Chapter Three
DISCRETE RANDOM VARIATES

1. INTRODUCTION.

A discrete random variable is a random variable taking only values on the nonnegative integers. In probability theoritical texts, a discrete random variable is a random variable which takes with probability one values in a given countable set of points. Since there is a one-to-one correspondence between any countable set and the nonnegative integers, it is clear that we need not consider the general case. In most cases of interest to the practitioner, this one-to-one correspondence is obvious. For example, for the countable set $1,\frac{1}{2},\frac{1}{4},\frac{1}{8},\ldots$, the mapping is trivial.

The distribution of a discrete random variable X is determined by the probability vector $p_0,p_1,\ldots$:

$$P(X=i) = p_i \qquad (i=0,1,2,\ldots) .$$

The probability vector can be given to us in several ways, such as

A. A table of values $p_0,p_1, \ldots, p_K$. Note that here it is necessary that X can only take finitely many values.

B. An analytical expression such as $p_i=2^{-i}$ $(i \geq 1)$. This is the standard form in statistical applications, and most popular distributions such as the binomial, Poisson and hypergeometric distributions are given in this form.

C. A subprogram which allows us to compute p_i for each i. This is the "black box" model.

D. Indirectly.. For example, the **generating function**

$$m(s) = \sum_{i=0}^{\infty} p_i s^i \qquad (s \in R)$$

can be given. Sometimes, a recursive equation allowing us to compute p_i from $p_j, j<i$, is given.

In cases B, C and D, we should also distinguish between methods for the generation of X when X has a fixed distribution, and methods that should be applicable when X belongs to a certain family of integer-valued random variables.

The methods that will be described below apply usually to only one or two of the cases listed above. Some of these are based on principles that are equally applicable to continuous random variate generation like inversion, composition and rejection. Other principles are unique to discrete random variate generation like the alias principle and the method of guide tables. In any case, this chapter goes hand in hand with chapter II. Very often, the best generator for a certain density uses a clever combination of discrete random variate generation principles and standard methods for continuous random variates. The actual discussion of such combinations is deferred until chapter VIII.

When we give examples in this chapter, we will refer to well-known discrete distributions. At this point, it is instructive to summarize some of these distributions.

Name of distribution	Parameters	$P(X=i)$	Range for i
Poisson(λ)	$\lambda>0$	$\frac{e^{-\lambda}\lambda^i}{i!}$	$i\geq 0$
Binomial(n,p)	$n\geq 1; 0\leq p\leq 1$	$\binom{n}{i} p^i(1-p)^{n-i}$	$0\leq i\leq n$
Negative binomial(n,p)	$n\geq 1; p>0$	$\binom{n+i-1}{i} p^i(1+p)^{n+i}$	$i\geq 0$
Logarithmic series(θ)	$0<\theta<1$	$\frac{\theta^i}{-\log(1-\theta)\, i}$	$i\geq 1$
Geometric(p)	$0<p<1$	$p(1-p)^{i-1}$	$i\geq 1$

We refer the reader to Johnson and Kotz (1969, 1982) or Ord (1972) for a survey of the properties of the most frequently used discrete distributions in statistics. For surveys of generators, see Schmeiser (1983), Ahrens and Kohrt (1981) or Ripley (1983).

Some of the methods described below are extremely fast: this is usually the case for well-designed table methods, and for the alias method or its variant, the alias-urn method. The method of guide tables is also very fast. Only finite-valued discrete random variates can be generated by table methods because tables must be set-up beforehand. In dynamic situations, or when distributions are infinite-tailed, slower methods such as the inversion method can be used.

avoided altogether as can be seen from the following example.

Example 2.1. Poisson random variates by sequential search.

We can quickly verify that for the Poisson (λ) distribution,

$$p_{i+1} = \frac{\lambda}{i+1} p_i \ , \ p_0 = e^{-\lambda} \ .$$

Thus, the sequential search algorithm can be simplified somewhat by recursively computing the values of p_i during the search:

Poisson generator using sequential search

Generate a uniform [0,1] random variate U.
Set $X \leftarrow 0$, $P \leftarrow e^{-\lambda}$, $S \leftarrow P$.
WHILE $U > S$ DO
  $X \leftarrow X+1$, $P \leftarrow \frac{\lambda P}{X}$, $S \leftarrow S+P$.
RETURN X

We should note here that the expected number of comparisons is equal to $E(X+1)=\lambda+1$. ■

A slight improvement in which the variable S is not needed was suggested by Kemp(1981). Note however that this forces us to destroy U:

Inversion by sequential search (Kemp, 1981)

Generate a uniform [0,1] random variate U.
$X \leftarrow 0$
WHILE $U > p_X$ DO
  $U \leftarrow U - p_X$
  $X \leftarrow X+1$
RETURN X

2. THE INVERSION METHOD.

2.1. Introduction.

In the **inversion method**, we generate one uniform [0,1] random variate U and obtain X by a monotone transformation of U which is such that $P(X=i)=p_i$. If we define X by

$$F(X-1) = \sum_{i<X} p_i < U \leq \sum_{i \leq X} p_i = F(X) ,$$

then it is clear that $P(X=i)=F(i)-F(i-1)=p_i$. This is comparable to the inversion method for continuous random variates. The solution of the inequality shown above is uniquely defined with probability one. An exact solution of the inversion inequalities can always be obtained in finite time, and the inversion method can thus truly be called universal. Note that for continuous distributions, we could not invert in finite time except in special cases.

There are several possible techniques for solving the inversion inequalities. We start with the simplest and most universal one, i.e. a method which is applicable to all discrete distributions.

Inversion by sequential search

Generate a uniform [0,1] random variate U.

Set $X \leftarrow 0$, $S \leftarrow p_0$.

WHILE $U > S$ DO

  $X \leftarrow X+1; S \leftarrow S+p_X$

RETURN X

Note that S is adjusted as we increase X in the **sequential search algorithm.** This method applies to the "black box" model, and it can handle infinite tails. The time taken by the algorithm is a random variable N, which can be equated in first approximation with the number of comparisons in the WHILE condition. But

$$P(N=i) = P(X=i-1) = p_{i-1} \quad (i \geq 1) .$$

Thus, $E(N)=E(X)+1$. In other words, the tail of the distribution of X determines the expected time taken by the algorithm. This is an uncomfortable situation in view of the fact that $E(X)$ can possibly be ∞. There are other more practical objections: p_i must be computed many times, and the consecutive additions $S \leftarrow S+p_X$ may lead to inadmissible accumulated errors. For these reasons, the sequential search algorithm is only recommended as a last resort. In the remainder of this section, we will describe various methods for improving the sequential search algorithm. In particular cases, the computation of p_X can be

2.2. Inversion by truncation of a continuous random variate.

If we know a continuous distribution function G on $[0,\infty)$ with the property that G agrees with F on the integers, i.e.

$$G(i+1) = F(i) \qquad (i=0,1,\ldots),\ G(0) = 0\ ,$$

then we could use the following algorithm for generating a random variate X with distribution function F:

Inversion by truncation of a continuous random variate

Generate a uniform [0,1] random variate U.

RETURN $X \leftarrow \lfloor G^{-1}(U) \rfloor$

This method is extremely fast if G^{-1} is explicitly known. That it is correct follows from the observation that for all $i \geq 0$,

$$P(X \leq i) = P(G^{-1}(U) < i+1) = P(U < G(i+1)) = G(i+1) = F(i)\ .$$

The task of finding a G such that $G(i+1)-G(i)=p_i$, all i, is often very simple, as we illustrate below with some examples.

Example 2.2. The geometric distribution.

