CAPÍTULO

9 Números aleatorios

En muchas ocasiones en la computación se requiere el uso de números aleatorios. Por ejemplo, la criptografía moderna, los sistemas de simulación y, sorprendentemente, incluso algunos algoritmos de búsqueda y ordenación, se basan en la generación de números aleatorios. Sin embargo, no es tan fácil implementar buenos generadores de números aleatorios. Por todo ello, en este capítulo estudiaremos la generación y el uso de números aleatorios.

Más concretamente, en este capítulo veremos:

- Cómo se generan números aleatorios.
- Cómo se generan permutaciones aleatorias.
- Cómo los números aleatorios nos permiten diseñar algoritmos eficientes, mediante el uso de una técnica conocida como *algoritmos aleatorios*.

9.1 ¿Por qué son necesarios los números aleatorios?

Los números aleatorios se emplean en numerosas aplicaciones. Esta sección discute varias de las más comunes.

Los números aleatorios tienen importantes aplicaciones en programación, entre las que se incluyen la criptografía, la simulación y la comprobación de programas.

Una importante aplicación de los números aleatorios es la comprobación de programas (*testing*). Supongamos, por ejemplo, que deseamos comprobar si los algoritmos de búsqueda diseñados en el Capítulo 8 son correctos. Es fácil probarlos tomando entradas con pocos datos, pero si deseamos comprobar su corrección para las entradas de gran tamaño para las que han sido diseñados, necesitamos suministrar un montón de datos. Generar una entrada de datos ordenada, como por ejemplo la secuencia 1, 2, ..., N, sería sencillo, pero con ello sólo comprobaríamos el buen funcionamiento de los algoritmos en un caso límite. Estos algoritmos necesitan tests generales, por medio de los cuales obtengamos comprobaciones más convincentes. Por ejemplo, podríamos probar los programas haciendo 5.000 ordenaciones sobre entradas de tamaño 1.000. Esto requiere la escritura de una rutina para generar datos arbitrarios, para lo cual se necesita a su vez, la generación de números aleatorios.

Una permutación de 1, 2, ..., *N* es una secuencia de *N* enteros que incluye cada uno de ellos exactamente una vez.

Continuando con el ejemplo, una vez que tenemos los datos generados aleatoriamente, ¿cómo podemos saber si el algoritmo ha funcionado correctamente? Una condición necesaria es que el resultado de la ordenación sea un vector ordenado en orden no decreciente, lo que puede ser comprobado en tiempo lineal. Pero, ¿cómo sabemos que los elementos de la salida son exactamente los elementos de la entrada? Una forma de comprobarlo es exigir que la entrada sea una permutación de los *N* primeros enteros. Recuérdese del Capítulo 8, que una *permutación* de 1, 2, ..., *N* es una secuencia de *N* enteros en la que cada uno de ellos aparece (exactamente) una vez. De este modo, no importa con qué permutación comencemos, el resultado de la ordenación debe ser la secuencia 1, 2, ..., *N*, lo que se comprueba fácilmente.

Además de ayudarnos a generar datos de entrada para comprobar la corrección de los programas, los números aleatorios nos son útiles para comparar la eficiencia de distintos algoritmos. Esto es debido, una vez más, al hecho de que pueden emplearse para generar multitud de entradas representativas.

Otra utilidad de los números aleatorios se encuentra en la simulación. Si deseamos conocer el tiempo medio que tarda un servidor (como el servicio de información de un banco) en procesar una secuencia de peticiones, podemos modelar el sistema empleando un computador. En esta simulación por computador, la secuencia de peticiones se genera empleando números aleatorios.

Aún queda otro uso importante de los números aleatorios, que corresponde a la técnica de los *algoritmos aleatorios*. En ellos se emplea un número aleatorio para determinar (de forma probabilística) el siguiente paso que será ejecutado por el algoritmo. La clase más común de estos algoritmos se basa en la selección (aleatoria) de una alternativa entre varias, más o menos indistinguibles. Por ejemplo, en los programas comerciales de ajedrez, el computador generalmente elige su primer movimiento de forma aleatoria, en lugar de jugar de forma determinista (es decir, en lugar de realizar siempre el mismo movimiento). Este capítulo examina muchos de los problemas que pueden resolverse de modo más eficiente empleando algoritmos aleatorios.

9.2 Generadores de números aleatorios

Los *números pseudoaleatorios* poseen muchas de las propiedades de los números aleatorios. Es difícil encontrar buenos generadores de números aleatorios.

¿Cómo se generan números aleatorios? La verdadera aleatoriedad es imposible de alcanzar en un computador, ya que los números obtenidos dependen del algoritmo empleado en su generación y esto los convierte en no aleatorios. Ahora bien, generalmente, es suficiente producir *números pseudoaleatorios*, esto es, números que *parecen* aleatorios, en el sentido de que satisfacen muchas de las propiedades que cumplen los números aleatorios. Aunque esto es mucho más fácil de decir que de conseguirlo.

Supongamos que necesitamos simular el lanzamiento de una moneda. Una forma de hacerlo es examinar el reloj del sistema. Este reloj indica el número de segundos de la hora actual. Si esta cantidad es par podemos devolver 0 (cara), y si es impar podemos devolver 1 (cruz). El problema es que esta estrategia no funciona correctamente si necesitamos generar una secuencia de números aleatorios. En tal caso, lo más probable es que se genere una secuencia de todo ceros o todo unos y, evidentemente, esto dista mucho de ser aleatorio. Un segundo es, para el computa-

dor, mucho tiempo, de modo que probablemente el reloj no cambiará mientras la secuencia está generándose. Incluso si el tiempo se mide en milisegundos y el programa está haciendo más cosas entre las llamadas al generador, la secuencia de números generada no sería aleatoria, ya que el tiempo transcurrido entre las llamadas al generador sería prácticamente idéntico en cada invocación del programa, con lo que los resultados que se producirían al realizar distintas ejecuciones serían demasiado parecidas.

En una *distribución uniforme*, todos los números en el rango especificado son equiprobables.

Lo que necesitamos es una *secuencia* de números pseudoaleatorios, es decir, una secuencia con las mismas propiedades que una secuencia aleatoria. Supongamos que necesitamos generar números aleatorios entre 0 y 999, uniformemente distribuidos. En una *distribución uniforme*, todos los números en el rango especificado tienen la misma probabilidad de ser elegidos. Muchas distribuciones pueden derivarse a partir de la distribución uniforme, por esta razón se considera en primer lugar. En el esquema de clase mostrado en la Figura 9.1 aparecen muchas de ellas. Las siguientes propiedades son verificadas por toda distribución uniforme de valores del intervalo 0 ... 999:

- El primer número puede ser cualquiera entre 0 y 999. Todos ellos son equiprobables.
- El número *i*-ésimo puede ser cualquiera entre 0 y 999. De nuevo, todos son equiprobables.
- La media de todos los números generados es 499,5.

