# 6.2 Generating Discrete Random Variables

In this section we will consider the development of correct, exact algorithms for the generation of discrete random variates. We begin with an important definition, associated theorem, and algorithm.

## 6.2.1 INVERSE DISTRIBUTION FUNCTION

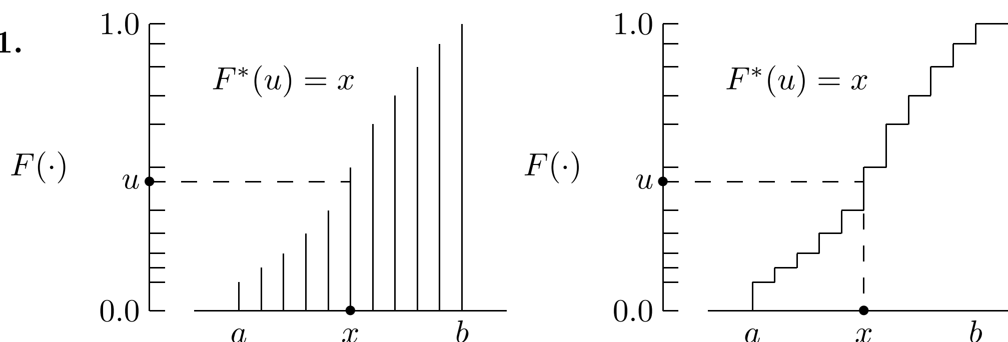
**Definition 6.2.1** Let $X$ be a discrete random variable with cdf $F(\cdot)$. The *inverse distribution function* (*idf*) of $X$ is the function $F^*$: $(0,1) \rightarrow \mathcal{X}$ defined for all $u \in (0,1)$ as

$$F^*(u) = \min_x \{x : u < F(x)\}$$

where the minimum is over all possible values $x \in \mathcal{X}$. That is, if $F^*(u) = x$ then $x$ is the smallest possible value of $X$ for which $F(x)$ is greater than $u$.*

**Example 6.2.1** Figure 6.2.1 provides a graphical illustration of the idf for a discrete random variable with $\mathcal{X} = \{a, a+1, \ldots, b\}$ for two common ways of plotting the same cdf. The value of $x \in \mathcal{X}$ corresponding to a given $u$, in this case $u = 0.45$, is found by extending the dashed line horizontally until it strikes one of the cdf "spikes" in the left-hand cdf or one of the "risers" on the right-hand cdf. The value of $X \in \mathcal{X}$ corresponding to that cdf spike defines the value of $x = F^*(u)$.

**Figure 6.2.1. Generating discrete random variables.**




The following theorem is based on Definition 6.2.1 and the insight provided by the previous example. The significance of the theorem is that it provides an important algorithmic characterization of the idf.

**Theorem 6.2.1** Let $X$ be an integer-valued random variable with $\mathcal{X} = \{a, a+1, \ldots, b\}$ where $b$ may be $\infty$ and let $F(\cdot)$ be the cdf of $X$. For any $u \in (0,1)$, if $u < F(a)$ then $F^*(u) = a$, else $F^*(u) = x$ where $x \in \mathcal{X}$ is the (unique) possible value of $X$ for which $F(x-1) \leq u < F(x)$.

Although Theorem 6.2.1 (and others that follow in this section) are written for *integer-valued* discrete random variables, they are easily extended to the more general case.

---

* The rather unusual definition of the inverse [and the avoidance of the notation $F^{-1}(u)$] is to account for the fact that $F(x)$ is not 1–1. A specific value of the idf is also called a *fractile*, *quantile*, or *percentile* of the distribution of $X$. For example, $F^*(0.95)$ is often called the 95$^{\text{th}}$ percentile of the distribution of $X$.

