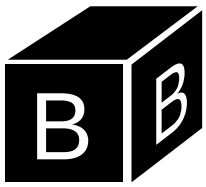


Chapter *14*

Random Numbers

Many events in our lives are random. When you enter a full parking lot, for example, you drive around until someone pulls out of a parking space that you can claim. The event of someone pulling out of a parking space is random. You do not know when or where it will occur. Consequently, you cannot predict exactly how long you will drive before you find a place.

Computers are useful in part because they can behave like the real world. For example, the popular flight-simulator computer programs can give the illusion of piloting an airplane. Large, sophisticated flight simulators are even used to train airline pilots. But how can a computer behave like the real world when some events in the real world are random? Every algorithm you have encountered thus far in this book has no random element. Given the input and the processing statements, you can always predict the output. With random events, you cannot predict the outcome.

A random number module

The solution to the problem of simulating random elements is to design an algorithm whose output appears random, even though it is not. The PboxRandom module contains four procedures that provide the client module the ability to behave in a seemingly random fashion, and thus to simulate random events in the real world. Figure 14.1 is the interface for the random number module PboxRandom.

```
DEFINITION PboxRandom;
   CONST
      seedLimit = 2147483647;

   PROCEDURE Int (n: INTEGER): INTEGER;
   PROCEDURE Randomize;
   PROCEDURE Real (): REAL;
   PROCEDURE SetSeed (n: INTEGER);

END PboxRandom.
```

Figure 14.1
The interface of module PboxRandom.

The algorithm that produces seemingly random numbers depends on maintaining an integer value known as a seed. Each time the importing module requests a random number the number is computed from the seed, and the next value of the seed is

computed from the current value of the seed. Procedure SetSeed allows the importing module to set the value of the seed before requesting a sequence of random numbers. Its documentation is

```
PROCEDURE SetSeed (n: INTEGER);
Pre
0 < n   20
n < seedLimit   21
Post
The random number seed is initialized to n.
```

A precondition for SetSeed to work correctly is for n to be greater than zero and less than seedLimit. If the seed is set to the same value before requesting a second sequence of random numbers, the second sequence of numbers will be identical to the first sequence.

Procedure Randomize sets the seed to some random number not predictable by the importing module. It gets the value from a clock inside the computer that keeps track of the date to the nearest second. Its documentation is

```
PROCEDURE Randomize
Post
The random number seed is initialized to a value derived from the system clock.
Two calls to Randomize should be separated by more than one second to guarantee
different values of seed.
```

Randomize has no precondition. If you call Randomize twice, the values that the seed is set to will be different, because you will have called the procedure at different times.

Random reals

Figure 14.3 shows the dialog box for a module that illustrates the behavior of procedure PboxRandom.Real(), which is a function procedure. The dialog box has an input field for the user to enter a seed value. When the user clicks the button labeled Set Seed, procedure SetSeed executes with its parameter value equal to the value the user has entered in the dialog box. If the user sets the seed to 4831 as shown in the figure then clicks the button labeled Display, ten real numbers are printed to the Log as shown.

```
MODULE Pbox14A;
   IMPORT Dialog, PboxRandom, PboxStrings, StdLog;
   VAR
      d*: RECORD
         seed*: INTEGER;
      END;
```

Figure 14.2
A procedure that prints ten random real numbers to the Log.

Figure 14.2
Continued.

```
    PROCEDURE SetSeed*;
    BEGIN
        IF (0 < d.seed) & (d.seed < PboxRandom.seedLimit) THEN
            PboxRandom.SetSeed(d.seed)
        ELSE
            StdLog.String("Seed must be greater than 0 and less than 2147483647."); StdLog.Ln
        END
    END SetSeed;

    PROCEDURE Randomize*;
    BEGIN
        PboxRandom.Randomize
    END Randomize;

    PROCEDURE Display*;
    VAR
        i: INTEGER;
        x: REAL;
        realString: ARRAY 8 OF CHAR;
    BEGIN
        FOR i := 1 TO 10 DO
            StdLog.String("i = "); StdLog.Int(i);
            x := PboxRandom.Real();
            PboxStrings.RealToString(x, 5, 3, realString);
            StdLog.String("    x = "); StdLog.String(realString); StdLog.Ln
        END;
        StdLog.Ln
    END Display;

BEGIN
    d.seed := 1
END Pbox14A.
```

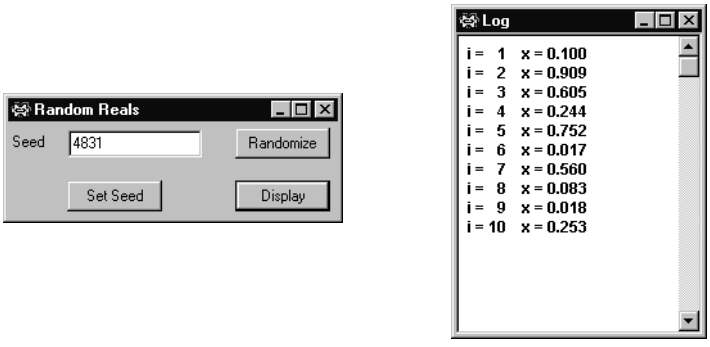


Figure 14.3
The output of procedure Display of Figure 14.2.