When $G(x)=1-e^{-\lambda x}$, $x \geq 0$, we have

$$G(i+1)-G(i) = e^{-\lambda i}-e^{-\lambda(i+1)}$$
$$= e^{-\lambda i}(1-e^{-\lambda})$$
$$= (1-q)q^i \qquad (i \geq 0)\ ,$$

where $q=e^{-\lambda}$. From this, we conclude that

$$\left\lceil -\frac{1}{\lambda}\log U \right\rceil$$

is geometrically distributed with parameter $e^{-\lambda}$. Equivalently, $\left\lceil \frac{\log U}{\log(1-p)} \right\rceil$ is geometrically distributed with parameter p. Equivalently, $\left\lceil -\frac{E}{\log(1-p)} \right\rceil$ is geometrically distributed with the same parameter, when E is an exponential random variate. ■

Example 2.3. A family of monotone distributions.

Consider $G(x)=1-x^{-b}$, $x \geq 1$, $G(1)=0$, $b>0$. We see that $G(i+1)-G(i)=i^{-b}-(i+1)^{-b}$. Thus a random variate X with probability vector

$$p_i = \frac{1}{i^b}-\frac{1}{(i+1)^b} \quad (i \geq 1)$$

can be generated as $\left\lfloor U^{-\frac{1}{b}} \right\rfloor$. In particular, $\left\lfloor \frac{1}{U} \right\rfloor$ has probability vector

$$p_i = \frac{1}{i(i+1)} \quad (i \geq 1) . ■$$

Example 2.4. Uniformly distributed discrete random variates.

A discrete random variable is said to be uniformly distributed on $\{1,2, \ldots, K\}$ when $p_i=\frac{1}{K}$ for all $1 \leq i \leq K$. Since $p_i=G(i+1)-G(i)$ where $G(x)=\frac{x-1}{K}$, $1 \leq x \leq K+1$, we see that $X \leftarrow \lfloor 1+KU \rfloor$ is uniformly distributed on the integers 1 through K. ■

2.3. Comparison-based inversions.

The sequential search algorithm uses comparisons only (between U and certain functions of the p_j's). It was convenient to compare U first with p_0, then with p_0+p_1 and so forth, but this is not by any means an optimal strategy. In this section we will highlight some other strategies that are based upon comparisons only. Some of these require that the probability vector be finite.

For example, if we were allowed to permute the integers first and then perform sequential search, then we would be best off if we permuted the integers in such a way that $p_0 \geq p_1 \geq p_2 \geq \cdots$. This is a consequence of the fact that the number of comparisons is equal to $1+X$ where X is the random variate generated. Reorganizations of the search that result from this will usually not preserve the monotonicity between U and X. Nevertheless, we will keep using the term inversion.

The improvements in expected time by reorganizations of sequential search can sometimes be dramatic. This is the case in particular when we have peaked distributions with a peak that is far removed from the origin. A case in point is the binomial distribution which has a mode at $\lfloor np \rfloor$ where n and p are the parameters of the binomial distribution. Here one could first verify whether $U \leq F(\lfloor np \rfloor)$, and then perform a sequential search "up" or "down" depending upon the outcome of the comparison. For fixed p, the expected number of comparisons grows as $\sqrt{n}$ instead of as n as can easily be checked. Of course, we have to compute either directly or in a set-up step, the value of F at $\lfloor np \rfloor$. A similar improvement can be implemented for the Poisson distribution. Interestingly, in this simple case, we do preserve the monotonicity of the transformation.

Other reorganizations are possible by using ideas borrowed from computer science. We will replace linear search (i.e., sequential search) by tree search. For good performance, the search trees must be set up in advance. And of course, we will only be able to handle a finite number of probabilities in our probability vector.

One can construct a binary search tree for generating X. Here each node in the tree is either a leaf (terminal node), or an internal node, in which case it has two children, a left child and a right child. Furthermore, each internal node has associated with it a real number, and each leaf contains one value, an integer between 0 and K. For a given tree, we obtain X from a uniform [0,1] random variate U in the following manner:

Inversion by binary search

Generate a uniform [0,1] random variate U.
Ptr $\leftarrow$ Root of tree (Ptr points to a node).
WHILE Ptr $\neq$ Leaf DO
    IF Value (Ptr) $> U$
        THEN Ptr $\leftarrow$ Leftchild (Ptr)
        ELSE Ptr $\leftarrow$ Rightchild (Ptr).
RETURN $X \leftarrow$ Value (Ptr)

Here, we travel down the tree, taking left and right turns according to the comparisons between U and the real numbers stored in the nodes, until we reach a leaf. These real numbers must be chosen in such a way that the leafs are reached with the correct probabilities. There is no particular reason for choosing $K+1$ leaves, one for each possible outcome of X, except perhaps economy of storage. Having fixed the shape of the tree and defined the leaves, we are left with the task of determining the real numbers for the K internal nodes. The real number for a given internal node should be equal to the probabilities of all the leaves encountered before the node in an inorder traversal. At the root, we turn left with the correct probability, and by induction, it is obvious that we keep on doing so when we travel to a leaf. Of course, we have quite a few possibilities where the shape of the tree is concerned. We could make a complete tree, i.e. a tree where all levels are full except perhaps the lowest level (which is filled from left to right). Complete trees with $2K+1$ nodes have

$$L = 1+ \lfloor \log_2(2K+1) \rfloor$$

levels, and thus the search takes at most L comparisons. In linear search, the worst case is always $\Omega(K)$, whereas now we have $L \sim \log_2 K$. The data structure that can be used for the inversion is as follows: define an array of $2K+1$ records. The last $K+1$ records correspond to the leaves (record $K+i$ corresponds to integer $i-1$). The first K records are internal nodes. The j-th record has as children records $2j$ and $2j+1$, and as father $\lfloor \frac{j}{2} \rfloor$. Thus, the root of the tree is record 1, its children are records 2 and 3, etcetera. This gives us a complete binary tree structure. We need only store one value in each record, and this can be done for the entire tree in time $O(K)$ by noting that we need only do an inorder traversal and keep track of the cumulative probability of the leaves visited when a node is encountered. Using a stack traversal, and notation similar to that of Aho, Hopcroft and Ullman (1982), we can do it as follows:

Set-up of the binary search tree

(BST[1] ,..., BST[2K+1] is our array of values. To save space, we can store the probabilities $p_0, \ldots, p_K$ in BST[K+1] ,..., BST[2K+1].)

(S is an auxiliary stack of integers.)

MAKENULL(S) (create an empty stack).

Ptr←1, PUSH(Ptr,S) (start at the root).

$P \leftarrow 0$ (set cumulative probability to zero).

```
REPEAT
    IF Ptr≤K
        THEN PUSH(Ptr,S), Ptr←2 Ptr
        ELSE
            P←P + BST[Ptr]
            Ptr←TOP(S), POP(S)
            BST[Ptr] ←P
            Ptr←2 Ptr+1
UNTIL EMPTY (S)
```

The binary search tree method described above is not optimal with respect to the expected number of comparisons required to reach a decision. For a fixed binary search tree, this number is equal to $\sum_{i=0}^{K} p_i D_i$ where D_i is the depth of the i-th leaf (the depth of the root is one, and the depth of a node is the number of nodes encountered on the path from that node to the root). A binary search tree is optimal when the expected number of comparisons is minimal. We now define **Huffman's tree** (Huffman, 1952, Zimmerman, 1959), and show that it is optimal.