Habitualmente es necesario generar una secuencia aleatoria en lugar de un solo número aleatorio.

Estas propiedades por sí mismas no son suficientemente restrictivas. Por ejemplo, podríamos generar el primer número examinando el reloj del sistema, cuya precisión es de un milisegundo, empleando el número de milisegundos. Los siguientes valores se irían obteniendo añadiendo una unidad al número anterior. Es claro que tras generar mil números, se verifican todas las propiedades anteriores. Sin embargo, no se cumplen otras más fuertes. Algunas de estas propiedades que también deben verificar los números aleatorios uniformemente distribuidos son las siguientes:

- La suma de dos números aleatorios generados consecutivamente debe ser par o impar con la misma probabilidad.
- Si se generan aleatoriamente mil números, algunos de ellos estarán duplicados (en concreto, aproximadamente 368 de ellos no aparecerán).

Nuestros valores no satisfacen estas propiedades. La suma de dos números consecutivos siempre es impar, y nuestra secuencia está libre de duplicaciones. Como consecuencia, nuestro generador de números pseudoaleatorios no ha pasado dos pruebas estadísticas. Es bien cierto que todos los generadores de este tipo acaban por no pasar algún test estadístico, aunque los mejores sí que pasan los tests que fallan los menos buenos. El Ejercicio 9.14 describe un test estadístico muy utilizado en la práctica.

El *generador lineal de congruencias* es un buen algoritmo para generar distribuciones uniformes.

En esta sección estudiaremos el generador uniforme más sencillo que satisface un número razonable de tests estadísticos. Desde luego no es el mejor generador posible, pero resulta apropiado para ser empleado en aplicaciones en las que se aceptan buenas aproximaciones de secuencias aleatorias. El método seguido es el *generador lineal de congruencias*, definido por primera vez en 1951. Dicho gene-

```
1  // Clase Random
2  //
3  // CONSTRUCCIÓN: con (a) ninguna inicialización o (b) un entero
4  //     que especifica el estado inicial del generador
5  //
6  // ******************OPERACIONES PÚBLICAS**********************
7  //     Devuelve un número aleatorio siguiendo una cierta
       distribución
8  // int randomInt( )                    --> Uniforme, 1 a 2^31-1
9  // double randomReal( )                --> Uniforme, 0..1
10 // int randomInt ( int linf, int lsup ) --> Uniforme linf..lsup
11 // int poisson( double valorEsperado ) --> Poisson
12 // double negExp( double valorEsperado ) --> Exponencial
13 //     Un método estático relacionado:
14 // void permute( Object [ ] a )        --> Permutación aleatoria
15
16 /**
17  * Clase de números aleatorios que emplea un
18  * generador lineal de congruencias 31-bit.
19  * Obsérvese que java.util contiene una clase Random,
20  * por lo que se deben vigilar los conflictos de nombres.
21  */
22 public class Random
23 {
24     public Random( )
25     { /* Figura 9.2 */ }
26     public Random( int valorInicial )
27     { /* Figura 9.2 */ }
28     public int randomInt( )
29     { /* Figura 9.3 */ }
30     public double randomReal( )
31     { return randomInt( ) / ( double ) M; }
32     public int randomInt( int linf, int lsup )
33     { /* Figura 9.8 */ }
34     public int poisson( double valorEsperado )
35     { /* Figura 9.5 */ }
36     public double negExp( double valorEsperado )
37     { /* Figura 9.6 */ }
38     public static final void permute( Object [ ] a )
39     { /* Figura 9.7 */ }
40
41     private int estado;
42     }
```

Figura 9.1 Esquema de una clase generadora de números aleatorios.

rador es un generador de números aleatorios en el que son generados los valores $X_1, X_2, \ldots$, verificándose

$$X_{i+1} = AX_i (\text{mod}\, M). \tag{9.1}$$

Esta ecuación indica que el $(i + 1)$-ésimo número se obtiene multiplicando el i-ésimo número por una cierta constante A, para calcular después el resto de dividir el resultado por M. En Java lo escribiríamos en la forma siguiente:

```
x[ i +1 ] = A * x[ i ] % M;
```

Los valores adecuados de las constantes A y M se concretarán en breve. Nótese que todos los números generados serán menores que M. Para iniciar la secuencia necesitamos un valor X_0. Este valor se conoce como *semilla*. Si $X_0 = 0$ la secuencia no sería aleatoria, ya que se generarían únicamente ceros. Pero si A y M son cuidadosamente elegidos, entonces cualquier semilla que cumpla $1 \leqslant X_0 < M$ es válida. En la Figura 9.2 se construye un objeto `Random` en el que se inicializa el valor de la semilla. Si M es primo entonces X_i no puede ser 0. En particular, si $M = 11$, $A = 7$ y la semilla $X_0 = 1$, los números generados son

La *semilla* es el valor inicial de un generador de números aleatorios.

7, 5, 2, 3, 10, 4, 6, 9, 8, 1, 7, 5, 2, ...

Generar un número cada segundo produce una secuencia repetitiva. En nuestro caso, la secuencia se repite después de $M - 1 = 10$ números. La longitud de la secuencia hasta que se encuentra un número generado anteriormente se conoce como *periodo* de la secuencia. El periodo obtenido con esta elección de A es, claramente, el mejor posible ya que todos los números no nulos menores que M son generados. (Necesariamente debemos tener un número repetido en la 11-ésima iteración.)

Un generador de números aleatorios con *periodo P* genera la misma secuencia de números tras P iteraciones.

Si M es primo, distintas elecciones de A permiten generar un periodo completo de $M - 1$ valores. Esta clase de generadores reciben el nombre de *generadores lineales de congruencias con periodo completo*. Otras elecciones de A no generan un periodo completo. Por ejemplo, si $A = 5$ y $X_0 = 1$, la secuencia tiene como periodo 5, al obtenerse:

Un *generador lineal de congruencias con periodo completo* tiene periodo $M - 1$.

5, 3, 4, 9, 1, 5, 3, 4, ...

Si se elige como valor de M un primo 31-bit muy alto, el periodo es suficientemente alto para la mayoría de las aplicaciones. El primo $M = 2^{31} - 1 = 2.147.483.647$

```
1  /**
2   * Construcción de un objeto Random cuyo
3   * estado inicial se obtiene del reloj del sistema.
4   */
5  public Random( )
6  {
7      this( (int) ( System.currentTimeMillis( )
8                    % Integer.MAX_VALUE ) );
9  }
10
11 /**
12  * Construcción de un objeto Random
13  * especificando un estado inicial.
14  * @param valorInicial el estado inicial.
15  */
16 public Random( int valorInicial )
17 {
18     if( valorInicial < 0 )
19         valorInicial += M;
20
21     estado = valorInicial;
22     if( estado == 0 )
23         estado = 1;
24 }
```

Figura 9.2 Constructores de la clase `Random`.

es una elección usual. Para este número, $A = 48.271$ es uno de los muchos valores que permiten obtener un generador lineal de congruencias con periodo completo. Su utilización ha sido ampliamente estudiada, siendo recomendada por los expertos en este campo. Como mostraremos más tarde en este mismo capítulo, jugar con generadores de números aleatorios significa en muchos casos romper la aleatoriedad, por lo que conviene emplear la fórmula indicada, a menos que tengamos garantizado que es posible obrar de otra forma.

