in (0,1)$ is a constant). In case 2, we have $c_2 = \frac{k}{k-1} \frac{n-1}{n}$, and this is uniformly bounded over all $k \geq 2$ and all $n \geq 1$.

First rejection algorithm

REPEAT

Generate two iid uniform [0,1] random variates U, V.

$Y \leftarrow 1+n\,(1-U^{\frac{1}{k}})$ (Y has density g_1)

$X \leftarrow \lfloor Y \rfloor$

IF $X \leq n-k+1$

THEN

$$\text{Accept} \leftarrow [V \leq \frac{n-k+1}{n} \left(\frac{1-\frac{X-1}{n-k+1}}{1-\frac{Y-1}{n}} \right)^{k-1}]$$

IF NOT Accept THEN

$$\text{Accept} \leftarrow [V \leq \frac{p_X}{c_1 g_1(Y)}]$$

UNTIL Accept

RETURN X

Second rejection algorithm

REPEAT

Generate an exponential random variate E and a uniform [0,1] random variate V.

$X \leftarrow \left\lceil -E / \log(1-\frac{k-1}{n-1}) \right\rceil$ (X has probability vector g_2)

IF $X \leq n-k+1$

THEN

$$\text{Accept} \leftarrow [V \leq \left(\frac{1-\frac{k-1}{n-X+1}}{1-\frac{k-1}{n-1}} \right)^{X-1}]$$

IF NOT Accept THEN

$$\text{Accept} \leftarrow [V \leq \frac{p_X}{c_2 g_2(X)}]$$

UNTIL Accept

RETURN X

3.7. Exercises.

1. Assume that in the standard sequential sampling algorithm, each element is chosen with equal probability $\frac{k}{n}$. The sample size is a binomial $(n, \frac{k}{n})$ random variable N. Show that as $k \to \infty, n \to \infty, n-k \to \infty$, we have
$$P(N=k) \sim \sqrt{\frac{n}{2\pi k(n-k)}}.$$
2. Assume that $k \leq \rho n$ for some fixed $\rho \in (0,1)$. Show that if the ghost point algorithm is used to generate a random sample of size k out of n, the expected time is bounded by a function of ρ only. Assume that a vector of membership flags is used in the algorithm, but do not switch to the standard sequential method when during the generation process, the current value of k temporarily exceeds ρ times the current value of n (as is suggested in the text).
3. Assume that in the ghost point algorithm, membership checking is done by traversing the list L. Show that to generate a random variate X with distribution $D(k,n)$, the algorithm takes expected time bounded by a function of $\frac{k}{n}$ only.
4. If X is $D(k,n)$ distributed, then
$$Var(X) = \frac{(n+1)(n-k)k}{(k+2)(k+1)^2}.$$
5. Consider the explicit factorial model in the rejection algorithm. Noting that the value of p_X can be computed in time $\min(k, X+1)$, find good upper bounds for the expected time complexity of the two rejection algorithms given in the text. In particular, prove that for the first algorithm, the expected time complexity is uniformly bounded over $k \leq \rho n$ where $\rho \in (0,1)$ is a constant (Vitter, 1984).

4. OVERSAMPLING.

4.1. Definition.

If we are given a random sequence of k uniform order statistics, and transform it via truncation into a random sequence of ordered integers in $\{1, \ldots, n\}$, then we are almost done. Unfortunately, some integers could appear more than once, and it is necessary to generate a few more observations. If we had started with $k_1 > k$ uniform order statistics, then with some luck we could have ended up with at least k different integers. The probability of this increases rapidly with k_1. On the other hand, we do not want to take k_1 too large, because then we will be left with quite a bit of work trying to eliminate some values to obtain a sample of precisely size k. This method is called oversampling. The

main issue at stake is the choice of k_1 as a function of k and n so that not only the total expected time is $O(k)$, but the total expected time is approximately minimal. One additional feature that makes oversampling attractive is that we will obtain an ordered random sample. Because the method is basically a two step method (uniform sample generator, followed by excess eliminator), it is not included in the section on sequential methods.

The oversampling algorithm

REPEAT

Generate $U_{(1)} < \cdots < U_{(k_1)}$, the order statistics of a uniform sample of size k_1 on [0,1].