The two smallest probability leaves should be furthest away from the root, for if they are not, then we can always swap one or both of them with other nodes at a deeper level, and obtain a smaller value for $\sum p_i D_i$. Because internal nodes have two children, we can always make these leaves children of the same internal node. But if the indices of these nodes are j and k, then we have

$$\sum_{i=0}^{K} p_i D_i = \sum_{i \neq j,k} p_i D_i + (p_j + p_k) D* + (p_j + p_k) .$$

Here $D*$ is the depth of the internal father node. We see that minimizing the right-hand-side of this expression reduces to a problem with K instead of $K+1$ nodes, one of these nodes being the new internal node with probability $p_j + p_k$ associated with it. Thus, we can now construct the entire (Huffman) tree. Perhaps a small example is informative here.

Example 2.5.
Consider the probabilities

p_0	0.11
p_1	0.30
p_2	0.25
p_3	0.21
p_4	0.13

We note that we should join nodes 0 and 4 first and form an internal node of cumulative weight 0.24. Then, this node and node 3 should be joined into a supernode of weight 0.45. Next, nodes 1 and 2 are made children of the same internal node of weight 0.55, and the two leftover internal nodes finally become children of the root. ■

For a data structure, we can no longer use a complete binary tree, but we can make use of the array implementation in which entries 1 through K denote internal nodes, and entries $K+1$ through $2K+1$ define leaves. For leaves, the entries are the given probabilities, and for the internal nodes, they are the threshold values as defined for general binary search trees. Since the shape of the tree must also be determined, we are forced to add for entries 1 through K two fields, a leftchildpointer and a rightchildpointer. For the sake of simplicity, we use BST[.] for the threshold values and probabilities, and Left[.], Right[.] for the pointer fields. The tree can be constructed in time $O(K \log K)$ by the Hu-Tucker algorithm (Hu,Tucker, 1971):

Construction of the Huffman tree

Create a heap H with elements $(K+1,p_0), \ldots, (2K+1,p_K)$ and order defined by the keys p_i (the smallest key is at the top of the heap). (For the definition of a heap, we refer to Aho, Hopcroft and Ullman (1982)). Note that this operation can be done in $O(K)$ time.

FOR i:=1 TO K DO

Take top element (j,p) off the heap H and fix the heap.

Take top element (k,q) off the heap H and fix the heap.

Left[i]$\leftarrow j$, Right[i]$\leftarrow k$.

Insert $(i,p+q)$ in the heap H.

Compute the array BST by an inorder traversal of the tree. (This is analogous to the traversal seen earlier, except that for travel down the tree, we must make use of the fields Left[.] and Right[.] instead of the positional trick that in a complete binary tree the index of the leftchild is twice that of the father. The time taken by this portion is $O(K)$.)

The entire set-up takes time $O(K \log K)$ in view of the fact that insertion and deletion-off-the-top are $O(\log K)$ operations for heaps.

It is worth pointing out that for families of discrete distributions, the extra cost of setting up a binary search tree is often inacceptable.

We close this section by showing that for most distributions the expected number of comparisons ($E(C)$) with the Huffman binary search tree is much less than with the complete binary search tree. To understand why this is possible, consider for example the simple distribution with probability vector $\frac{1}{2},\frac{1}{4}, \ldots, \frac{1}{2^K},\frac{1}{2^K}$. It is trivial to see that the Huffman tree here has a linear shape: we can define it recursively by putting the largest probability in the right child of the root, and putting the Huffman tree for the leftover probabilities in the left subtree of the root. Clearly, the expected number of comparisons is $(\frac{1}{2})2+(\frac{1}{4})3+(\frac{1}{8})4+ \cdots$. For any K, this is less than 3, and as $K \to \infty$, the value 3 is approached. In fact, this finite bound also applies to the extended Huffman tree for the probability vector $\frac{1}{2^i}$ $(i \geq 1)$. Similar asymmetric trees are obtained for all distributions for which $E(e^{tX})<\infty$ for some $t>0$: these are distributions with roughly speaking exponentially or subexponentially decreasing tail probabilities. The relationship between the tail of the distribution and $E(C)$ is clarified in Theorem 2.1.

Theorem 2.1.

Let $p_1,p_2,\ldots$ be an arbitrary probability vector. Then it is possible to construct a binary search tree (including the Huffman tree) for which

$$E(C) \le 1+4\left\lceil \log_2(1+E(X))\right\rceil ,$$

where X is the discrete random variate generated by using the binary search tree for inversion.

Proof of Theorem 2.1.

The tree that will be considered here is as follows: choose first an integer $k \ge 1$. We put leaves at levels $k+1, 2k+1, 3k+1, \ldots$ only. At level $k+1$, we have 2^k slots, and all but one is filled from left to right. The extra slot is used as a root for a similar tree with 2^k-1 leaves at level $2k+1$. Thus, C is equal to:

$$k+1 \quad \text{with probability } \sum_{i=1}^{2^k-1} p_i$$

$$2k+1 \quad \text{with probability } \sum_{i=2^k}^{2(^k-1)} p_i \ .$$

$$\cdots$$

Taking expected values gives

$$E(C) = 1+k\sum_{j=1}^{\infty} j \sum_{i=(j-1)(2^k-1)+1}^{j(2^k-1)} p_i$$

$$= 1+k\sum_{i=1}^{\infty} p_i \sum_{\frac{i}{2^k-1} \le j \le 1+\frac{i-1}{2^k-1}} j$$

$$\le 1+k\sum_{i=1}^{\infty} p_i\,(1+\frac{i-1}{2^k-1})(2-\frac{1}{2^k-1})$$

$$\le 1+2k\sum_{i=1}^{\infty} p_i\ (1+\frac{i}{2^k-1})$$

$$\le 1+2k+\frac{2k}{2^k-1}\sum_{i=1}^{\infty} ip_i$$

$$= 1+2k+\frac{2k}{2^k-1}\,E(X)\ .$$

If we take $k=\left\lceil \log_2(1+E(X))\right\rceil$, then $2^k-1 \ge E(X)$, and thus,

$$E(C) \le 1+2\left\lceil \log_2(1+E(X))\right\rceil(1+\frac{E(X)}{E(X)}) = 1+4\left\lceil \log_2(1+E(X))\right\rceil .$$

This concludes the proof of Theorem 2.1. ■

We have shown two things in this theorem. First, of all, we have exhibited a particular binary search tree with design constant $k \geq 1$ (k is an integer) for which

$$E(C) \leq 1+2k+\frac{2k}{2^k-1} E(X) .$$

Next, we have shown that the value of $E(C)$ for the Huffman tree does not exceed the upper bound given in the statement of the theorem by manipulating the value of k and noting that the Huffman tree is optimal. Whether in practice we can use the construction successfully depends upon whether we have a fair idea of the value of $E(X)$, because the optimal k depends upon this value. The upper bound of the theorem grows logarithmically in $E(X)$. In contrast, the expected number of comparisons for inversion by sequential search grows linearly with $E(X)$. It goes without saying that if the p_i's are not in decreasing order, then we can permute them to order them. If in the construction we fill empty slots by borrowing from the ordered vector $p_{(1)},p_{(2)},\ldots$, then the inequality remains valid if we replace $E(X)$ by $\sum_{i=1}^{\infty} i p_{(i)}$. We should also note that Theorem 2.1 is useless for distributions with $E(X)=\infty$. In those situations, there are other possible constructions. The binary tree that we construct has once again leaves at levels $k+1,2k+1,\ldots$, but now, we define the leaf positions as follows: at level $k+1$, put one leaf, and define 2^k-1 roots of subtrees, and recurse. This means that at level $2k+1$ we find 2^k-1 leaves. We associate the p_i's with leaves in the order that they are encountered in this construction, and we keep on going until K leaves are accommodated.

Theorem 2.2.