Debido al desbordamiento, los cálculos deben reordenarse.

Esta rutina parece sencilla de implementar. Si `estado` representa el último valor calculado por la rutina `Random` entonces el nuevo valor de `estado` viene definido por

```
estado = ( A * estado ) % M;        // Incorrecto
```

Desafortunadamente, si este cálculo se realiza con enteros 32-bit, es casi seguro que el producto provocará un error de desbordamiento. Aunque Java tiene definido un tipo `long` 64-bit, emplearlo es computacionalmente costoso. Si nos restringimos al tipo `int` 32-bit, podríamos argumentar que el desbordamiento, una vez anulado el correspondiente error, mantiene la aleatoriedad. Sin embargo, el desbordamiento es inaceptable, pues con él perdemos la garantía del periodo completo. Afortunadamente, para evitar el desbordamiento sólo es necesaria una pequeña modificación. En concreto, si Q y R son el cociente y el resto de M/A, podemos rescribir la Ecuación 9.1 en la forma

$$X_{i+1} = A(X_i (\text{mod } Q)) - R\lfloor X_i/Q \rfloor + M\delta(X_i) \qquad \textbf{(9.2)}$$

con lo que se verifica lo siguiente (*véase* el Ejercicio 9.5):

- El primer término siempre puede evaluarse sin desbordamiento.
- El segundo término puede evaluarse sin desbordamiento, siempre que $R < Q$.
- $\delta(X_i)$ se evalúa a 0 si el resultado de la diferencia entre los dos primeros términos es positiva; se evalúa a 1 si dicha diferencia es negativa.

Debemos ceñirnos a esta elección de *A* y *M*, a menos que conozcamos otra mejor.

Para los valores de M y A fijados, se tiene que $Q = 44.488$ y $R = 3.399$. Como consecuencia, $R < Q$ y una aplicación directa de (9.2) nos da una implementación de una clase de números aleatorios. El código que resulta se muestra en la Figura 9.3. En ella, la rutina `randomInt` devuelve el valor de `estado`.

En la Figura 9.1 aparecen dos métodos adicionales: uno de ellos genera aleatoriamente un real en el intervalo de 0 a 1, mientras que el segundo genera un entero aleatorio en el intervalo cerrado que se especifica como argumento (*véase* el Ejercicio 9.8).

Por último, la clase también incluye un generador para el caso en el que se necesiten números aleatorios no uniformemente distribuidos. Al respecto, en la Sección 9.3 se muestra el código de los métodos `poisson` y `negExp`.

Se puede estar tentado a asumir que cualquier sistema computacional tiene un generador de números aleatorios al menos tan bueno como el de la Figura 9.3. Tristemente, esto no es cierto. Muchas librerías poseen generadores basados en la función

$$X_{i+1} = (AX_i + C) \text{mod } 2^B$$

```
1  private static final int A = 48271;
2  private static final int M = 2147483647;
3  private static final int Q = M / A;
4  private static final int R = M % A;
5
6  /**
7   * Devuelve un entero pseudoaleatorio y cambia el
8   * estado interno.
9   * @return un entero pseudoaleatorio.
10  */
11 public int randomInt( )
12 {
13     int estadoTmp = A * ( estado % Q ) - R * ( estado / Q );
14     if( estadoTmp >= 0 )
15         estado = estadoTmp;
16     else
17         estado = estadoTmp + M;
18
19     return estado;
20 }
```

Figura 9.3 Generador de números aleatorios.

donde B es el número de bits de los enteros de la máquina y C es impar. Estas librerías, al igual que la rutina de la Figura 9.3, también devuelven directamente el nuevo valor calculado en `estado`, en lugar de (por ejemplo) un valor entre 0 y 1. Desafortunadamente, estos generadores siempre producen valores de X_i que se alternan entre pares e impares —propiedad indeseable en un generador—. Más en general, cada sufijo de k bits cicla con un periodo de 2^k, en el mejor de los casos. Muchos otros generadores de números aleatorios tienen ciclos bastante menores que el mostrado aquí. Éstos no son adecuados en aplicaciones que requieran secuencias largas de números aleatorios. La librería de Java tiene un generador de este tipo; sin embargo, emplea un generador lineal de congruencias de 48 bits, devolviendo sólo los primeros 32 bits, evitando así el problema que representa la ciclicidad de los bits de menor orden. En concreto, las constantes que utiliza son $A = 25.214.903.917$, $B = 48$ y $C = 13$[1].

Finalmente, podría parecer que es posible obtener un mejor generador de números aleatorios añadiendo una constante a la ecuación. Por ejemplo, podría pensarse que

$$X_{i+1} = (48.271X_i + 1) \bmod (2^{31} - 1)$$

generará valores más aleatorios. Sin embargo, cuando usamos esta ecuación podemos observar que

$$(48.271 \cdot 179.424.105 + 1) \bmod (2^{31} - 1) = 179.424.105.$$

De modo que, si la semilla es 179.424.105, el generador se bloquea en un ciclo de periodo 1. Esto muestra lo frágiles que pueden ser estos generadores.

[1] Éste es el mismo generador que el conocido como `drand48`, existente en los sistemas Unix.

9.3 Números aleatorios no uniformes

No todas las aplicaciones requieren números aleatorios uniformemente distribuidos. Por ejemplo, las calificaciones de los alumnos de un curso numeroso no suelen distribuirse de manera uniforme, sino que se ajustan a la clásica distribución en forma de campana, conocida formalmente como *distribución normal* o *de Gauss*. Un generador uniforme de números aleatorios puede emplearse para generar números que satisfacen otro tipo de distribuciones.

La *distribución de Poisson* modela el número de veces que se produce un evento extraño y suele emplearse en simulación.

Una importante distribución no uniforme que se produce durante las simulaciones es la *distribución de Poisson*. Los sucesos que se producen bajo las siguientes circunstancias satisfacen la distribución de Poisson:

- La probabilidad de un suceso en una región pequeña es proporcional al tamaño de la región.
- La probabilidad de dos sucesos en una región pequeña es proporcional al cuadrado del tamaño de la región, con lo que normalmente es lo suficientemente pequeña como para poder ser ignorada.
- Los hechos consistentes en obtener k sucesos en una región determinada y obtener j sucesos en una región disjunta a la anterior son independientes. (Técnicamente, esto quiere decir que la probabilidad de que ambos sucesos se produzcan simultáneamente se calcula multiplicando las probabilidades de los eventos individuales.)
- La cantidad media de sucesos en una región de un tamaño dado es conocida.