Determine $X_i \leftarrow \lfloor 1+nU_{(i)} \rfloor$ for all i, and construct, after elimination of duplicates, the ordered array $X_{(1)}, \ldots, X_{(K_1)}$.

UNTIL $K_1 \geq k$

Mark a random sample of size K_1-k of the sequence $X_{(1)}, \ldots, X_{(K_1)}$ by the standard sequential sampling algorithm.

RETURN the sequence of k unmarked X_i's.

The amount of extra storage needed is K_1-k. Note that this is always bounded by k_1-k. For the expected time analysis of the algorithm, we observe that the uniform sample generation takes expected time $c_u k_1$, and that the elimination step takes expected time $c_e K_1$. Here c_u and c_e are positive constants. If the standard sequential sampling algorithm is replaced by classical sampling for elimination (i.e., to mark one integer, generate random integers on $\{1, \ldots, K_1\}$ until a nonmarked integer is found), then the expected time taken by the elimination algorithm is

$$c_e \sum_{i=1}^{K_1-k} \frac{K_1}{K_1-i+1}$$

$$\leq c_e (K_1-k) \frac{K_1}{k+1} .$$

What we should also count in the expected time complexity is the probability of accepting a sequence. The results are combined in the following theorem:

Theorem 4.1.

Let c_u, c_e be as defined above. Assume that $n > k$ and that

$$k_1 = k + (k+a)/\log(\frac{n}{k})$$

for some constant $a > 0$. Then the expected time spent on the uniform sample is

$$E(N) c_u k_1$$

where $E(N)$ is the expected number of iterations. We have the following inequality:

$$E(N) = \frac{1}{P(K_1 \geq k)} \leq \frac{1}{1-e^{-a}} .$$

The expected time spent marking does not exceed $c_e k_1$, which, when $a = O(k), \frac{k}{n} \to 0$, is asymptotic to $c_e k$. If classical sampling is used for marking, then it is not greater than

$$\frac{k_1}{k+1} \frac{k+a}{\log(\frac{n}{k})} .$$

Proof of Theorem 4.1.

The expression for the expected time spent generating order statistics is based upon Wald's equation. Furthermore, $E(N) = 1/P(K_1 \geq k)$. But

$$P(K_1 < k) \leq \binom{n}{k} \left(\frac{k}{n}\right)^{k_1} \leq \left(\frac{en}{k}\right)^{k_1}$$

$$= \left(\frac{n}{k}\right)^{k-k_1} e^k$$

$$= e^{-a} .$$

The only other statement in the theorem requiring some explanation is the statement about the marking scheme with classical sampling. The expected time spent doing so does not exceed c_e times

$$E((K_1 - k)\frac{K_1}{k+1} \mid K_1 \geq k)$$

$$\leq \frac{(k_1 - k)k_1}{k+1} . \blacksquare$$

Once again, we see that uniformly over $k \leq \rho n$, the expected time is bounded by a constant times k, for all fixed $\rho \in (0,1)$ and for all choices of a that are either fixed or vary with k in such a manner that $a = O(k)$. We recommend that a be taken large but fixed, say $a = 10$. Note that in the special case that $\frac{n}{k} \to \infty$, $a = O(k)$, $k_1 \sim k$. Thus, the expected time of the marking section based upon classical sampling is $o(k)$, i.e. it is asymptotically negligible. Also, if $a \to \infty$, $E(N) \to 1$ for all choices of n, k. In those cases, the main contributions to the expected time complexity come from the generation of the k_1 uniform order statistics, and the elimination of the marked values (not the marking itself).

4.2. Exercises.

1. Show that for the choice of k_1 given in Theorem 4.1, we have $E(N) \to 1$ as $n, k \to \infty$, $\frac{k}{n} \to \rho \in (0,1)$. Do this by proving the existence of a universal constant A depending upon ρ only such that $E(N) \leq 1 + \frac{A}{\sqrt{n}}$.