**Algorithm 6.2.1** Let $X$ be an integer-valued random variable with $\mathcal{X} = \{a, a+1, \ldots, b\}$ where $b$ may be $\infty$ and let $F(\cdot)$ be the cdf of $X$. For any $u \in (0, 1)$ the following linear search algorithm defines the idf $F^*(u)$

```
x = a;
while (F(x) <= u)
   x++;
return x;                                        /* x is F*(u) */
```

Because $u < 1.0$ and $F(x) \to 1.0$ as $x$ increases, the loop in Algorithm 6.2.1 is guaranteed to terminate. Note, however, that the search is linear and begins at the smallest possible value of $X$ so that *average case* efficiency can be a real problem if $\mu = E[X]$ is large relative to $a$. More specifically, in repeated application of this algorithm using values of $u$ generated by calls to `Random`, the average number of passes through the `while` loop will be $\mu - a$. To see this, let $u$ be a *Uniform*$(0, 1)$ random variate and let $X$ be the discrete random variable corresponding to the random variate $x$ generated by Algorithm 6.2.1. If the discrete random variable $Y$ represents the number of `while` loop passes then $Y = X - a$. Therefore, from Definition 6.1.6, $E[Y] = E[X - a] = E[X] - a = \mu - a$.

Algorithm 6.2.1 can be made more efficient by starting the search at a better (more likely) point. The best (most likely) starting point is the mode.* We then have the following more efficient version of Algorithm 6.2.1.

**Algorithm 6.2.2** Let $X$ be an integer-valued random variable with $\mathcal{X} = \{a, a+1, \ldots, b\}$ where $b$ may be $\infty$ and let $F(\cdot)$ be the cdf of $X$. For any $u \in (0, 1)$, the following linear search algorithm defines the idf $F^*(u)$

```
x = mode;                      /* initialize with the mode of X */
if (F(x) <= u)
   while (F(x) <= u)
     x++;
else if (F(a) <= u)
   while (F(x − 1) > u)
     x--;
else
   x = a;
return x;                                        /* x is F*(u) */
```

Although Algorithm 6.2.2 is still a linear search, generally it is more efficient than Algorithm 6.2.1, perhaps significantly so, unless the mode of $X$ is $a$. If $|\mathcal{X}|$ is very large, even more efficiency may be needed. In this case, a *binary* search should be considered. (See Exercise 6.2.9.)

* The mode of $X$ is the value of $x \in \mathcal{X}$ for which $f(x)$ is largest. For many discrete random variables, but *not* all, $\lfloor \mu \rfloor$ is an essentially equivalent choice.

### Idf Examples

In some important cases the idf $F^*(u)$ can be determined explicitly by using Theorem 6.2.1 to solve the equation $F(x) = u$ for $x$. The following three examples are illustrations.

**Example 6.2.2** If $X$ is *Bernoulli*$(p)$ and $F(x) = u$ then $x = 0$ if and only if $0 < u < 1-p$ and $x = 1$ otherwise. Therefore

$$F^*(u) = \begin{cases} 0 & 0 < u < 1-p \\ 1 & 1-p \le u < 1. \end{cases}$$

**Example 6.2.3** If $X$ is *Equilikely*$(a, b)$ then

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

Therefore, provided $u \ge F(a)$,

$$\begin{aligned} F(x-1) \le u < F(x) &\iff \frac{(x-1)-a+1}{b-a+1} \le u < \frac{x-a+1}{b-a+1} \\ &\iff x-a \le (b-a+1)u < x-a+1 \\ &\iff x \le a+(b-a+1)u < x+1. \end{aligned}$$

Thus, for $F(a) \le u < 1$ the idf is

$$F^*(u) = a + \lfloor (b-a+1)u \rfloor.$$

Moreover, if $0 < u < F(a) = 1/(b-a+1)$ then $0 < (b-a+1)u < 1$ so that $F^*(u) = a$, as required. Therefore, the idf equation is valid for all $u \in (0, 1)$.