Entonces, si el número medio de sucesos viene dado por la constante a, la probabilidad de que se presenten exactamente k sucesos es $a^k e^{-a}/k!$.

La distribución de Poisson normalmente se aplica a sucesos que tienen una probabilidad reducida de producirse. Por ejemplo, considérese el evento de tener un boleto ganador de la lotería primitiva, en el que la probabilidad de ganar el bote es de 1 entre 14 millones. Supondremos que los números escogidos al rellenar un boleto son, más o menos, aleatorios e independientes. Si una persona rellena 100 boletos, sus probabilidades de ganar son ahora de 1 entre 140.000 (han mejorado en un factor de 100), de modo que se verifica la primera condición. La probabilidad de que una persona tenga dos boletos ganadores es casi nula, por lo que también se cumple la segunda condición. Si alguna otra persona compra 10 boletos, sus probabilidades de ganar son de 1 entre 1.400.000 , y éstas son independientes de la primera persona, de modo que la tercera propiedad también se cumple. Supóngase que se venden 28 millones de boletos. La cantidad media de boletos ganadores en esta situación es 2 (ésta es la cifra que necesitamos para satisfacer la última condición). El número previsible de boletos ganadores es una variable aleatoria con un valor esperado de 2, que satisface la distribución de Poisson. De este modo, la probabilidad de que se hayan vendido exactamente k boletos

Boletos ganadores	0	1	2	3	4	5
Frecuencia	0,135	0,271	0,271	0,180	0,090	0,036

Figura 9.4 Distribución de los ganadores de la lotería cuando el número esperado de ganadores es 2.

ganadores es $2^k e^{-2}/k!$. Esto nos da la distribución de la Figura 9.4. En general, si el número esperado de boletos ganadores es a, entonces la probabilidad de que aparezcan k boletos ganadores es $a^k e^{-a}/k!$.

Para generar de forma aleatoria un entero siguiendo una distribución de Poisson con media a, podemos seguir la siguiente estrategia (su justificación matemática queda fuera de los objetivos del texto): se generan repetidamente números aleatorios en el intervalo (0, 1) distribuidos de manera uniforme hasta que su producto sea menor (o igual) que e^{-a}. Esto es lo que se hace en la rutina de la Figura 9.5.

Otra importante distribución no uniforme es la *distribución exponencial*, generada en la Figura 9.6. En esta distribución la media y la varianza coinciden. La distribución exponencial se emplea para modelar el tiempo que transcurre entre dos eventos aleatorios independientes, como en el ejemplo de simulación de la Sección 13.2.

En la *distribución exponencial* la media y la varianza coinciden. Se emplea para modelar el tiempo que transcurre entre dos eventos aleatorios independientes.

Muchas otras distribuciones aparecen con frecuencia. Hemos tratado aquí de mostrar como una mayoría de ellas pueden obtenerse a partir de la distribución uniforme. Puede consultarse cualquier libro sobre probabilidades y estadística para conocer más detalles acerca de estas distribuciones.

```
1  /**
2   * Devuelve un entero usando una distribución de Poisson y
3   * cambia el estado interno.
4   * @param valorEsperado la media de la distribución.
5   * @return el int pseudoaleatorio.
6   */
7  public int poisson( double valorEsperado )
8  {
9      double limite = -valorEsperado;
10     double producto = Math.log( randomReal( ) );
11     int contador;
12
13     for( contador = 0; producto > limite; contador++ )
14         producto += Math.log( randomReal( ) );
15
16     return contador;
17 }
```

Figura 9.5 Generación de un número aleatorio según la distribución de Poisson.

```
1  /**
2   * Devuelve un valor de tipo double que sigue una distribución
3   * exponencial y cambia el estado interno.
4   * @param valorEsperado la media de la distribución.
5   * @return el double pseudoaleatorio.
6   */
7  public double negExp( double valorEsperado )
8  {
9      return - valorEsperado * Math.log( randomReal( ) );
10 }
```

Figura 9.6 Generación de un número aleatorio siguiendo una distribución exponencial.

9.4 Generación de una permutación aleatoria

Considérese el problema de simular un juego de cartas. La baraja tiene 52 cartas distintas. En el transcurso de una partida debemos generar cartas de la baraja sin repetir ninguna. Esto corresponde al proceso de barajar las cartas para después repartir cartas del mazo resultante. Naturalmente deseamos que el barajeo de las cartas sea justo, es decir, que cada una de las 52! posibles combinaciones tenga la misma probabilidad de aparecer tras barajar las cartas.

Las *permutaciones aleatorias* pueden generarse en tiempo lineal empleando un valor aleatorio para generar cada elemento.

Este tipo de problemas precisa la generación de *permutaciones aleatorias*. Una permutación aleatoria emplea números aleatorios distintos para cada elemento. En general, el problema es el siguiente: generar una permutación aleatoria de 1, 2, ..., N. Todas las permutaciones deben ser equiprobables. La aleatoriedad de la permutación está limitada por la aleatoriedad del generador de números pseudoaleatorios. De este modo, que todas las permutaciones sean igualmente probables depende de que todos los números generados sean independientes y se distribuyan uniformemente. Las permutaciones aleatorias pueden generarse en tiempo lineal.

En la Figura 9.7 se muestra una rutina, `permute`, que sirve para generar una permutación aleatoria. Al efecto procedemos del siguiente modo: partimos de un vector en el que aparecen los elementos de 1, ..., N en un orden cualquiera. El bucle realiza la mezcla aleatoria. En cada iteración se intercambia `a[ j ]` con un cierto elemento del vector entre las posiciones 0 y `j` (en particular es posible que no se realice ningún intercambio, al escogerse el elemento `j`-ésimo). Tras ejecutar la rutina se genera una permutación aleatoria de 1, 2, ..., N.

La corrección de `permute` es sutil.

Es evidente que `permute` genera permutaciones arbitrarias. Pero, ¿todas las permutaciones son igualmente probables? La respuesta es sí y no, a la vez. Si nos basamos en el algoritmo la respuesta es sí. Existen $N!$ permutaciones posibles, y el número de posibles salidas diferentes de las $N - 1$ llamadas a `randomInt` en la linea 11 es también $N!$. Esto es debido a que la primera llamada produce 0 o 1, luego tiene dos salidas. La segunda llamada produce 0, 1 o 2, luego tiene tres salidas. La llamada i-ésima tiene N salidas. El número total de salidas es el producto de todas estas posibilidades, ya que cada número aleatorio es independiente del resto. En consecuencia, basta demostrar que cada secuencia de números aleatorios corresponde a una y sólo una permutación. Esto puede demostrarse razonando *hacia atrás* (*véase* el Ejercicio 9.6.)