For the binary search tree constructed above with fixed design constant $k \geq 1$, we have

$$E(C) \leq 1+kp_1+\frac{2k}{\log(2^k-1)} E(\log X)$$

and, for $k=2$,

$$E(C) \leq 1+2p_1+\frac{4}{\log 3} E(\log X) \leq 3+\frac{4}{\log 3} E(\log X) ,$$

where X is a random variate with the probability vector $p_1, \ldots, p_K$ that is used in the construction of the binary search tree, and C is the number of comparisons in the inversion method.

Proof of Theorem 2.2.

Let us define $m=2^k-1$ to simplify the notation. It is clear that

$$C=\begin{cases} k+1 & \text{with probability } p_1 \\ 2k+1 & \text{with probability } p_2+\cdots+p_{m+1} \\ 3k+1 & \text{with probability } p_{m+2}+\cdots+p_{m^2+m+1} \\ \cdots & \end{cases} .$$

In such expressions, we assume that $p_i=0$ for $i>K$. The construction also works for infinite-tailed distributions, so that we do not need K any further. Now,

$$E(C)\le 1+kp_1+k\sum_{j=2}^{\infty} j \sum_{i=1+1+\cdots+m^{j-2}}^{1+\cdots+m^{j-1}} p_i$$

$$=1+kp_1+k\sum_{i=2}^{\infty} p_i \sum_{1+1+\cdots+m^{j-2}\le i\le 1+\cdots+m^{j-1}} j$$

$$\le 1+kp_1+k\sum_{i=2}^{\infty} p_i \sum_{m^{j-2}<i\le m^j} j$$

$$\le 1+kp_1+k\sum_{i=2}^{\infty}(2\frac{\log i}{\log m})$$

$$=1+kp_1+\frac{2k}{\log m}\sum_{i=2}^{\infty} p_i \log i$$

$$=1+kp_1+\frac{2k}{\log m}E(\log X) .$$

This proves the first inequality of the theorem. The remainder follows without work. ■

The bounds of Theorem 2.2 grow as $E(\log X)$ and not as $\log(E(X))$. The difference is that $E(\log X)\le\log(E(X))$ (by Jensen's inequality), and that for long-tailed distributions, the former expression can be finite while the second expression is ∞.

2.4. The method of guide tables.

We have seen that inversion can be based upon sequential search, ordinary binary search or modified binary search. All these techniques are comparison-based. Computer scientists have known for a long time that hashing methods are ultra fast for searching data structures provided that the elements are evenly distributed over the range of values of interest. This speed is bought by the exploitation of the truncation operation.

Chen and Asau (1974) first suggested the use of hashing techniques to handle the inversion. To insure a good expected time, they introduced an ingenious trick, which we shall describe here. Their method has come to be known as the **method of guide tables.** Again, we have a monotone relationship between X, the generated random variate, and U, the uniform [0,1] random variate which is inverted.

We assume that a probability vector $p_0, p_1, \ldots, p_K$ is given. The cumulative probabilities are defined as

$$q_i = \sum_{j=0}^{i} p_j \qquad (0 \le i \le K) .$$

If we were to throw a dart (in this case U) at the segment [0,1], which is partitioned into $K+1$ intervals $[0,q_0),[q_0,q_1), \ldots, [q_{K-1},1]$, then it would come to rest in the interval $[q_{i-1},q_i)$ with probability $q_i-q_{i-1}=p_i$. This is another way of rephrasing the inversion principle of course. It is another matter to find the interval to which U belongs. This can be done by standard binary search in the array of q_i's (this corresponds roughly to the complete binary search tree algorithm). If we are to exploit truncation however, then we somehow have to consider equi-spaced intervals, such as $[\frac{i-1}{K+1},\frac{i}{K+1})$, $1 \le i \le K+1$. The method of guide tables helps the search by storing in each of the $K+1$ intervals a "guide table value" g_i where

$$g_i = \max_{q_j < \frac{i}{K+1}} j .$$

This helps the inversion tremendously:

Method of guide tables

Generate a uniform [0,1] random variate U.
Set $X \leftarrow \lfloor (K+1)U+1 \rfloor$ (this is the truncation).
Set $X \leftarrow g_X+1$ (guide table look-up).
WHILE $q_{X-1} > U$ DO $X \leftarrow X-1$.
RETURN X

It is easy to determine the validity of this algorithm. Note also that no expensive computations are involved.

Theorem 2.3.

The expected number of comparisons (of q_{X-1} and U) in the method of guide tables is always bounded from above by 2.

Proof of Theorem 2.3.

Observe that the number of comparisons C is not greater than the number of q_i values in the interval X (the returned random variate) plus one. But since all intervals are equi-spaced, we have

$$E(C) \leq 1+\frac{1}{K+1}\sum_{i=0}^{K}(\text{number of values of } q_j \text{ in interval } i)$$

$$\leq 1+1 = 2 . \blacksquare$$

Theorem 2.3 is very important because it guarantees a uniformly good performance for all distributions as long as we make sure that the number of intervals and the number of possible values of the discrete random variable are equal.

This inversion method too requires a set-up step. The table of values $g_1 \cdot g_2, \ldots, g_{K+1}$ can be found in time $O(K)$:

Set-up of guide table

FOR i :=1 TO $K+1$ DO $g_i \leftarrow 0$.
$S \leftarrow 0$.
FOR i := 0 TO K DO
    $S \leftarrow S+p_i$ (S is now q_i).
    $j \leftarrow \lfloor S(K+1)+1 \rfloor$. (Determine interval for q_i.)
    $g_j \leftarrow i$.
FOR i :=2 TO $K+1$ DO $g_i \leftarrow \max(g_{i-1}, g_i)$.

There is a trade-off between expected number of comparisons and the size of the guide table. It is easy to see that if we have a guide table of $\alpha(K+1)$ elements for some $\alpha>0$, then we have

$$E(C) \leq 1+\frac{1}{\alpha} .$$

If speed is extremely important, one should not hesitate to set α equal to 5 or 10. Of all the inversion methods discussed so far, the method of guide tables shows clearly the greatest potential in terms of speed. This is confirmed in Ahrens and Kohrt(1981).

2.5. Inversion by correction.

It is sometimes possible to find another distribution function G that is close to the distribution function F of the random variable X. Here G is the distribution function of another discrete random variable, Y. It is assumed that G is an easy distribution. In that case, it is possible to generate X by first generating Y and then applying a small correction. It should be stressed that the fact that G is close to F does not imply that the probabilities $G(i)-G(i-1)$ are close to the probabilities $F(i)-F(i-1)$. Thus, other methods that are based upon the closeness of these probabilities, such as the rejection method, are not necessarily applicable. We are simply using G to obtain an initial estimate of X.

Inversion by correction; direct version

Generate a uniform [0,1] random variate U.

Set $X \leftarrow G^{-1}(U)$ (i.e. X is an integer such that $G(X-1)<U \leq G(X)$. This usually means that X is obtained by truncation of a continuous random variable.)

IF $U \leq F(X)$

THEN WHILE $U \leq F(X-1)$ DO $X \leftarrow X-1$.

ELSE WHILE $U > F(X+1)$ DO $X \leftarrow X+1$.

RETURN X

We can measure the time taken by this algorithm in terms of the number of F-computations. We have:

Theorem 2.4.

The number of computations C of F in the inversion algorithm shown above is

$$2+|Y-X|$$

where X,Y are defined by

$$F(X-1)<U \leq F(X) \ , \ G(Y-1)<U \leq G(Y) \ .$$

It is clear that $E(C)=2+E(|Y-X|)$ where Y,X are as defined in the theorem. Note that Y and X are dependent random variables in this definition. We observe that in the algorithm, we use inversion by sequential search and start this search from the initial guess Y. The correction is $|Y-X|$.

There is one important special case, occurring when F and G are stochastically ordered, for example, when $F \leq G$. Then one computation of F can be saved by noting that we can use the following implementation.