**Example 6.2.4** If $X$ is *Geometric*$(p)$ then

$$F(x) = 1 - p^{x+1} \qquad x = 0, 1, 2, \ldots$$

Therefore, provided $u \ge F(0)$,

$$\begin{aligned} F(x-1) \le u < F(x) &\iff 1-p^x \le u < 1-p^{x+1} \\ &\iff -p^x \le u-1 < -p^{x+1} \\ &\iff p^x \ge 1-u > p^{x+1} \\ &\iff x\ln(p) \ge \ln(1-u) > (x+1)\ln(p) \\ &\iff x \le \frac{\ln(1-u)}{\ln(p)} < x+1. \end{aligned}$$

Thus, for $F(0) \le u < 1$ the idf is

$$F^*(u) = \left\lfloor \frac{\ln(1-u)}{\ln(p)} \right\rfloor$$

Moreover, if $0 < u < F(0) = 1-p$ then $p < 1-u < 1$ so that $F^*(u) = 0$, as required. Therefore, the idf equation is valid for all $u \in (0, 1)$.

### 6.2.2 RANDOM VARIATE GENERATION BY INVERSION

The following theorem is of fundamental importance in random variate generation applications. Because of the importance of this theorem, a detailed proof is presented. The proof makes use of a definition and two results, listed here for reference.

- Two random variables $X_1$ and $X_2$ with corresponding pdf's $f_1(\cdot)$ and $f_2(\cdot)$ defined on $\mathcal{X}_1$ and $\mathcal{X}_2$ respectively, are *identically distributed* if and only if they have a common set of possible values $\mathcal{X}_1 = \mathcal{X}_2$ and for all $x$ in this common set $f_1(x) = f_2(x)$.
- The idf $F^*\colon (0,1) \to \mathcal{X}$ maps the interval $(0,1)$ *onto* $\mathcal{X}$.
- If $U$ is *Uniform*$(0,1)$ and $0 \le \alpha < \beta \le 1$ then $\Pr(\alpha \le U < \beta) = \beta - \alpha$.*

**Theorem 6.2.2** (Probability integral transformation) If $X$ is a discrete random variable with idf $F^*(\cdot)$ and the continuous random variable $U$ is *Uniform*$(0,1)$ and $Z$ is the discrete random variable defined by $Z = F^*(U)$, then $Z$ and $X$ are identically distributed.

**Proof** Let $\mathcal{X}$, $\mathcal{Z}$ be the set of possible values for $X$, $Z$ respectively. If $x \in \mathcal{X}$ then, because $F^*$ maps $(0,1)$ onto $\mathcal{X}$, there exists $u \in (0,1)$ such that $F^*(u) = x$. From the definition of $Z$ it follows that $x \in \mathcal{Z}$ and so $\mathcal{X} \subseteq \mathcal{Z}$. Similarly, if $z \in \mathcal{Z}$ then from the definition of $Z$ there exists $u \in (0,1)$ such that $F^*(u) = z$. Because $F^*\colon (0,1) \to \mathcal{X}$, it follows that $z \in \mathcal{X}$ and so $\mathcal{Z} \subseteq \mathcal{X}$. This proves that $\mathcal{X} = \mathcal{Z}$. Now, let $\mathcal{X} = \mathcal{Z} = \{a, a+1, \ldots, b\}$ be the common set of possible values. To prove that $X$ and $Z$ are identically distributed we must show that $\Pr(Z = z) = f(z)$ for any $z$ in this set of possible values, where $f(\cdot)$ is the pdf of $X$. From the definition of $Z$ and $F^*(\cdot)$ and Theorem 6.2.1, if $z = a$ then

$$\Pr(Z = a) = \Pr\big(U < F(a)\big) = F(a) = f(a).$$

Moreover, if $z \in \mathcal{Z}$ with $z \ne a$ then

$$\Pr(Z = z) = \Pr\big(F(z-1) \le U < F(z)\big) = F(z) - F(z-1) = f(z)$$

which proves that $Z$ and $X$ have the same pdf and so are identically distributed.

Theorem 6.2.2 is the basis for the following algorithm by which *any* discrete random variable can be generated with just *one* call to `Random`. In Section 7.2 we will see that there is an analogous theorem and algorithm for continuous random variables. This elegant algorithm is known as the *inversion* method for random variate generation.

**Algorithm 6.2.3** If $X$ is a discrete random variable with idf $F^*(\cdot)$ then a corresponding discrete random variate $x$ can be generated as follows

```
u = Random();
return F*(u);
```

* See Example 7.1.2 in the next chapter for more discussion of this result.