Sin embargo la respuesta real es no, pues todas las permutaciones no son equiprobables. Como sólo hay $2^{31} - 2$ estados iniciales posibles para el generador de

```
1  /**
2   * Reordena un vector de forma aleatoria.
3   * Los números aleatorios empleados dependen de la hora y el día.
4   * @param a el vector.
5   */
6  public static final void permute( Object [ ] a )
7  {
8      Random r = new Random( );
9
10     for( int j = 1; j < a.length; j++ )
11         Ordenacion.intercambioRef( a, j, r.randomInt( 0, j ) );
12 }
```

Figura 9.7 Rutina para generar permutaciones aleatorias.

números aleatorios, sólo pueden generarse $2^{31} - 2$ permutaciones distintas. Esto podría representar un problema en algunas aplicaciones. Por ejemplo, un programa que generara 1.000.000 permutaciones (dividiendo quizás el trabajo entre varios computadores) para medir la eficiencia de un algoritmo de ordenación, generará, con total seguridad, algunas permutaciones dos veces. Se necesitan generadores de números aleatorios mejores para hacer que la teoría y la práctica coincidan absolutamente.

Observe que substituir la llamada al método de intercambiar por la llamada `r.randomInt(0,a.length-1)` no funciona, incluso en el caso de tener tres elementos. Existen $3! = 6$ posibles permutaciones y el número de secuencias diferentes que puede calcularse con las tres llamadas de `randomInt` es $3^3 = 27$. Como 6 no es divisor de 27, unas permutaciones tienen mayor probabilidad de aparecer que otras.

9.5 Algoritmos aleatorios

Suponga que es un profesor que da clases de programación cada semana. Quiere asegurarse de que los estudiantes están escribiendo sus propios programas o, al menos, que entienden el código que presentan. Una solución es preguntar a los alumnos acerca de los programas cada día de clase. Sin embargo, estas preguntas reducen el tiempo disponible, por lo que sólo sería práctico cuestionar la mitad de los programas. Su problema es decidir cuándo preguntar a los estudiantes.

Si avisa con antelación de que va a preguntar, los alumnos tendrán la posibilidad de hacer trampas no realizando el 50 por ciento de los programas que no van a ser examinados. Alternativamente puede adoptar la estrategia de examinar un programa sí y otro no, sin avisar previamente, pero los alumnos podrían descifrar dicha estrategia rápidamente. Otra posibilidad es examinar a los alumnos sobre los programas que parecen más importantes, pero esto probablemente conducirá a patrones similares de año en año, por lo que de nuevo la estrategia resultaría predecible a partir del primer año.

Un método que parece eliminar estos problemas es lanzar una moneda. Es decir, hace un cuestionario para cada programa (redactar las preguntas no consume tanto tiempo como corregirlas), y al comienzo de la clase lanza una moneda para decidir si va a haber examen. De esta manera es imposible saber antes de cada clase si ese día habrá cuestionario. Además, este patrón no se repite cada año. Los estudiantes pueden ser examinados con un 50 por ciento de posibilidades, independientemente de lo sucedido antes. La desventaja de esta estrategia es que el año podría acabar sin que haya habido ningún examen. Aunque si se asume que los estudiantes tienen que realizar muchos programas, esto es improbable, a menos que la moneda esté trucada. Cada año el número esperado de exámenes es la mitad del número de programas implementados, y con una elevada probabilidad, el número de exámenes realizados será muy parecido a dicha cantidad.

Este ejemplo ilustra los llamados *algoritmos aleatorios*. Durante su ejecución se emplea, en alguna ocasión, un número aleatorio para tomar una decisión, en lugar de tomarlas siempre de forma determinista. La duración de la ejecución del algoritmo depende no sólo de los datos de entrada sino también de los números aletorios generados.

Los *algoritmos aleatorios* emplean números aleatorios para tomar decisiones.

La duración de la ejecución de un algoritmo aleatorio depende tanto de los números aleatorios generados como de los datos de entrada.

En el peor de los casos la duración de la ejecución es, casi siempre, la misma que la que la del algoritmo no aleatorizado. La diferencia más importante es que un algoritmo bien aleatorizado no tiene datos de entrada malos sino números aleatorios (para cada entrada en particular) inadecuados. Esto parece sólo una diferencia teórica, pero tiene una gran importancia, tal y como muestra el siguiente ejemplo.

Considérese el problema siguiente. Su jefe le pide escribir un programa que calcule la media de un conjunto de 1.000.000 de números. Debe entregar el programa y ejecutarlo con los datos de entrada que el jefe decida. Si aparece la respuesta correcta en unos pocos segundos (lo que se espera de un algoritmo lineal), su jefe estará muy satisfecho por el trabajo realizado y le premiará de algún modo. Pero si su programa no funciona o es demasiado lento, su jefe le despedirá por incompetente. Su jefe piensa que gana demasiado y está deseando tomar la segunda opción. ¿Qué debería hacer?

El algoritmo de selección rápida descrito en la Sección 8.7 parece ser la mejor solución al problema. Aunque el algoritmo (*véase* la Figura 8.20) es muy rápido en media, es cuadrático en el caso peor si el pivote es pobre. Usando la mediana de tres, tenemos garantizado que este caso peor no se dará nunca para los datos de entrada más comunes, al igual que para aquellos que ya están ordenados o tienen algún elemento duplicado. Sin embargo, aún existe un caso peor cuadrático mostrado en el Ejercicio 8.8. De modo que su malévolo jefe tras leer su programa, podría ver cómo se elige el pivote, y será capaz de construir el caso peor. Como consecuencia, será despedido.

La selección aleatoria rápida funciona (en media) en tiempo lineal.

Empleando números aleatorios puede garantizar estadísticamente la seguridad de su trabajo. Puede comenzar el algoritmo de selección rápida mezclando aleatoriamente la entrada empleando el programa de la Figura 9.7[2]. Como consecuencia, su jefe ha perdido la capacidad de especificar una mala secuencia de entrada. Cuando ejecute el algoritmo de selección rápida, éste procesará una entrada aleatoria, por lo que es de esperar un tiempo lineal en su ejecución. ¿Puede durar un tiempo cuadrático? La respuesta es sí. Es posible que la mezcla de cualquier entrada inicial devuelva el peor caso para la selección rápida, por lo que la secuencia resultante sería ordenada en tiempo cuadrático. Si es lo suficientemente desafortunado como para que suceda esto, perderá su trabajo (¡aunque su jefe no sea en principio tan malicioso!). Sin embargo, este suceso es estadísticamente imposible: para un millón de elementos, la probabilidad de tener que emplear tan sólo el doble del tiempo del que indica la media es tan pequeña que puede ser ignorada. Es mucho, pero que mucho más probable que su computador se equivoque o se estropee inoportunamente. Así que su puesto de trabajo estaría seguro.