If the user clicks the Display button a second time without first setting the seed back to 4831 a different set of ten real values will be displayed as follows:

```
i =  1   x = 0.329
i =  2   x = 0.699
i =  3   x = 0.301
i =  4   x = 0.252
i =  5   x = 0.473
i =  6   x = 0.637
i =  7   x = 0.312
i =  8   x = 0.166
i =  9   x = 0.728
i = 10   x = 0.937
```

The program in Figure 14.2 implements the dialog box of Figure 14.3. The Set Seed and Randomize buttons simply call the corresponding procedures from module PboxRandom, with procedure SetSeed insuring that the precondition for procedure PboxRandom.SetSeed is not violated.

The Display button is linked to a procedure that executes a FOR loop ten times. Each time the body of the loop executes it calls procedure PboxRandom.Real(), which is a function procedure that returns a random real value between zero and one. Its documentation is

```
PROCEDURE Real (): REAL;
Post
Returns a random real between 0.0 and 1.0.
```

PboxRandom.Real has no precondition. Because it is a function, you must use it within another statement. Procedure Display uses it within the assignment statement

```
x := PboxRandom.Real()
```

This function call shows a curious requirement of procedure calls. When you call a proper procedure that has no parameters you omit the parentheses in the call. However, when you call a function procedure that has no parentheses you must include the parentheses with nothing between them.

Function procedure calls with no parameters still need parentheses.

Random integers

For simulation purposes it is frequently useful to have a series of random integers rather than random reals. Figure 14.4 shows a dialog box that produces a sequence of ten seemingly random integers. It outputs a different list of random integers depending on the initial value for the seed.

The dialog box has an additional input field labeled Limit. In the figure, the user entered 10 for the limit before clicking on the Display button. That choice caused each random integer to have one of 10 values between 0 and 9.

Module Pbox14B in Figure 14.5 produces the output shown in Figure 14.4. As with the previous module, it consists mostly of simple calls to the procedures of PboxRandom.

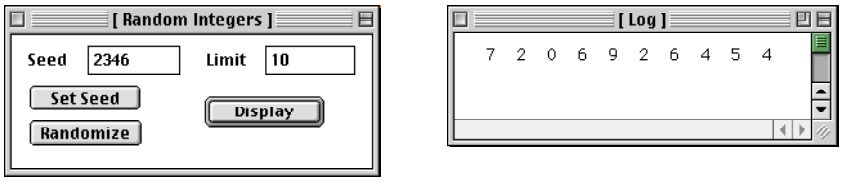


Figure 14.4
The output for the procedure of Figure 14.5.

```
MODULE Pbox14B;
   IMPORT Dialog, PboxRandom, PboxStrings, StdLog;
   VAR
      d*: RECORD
         seed*: INTEGER;
         limit*: INTEGER;
      END;

   PROCEDURE SetSeed*;
   BEGIN
      IF (0 < d.seed) & (d.seed < PboxRandom.seedLimit) THEN
         PboxRandom.SetSeed(d.seed)
      ELSE
         StdLog.String("Seed must be greater than 0 and less than 2147483647."); StdLog.Ln
      END
   END SetSeed;

   PROCEDURE Randomize*;
   BEGIN
      PboxRandom.Randomize
   END Randomize;

   PROCEDURE Display*;
   VAR
      i: INTEGER;
      m: INTEGER;
   BEGIN
      IF (0 < d.limit) & (d.limit < PboxRandom.seedLimit) THEN
         FOR i := 1 TO 10 DO
            m := PboxRandom.Int(d.limit);
            StdLog.Int(m)
         END;
         StdLog.Ln
      ELSE
         StdLog.String("Limit must be greater than 0 and less than 2147483647."); StdLog.Ln
      END
   END Display;

BEGIN
   d.seed := 1;
   d.limit := 0
END Pbox14B.
```

Figure 14.5
A procedure that prints ten random integers to the Log.

Procedure PboxRandom.Int is a function that returns an integer. Unlike PboxRandom.Real, it requires a parameter that specifies the range of possible integer values to be returned. The documentation for PboxRandom.Int is

```
PROCEDURE Int (n: INTEGER): INTEGER;
Pre
0 < n   20
n < seedLimit   21
Post
Returns a random integer in the range 0..n-1.
```

The precondition is that n is positive and less than seedLimit, which is checked by the calling procedure.