### Inversion Examples

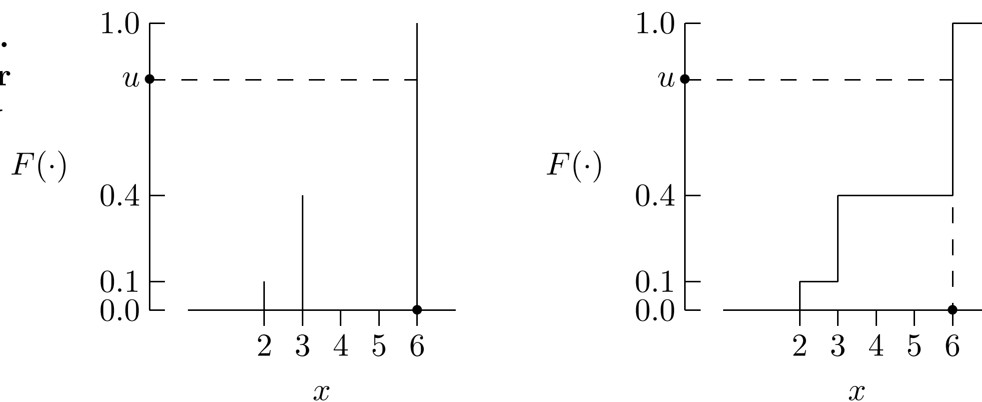
Random variate generation by inversion has a clear intuitive appeal and always works. It is *the* method of choice for discrete random variate generation, provided the idf can be manipulated into a form that is mathematically tractable for algorithmic implementation. We begin with an example for an arbitrary discrete distribution.

**Example 6.2.5** Consider the discrete random variable $X$ with pdf:

$$f(x) = \begin{cases} 0.1 & x = 2 \\ 0.3 & x = 3 \\ 0.6 & x = 6. \end{cases}$$

The cdf for $X$ is plotted using two different formats in Figure 6.2.2.

**Figure 6.2.2. Inversion for an arbitrary discrete random variable.**




For a random number $u$ (which strikes the vertical axes in Figure 6.2.2, for example, at $u = 0.803$), $F^*(u)$ assumes a value of 2, 3, or 6 depending on the intersection point of the associated horizontal line with the cdf. The reader is encouraged to reflect on the geometry in Figure 6.2.2 to appreciate the fact that the algorithm

```
u = Random();
if (u < 0.1)
   return 2;
else if (u < 0.4)
   return 3;
else
   return 6;
```

returns 2 with probability 0.1, returns 3 with probability 0.3, and returns 6 with probability 0.6, which corresponds to the pdf of $X$. The inversion method for generating discrete random variates always carries this intuitive geometric interpretation. Problems arise, however, when $|\mathcal{X}|$ is large or countably infinite. The average execution time for this particular example can be minimized by checking the ranges for $u$ associated with $x = 6$ (the mode) first, then $x = 3$, then $x = 2$. This way, 60% of the invocations of this algorithm require the evaluation of just one condition.

Random variate generation algorithms for popular discrete probability models that include parameters follow.

**Example 6.2.6** This algorithm uses inversion and the idf expression from Example 6.2.2 to generate a *Bernoulli*$(p)$ random variate (see Example 6.1.13)

```
u = Random();
if (u < 1 - p)
   return 0;
else
   return 1;
```

**Example 6.2.7** This algorithm uses inversion and the idf expression from Example 6.2.3 to generate an *Equilikely*$(a, b)$ random variate

```
u = Random();
return a + (long) (u * (b - a + 1));
```

This is the algorithm used to define the function `Equilikely`$(a,\ b)$ (see Definition 2.3.4).

**Example 6.2.8** This algorithm uses inversion and the idf expression from Example 6.2.4 to generate a *Geometric*$(p)$ random variate

```
u = Random();
return (long) (log(1.0 - u) / log(p));
```

This is the algorithm used to define the function `Geometric`$(p)$ (see Definition 3.1.2).

Examples 6.2.6, 6.2.7, and 6.2.8 illustrate random variate generation by inversion for three of the six parametric discrete random variable models presented in the previous section. For the other three models, inversion is not so easily applied.