En lugar de mezclar los datos, podemos obtener un resultado equivalente eligiendo el pivote de forma aleatoria en lugar de hacerlo de forma determinista. Elegimos aleatoriamente un elemento del vector y lo intercambiamos con el elemento en la posición `linf`. Elegimos de forma aleatoria otro elemento y lo intercambiamos con el elemento en la posición `lsup`. Escogemos un tercer elemento y lo intercambiamos con el situado en la posición media. Después el algoritmo continúa como de ordinario. Como en el algoritmo original, las particiones degeneradas

[2] Es preciso asegurarse de que el generador de números aleatorios es lo suficientemente aleatorio como para que su salida no pueda ser predicha en absoluto por su jefe.

siguen siendo posibles, pero ahora aparecen como resultado de números aleatorios inadecuados, y no por culpa de malas entradas.

Comentamos a continuación las principales diferencias entre los algoritmos aleatorizados y los no aleatorizados. Hasta ahora hemos venido estudiando los algoritmos no aleatorizados. Cuando calculamos su tiempo medio de ejecución, suponemos que todas las entradas son equiprobables. Sin embargo, esta suposición podría no ser cierta ya que, por ejemplo, las entradas casi ordenadas son más habituales de lo esperado estadísticamente. Esto puede causar problemas para algunos algoritmos como el quicksort. Pero usando un algoritmo aleatorizado el tener unos datos de entrada u otros no es tan importante. En este caso lo que sí es importante es poder contar con verdaderos números aleatorios, ya que para obtener una *estimación* del tiempo de ejecución se hace una media de todos los comportamientos posibles correspondientes a los números aleatorios asociados a cualquier entrada. Empleando la selección rápida con elección aleatoria de pivotes (o un paso de preprocesamiento de la entrada) obtenemos un algoritmo $O(N)$. Esto significa que para cualquier entrada, incluyendo aquella que ya está ordenada, el tiempo de ejecución estimado, basándonos en el comportamiento estadístico de los números aleatorios, es $O(N)$. Una cota del tiempo estimado será habitualmente algo mayor que una cota del tiempo en el caso promedio ya que las suposiciones hechas para generar los números aleatorios son más débiles (números aleatorios versus entrada aleatoria), pero será menor que la correspondiente cota en el caso peor. Por otra parte, en muchas ocasiones las soluciones que tienen buenas cotas en el caso peor exigen con frecuencia un trabajo añadido para asegurar que no se produzca el caso peor. El algoritmo de selección con coste $O(N)$ en el caso peor es un resultado teórico formulable pero poco útil en la práctica.

Algunos algoritmos aleatorios se ejecutan en una cantidad fija de tiempo, pero cometen errores de forma aleatoria (presumiblemente con una baja probabilidad). Estos errores se clasifican en *falsos positivos* y *falsos negativos*.

Los algoritmos aleatorios se clasifican en dos grandes familias. Los primeros, como el arriba estudiado, siempre devuelven la respuesta correcta, aunque pueden tardar en hacerlo, dependiendo de los números aleatorios generados. Los segundos son los que estudiaremos en el resto de este capítulo. Este segundo tipo de algoritmos aleatorios se ejecutan en una cantidad fija de tiempo pero pueden cometer errores de forma aleatoria (presumiblemente con una probabilidad reducida). Estos errores se clasifican en *falsos positivos* o *falsos negativos*. Ésta es una técnica ampliamente aceptada en medicina. Los falsos positivos y los falsos negativos son bastante comunes: algunos tests médicos tienen tasas de error sorprendentemente elevadas. Más aún, para algunos tests los errores dependen de las características del individuo y no de las circunstancias aleatorias generadas, por lo que repetir el test producirá un nuevo resultado erróneo. En los algoritmos aleatorios podemos repetir el test con la misma entrada usando números aleatorios diferentes. Si ejecutamos el algoritmo diez veces y obtenemos diez positivos, y si obtener un solo falso positivo es un suceso improbable (por ejemplo una vez de cada cien), entonces la probabilidad de obtener consecutivamente diez positivos falsos (una vez en 100^{10}, o sea cien trillones) es prácticamente nula.

9.6 Test aleatorio de primalidad

Recordemos que en la Sección 7.4 se han descrito muchos algoritmos numéricos que vimos cómo pueden emplearse para implementar el esquema de encriptación

basado en el algoritmo RSA. Un paso importante en el algoritmo RSA es la producción de dos números primos p y q. Podemos encontrar un número primo probando reiteradamente números impares sucesivos hasta encontrar uno primo. Así que el punto crucial del algoritmo es comprobar si un número dado es primo.

El algoritmo más sencillo para comprobar si un número impar N es primo consiste en estudiar su *divisibilidad*. En este algoritmo, se utiliza el hecho de que un número mayor que 3 es primo si no es divisible entre ningún otro impar menor o igual que $\sqrt{N}$. En la Figura 9.8 se implementa directamente este método.

El algoritmo de la *divisibilidad* es rápido para números pequeños (32-bit), pero no puede emplearse para números grandes.

El test de la divisibilidad es razonablemente rápido para números pequeños (32-bit), pero no se puede emplear ni siquiera para el tipo `long` 64-bit, pues se necesitan probar cerca de $\sqrt{N}/2$ divisores, empleándose para ello una cantidad de tiempo del orden $O(\sqrt{N})$. Lo que deseamos es un test del mismo orden de magnitud que la rutina `potencia` de la Sección 7.4.2. Un teorema muy conocido que puede llevarnos a resolver la cuestión es el *Teorema pequeño de Fermat*. (En aras de la completitud del presente texto el resultado se demuestra en el Teorema 9.1, aunque no es necesario para entender la demostración del test de primalidad.)

Teorema 9.1 *(Teorema pequeño de Fermat): Si P es primo y $0 < A < P$ entonces $A^{P-1} \equiv 1 (\text{mod } P)$.*

Demostración *Considérese un k cualquiera verificando $1 \leqslant k < P$. Como P es primo y menor que A y k, $Ak \equiv 0 (\text{mod } P)$ es imposible. Considérese ahora cualquier $1 \leqslant i < j < P$. $Ai \equiv Aj (\text{mod } P)$ implicaría $A(j - i) \equiv 0 (\text{mod } P)$ pero esto es imposible por el argumento anterior, ya que $1 \leqslant j - i < P$. Además, cuando trabajamos* mod P, *la secuencia A, $2A$, ..., $(P - 1)A$ es una permutación de $1, 2, ..., P - 1$. El producto de ambas secuencias* (mod P) *debe ser equivalente, derivándose entonces la equivalencia $A^{P-1}(P - 1)! \equiv (P - 1)! (\text{mod } P)$, de la que se sigue el teorema.*

El *Teorema pequeño de Fermat* nos da una condición necesaria pero no suficiente para establecer la primalidad de un número.