Example 14.1 Had the user entered 8 for the limit and set the value of seed to 2346 the sequence

```
6 2 0 5 7 1 5 3 4 3
```

would be printed on the Log. In this sequence, each random integer has one of eight values between 0 and 7. ▮

The REPEAT statement

When the programs we have written up until now have required loops we were always able to solve the problem at hand with either the WHILE loop or the FOR loop. One characteristic that is common to both of these loops is that the test is at the beginning of the loop. Hence, the body of a WHILE or FOR loop will not execute at all if the first test of the boolean condition is false. It is usually desirable to permit the possibility of the body never executing. For example, if you are processing a list of values in the focus window the body of the loop contains the statements necessary to process one value. Each time the body executes it processes another value. If the focus window has no values, you do not want the body of the loop to execute at all.

Although not as common, the situation sometimes occurs where you always want the body of the loop to execute at least one time. For these cases it would be more convenient to have the test for loop termination be at the end of the loop rather than at the beginning. Component Pascal provides such a loop in the form of the REPEAT statement. It differs from the WHILE statement in two respects. Not only is the test for termination at the end of the loop instead of the beginning, the loop terminates when the test condition is true rather than false. Figure 14.6 is a flowchart for the REPEAT statement

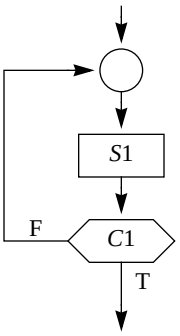


Figure 14.6
The flowchart for the REPEAT statement.

```
REPEAT
   Statement1
UNTIL Condition1
```

which you should compare with Figure 10.1 for the WHILE statement. Statement1 always executes the first time regardless of Condition1. If Condition1 is false control

branches up to the REPEAT and Statement1 executes again. The loop repeats until Condition1 is true.

Rolling a pair of dice

Figure 14.7 shows how module PboxRandom can be used to simulate a random event in the real world. Suppose you are playing a game with a pair of dice. Each die has six sides. When you roll one die, it will come to rest with some random integer between 1 and 6 showing on its top side. The figure shows a simulation of a player who rolls the dice until a total of 7 or 11 appears.

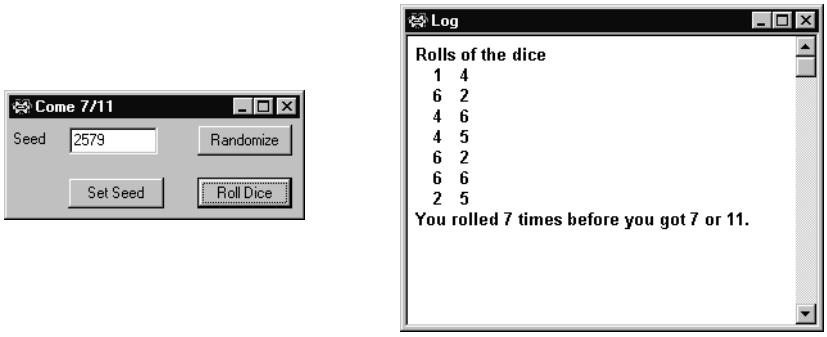


Figure 14.7
The output for the procedure in Figure 14.8.

You can simulate the toss of a single die by calling procedure PboxRandom.Int(6), which will return a random number between 0 and 5. If you add one to that value you will have a random number between 1 and 6. The program in Figure 14.8 simulates the rolls by executing a REPEAT statement. Each time the body of the loop executes it makes two calls to PboxRandom.Int and outputs the result of the rolls on the Log. The loop repeats until the sum of the numbers on the dice equals 7 or 11. You can see how convenient the REPEAT statement is in this situation, because you always want the body to execute at least one time. In the simulation, there is always at least one roll of the dice.

To simulate the toss of a coin, you would need random values with two possibilities, one for heads and one for tails.

Example 14.2 You could call

```
PboxRandom.Int(2)
```

and let 0 represent heads and 1 represent tails. The sequence of 20 calls will produce

0 0 1 0 0 0 0 1 0 0 1 1 0 1 1 1 0 1 0 1

with an initial value of 9735 for Seed. ∎

```
MODULE Pbox14C;
   IMPORT Dialog, PboxRandom, PboxStrings, StdLog;
   VAR
      d*: RECORD
         seed*: INTEGER;
      END;

   PROCEDURE SetSeed*;
   BEGIN
      IF (0 < d.seed) & (d.seed < PboxRandom.seedLimit) THEN
         PboxRandom.SetSeed(d.seed)
      ELSE
         StdLog.String("Seed must be greater than 0 and less than 2147483647."); StdLog.Ln
      END
   END SetSeed;

   PROCEDURE Randomize*;
   BEGIN
      PboxRandom.Randomize
   END Randomize;

   PROCEDURE RollDice*;
   VAR
      die1, die2: INTEGER;
      sum, numRolls: INTEGER;
   BEGIN
      numRolls := 0;
      StdLog.String("Rolls of the dice"); StdLog.Ln;
      REPEAT
         die1 := PboxRandom.Int(6) + 1;
         die2 := PboxRandom.Int(6) + 1;
         INC(numRolls);
         sum := die1 + die2;
         StdLog.Int(die1); StdLog.Int(die2); StdLog.Ln
      UNTIL (sum = 7) OR (sum = 11);
      StdLog.String("You rolled "); StdLog.Int(numRolls);
      StdLog.String(" times before you got 7 or 11."); StdLog.Ln
   END RollDice;

BEGIN
   d.seed := 1
END Pbox14C.
```

Figure 14.8
A procedure that simulates rolls of a pair of dice.