**Example 6.2.9** If $X$ is a *Binomial*$(n, p)$ random variable the cdf is

$$F(x) = \sum_{t=0}^{x} \binom{n}{t} p^t (1-p)^{n-t} \qquad x = 0, 1, 2, \ldots, n.$$

It can be shown that this sum is the complement of an *incomplete beta function* (see Appendix D)

$$F(x) = \begin{cases} 1 - I(x+1, n-x, p) & x = 0, 1, \ldots, n-1 \\ 1 & x = n. \end{cases}$$

Except for special cases, an incomplete beta function cannot be inverted algebraically to form a "closed form" expression for the idf. Therefore, inversion is not easily applied to the generation of *Binomial*$(n, p)$ random variates.

### Algorithm Design Criteria

As suggested by Example 6.2.9, for many discrete random *variables* the design of a correct, exact, and efficient algorithm to generate corresponding random *variates* is significantly more complex than just a straight-forward implementation of Algorithm 6.2.3. There are some important, generally accepted algorithm design criteria listed below. With minor modifications, these criteria apply equally well to continuous random variate generation algorithms also.

- *Portability* — a random variate generation algorithm should be implementable in a high-level language in such a way that it ports easily to any common contemporary computing environment. Hardware-dependent implementations should be rejected, even if the cost to do so is an increase in execution time.
- *Exactness* — assuming a perfect random number generator, the histogram that results from generating random variates from an *exact* algorithm converges to the corresponding random variable pdf as the number of random variates generated goes to infinity. Example 6.2.10, presented later, is an illustration of a random variate generator that is approximate, not exact.
- *Robustness* — the performance of a random variate generation algorithm should be insensitive to small changes in the random variable model parameters (e.g., $n$ and $p$ for a $Binomial(n, p)$ model) and work properly for all reasonable values of the parameters.
- *Efficiency* — although this criteria is commonly over-rated, a random variate generation algorithm should be both time and memory efficient. Execution time often consists of two parts. *Set-up* time occurs once (for example, to compute and store a cdf array); *marginal* execution time occurs each time a random variate is generated. Ideally both times are small. If not, an implementation-dependent algorithm judgment must be made about which is more important.
- *Clarity* — if all other things are equal, a random variate generation algorithm that is easy to understand and implement is always preferred. For some people this is the most important criteria; however, for random variate generation specialists, portability, exactness, robustness, and efficiency tend to be most important.
- *Synchronization* — a random variate generation algorithm is synchronized if exactly one call to `Random` is required for each random variate generated. This property and monotonicity are often needed to implement certain *variance reduction techniques* like, for example, the *common random numbers technique*, illustrated in Example 3.1.7.
- *Monotonicity* — a random variate generation algorithm is monotone if it is synchronized and, like Algorithm 6.2.3, the transformation from $u$ to $x$ is monotone increasing (or monotone decreasing).

Although some of these criteria will be automatically satisfied if inversion is used, others (e.g., efficiency) may not. Generally, however, like the algorithms in Examples 6.2.6, 6.2.7, and 6.2.8, the best random variate generation algorithm is based on Algorithm 6.2.3.

**Example 6.2.10** In practice, an algorithm that uses inversion but is only approximate, may be satisfactory, provided the approximation is sufficiently good. As an illustration, suppose we want to generate *Binomial*(10, 0.4) random variates. To the 0.*ddd* precision indicated, the corresponding pdf is

| $x$ : | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $f(x)$ : | 0.006 | 0.040 | 0.121 | 0.215 | 0.251 | 0.201 | 0.111 | 0.042 | 0.011 | 0.002 | 0.000 |

Consistent with these *approximate* pdf values, random variates can be generated by filling a 1000-element integer-valued array $a[\cdot]$ with 6 zeros, 40 ones, 121 twos, etc. Then the algorithm

```
j = Equilikely(0, 999);
return a[j];
```

can be used each time a *Binomial*(10, 0.4) random variate is needed. The non-exactness of this algorithm is evident. A ten, for example, will never be generated, even though $f(10) = \binom{10}{10}(0.4)^{10}(0.6)^0 = 1/9765625$. It is "approximately exact," however, with an accuracy that may be acceptable in some applications. Moreover, the algorithm is portable, robust, clear, synchronized, and monotone. Marginal execution time efficiency is good. Set-up time efficiency is a potential problem, however, as is memory efficiency. For example, if 0.*ddddd* precision is desired, then it would be necessary to use a 100 000-element array.