Inversion by correction; $F \le G$

Generate a uniform [0,1] random variate U. Set $X \leftarrow G^{-1}(U)$.
WHILE $U > F(X)$ DO $X \leftarrow X+1$.
RETURN X

What is saved here is the comparison needed to decide whether we should search up or down. Since in the notation of Theorem 2.4, $Y \le X$, we see that

$$E(C) = 1+E(X-Y) .$$

When $E(X)$ and $E(Y)$ are finite, this can be written as $1+E(X)-E(Y)$. In any case, we have

$$E(C) = 1+\sum_i | F(i)-G(i) | .$$

To see this, use the fact that $E(X)=\sum_i(1-F(i))$ and $E(Y)=\sum_i(1-G(i))$. When $F \ge G$, we have a symmetric development of course.

In some cases, a random variate with distribution function G can more easily be obtained by methods other than inversion. Because we still need a uniform [0,1] random variate, it is necessary to cook up such a random variate from the previous one. Thus, the initial pair of random variates (U,X) can be generated indirectly:

Inversion by correction; indirect version

Generate a random variate X with distribution function G.
Generate an independent uniform [0,1] random variate V, and set $U \leftarrow G(X-1)+V(G(X)-G(X-1))$.
IF $U \le F(X)$
THEN WHILE $U \le F(X-1)$ DO $X \leftarrow X-1$.
ELSE WHILE $U > F(X+1)$ DO $X \leftarrow X+1$.
RETURN X

It is easy to verify that the direct and indirect versions are equivalent because the joint distributions of the starting pair (U,X) are identical. Note that in both cases, we have the same monotone relation between the generated X and the random variate U, even though in the indirect version, an auxiliary uniform [0.1] random variate V is needed.

Example 2.6.

Consider

$$F(i) = 1-\frac{1+a}{i^p+ai} \quad (i \geq 1),$$

where $a > 0$ and $p > 1$ are given constants. Explicit inversion of F is not feasible except perhaps in special cases such as $p=2$ or $p=3$. If sequential search is used started at 0, then the expected number of F computations is

$$1+\sum_{i=1}^{\infty}(1-F(i)) = 1+\sum_{i=1}^{\infty}\frac{1+a}{i^p+ai} \geq 1+\sum_{i=1}^{\infty}\frac{1}{i^p} .$$

Assume next that we use inversion by correction, and that as easy distribution function we take $G(i)=1-\frac{1}{i^p}$, $i \geq 1$. First, we have stochastic ordering because $F \leq G$. Note first that $G^{-1}(U)$ (the inverse being defined as in Theorem 2.4) is equal to $\left\lfloor 1+U^{-\frac{1}{p}} \right\rfloor$. Furthermore, the expected number of computations of F is

$$1+\sum_{i=1}^{\infty} G(i)-F(i) = 1+\sum_{i=2}^{\infty}\frac{ai^p-ai}{i^p(i^p+ai)} \leq 1+\sum_{i=2}^{\infty}\frac{a}{i^p} .$$

Thus, the improvement in terms of expected number of computations of F is at least $1+(1-a)\sum_{i=2}^{\infty}\frac{1}{i^p}$, and this can be considerable when a is small. ■

2.6. Exercises.

1. Give a one-line generator (based upon inversion via truncation of a continuous random variate) for generating a random variate X with distribution

$$P(X=i) = \frac{i}{\frac{n(n+1)}{2}} \quad (1 \leq i \leq n) .$$

2. By empirical measurement, the following discrete cumulative distribution function was obtained by Nigel Horspool when studying operating systems:

$$F(i) = \min(1 , 0.114 \log(1+\frac{i}{0.731})-0.069) \quad (i \geq 1) .$$

Give a one-line generator for this distribution which uses truncation of a continuous random variate.

3. Give one-line generators based upon inversion by truncation of a continuous random variate for the following probability distributions on the positive integers:

p_n
$\frac{a}{(n+a)(n+a+1)}$ $(a>0)$
$\frac{1}{2^n}$
$\frac{3n^2+3n+1}{n^3(n+1)^3}$
$\frac{2n+1}{n^2(n+1)^2}$
$\frac{1}{\sqrt{n(n+1)}(\sqrt{n}+\sqrt{n+1})}$

3. TABLE LOOK-UP METHODS.

3.1. The table look-up principle.

We can generate a random variate X very quickly if all probabilities p_i are rational numbers with common denominator M. It suffices to note that the sum of the numerators is also M. Thus, if we were to construct an array A of size M with Mp_0 entries 0, Mp_1 entries 1, and so forth, then a uniformly picked element of this array would yield a random variate with the given probability vector $p_0, p_1, \ldots$. Formally we have:

Table look-up method

[SET-UP]

Given the probability vector $(p_0=\frac{k_0}{M}, p_1=\frac{k_1}{M}, \ldots)$, where the k_i's and M are nonnegative integers, we define a table $A=(A[0], \ldots, A[M-1])$ where k_i entries are i, $i \geq 0$.

[GENERATOR]

Generate a uniform [0,1] random variate U.

RETURN $A[\lfloor MU \rfloor]$

The beauty of this technique is that it takes a constant time. Its disadvantages include its limitation (probabilities are rarely rational numbers) and its large table size (M can be phenomenally big).

We will give two important examples to illustrate its use.

Example 3.1. Simulating dice.

We are asked to generate the sum of n independently thrown unbiased dice. This can be done naively by using $X_1+X_2+ \cdots +X_n$ where the X_i's are iid uniform $\{1,2, \ldots, 6\}$ random variates. The time for this algorithm grows as n. Usually, n will be small, so that this is not a major drawback. We could also proceed as follows: first we set up a table $A[0], \ldots, A[M-1]$ of size $M=6^n$ where each entry corresponds to one of the 6^n possible outcomes of the n throws (for example, the first entry corresponds to 1,1,1,1, . . . , 1, the second entry to 2,1,1,1, . . . , 1, etcetera). The entries themselves are the sums. Then $A[\lfloor MU \rfloor]$ has the correct distribution when U is a uniform [0,1] random variate. Note that the time is $O(1)$, but that the space requirements now grow exponentially in n. Interestingly, we have one uniform random variate per random variate that is generated. And if we wish to implement the inversion method, the only thing that we need to do is to sort the array according to increasing values. We have thus bought time and paid with space. It should be noted though that in this case the space requirements are so outrageous that we are practically limited to $n \leq 5$. Also, the set-up is only admissible if very many iid sums are needed in the simulation. ■

Example 3.2. The histogram method.

Statisticians often construct histograms by counting frequencies of events of a certain type. Let events 0,1, . . . , K have associated with them frequencies $k_0, k_1, \ldots, k_K$. A question sometimes asked is to generate a new event with the probabilities defined by the histogram, i.e. the probability of event i should be $\frac{k_i}{M}$ where $M=\sum_{i=0}^{K} k_i$. In this case, we are usually given the original events in table form $A[0], \ldots, A[M-1]$, and it is obvious that the table method can be applied here without set-up. We will refer to this special case as the **histogram method.** Note that for Example 3.1, we could also construct a histogram, but it differs in that a table must be set up. ■

Assume next that we wish to generate the number of heads in n perfect coin tosses. It is known that this number is binomially distributed with parameters n and $\frac{1}{2}$. By the method of Example 3.1, we can use a table look-up method with table of size 2^n, so for $n \leq 10$, this is entirely reasonable. Unfortunately, when the coin is not perfect and the probability of heads is an irrational number p, the table look-up method cannot be used.

3.2. Multiple table look-ups.

The table look-up method has a geometric interpretation. When the table size is M, then we can think of the algorithm in terms of the selection of one of M equi-spaced intervals of [0,1] by finding the interval to which a uniform [0,1] random variate U belongs. Each interval has an integer associated with it, which should be returned.