Random number generators

The random number generators in the preceding programs are all based on the general computation

$$z_{n+1} = az_n \text{ MOD } m$$

where $z_1, z_2, z_3, \ldots$ are the successive values of the seed, m is the modulus, and a is the multiplier with $2 < a < m$. Generators of this form are called Lehmer generators after the person who proposed them. To design a Lehmer generator you select values of m and a. An (m, a) Lehmer generator is one with a modulus of m and a multiplier of a.

Lehmer generators

Example 14.3 If you select values of (17, 5) for (m, a), and the current value of seed z_n is 11, then the next seed is computed as

$$\begin{aligned} z_{n+1} &= az_n \bmod m \\ &= 5 \cdot 11 \bmod 17 \\ &= 55 \bmod 17 \\ &= 4 \end{aligned}$$

The 20 successive seed values from the (17, 5) Lehmer generator are

11 4 3 15 7 1 5 8 6 13 14 2 10 16 12 9 11 4 3 15

with an initial seed of 11. ∎

Example 14.4 The (17, 13) Lehmer generator produces the sequence

4 1 13 16 4 1 13 16 4 1 13

starting from 4. ∎

These examples show an unavoidable feature of all pseudorandom number generators. Because each value is computed from the previous value, once the initial value reappears in the sequence, the sequence must repeat. The period of the generator is the maximum number of values before the sequence begins to repeat. The (17, 5) Lehmer generator has a period of 16, and the (17, 13) generator has a period of 4.

A truly random sequence would contain no repeating cycles. But a pseudorandom sequence must repeat eventually, because there are only a finite number of values less than the modulus. The best you can do is to pick the modulus and multiplier to make the cycle as long as possible and to make the output appear random. The longest possible period with a modulus of m is $m - 1$. A generator with this period is known as a full-period generator.

Full-period generators

Computer scientists have devised a set of statistical tests that measure the randomness of proposed generators. They have investigated the pseudorandom sequences generated by different choices of m and a in an effort to discover the best generators. One standard Lehmer generator that is among the best known has (m, a) values of (2147483647, 48271), which is a full-period generator with good pseudorandom behavior. The modulus m is a Mersenne prime equal to $2^{31} - 1$. This is the generator that is used in module PboxRandom in Figure 14.9.

```
MODULE PboxRandom;
    IMPORT Dates;
    CONST
        multiplier = 48271;
        modulus = 2147483647;
        quotient = modulus DIV multiplier;
        remainder = modulus MOD multiplier;
        seedLimit* = modulus;
    VAR
        seed: INTEGER;

    PROCEDURE ComputeNextSeed;
        VAR
            low, high: INTEGER;
    BEGIN
        low := seed MOD quotient;
        high := seed DIV quotient;
        seed := multiplier * low - remainder * high;
        IF seed <= 0 THEN
            seed := seed + modulus
        END
    END ComputeNextSeed;

    PROCEDURE Int* (n: INTEGER): INTEGER;
    BEGIN
        ASSERT(0 < n, 20);
        ASSERT(n < seedLimit, 21);
        ComputeNextSeed;
        RETURN SHORT(ENTIER(seed / modulus * n))
    END Int;

    PROCEDURE Randomize*;
        VAR
            date: Dates.Date;
            time: Dates.Time;
            i: INTEGER;
    BEGIN
        Dates.GetDate(date); Dates.GetTime(time);
        seed := 86400 * Dates.Day(date) + 3600 * time.hour + 60 * time.minute
            + time.second; (* Elapsed time this year in seconds *)
        FOR i := 0 TO 7 DO
            ComputeNextSeed
        END
    END Randomize;

    PROCEDURE Real* (): REAL;
    BEGIN
        ComputeNextSeed;
        RETURN seed / modulus
    END Real;
```

Figure 14.9
An implementation of the standard (2147483647, 48271) Lehmer random number generator.

```
    PROCEDURE SetSeed* (n: INTEGER);
        VAR
            i: INTEGER;
    BEGIN
        ASSERT(0 < n, 20);
        ASSERT(n < seedLimit, 21);
        seed := n MOD modulus;
        FOR i := 0 TO 7 DO
            ComputeNextSeed
        END
    END SetSeed;

BEGIN
    Randomize
END PboxRandom.
```

Figure 14.9
Continued.