Si el recíproco del *Teorema pequeño de Fermat* fuese cierto, entonces tendríamos un test de primalidad computacionalmente equivalente a la exponenciación modular (esto es, $O(\log N)$). Desafortunadamente, el resultado recíproco no es cierto. Se comprueba fácilmente que $2^{340} \equiv 1 (\text{mod } 341)$, pero 341 es factorizable (11×31).

```
1  // Devuelve true si el número impar n es primo
2
3  public static boolean esPrimo( long n )
4  {
5      for( long i = 3; i * i <= n; i += 2 )
6          if( n % i == 0 )
7              return false;     // No primo
8
9      return true;              // Primo
10 }
```

Figura 9.8 Test de primalidad basado en la divisibilidad.

Para obtener un test de primalidad necesitamos un teorema adicional: el Teorema 9.2.

Teorema 9.2

Si P es primo y $X^2 \equiv 1 (\mathrm{mod}\, P)$ entonces $X \equiv \pm 1 (\mathrm{mod}\, P)$.

Demostración

Como $X^2 - 1 \equiv 0 (\mathrm{mod}\, P)$ implica $(X - 1)(X + 1) \equiv 0 (\mathrm{mod}\, P)$ y P es primo, $X - 1$ o $X + 1 \equiv 0 (\mathrm{mod}\, P)$.

Una combinación de los Teoremas 9.1 y 9.2 resulta útil. Sea A un entero entre 2 y $N - 2$. Si calculamos $A^{N-1} (\mathrm{mod}\, N)$ y el resultado no es 1, entonces N no puede ser primo, ya que de otro modo se contradice el Teorema pequeño de Fermat. Decimos entonces que A es un *testigo* de la factorizabilidad de N, ya que A demuestra que N es factorizable. Cada número compuesto N tiene algún testigo A, pero para algunos números, llamados *números de Carmichael*, estos testigos son difíciles de encontrar. Necesitamos tener la seguridad de que existe una elevada probabilidad de encontrar un testigo, sea cual sea la elección de N. Para mejorar nuestras posibilidades aplicamos el Teorema 9.2.

Durante el cálculo de A^i también calculamos $(A^{\lfloor i/2 \rfloor})^2$. Así tenemos $X = A^{\lfloor i/2 \rfloor}$ y $Y = X^2$. Obsérvese que X e Y se calculan automáticamente, como parte de la rutina `potencia`. Si Y es 1 y X no es $\pm 1 (\mathrm{mod}\, N)$, entonces, por el Teorema 9.2, N no puede ser primo. Podemos devolver 0 como el valor de A^i cuando esto se detecte. Entonces N habrá fallado el test de primalidad, basándonos en el Teorema pequeño de Fermat.

La rutina `testigo`, mostrada en la Figura 9.9, calcula $A^i (\mathrm{mod}\, P)$, salvo que se devuelva 0 cuando se detecta la contradicción del Teorema 9.2[3]. Si `testigo` no devuelve 1, entonces A es un testigo del hecho de que N no puede ser primo. En las líneas desde la 12 hasta la 14 se hace una llamada recursiva y se produce X. Entonces calculamos X^2, como en una computación normal de `potencia`. Comprobamos si el Teorema 9.2 se contradice, y se devuelve 0 si es así. En cualquier otro caso se completa la ejecución de `potencia`.

Si el algoritmo decide que un número no es primo, no lo es con una certeza del 100 por cien. Cada intento aleatorio tiene, a lo sumo, una tasa de falsos positivos del 25 por ciento.

El único tema pendiente es la corrección. Si nuestro algoritmo decide que N es factorizable, entonces N debe serlo. Si N es factorizable, ¿es cierto que toda A con $2 \leqslant A \leqslant N - 2$ es un testigo? Desafortunadamente, la respuesta es no. Esto significa que existen algunos valores de A que pueden falsear nuestro algoritmo haciéndonos decir que N es primo. De hecho, si se elige A de forma aleatoria, tenemos una probabilidad de 1/4 de fallar en la detección de un número factorizable, cometiendo así un error. Obsérvese que esto es cierto para cualquier N. Si sólo se tuviera esta conclusión en media para los distintos valores de N, entonces no podríamos obtener una rutina lo suficientemente buena. Si recuperamos la analogía con los tests médicos, nuestro algoritmo genera falsos positivos para cualquier N, a lo sumo el 25 por cierto del tiempo.

[3] Obsérvese que este pseudocódigo no funciona para `longs` grandes debido a los posibles errores de desbordamiento. Debemos forzar que los números testeados sean 32-bit.

```
1  /**
2   * Método que implementa el test básico de primalidad.
3   * Si testigo devuelve 1, n es factorizable.
4   * Para ello se calcula a^i (mod n), buscándose a la vez
5   * las raíces no triviales de 1.
6   */
7  private static long testigo( long a, long i, long n )
8  {
9      if( i == 0 )
10         return 1;
11
12     long x = testigo( a, i / 2, n );
13     if( x == 0 )   // Si n es recursivamente factorizable, parar
14        return 0;
15
16     // n no es primo si encontramos una raíz no trivial de 1
17     long y = ( x * x ) % n;
18     if( y == 1 && x != 1 && x != n - 1 )
19         return 0;
20
21     if( i % 2 != 0 )
22         y = ( a * y ) % n;
23     return y;
24 }
25
26 public static final int INTENTOS = 5;
27
28 /**
29  * Test aleatorio de primalidad.
30  * El ajuste de INTENTOS aumenta el grado de confianza.
31  * @param n el número a testear.
32  * @return false si n no es primo (con seguridad).
33  *      true si n seguramente es primo.
34  */
35 public static boolean esPrimo( long n )
36 {
37     Random r = new Random( );
38
39     for( int contador = 0; contador < INTENTOS; contador++ )
40        if( testigo( r.randomInt( 2, (int) n - 2 ),
41                                          n - 1, n ) != 1 )
42            return false;
43
44     return true;
45 }
```

Figura 9.9 Test aleatorio de primalidad.

Algunos números compuestos pasarán el test y serán declarados primos. Pero es muy improbable que un número compuesto pase 20 tests aleatorios independientes.

Esto no parece una estadística muy buena, ya que una tasa de error del 25 por ciento es ciertamente muy alta. Sin embargo, si usamos 20 valores independientes de A, entonces la probabilidad de que ninguno de ellos sea testigo de un número factorizable es $1/4^{20}$, que es cerca de una entre un millón de millones. Este grado de certeza es mucho más razonable, y además podría mejorarse tanto como deseásemos realizando más intentos. La rutina `esPrimo`, mostrada en la Figura 9.9, se limita a hacer cinco intentos.

Resumen

Este capítulo describe cómo se generan y utilizan los números aleatorios. El generador lineal de congruencias es una buena elección para aplicaciones sencillas, siempre que se elijan con cuidado los parámetros A y M. Podemos obtener números aleatorios que sigan distribuciones no uniformes, como la distribución de Poisson o distribuciones exponenciales, a partir de un generador de números aleatorios uniforme.