One of the problems highlighted in the previous section is the table size. One should also recognize that there normally are many identical table entries. These duplicates can be grouped together to reduce the table size. Assume for example that there are k_i entries with value i where $i \geq 0$ and $\sum_{i \geq 0} k_i = M$. Then, if $M = M_0 M_1$ for two integers M_0, M_1, we can set up an auxiliary table $B[0], \ldots, B[M_0-1]$ where each $B[i]$ points to a block of M_1 entries in the true table $A[0], \ldots, A[M-1]$. If this block is such that all values are identical, then it is not necessary to store the block. If we think geometrically again, then this corresponds to defining a partition of [0,1] into M_0 intervals. The original partition of M intervals is finer, and the boundaries are aligned because M is a multiple of M_0. If for the i-th big interval, all M_1 values of $A[j]$ are identical, then we can store that value directly in $B[i]$ thereby saving M_1-1 entries in the A table. By rearranging the A table, it should be possible to repeat this for many large intervals. For the few large intervals covering small intervals with non-identical values for A, we do store a placeholder such as $*$. In this manner, we have built a three-level tree. The root has M_0 children with values $B[i]$. When $B[i]$ is an integer, then i is a terminal node. When $B[i]=*$, we have an internal node. Internal nodes have in turn M_1 children, each carrying a value $A[j]$. It is obvious that this process can be extended to any number of levels. This structure is known as a trie (Fredkin, 1960) or an extendible hash structure (Fagin, Nievergelt, Pippenger and Strong, 1979). If all internal nodes have precisely two children, then we obtain in effect the binary search tree structure of section III.2. Since we want to get as much as possible from the truncation operation, it is obvious that the fan-out should be larger than 2 in all cases.

Consider for example a table for look-up with 1000 entries defined for the following probability vector:

Probability		Number of entries in table A
p_1	0.005	5
p_2	0.123	123
p_3	0.240	240
p_4	0.355	355
p_5	0.277	277

Suppose now that we set up an auxiliary table B which will allow us to refer to sections of size 100 in the table A. Here we could set

B[0]	2
B[1]	3
B[2]	3
B[3]	4
B[4]	4
B[5]	4
B[6]	5
B[7]	5
B[8]	*
B[9]	*

The interpretation is that if $B[i]=j$ then j appears 100 times in table A, and if $B[i]=*$ then we must consult a block of 100 entries of A which are not all identical. Thus, if $B[8]$ or $B[9]$ are chosen, then we need to consult $A[800], \ldots, A[999]$, where we make sure that there are 5 "1"'s, 23 "2"'s, 40 "3"'s, 55 "4"'s and 77 "5"'s. Note however that we need no longer store $A[0], \ldots, A[799]$! Thus, our space requirements are reduced from 1000 words to 210 words.

After having set-up the tables $B[0], \ldots, B[9]$ and $A[800], \ldots, A[999]$, we can generate X as follows:

Example of a multiple table look-up

Generate a uniform [0,1] random variate U.
Set $X \leftarrow B[\lfloor 10U \rfloor]$.
IF $X \neq *$
    THEN RETURN X
    ELSE RETURN $A[\lfloor 1000U \rfloor]$

Here we have exploited the fact that the same U can be reused for obtaining a random entry from the table A. Notice also that in 80% of the cases, we need not access A at all. Thus, the auxiliary table does not cost us too much timewise. Finally, observe that the condition $X \neq *$ can be replaced by $X > 7$, and that therefore $B[8]$ and $B[9]$ need not be stored.

What we have described here forms the essence of **Marsaglia's table look-up method** (Marsaglia, 1963; see also Norman and Cannon, 1972). We can of course do a lot of fine-tuning. For example, the table $A[800], \ldots, A[999]$ can in turn be replaced by an auxiliary table C grouping now only 10 entries, which could be picked as follows:

C[80]	2
C[81]	2
C[82]	3
C[83]	3
C[84]	3
C[85]	3
C[86]	4
C[87]	4
C[88]	4
C[89]	4
C[90]	4
C[91]	5
C[92]	5
C[93]	5
C[94]	5
C[95]	5
C[96]	5
C[97]	5
C[98]	*
C[99]	*

Given that $B[i]=*$ for our value of U, we can in 90% of the cases return $C[\lfloor 100U \rfloor]$. Only if once more an entry $*$ is seen do we have to access the table $A[980], \ldots, A[999]$ at position $\lfloor 1000U \rfloor$. The numbering in our arrays is convenient for accessing elements for our representation, i.e. $B[i]$ stands for $C[10i], \ldots, C[10i+9]$, or for $A[100i], \ldots, A[100i+99]$. Some high level languages do not permit the use of subranges of the integers as indices. It is also convenient to combine A, B and C into one big array. All of this requires additional work during the set-up stage.

We observe that in the multilevel table look-up we must group identical entries in the original table, and this forces us to introduce a nonmonotone relationship between U and X.

The method described here can be extended towards the case where all p_i's are multiples of either 10^{-7} or 2^{-32}. In these cases, the p_i's are usually approximations of real numbers truncated by the wordsize of the computer.

4. THE ALIAS METHOD.

4.1. Definition.

Walker (1974, 1977) proposed an ingenious method for generating a random variate X with probability vector $p_0, p_1, \ldots, p_{K-1}$ which requires a table of size $O(K)$ and has a worst-case time that is independent of the probability vector and K. His method is based upon the following property:

Theorem 4.1.

Every probability vector $p_0, p_1, \ldots, p_{K-1}$ can be expressed as an equiprobable mixture of K two-point distributions.

Proof of Theorem 4.1.

We have to show that there are K pairs of integers $(i_0, j_0), \ldots, (i_{K-1}, j_{K-1})$ and K probabilities $q_0, \ldots, q_{K-1}$ such that

$$p_i = \frac{1}{K} \sum_{l=0}^{K-1} (q_l I_{[i_l=i]} + (1-q_l) I_{[j_l=i]}) \qquad (0 \le i < K) .$$

This can be shown by induction. It is obviously true when $K=1$. Assuming that it is true for $K<n$, we can show that it is true for $K=n$ as follows. Choose the minimal p_i. Since it is at most equal to $\frac{1}{K}$, we can take i_0 equal to the index of this minimum, and set q_0 equal to Kp_{i_0}. Then choose the index j_0 which corresponds to the largest p_i. This defines our first pair in the equiprobable mixture. Note that we used the fact that $\frac{(1-q_0)}{K} \le p_{j_0}$ because $\frac{1}{K} \le p_{j_0}$. The other $K-1$ pairs in the equiprobable mixture have to be constructed from the leftover probabilities

$$p_0, \ldots, p_{i_0}-p_{i_0}, \ldots, p_{j_0}-\frac{1}{K}(1-q_0), \ldots, p_{K-1}$$

which, after deletion of the i_0-th entry, is easily seen to be a vector of $K-1$ nonnegative numbers summing to $\frac{K-1}{K}$. But for such a vector, an equiprobable mixture of $K-1$ two-point distributions can be found by our induction hypothesis. ■

To turn this theorem into profit, we have two tasks ahead of us: first we need to actually construct the equiprobable mixture (this is a set-up problem), and then we need to generate a random variate X. The latter problem is easy to solve. Theorem 4.1 tells us that it suffices to throw a dart at the unit square in the plane and to read off the index of the region in which the dart has landed.