**Example 6.2.11** The algorithm in Example 6.2.10 for generating *Binomial*(10, 0.4) random variates is inferior to an *exact* algorithm based on filling an 11-element floating-point array with cdf values and then using Algorithm 6.2.2 with $x = 4$ (the mode) to initialize the search.

In general, inversion can be used to generate *Binomial*$(n, p)$ random variates by computing a floating-point array of $n + 1$ cdf values and then using Algorithm 6.2.2 with $x = \lfloor np \rfloor$ to initialize the search. The capability provided by the library `rvms` (see Appendix D) can be used in this case to compute the cdf array by calling the cdf function `cdfBinomial(`$n, p, x$`)` for $x = 0, 1, \ldots, n$. Because this approach is inversion, in many respects it is ideal. The only drawback to this approach is some inefficiency in the sense of set-up time (to compute the cdf array) and memory (to store the cdf array), particularly if $n$ is large.*

**Example 6.2.12** The need to store a cdf array can be eliminated completely, at the expense of increased marginal execution time. This can be done by using Algorithm 6.2.2 with the cdf capability provided by the library `rvms` to compute cdf values only as needed. Indeed, in the library `rvms` this is how the idf function `idfBinomial(`$n, p, u$`)` is evaluated. Given that `rvms` provides this capability, *Binomial*$(n, p)$ random variates can be generated by inversion as

---

* As discussed in the next section, if $n$ is large, the size of the cdf array can be *truncated* significantly and thereby partially compensate for the memory and set-up time inefficiency.

```
u = Random();
return idfBinomial(n, p, u);                    /* use the library rvms */
```

With appropriate modifications, the approach in Examples 6.2.11 and 6.2.12 can be used to generate *Poisson*($\mu$) and *Pascal*($n, p$) random variates. Therefore, inversion can be used to generate random variates for all six of the parametric random variable models presented in the previous section. For *Equilikely*($a, b$), *Bernoulli*($p$), and *Geometric*($p$) random variates inversion is essentially ideal. For *Binomial*($n, p$), *Pascal*($n, p$), and *Poisson*($\mu$) random variates, however, time and memory efficiency can be a problem if inversion is used. In part, this justifies the development of alternative generation algorithms for *Binomial*($n, p$), *Pascal*($n, p$), and *Poisson*($\mu$) random variates.

### 6.2.3 ALTERNATIVE RANDOM VARIATE GENERATION ALGORITHMS

#### Binomial Random Variates

As an alternative to inversion, a *Binomial*($n, p$) random variate can be generated by summing an *iid* sequence of *Bernoulli*($p$) random variates. (See Example 6.1.15.)*

**Example 6.2.13** This algorithm uses a *Bernoulli*($p$) random variate generator to generate a *Binomial*($n, p$) random variate

```
x = 0;
for (i = 0; i < n; i++)
   x += Bernoulli(p);
return x;
```

This algorithm is portable, exact, robust, and clear. It is not synchronized or monotone and the $O(n)$ marginal execution time complexity can be a problem if $n$ is large.

#### Poisson Random Variates

A *Poisson*($\mu$) random variable is the limiting case of a *Binomial*($n$, $\mu/n$) random variable as $n \to \infty$. Therefore, if $n$ is large then one of these random variates can be generated as an approximation to the other. Unfortunately, because the algorithm in Example 6.2.13 is $O(n)$, if $n$ is large we must look for other ways to generate a `Poisson`($\mu$) random variate. The *Poisson*($\mu$) cdf $F(\cdot)$ is equal to the complement of an *incomplete gamma function* (see Appendix D)

$$F(x) = 1 - P(x + 1, \mu) \qquad x = 0, 1, 2 \ldots$$

Except for special cases, an incomplete gamma function cannot be inverted to form an idf. Therefore, inversion can be used to generate a *Poisson*($\mu$) random variate, but the cdf is not simple enough to avoid the need to use a numerical approach like those in Examples 6.2.11 or 6.2.12.