In PboxRandom, the constant multiplier corresponds to *a*, and the constant modulus corresponds to *m* in the equation $z_{n+1} = az_n \text{ MOD } m$. Module PboxRandom contains a global variable seed, which corresponds to z_n. It must be global, because its value must persist between calls of the procedures. Procedure ComputeNextSeed computes the next seed z_{n+1} from the current seed. A direct translation from the equation to Component Pascal would be one assignment statement,

```
seed := multiplier * seed MOD modulus
```

Instead, module ComputeNextSeed is implemented as

```
low := seed MOD quotient;
high := seed DIV quotient;
seed := multiplier * low - remainder * high;
IF seed <= 0 THEN
    seed := seed + modulus
END
```

where quotient and remainder are defined as the constants

```
quotient = modulus DIV multiplier;
remainder = modulus MOD multiplier;
```

and low and high are local variables. Why is this more complicated algorithm used instead of the more direct translation of a single assignment statement?

The problem is that the range of possible values for a Component Pascal integer is –2147483648 to 2147483647, which not so coincidentally is the modulus of the generator. When z_n gets close to 2147483647 and gets multiplied by 48271 before the MOD operation, the product lies outside the range and an overflow error occurs.

Fortunately, Schrage developed an algorithm (published in 1979) to implement a Lehmer generator in spite of the limited range of the integer type. Instead of multiplying the value of z_n by *a*, the algorithm computes two intermediate numbers from z_n, each of which is guaranteed to be smaller than *m*. It then combines these smaller

numbers (low and high in procedure ComputeNextSeed) to compute z_{n+1} in a way that is guaranteed mathematically to be equivalent to multiplying z_n by a and then doing the MOD operation. The last section of this chapter provides a more detailed explanation of Schrage's algorithm.

Procedure Real works by computing the next integer value of seed and then converting that value to a real number between 0.0 and 1.0. Because every value of seed is between zero and the modulus, Real simply returns seed divided by the modulus.

Procedure Int is a bit more complicated. Suppose the calling procedure calls Int with value 5 for formal parameter n. Because seed has a value between 1 and the modulus minus 1, the quantity seed / modulus has a value between just above 0.0 and just below 1.0. Therefore, the quantity seed / modulus * n has a value between just above 0.0 and just below 5.0. The ENTIER function truncates this value, producing an integer value between 0 and 4. Figure 14.10 shows the transformation from the real number line to the integer number line.

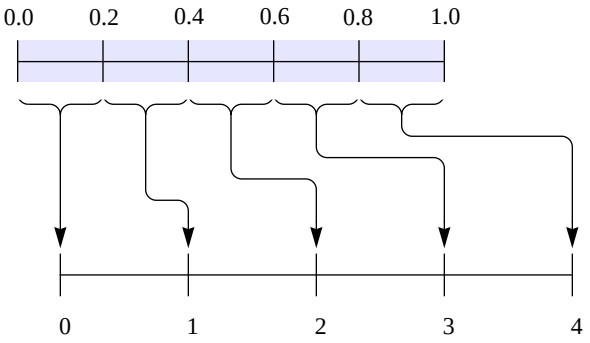


Figure 14.10
The transformation PboxRandom.Int makes from the real number line to the integer number line.

Procedure Randomize sets the value of seed based on the date and time from the system clock. BlackBox provides a module named Dates that links to the clock in your computer. The module provides abstract data types (ADTs) Date and Time. There is a procedure named GetDate that gives the actual parameter the current date and another named GetTime that gives the actual parameter the current time of day. Function Day returns the day of the year, beginning with 1 for the first day of January, 2 for the second of January, and up to 365 for the last day of December (provided the current year is not a leap year). Randomize uses Day together with the current time to compute how many seconds has elapsed since the first of the year. The seed variable is initialized to that number of seconds, and then run through eight cycles of the Lehmer algorithm.

Although you can have fun trying to design your own generator, finding good values of m and a is not an easy task. If the modulus m is a prime number, then zero will never appear in the sequence, a desirable feature indeed. Even so, Example 14.4 shows that a prime modulus is no guarantee of a full-period generator. Actually, making a Lehmer generator full period is the easy part. It is much more difficult to find values of m and a that produce sequences that are sufficiently pseudorandom.

★ Schrage's algorithm

This section uses the following mathematical symbols corresponding to the quantities in module PboxRandom.

z = seed
a = multiplier
m = modulus
q = quotient
r = remainder
l = low
h = high

With these abbreviations, the computation of the quotient and remainder are

$q = m \text{ div } a$
$r = m \bmod a$

and Schrage's algorithm in GCL is

$l := z \bmod q$
$h := z \text{ div } q$
$z := a * l - r * h$
if $z \leq 0 \rightarrow z := z + m$
[] $z > 0 \rightarrow$ **skip**
fi

Before describing Schrage's algorithm in general, consider a specific computation of the next seed for the Lehmer generator of Example 14.3.