5. RESERVOIR SAMPLING

5.1. Definition.

There is one particular sequential sampling problem deserving special attention, namely the problem of sampling records from large (presumably external) files with an unknown total population. While k is known, n is not. Knuth (1969) gives a particularly elegant solution for drawing such a random sample called the reservoir method. See also Vitter (1985). Imagine that we associate with each of the records an independent uniform [0,1] random variable U_i. If the object is simply to draw a random set of size k, it suffices to pick those k records that correspond to the k largest values of the U_i's. This can be done sequentially:

Reservoir sampling

[NOTE: S is a set of pairs (i, U_i).]
FOR $i := 1$ TO k DO
    Generate a uniform [0,1] random variate U_i, and add (i, U_i) to S. Keep track of the pair (m, U_m) with the smallest value for the uniform random variate.
$i \leftarrow k+1$ (i is a record counter)
WHILE NOT end of file DO
    Generate a uniform [0,1] random variate U_i.
    IF $U_i \geq U_m$
        THEN
            Delete (m, U_m) from S.
            Insert (i, U_i) in S.
            Find a new smallest pair (m, U_m).
    $i \leftarrow i+1$
RETURN all integers i for which $(i, U_i) \in S$.

The general algorithm of reservoir sampling given above returns integers (indices); it is trivial to modify the algorithm so that actual records are returned. It is clear that n uniform random variates are needed. In addition, there is a cost for updating S. The expected number of deletions in S (which is equal to the number of insertions minus k) is

$$\sum_{i=k+1}^{n} P((i, U_i) \text{ is inserted in } S)$$
$$= \sum_{i=k+1}^{n} \frac{k}{i}$$
$$= k \log(\frac{n}{k}) + o(1)$$

as $k \rightarrow \infty$. Here we used the fact that the first n terms of the harmonic series are $\log(n) + \gamma + o(1/n)$ where γ is Euler's constant. There are several possible implementations for the set S. Because we are mainly interested in ordinary insertions and deletions of the minimum, the obvious choice should be a heap. Both the expected and worst-case times for a delete operation in a heap of size k are proportional to $\log(k)$ as $k \rightarrow \infty$. The overall expected time complexity for deletions is proportional to

$$k \log(\frac{n}{k}) \log(k)$$

as $k \rightarrow \infty$. This may or may not be larger than the $\theta(n)$ contribution from the uniform random variate generator. With ordered or unordered linked lists, the

time complexity is worse. In the exercise section, a hash structure exploiting the fact that the inserted elements are uniformly distributed is explored.

5.2. The reservoir method with geometric jumps.

In some applications, such as when records are stored on a sequential access device (e.g., a magnetic tape), there is no way that we can avoid traversing the entire file. When the records are in RAM or on a random access device, it is possible to skip over any number of records in constant time: in those cases, it should be possible to get rid of the $\theta(n)$ term in the time complexity. Given (m, U_m), we know that the waiting time until the occurrence of a uniform value greater than U_m is geometrically distributed with success probability $1-U_m$. It can be generated as $\left\lceil -E/\log(U_m) \right\rceil$ where E is an exponential random variate. The corresponding record-breaking value is uniformly distributed on $[U_m, 1]$. Thus, the reservoir method with geometric jumps can be summarized as follows:

Reservoir sampling with geometric jumps

[NOTE: S is a set of pairs (i, U_i).]
FOR i:=1 TO k DO
    Generate a uniform [0,1] random variate U_i, and add (i, U_i) to S. Keep track of the pair (m, U_m) with the smallest value for the uniform random variate.
$i \leftarrow k$ (i is a record counter)
WHILE True DO
    Generate an exponential random variate E.
    $i \leftarrow i + \left\lceil -E/\log(U_m) \right\rceil$.
    IF i not outside file
        THEN
            Generate a uniform $[U_m, 1]$ random variate U_i.
            Delete (m, U_m) from S.
            Insert (i, U_i) in S.
            Find a new smallest pair (m, U_m).
        ELSE RETURN all integers i for which $(i, U_i) \in S$.

The analysis of the previous section about the expected time spent updating S remains valid here. The difference is that the $\theta(n)$ has disappeared from the picture, because we only generate uniform random variates when insertions in S are needed.

5.3. Exercises.

1. Design a bucket-based dynamic data structure for the set S, which yields a total expected time complexity for N insertions and deletions that is $o(N\log(k))$ when $N,k\to\infty$. Note that inserted elements are uniformly distributed on $[U_m,1]$ where U_m is the minimal value present in the set. Initially, S contains k iid uniform [0,1] random variates. For the heap implementation of S, the expected time complexity would be $\theta(N\log(k))$.