---

* Random variate generation by summing an *iid* sequence of more elementary random variates is known as a *convolution method.*

There are standard algorithms for generating a *Poisson*($\mu$) random variate that do not rely on either inversion or a "large $n$" version of the algorithm in Example 6.2.13. The following is an example.

**Example 6.2.14** This algorithm generates a *Poisson*($\mu$) random variate

```
a = 0.0;
x = 0;
while (a < μ) {
   a += Exponential(1.0);
   x++;
}
return x - 1;
```

This algorithm is portable, exact, and robust. It is neither synchronized nor monotone, and marginal execution time efficiency can be a problem if $\mu$ is large because the expected number of passes through the `while` loop is $\mu + 1$. Although the algorithm is obscure at this point, clarity will be provided in Section 7.3.

### Pascal Random Variates

Like a *Binomial*($n, p$) cdf, a *Pascal*($n, p$) cdf contains an *incomplete beta function* (see Appendix D). Specifically, a *Pascal*($n, p$) cdf is

$$F(x) = 1 - I(x + 1,\, n,\, p) \qquad\qquad x = 0, 1, 2, \ldots$$

Except for special cases, an incomplete beta function cannot be inverted algebraically to form a closed-form idf. Therefore, inversion can be used to generate a *Pascal*($n, p$) random variate, but the cdf is not simple enough to avoid the need to use a numerical approach like those in Examples 6.2.11 or 6.2.12.

As an alternative to inversion, recall from Section 6.1 that the random variable $X$ is *Pascal*($n, p$) if and only if

$$X = X_1 + X_2 + \cdots + X_n$$

where $X_1$, $X_2$, ..., $X_n$ is an *iid* *Geometric*($p$) sequence. Therefore, a *Pascal*($n, p$) random variate can be generated by summing an *iid* sequence of $n$ *Geometric*($p$) random variates.

**Example 6.2.15** This algorithm uses a *Geometric*($p$) random variate generator to generate a *Pascal*($n, p$) random variate.

```
x = 0;
for (i = 0; i < n; i++)
   x += Geometric(p);
return x;
```

This algorithm is portable, exact, robust, and clear. It is neither synchronized nor monotone and the $O(n)$ marginal execution time complexity can be a problem if $n$ is large.

**Library rvgs**

See Appendix E for the library `rvgs` (Random Variate GeneratorS). This library consists of six functions for generating discrete random variates and seven functions for generating continuous random variates (see Chapter 7). The six discrete random variate generators in the library are:

- `long Bernoulli(double` $p$`)` — returns 1 with probability $p$ or 0 otherwise;
- `long Binomial(long` $n$`, double` $p$`)` — returns a *Binomial*$(n, p)$ random variate;
- `long Equilikely(long` $a$`, long` $b$`)` — returns an *Equilikely*$(a, b)$ random variate;
- `long Geometric(double` $p$`)` — returns a *Geometric*$(p)$ random variate;
- `long Pascal(long` $n$`, double` $p$`)` — returns a *Pascal*$(n, p)$ random variate;
- `long Poisson(double` $\mu$`)` — returns a *Poisson*$(\mu)$ random variate.

These random variate generators feature minimal set-up times, in some cases at the expense of potentially large marginal execution times.

**Library rvms**

The *Bernoulli*$(p)$, *Equilikely*$(a, b)$, and *Geometric*$(p)$ generators in `rvgs` use inversion and in that sense are ideal. The other three generators do not use inversion. If that is a problem, then as an alternative to the library `rvgs`, the idf functions in the library `rvms` (Random Variable ModelS, Appendix D) can be used to generate *Binomial*$(n, p)$, *Pascal*$(n, p)$, and *Poisson*$(\mu)$ random variates by inversion, as illustrated in Example 6.2.12.