Example 14.5 For the (17, 5) generator of Example 14.3, the constants q and r are computed as

$q = m \text{ div } a = 17 \text{ div } 5 = 3$
$r = m \bmod a = 17 \bmod 5 = 2$

and the computation of the next seed after 11 from Schrage's algorithm is

$l = z \bmod q = 11 \bmod 3 = 2$
$h = z \text{ div } q = 11 \text{ div } 3 = 3$
$z = a \cdot l - r \cdot h = 5 \cdot 2 - 2 \cdot 3 = 10 - 6 = 4$

The computation for z in Example 14.3 requires the intermediate computation of 5 times 11, which is 55. This computation for the same next seed requires 5 times 2, which is only 10. ∎

Example 14.6 To illustrate how the **if** statement works in Schrage's algorithm, consider the same (17, 5) generator to compute the next seed after 16.

$$l = z \bmod q = 16 \bmod 3 = 1$$
$$h = z \operatorname{div} q = 16 \operatorname{div} 3 = 5$$
$$z = a \cdot l - r \cdot h = 5 \cdot 1 - 2 \cdot 5 = 5 - 10 = -5$$

This time, –5 is less than or equal to 0. So, the **if** statement requires the addition of m as follows.

$$z = -5 + m = -5 + 17 = 12$$

which is the next seed after 16. ∎

The relation between the div and mod operators is based on the fact that $x \operatorname{div} y$ is the quotient when you divide x by y, and $x \bmod y$ is the remainder when you divide x by y. The quotient and remainder are related to x and y by

$$x = y \cdot (\text{quotient}) + \text{remainder} \qquad 0 \le \text{remainder} < y$$

so that

$$x = y \cdot (x \operatorname{div} y) + x \bmod y \qquad 0 \le x \bmod y < y$$

Solving this equation for $x \bmod y$

$$x \bmod y = x - y(x \operatorname{div} y)$$

Schrage's algorithm is an alternate way of computing the quantity $az \bmod m$, which can be manipulated as follows.

$$\begin{aligned}
& az \bmod m \\
= \quad & \langle x \bmod y = x - y(x \operatorname{div} y) \rangle \\
& az - m(az \operatorname{div} m) \\
= \quad & \langle \text{Add and subtract } m(z \operatorname{div} q) \rangle \\
& az - m(z \operatorname{div} q) + m(z \operatorname{div} q) - m(az \operatorname{div} m) \\
= \quad & \langle \text{Factor out } m \rangle \\
& az - m(z \operatorname{div} q) + m(z \operatorname{div} q - az \operatorname{div} m) \\
= \quad & \langle \text{Define } \gamma(z) = az - m(z \operatorname{div} q) \text{ and } \delta(z) = z \operatorname{div} q - az \operatorname{div} m \rangle \\
& \gamma(z) + m\delta(z)
\end{aligned}$$

Schrage's algorithm is based on the fact that $\gamma(z)$ is the computation $a \cdot l - r \cdot h$, which is computed just before the **if** statement, and that the quantity $\delta(z)$ is either zero or one. If $\delta(z)$ is zero, the algorithm does not add anything to $\gamma(z)$. If $\delta(z)$ is

one, the algorithm adds m to $\gamma(z)$.

To show that $\gamma(z)$ is the computation $a \cdot l - r \cdot h$, use the fact that q is the quotient and r is the remainder when you divide m by a. So, they are related by

$$m = q \cdot a + r \qquad 0 \le r < a$$

Therefore,

$$\begin{aligned}
& \gamma(z) \\
= \quad & \langle \text{Definition of } \gamma(z) \rangle \\
& az - m(z \text{ div } q) \\
= \quad & \langle m = qa + r \rangle \\
& az - (qa + r)(z \text{ div } q) \\
= \quad & \langle \text{Algebra} \rangle \\
& a[z - q(z \text{ div } q)] - r(z \text{ div } q) \\
= \quad & \langle \text{General relation between div and mod, } x = y \cdot (x \text{ div } y) + x \bmod y \rangle \\
& a(z \bmod q) - r(z \text{ div } q) \\
= \quad & \langle \text{Computation from algorithm } l := z \bmod q \text{ and } h := z \text{ div } q \rangle \\
& a \cdot l - r \cdot h
\end{aligned}$$

To show that $\delta(z)$ is either zero or one, you must prove both an upper and lower bound, $-1 < \delta(z) < 2$. The lower bound $-1 < \delta(x)$ is straightforward to prove. It depends only on the fact that for positive integers x and y

$$x/y - 1 < x \text{ div } y \le x/y$$

where / represents real division.