The unit square is of course partitioned into regions by cutting the x-axis up into K equi-spaced intervals which define slabs in the plane. These slabs are then cut into two pieces by the threshold values q_l. If

$$p_i = \frac{1}{K}\sum_{l=0}^{K-1}(q_l I_{[i_l=i]} + (1-q_l)I_{[j_l=i]}) \qquad (0 \le i < K),$$

then we can proceed as follows:

The alias method

Generate a uniform [0,1] random variate U. Set $X \leftarrow \lfloor KU \rfloor$. Generate a uniform [0,1] random variate V.

IF $V < q_X$

THEN RETURN i_X

ELSE RETURN j_X

Here one uniform random variate is used to select one component in the equiprobable mixture, and one uniform random variate is used to decide which part in the two-point distribution should be selected. This unsophisticated version of the alias method thus requires precisely two uniform random variates and two table look-ups per random variate generated. Also, three tables of size K are needed.

We observe that one uniform random variate can be saved by noting that for a uniform [0,1] random variable U, the random variables $X = \lfloor KU \rfloor$ and $V = KU - X$ are independent: X is uniformly distributed on $0, \ldots, K-1$, and the latter is again uniform [0,1]. This trick is not recommended for large K because it relies on the randomness of the lower-order digits of the uniform random number generator. With our idealized model of course, this does not matter.

One of the arrays of size K can be saved too by noting that we can always insure that $i_0, \ldots, i_{K-1}$ is a permutation of $0, \ldots, K-1$. This is one of the duties of the set-up algorithm of course. If a set-up gives us such a permuted table of i-values, then it should be noted that we can in time $O(K)$ reorder the structure such that $i_l = l$, for all l. The set-up algorithm given below will directly compute the tables j and q in time $O(K)$ and is due to Kronmal and Peterson (1979, 1980):

The alias method can further be improved by minimizing this expression, but this won't be pursued any further here. The main reason for not doing so is that there exists a simple generalization of the alias method, called the alias-urn method, which is designed to reduce the expected number of table accesses. Because of its importance, we will describe it in a separate section.

4.2. The alias-urn method.

Peterson and Kronmal (1982) suggested a generalization of the alias method in the following manner: think of the probability vector $p_0, p_1, \ldots, p_{K-1}$ as a special case of a probability vector with $K* \geq K$ components where $p_i = 0$ for all $i \geq K$. Everything that was said in the previous section remains valid for this case. In particular, if we use the linear set-up algorithm for the tables q and j, then it should be noted that $q_l > 0$ for at most K values of l. At least for all $l > K-1$ we must have $q_l = 0$. For these values of l, one table access is necessary:

The alias-urn method

Generate a random integer X uniformly distributed on $0, \ldots, K*-1$.
IF $X \geq K$
  THEN RETURN j_X
  ELSE
    Generate a uniform [0,1] random variate V.
    IF $V \leq q_X$
      THEN RETURN X
      ELSE RETURN j_X

Per random variate, we require either one or two table look-ups. It is easy to see that the expected number of table look-ups (not counting q_X) is

$$\frac{K*-K}{K*} + \frac{1}{K*} \sum_{l=0}^{K-1} (1-q_l) \leq 1 .$$

The upper bound of 1 may somehow seem like magic, but one should remember that instead of one comparison, we now have either one or two comparisons, the expected value being

$$1 + \frac{K}{K*} .$$

Thus, as $K*$ becomes large compared to K, the expected number of comparisons and the expected number of table accesses both tend to one, as for the urn method. In this light, the method can be considered as an urn method with slight

Set-up of tables for alias method

Greater $\leftarrow\emptyset$, Smaller $\leftarrow\emptyset$ (Greater and Smaller are sets of integers.)
FOR l :=0 TO K-1 DO
    $q_l \leftarrow Kp_l$.
    IF $q_l < 1$
        THEN Smaller $\leftarrow$ Smaller $+\{l\}$.
        ELSE Greater $\leftarrow$ Greater $+\{l\}$.
WHILE NOT EMPTY (Smaller) DO
    Choose $k \in$ Greater ,$l \in$ Smaller [q_l is finalized].
    Set $j_l \leftarrow k$ [j_l is finalized].
    $q_k \leftarrow q_k -(1-q_l)$.
    IF $q_k < 1$ THEN Greater $\leftarrow$ Greater - $\{k\}$, Smaller $\leftarrow$ Smaller $+\{k\}$.
    Smaller $\leftarrow$ Smaller $-\{l\}$.

The sets Greater and Smaller can be implemented in many ways. If we can do it in such a way that the operations "grab one element", "is set empty ?", "delete one element" and "add one element" can be done in constant time, then the algorithm given above takes time $O(K)$. This can always be insured if linked lists are used. But since the cardinalities sum to K at all times, we can organize it by using an ordinary array in which the top part is occupied by Smaller and the bottom part by Greater. The alias algorithm based upon the two tables computed above reads:

Alias method with two tables

Generate a random integer X uniformly distributed on 0, . . . , K-1.
Generate a uniform [0,1] random variate V.
IF $V \leq q_X$
    THEN RETURN X
    ELSE RETURN j_X

Thus, per random variate, we have either 1 or 2 table accesses. The expected number of table accesses is

$$1+\frac{1}{K}\sum_{l=0}^{K-1}(1-q_l) .$$

fine-tuning. We are paying for this luxury in terms of space, since we need to store $K*+K$ values: $j_0, \ldots, j_{K*-1}, q_0, \ldots, q_{K-1}$. Finally, note that the comparison $X \geq K$ takes much less time than the comparison $V \leq q_X$.

4.3. Geometrical puzzles.

We have seen the geometrical interpretation of the alias method: throw a dart at random and uniformly on the unit square of R^2 properly partitioned into $2K$ rectangles, and return the index that is associated with the rectangle that is hit. The indices, or aliases, are stored in a table, and so are the definitions of the rectangles. The power of the alias method is due to the fact that we can take K identical slabs of height 1 and base $\frac{1}{K}$ and then split each slab into two rectangles. It should be obvious that there are an unlimited number of ways in which the unit square can be cut up conveniently. In general, if the components are $A_1, \ldots, A_M$, and the aliases are $j_1, \ldots, j_M$, then the algorithm

General alias algorithm

Generate a random variate (X,Y) uniformly distributed in $[0,1]^2$.
Determine the index Z in $1, \ldots, M$ such that $(X,Y) \in A_Z$.
RETURN j_Z

produces a random variate which takes the value k with probability

$$\sum_{l : j_l = k} \text{area}(A_l) .$$

Let us illustrate this with an example. Let the probabilities for consecutive integers 1,2,... be $c, \frac{c}{2}, \frac{c}{2}, \frac{c}{4}, \frac{c}{4}, \frac{c}{4}, \frac{c}{4}, \ldots, \frac{c}{2^n}$, where n is a positive integer, and $c = \frac{1}{n+1}$ is a normalization constant. It is clear that we can group the values in groups of size $1,2,4, \ldots, 2^n$, and the probability weights of the groups are all equal to c. This suggests that we should partition the square first into $n+1$ equal vertical slabs of height 1 and base $\frac{1}{n+1}$. Then, the i-th slab should be further subdivided into 2^i equal rectangles to distinguish between different integers in the groups. The algorithm then becomes:

Generate a random variate X with a uniform distribution on $\{0,1, \ldots, n\}$.
Generate a random variate Y with a uniform distribution on $2^X, \ldots, 2^{X+1}-1$.
RETURN Y.

In this simple example, it is possible to combine the uniform variate generation and membership determination into one. Also, no table is needed.