Because the idf functions in the library `rvms` were designed for accuracy at the possible expense of marginal execution time inefficiency, use of this approach is generally not recommended when many observations need to be generated. Instead, in that case set up an array of cdf values and use inversion (Algorithm 6.2.2), as illustrated in Example 6.2.11. This approach is considered in more detail in the next section.

### 6.2.4 EXERCISES

**Exercise 6.2.1** Prove that $F^*(\cdot)$ is a monotone increasing function, i.e., prove that if $0 < u_1 \le u_2 < 1$ then $F^*(u_1) \le F^*(u_2)$.

**Exercise 6.2.2** If inversion is used to generate a *Geometric*$(p)$ random variate $x$, use your knowledge of the largest and smallest possible values of `Random` to determine (as a function of $p$) the largest and smallest possible value of $x$ that can be generated. (See also Exercise 3.1.3.)

**Exercise 6.2.3** Find the pdf associated with the random variate generation algorithm

```
u = Random();
return ⌈3.0 + 2.0 * u²⌉;
```

**Exercise 6.2.4** ($a$) Generate a *Poisson*(9) random variate sample of size 1 000 000 using the appropriate generator function in the library `rvgs` and form a histogram of the results. ($b$) Compare the resulting relative frequencies with the corresponding *Poisson*(9) pdf using the appropriate pdf function in the library `rvms`. ($c$) Comment on the value of this process as a test of correctness for the two functions used.

**Exercise 6.2.5** ($a$) If $X$ is a discrete random variable with possible values $x = 1, 2, \ldots, n$ and pdf $f(x) = \alpha x$, find an equation for the idf $F^*(u)$ as a function of $n$. (See Exercise 6.1.5.) ($b$) Construct an inversion function that will generate a value of $X$ with one call to `Random`. ($c$) How would you convince yourself that this random variate generator function is correct?

**Exercise 6.2.6** $X$ is a discrete random variable with cdf $F(\cdot)$ and idf $F^*(\cdot)$. Prove or disprove the following. ($a$) If $u \in (0,1)$ then $F(F^*(u)) = u$. ($b$) If $x \in \mathcal{X}$ then $F^*(F(x)) = x$.

**Exercise 6.2.7** ($a$) Implement Algorithm 6.2.1 for a *Poisson*($\mu$) random variable and use Monte Carlo simulation to verify that the expected number of passes through the `while` loop is $\mu$. Use $\mu = 1, 5, 10, 15, 20, 25$. ($b$) Repeat with Algorithm 6.2.2. ($c$) Comment. (Use the function `cdfPoisson` in the library `rvms` to generate the *Poisson*($\mu$) cdf values.)

**Exercise 6.2.8** Design and then implement a consistent "invalid input" error trapping mechanism for the functions in the library `rvgs`. Write a defense of your design.

**Exercise 6.2.9**[a] Suppose $X$ is a discrete random variable with cdf $F(\cdot)$ and possible values $\mathcal{X} = \{a, a+1, \ldots, b\}$ with both $a$ and $b$ finite. ($a$) Construct a binary search algorithm that will determine $F^*(u)$ for any $u \in (0,1)$. ($b$) Present convincing numerical evidence that your algorithm is correct when used to generate *Binomial*(100, 0.2) random variates and compare its efficiency with that of algorithms 6.2.1 and 6.2.2. ($c$) Comment.

**Exercise 6.2.10** ($a$) As an extension of Exercise 6.1.8, what is the idf of $X$? ($b$) Provide convincing numerical evidence that this idf is correct.

**Exercise 6.2.11**[a] Two integers $X_1, X_2$ are drawn at random, without replacement, from the set $\{1, 2, \ldots, n\}$ with $n \geq 2$. Let $X = |X_1 - X_2|$. ($a$) What are the possible values of $X$? ($b$) What are the pdf, cdf, and idf of $X$? ($c$) Construct an inversion function that will generate a value of $X$ with just one call to `Random`. ($d$) What did you do to convince yourself that this random variate generator function is correct?

**Exercise 6.2.12**[a] Same as the previous exercise, except that the draw is with replacement.