Example 14.7 With $x = 25$ and $y = 4$, the inequalities state that

$$25/4 - 1 < 25 \text{ div } 4 \le 25/4$$

which is equivalent to

$$5.25 < 6 \le 6.25$$

With $x = 24$ and $y = 4$, the inequalities state that

$$24/4 - 1 < 24 \text{ div } 4 \le 24/4$$

which is equivalent to

$$5 < 6 \le 6$$

From the definition of $\delta(z)$, the lower bound is

$$-1 < z \text{ div } (m \text{ div } a) - az \text{ div } m$$

which is equivalent to

$$z \text{ div } (m \text{ div } a) > az \text{ div } m - 1$$

To prove this inequality, start with the left hand side.

$$\begin{aligned}
& z \text{ div } (m \text{ div } a) \\
> \quad & \langle x \text{ div } y > x/y - 1 \rangle \\
& z/(m \text{ div } a) - 1 \\
\geq \quad & \langle x \text{ div } y \leq x/y \rangle \\
& z/(m/a) - 1 \\
= \quad & \langle \text{Algebra} \rangle \\
& az/m - 1 \\
\geq \quad & \langle x/y \geq x \text{ div } y \rangle \\
& az \text{ div } m - 1
\end{aligned}$$

The upper bound $\delta(z) < 2$ is not so straightforward to prove and will not be given here. It depends on the fact that $z < m$, which must be true because m is the modulus of the computation for the seed. But, the upper bound also depends on one additional assumption, namely that $r < q$, which is equivalent to $m \bmod a < m \text{ div } a$. This additional assumption is a restriction on Schrage's algorithm. Without it, $\delta(z)$ can be greater than one, and the algorithm would need to add some multiple of m, not just m, to the original computation of $a \cdot l - r \cdot h$. The beauty of Schrage's algorithm is that if you are careful to choose (m, a) for your Lehmer generator so that $m \bmod a < m \text{ div } a$, the computation of $\delta(z)$ is not necessary. By choosing (m, a) so that $\delta(z)$ is zero or one you are guaranteed that the computation $a \cdot l - r \cdot h$ will either be correct or, if it is not positive, will need to be adjusted only by the addition of m.

Exercises

1. When you toss two dice, the sum is one of the 11 numbers between 2 and 12. Would it be a good idea to simulate the toss of two dice by calling PboxRandom.Int once with a value of 11 for n? Explain.

2. **(a)** What are the values of r and q for the random number generator in PboxRandom? Do these values satisfy the assumption for the upper bound of $\delta(z)$? **(b)** Answer part (a) for the random number generator of Example 14.3.

3. Prove the upper bound $\delta(z) < 2$ for Schrage's algorithm for the implementation of the Lehmer random number generator. Send your proof to the author of this book and he

will include it in the next printed revision and credit you in the acknowledgments.

Problems

4. In the child's game of paper/scissors/rock, each child secretly chooses one of the objects. When the choices are revealed, paper loses to scissors, scissors loses to rock, and rock loses to paper. Equal choices are a draw. Design a dialog box that permits the user to set the seed or randomize it. Include a set of three radio buttons for the user to select paper, scissors, or rock. When the user clicks on a button labeled Play, compare her choice with a randomly selected object and print a message on the Log that states what the computer chose and who won, the computer or the user.

5. Add some output fields to the dialog box of Problem 4 to keep track of the score between the user and the computer. Each time the user plays a game, update the score. Include a button to reset the score to zero.

6. In the program of Listing 14.8, it took 8 rolls of the dice to get a 7 or 11. What do you think the average number of rolls would be? Can you calculate the average mathematically? Perform a computational experiment by writing a program to simulate 100 sequences of rolls. Design a dialog box for the user to set the seed or to randomize it. When the user clicks a button, simulate 100 sequences of rolls and output on the Log the fewest number of rolls (integer), the greatest number of rolls (integer) and the average number of rolls (real), to get a 7 or 11.

7. The game of craps is played as follows. Roll a pair of dice. If you get a 7 or 11 on the first roll you win, and if you get 2, 3 or 12 (called craps) you lose. Otherwise, the number you rolled becomes your point. You then keep rolling until you roll your point again, in which case you win, or until you roll a 7, in which case you lose. For example, a roll of 9, then 2, then 10, then 9 is a win, because the point (9 in this case) was rolled again before 7. As another example, a roll of 5, then 2, then 10, then 7 is a loss, because a 7 was rolled before the point (5 in this case).

Write a Component Pascal program to simulate a game of craps. Design a dialog box for the user to set the seed or randomize it. Include one other button for the user to press to play a game of craps. When the user clicks the button, output the result of the sequence of rolls for the game to the Log. In the end, announce if the user won or lost.

8. In the Problem 7, what do you think the probability of winning a game of craps is? Can you calculate it mathematically? Perform a computational experiment by writing a program to simulate 100 games. Design a dialog box that allows the user to set the seed or randomize it. When the user clicks a button, output to the Log the number of wins (integer) and losses (integer), and an estimate of the probability of winning a single game as the ratio (real) of the number of wins divided by the total number of games.