Consider next the probability vector

$$p_i = \frac{2}{n+1}(1-\frac{i}{n}) \qquad (0 \leq i \leq n) .$$

Now, we can partition the unit square into $n(n+1)$ equal rectangles and assign aliases as in the matrix shown below:

$$\left|\begin{matrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 2 & 2 & 2 \\ 0 & 1 & 2 & 3 & 3 \\ 0 & 1 & 2 & 3 & 4 \\ 0 & 1 & 2 & 3 & 4 \end{matrix}\right| .$$

We can verify first that the probabilities are correct. Then, it is easily seen that the alias method applied here requires no table either. Both examples illustrate the virtually unlimited possibilities of the alias method.

4.4. Exercises.

1. Give a simple linear time algorithm for sorting a table of records $R_1, \ldots, R_n$ if it is known that the vector of key values used for sorting is a permutation of $1, \ldots, n$.
2. Show that there exists a one-line FORTRAN or PASCAL language generator for random variates with probability vector $p_i = \frac{2}{n+1}(1-\frac{i}{n})$, $0 \leq i \leq n$ (Duncan McCallum).
3. Combine the rejection and geometric puzzle method for generating random variates with probability vector $p_i = \frac{c}{i}$, $1 \leq i \leq K$, where c is a normalization constant. The method should take expected time bounded uniformly over K. Hint: note that the vector $c, \frac{c}{2}, \frac{c}{2}, \frac{c}{4}, \frac{c}{4}, \frac{c}{4}, \frac{c}{4}, \ldots$ dominates the

given probability vector.

4. Repeat the previous exercise for the two-parameter class of probability vectors $p_i = \frac{c}{i^M}$, $1 \le i \le K$ where M is a positive integer.

5. OTHER GENERAL PRINCIPLES.

5.1. The rejection method.

The rejection principle remains of course valid for discrete distributions. If the probability vector p_i , $i \ge 0$, is such that

$$p_i \le c q_i \qquad (i \ge 0) ,$$

where $c \ge 1$ is the rejection constant and q_i , $i \ge 0$, is an easy probability vector, then the following algorithm is valid:

The rejection method

REPEAT

Generate a uniform [0,1] random variate U.

GENERATE a random variate X with discrete distribution determined by q_i , $i \ge 0$.

UNTIL $Ucq_X \le p_X$

RETURN X

We recall that the number of iterations is geometrically distributed with parameter $\frac{1}{c}$ (and thus mean c). Also, in each iteration, we need to compute $\frac{p_X}{cq_X}$. In view of the ultra fast methods described in the previous sections for finite-valued random variates, it seems that the rejection method is mainly applicable in one of two situations:

A. The distribution has an infinite tail.

B. The distribution changes frequently (so that we do not have the time to set up long tables every time).

Often, the body of a distribution can be taken care of by the guide table, alias or alias-urn methods, and the tail (which carries small probability anyway) is dealt with by the rejection method.

Example 5.1.

Consider the probability vector

$$p_i = \frac{6}{\pi^2 i^2} \quad (i \geq 1) .$$

Sequential search for this distribution is undesirable because the expected number of comparisons would be $1+\sum_{i=1}^{\infty} i p_i = \infty$. With the easy probability vector

$$q_i = \frac{1}{i(i+1)} \quad (i \geq 1) ,$$

we can apply the rejection method. The best possible rejection constant is

$$c = \sup_{i \geq 1} \frac{p_i}{q_i} = \frac{6}{\pi^2} \sup_{i \geq 1} \frac{i+1}{i} = \frac{12}{\pi^2} .$$

Since $\left\lfloor \frac{1}{U} \right\rfloor$ has probability vector q (where U is a uniform [0,1] random variable), we can proceed as follows:

REPEAT

Generate iid uniform [0,1] random variates U,V. Set $X \leftarrow \left\lfloor \frac{1}{U} \right\rfloor$.

UNTIL $2VX \leq X+1$

RETURN X ■

Example 5.2. Monotone distributions.

When the probability vector $p_1, \ldots, p_n$ is nonincreasing, then it is obvious that $p_i \leq \frac{1}{i}$ for all i. Thus, the following rejection algorithm is valid:

REPEAT

Generate a random variate X with probability vector proportional to $1, \frac{1}{2}, \ldots, \frac{1}{n}$.

Generate a uniform [0,1] random variate U.

UNTIL $U \le X p_X$

RETURN X

The expected number of iterations is $\sum_{i=1}^{n} \frac{1}{i} \le 1+\log(n)$. For example, a binomial (n,p) random variate can be generated by this method in expected time $O(\log(n))$ provided that the probabilities can be computed in time $O(1)$ (this assumes that the logarithm of the factorial can be computed in constant time). For the dominating distribution, see for example exercise III.4.3. ■

Example 5.3. The hybrid rejection method.

As in example 5.1, random variates with the dominating probability vector are usually obtained by truncation of a continuous random variate. Thus, it seems important to discuss very briefly how we can apply a hybrid rejection method based on the following inequality:

$$p_i \le cg(x) \quad (\text{all } x \in [i, i+1),\ i \ge 0) .$$

Here $c \ge 1$ is the rejection constant, and g is an easy density on $[0,\infty)$. Note that p can be extended to a density f in the obvious manner , i.e. $f(x)=p_i$, all $x \in [i, i+1)$. Thus, random variates with probability vector p can be generated as follows:

Hybrid rejection algorithm

REPEAT

Generate a random variate Y with density g. Set $X \leftarrow \lfloor Y \rfloor$.

Generate a uniform [0,1] random variate U.

UNTIL $Ucg(Y) \leq p_X$

RETURN X ■

5.2. The composition and acceptance-complement methods.

It goes without saying that the entire discussion of the composition and acceptance-complement methods for continuous random variates can be repeated for discrete random variates.

5.3. Exercises.

1. Develop a rejection algorithm for the generation of an integer-valued random variate X where

$$P(X=i) = \frac{c}{2i-1} - \frac{c}{2i} \qquad (i=1,2,\ldots)$$

and $c = \frac{1}{2\log 2}$ is a normalization constant. Analyze the efficiency of your algorithm. Note: the series $1-\frac{1}{2}+\frac{1}{3}-\frac{1}{4}+\frac{1}{5}-\cdots$ converges to log2. Therefore, the terms considered in pairs and divided by log2 can be considered as probabilities defining a probability vector.

2. Consider the family of probability vectors $\frac{c(a)}{(a+i)^2}$, $i \geq 1$, where $a \geq 0$ is a parameter and $c(a)>0$ is a normalization constant. Develop the best possible rejection algorithm that is based upon truncation of random variables with distribution function

$$F(x) = 1-\frac{a+1}{a+x} \qquad (x>1).$$

Find the probability of acceptance, and show that it is at least equal to $\frac{a}{a+2}$. Show that the infimum of the probability of acceptance over

$a \in [0,\infty)$ is nonzero.

3. **The discrete normal distribution.** A random variable X has the discrete normal distribution with parameter $\sigma>0$ when

$$P(X=i) = ce^{-\frac{(|i|+\frac{1}{2})^2}{2\sigma^2}} \qquad (i \text{ integer}) .$$

Here $c>0$ is a normalization constant. Show first that

$$c = \frac{1}{\sigma}\left(\frac{1}{\sqrt{2\pi}}+o(1)\right)$$

as $\sigma\to\infty$. Show then that X can be generated by the following rejection algorithm:

REPEAT

Generate a normal random variate Y, and let X be the closest integer to Y, i.e. $X \leftarrow \text{round}(Y)$. Set $Z \leftarrow |X|+\frac{1}{2}$.

Generate a uniform [0,1] random variate U.

UNTIL $-2\sigma^2\log(U) \geq Z^2-Y^2$

RETURN X

Note that $-\log(U)$ can be replaced by an exponential random variate. Show that the probability of rejection does not exceed $\frac{2}{\sigma}\sqrt{\frac{2}{\pi}}$. In other words, the algorithm is very efficient when σ is large.