9. The local town drunk gets thoroughly inebriated, climbs to the roof of a skyscraper, steps out onto the center of the ledge, and begins to walk. Each time he takes a step, the probability is 1/3 that he will step to the right, 1/3 that he will step straight ahead, and 1/3 that he will step to the left. If he takes a total of two steps to the left, he will fall safely onto the roof. But if he takes a total of two steps to the right, he will fall to the sidewalk below. Write a program that simulates one walk of the drunk. Design a dialog box that allows the user to set the seed or randomize it. When the user clicks a button,

output the random walk to the Log as shown below, with the roof on the left and the sidewalk on the right. At the end of the simulation announce which way he fell. After the program executes you can display the Log in Courier or some other monospaced font to make the spaces next to the x character distinct.

```
(  x  )
( x   )
( x   )
(  x  )
(   x )
(  x  )
(   x )
(   x )
(    x)
He fell to the sidewalk.
```

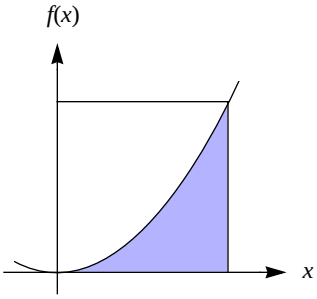
10. In Problem 9, suppose a whole army of drunks repeat the walk many times. Guess the average length of a walk. That is, how many steps on the average does a drunk take before he falls off one way or the other? Now perform a computational experiment by writing a program to simulate 100 walks. Design a dialog box that allows the user to set the seed or randomize it. When the user clicks a button, output the number of times he fell to the sidewalk (integer), the number of times he fell onto the roof (integer), and the average number of steps he took (real) before falling. How close is the computed value to your guess?

The ideas in this problem form the basis of an important mathematical technique called the Monte Carlo method. The method has application to problems in statistical physics. The "army" of drunks is called an ensemble, and the average is called an ensemble average.

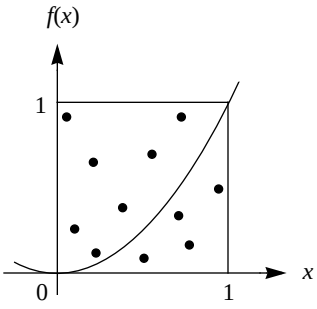
11. Figure 14.11(a) shows a graph of the equation $y = x^2$ between the points $x = 0.0$ and $x = 1.0$. The square of height 1.0 between these points has area 1.0. You can estimate the area under the curve by picking several points at random inside the square and counting the points that are below the curve. The area is approximately the number of points below the curve divided by the total number of points. For example, the estimate from Figure 14.11(b) is $4/11$ or 0.3636.

Design a dialog box that requests the user to enter a seed and the number of random points, then outputs the estimate of the area in the dialog box. Obtain the coordinates of a single point by calling procedure PboxRandom.Real twice, once for the x-coordinate and once for the y-coordinate. What do you think is the relationship between the number of random points and the accuracy of the estimate? Can you illustrate that relationship with your program?

12. You can use a random number generator to compute the value of π based on the fact that the area of a circle is πr^2. Figure 14.12 shows the area of one fourth of a circle with radius 1.0 whose area is $\pi(1.0)^2/4 = \pi/4$. Write a program that asks the user to enter a seed and the number of random points, then computes the estimate of the area using the technique of Problem 11. Output the estimate of π as four times the area. What do you think is the relationship between the number of random points and the accuracy of the estimate? Can you illustrate that relationship with your program?



(a) The area under the curve.



(b) Eleven random points for estimating the area.

Figure 14.11

The function $f(x) = x^2$ for Problem 11.

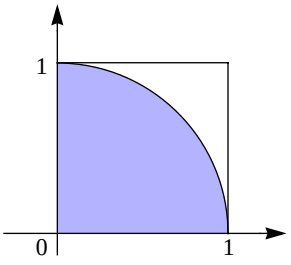


Figure 14.12

The quarter circle to estimate π for Problem 12.

13. Figure 10.7 shows an algorithm for finding the largest value from a list of values in the focus window. It contains the loop

```
WHILE ~sc.eot DO
    IF num > largest THEN
        largest := num
    END
    sc.ScanInt(num)
END
```

(a) If the numbers in the focus window are random, approximately what percentage of the executions of the loop do you suppose would include the statement largest := num? Explain the reasoning behind your supposition. **(b)** After answering part (a), write a program in Component Pascal to test your supposition. Modify the algorithm to take the numbers from a random number generator instead of from the focus window. Design a dialog box with seven controls—a field for the user to input a seed, a button to set the seed, a button to randomize the seed, a field for the user to input the number of random integers to process, a button to find the maximum of the random numbers, an output field to display the largest integer found, and an output field to display the percentage of the number of times that execution of the body of the loop includes the statement largest := num. When the user presses the button to find the maximum integer, invoke PboxRandom.Int with an actual parameter of one billion (1,000,000,000). That is, you will be testing to find the maximum of a set of integers, each one of which is between one and one billion. **(c)** Experiment with your program trying out various values of the seed and of the number of integers to test. Is your supposition from part (a) close to the results from your program? If not, explain how your reasoning must be modified to account for the results of your program.

14. Write a program that outputs to the Log all the values of the multiplier a that will produce a full-period Lehmer generator with a modulus m of 1021. Remember that the multiplier is restricted to $2 < a < m$.