Los números aleatorios tienen múltiples aplicaciones. Algunas de ellas incluyen el estudio empírico de algoritmos, la simulación de sistemas de la vida real y el diseño de algoritmos que eviten de forma probabilística el caso peor. Los números aleatorios se emplean en otras partes del texto, especialmente en la Sección 13.2 y el Ejercicio 20.18.

Con este capítulo se concluye la Parte II del libro. La Parte III muestra algunas aplicaciones sencillas, comenzando con una discusión acerca de los juegos en el Capítulo 10, en el que ilustramos tres técnicas importantes de resolución de problemas.

Elementos del juego

falsos positivos y falsos negativos Errores cometidos aleatoriamente (con baja probabilidad) por algunos algoritmos aleatorios de duración fija.

algoritmo aleatorio Algoritmo que emplea números aleatorios en lugar de decisiones deterministas, para decidir el siguiente paso de la ejecución.

distribución de Poisson Distribución que modela el número de veces que se produce un suceso raro.

distribución exponencial Distribución empleada para modelar el tiempo transcurrido entre dos ocurrencias de sucesos aleatorios. Su media y su varianza coinciden.

distribución uniforme Distribución en la que todos los números dentro del rango especificado son equiprobables.

generador lineal de congruencias Un buen algoritmo para generar distribuciones uniformes.

generador lineal de congruencias con periodo completo Generador lineal de congruencias cuyo periodo es $M - 1$.

números pseudoaleatorios Números que verifican muchas de las propiedades de los números aleatorios. Es difícil encontrar buenos generadores de números pseudoaleatorios.

periodo Un generador de números aleatorios con periodo P genera la misma secuencia aleatoria después de P iteraciones.

permutación aleatoria Recolocación aleatoria de N elementos. Puede generarse en tiempo lineal generando un número aleatorio por cada elemento.

permutación Una permutación de 1, 2, ..., N es una secuencia de N enteros que incluye cada elemento de 1, 2, ..., N exactamente una vez.

semilla Es el estado inicial de un generador de números aleatorios.

Teorema pequeño de Fermat Establece que si P es primo y $0 < A < P$ entonces $A^{P-1} \equiv 1 (\text{mod}\, P)$. Es una condición necesaria, pero no suficiente, para establecer la primalidad de un número.

testigo de composicionalidad Valor de A que prueba que un número no es primo, empleando el Teorema pequeño de Fermat.

trial división Algoritmo sencillo para comprobar la primalidad de un número. Es rápido para números pequeños (32-bit) pero no puede aplicarse para números grandes.

Errores comunes

1. La utilización de una semilla nula producirá números aleatorios inadecuados.
2. Los usuarios sin experiencia podrían reinicializar la semilla con un cierto valor prefijado antes de generar una nueva permutación aleatoria. Esto provocaría que se generara la misma permutación en distintas ocasiones, lo que probablemente no se pretendía.
3. Muchos generadores de números aleatorios son particularmente malos. En algunas aplicaciones que requieren largas secuencias de números aleatorios, incluso el generador lineal de congruencias es insatisfactorio.
4. Se sabe que, bajo ciertos criterios de aleatoriedad, los bits de menor orden de los generadores lineales de congruencias no son aleatorios. En consecuencia, debe evitarse su uso. Por ejemplo, `randomInt()%2` es una mala táctica para simular el lanzamiento de una moneda.
5. Cuando se generan números aleatorios en un intervalo determinado, un error muy común es manejar valores ligeramente fuera de los límites e incluso permitir la generación de valores fuera del intervalo, o no permitir que el valor más pequeño se genere con probabilidad ajustada.
6. Muchos generadores de permutaciones aleatorias no generan todas las permutaciones posibles con la misma probabilidad. Como se ha comentado en el texto, nuestro algoritmo está limitado por el generador de números aleatorios.
7. Cambiar los valores de un generador de números aleatorios equivale, muy probablemente, a debilitar sus propiedades estadísticas.

En Internet

Todo el código que aparece en este capítulo forma parte del paquete `Supporting` y puede encontrarse en el directorio **Supporting**. Éstos son los nombres de los ficheros:

Random.java	Contiene la clase `Random`.
Numerical.java	Contiene la rutina de test de primalidad de la Figura 9.9 además de las rutinas matemáticas de la Sección 7.4.

Ejercicios

Cuestiones breves

9.1. Con el generador de números aleatorios descrito en el texto, determine los primeros 10 valores de `estado`, suponiendo que su valor inicial es 1.

9.2. Muestre el resultado de ejecutar el algoritmo del test de primalidad para $N = 561$, con valores de A entre 2 y 5 (ambos inclusive).

9.3. Si se venden 42.000.000 boletos de la lotería primitiva, ¿cuál es el número esperado de ganadores? ¿Qué probabilidad hay de que el premio quede desierto?, ¿y las probabilidades de que haya exactamente un ganador?

9.4. ¿Por qué no puede emplearse una semilla nula en el generador lineal de congruencias?

Problemas teóricos

9.5. Demuestre que la Ecuación 9.2 es equivalente a la Ecuación 9.1 y que el programa resultante, presentado en la Figura 9.3, es correcto.

9.6. Complete la prueba de que cada permutación obtenida en la Figura 9.7 es equiprobable.

9.7. Supóngamos que tenemos una moneda sesgada, de modo que al lanzarla la probabilidad de que salga cara es p y la probabilidad de que salga cruz es $1 - p$. Muestre cómo podría diseñarse un algoritmo que emplee dicha moneda para generar un 0 o un 1, con la misma probabilidad.

Problemas prácticos

9.8. Escriba los métodos `randomReal` y `randomInt`. Después, implemente un programa que llame a `randomInt` 1.000 veces para generar números entre 1 y 1.000. ¿Satisface los tests estadísticos más fuertes de la Sección 9.2?

9.9. Ejecute un millón de veces el generador de Poisson de la Figura 9.5 tomando como valor esperado 2. ¿Concuerda la distribución obtenida con la Figura 9.4?

9.10. Considere una elección entre dos candidatos en la que el ganador ha obtenido una fracción p del voto. Si el recuento de votos se realiza secuencialmente, ¿cuál es la probabilidad de que el ganador esté siempre en cabeza (o empatado con su rival) durante todo el recuento? Este problema se conoce con el nombre del *problema de la votación*. La respuesta es p. Escriba un programa que lo verifique. *Indicación*: Simule una elección con 10.000 votantes. Genere aleatoriamente vectores de $10000p$ unos y $1000(1 - p)$ ceros. Después estudie secuencialmente si la diferencia entre ceros y unos no es nunca negativa (ello deberá ocurrir $100p$ de cada cien veces).

Prácticas de programación

9.11. Un algoritmo alternativo para generar permutaciones es rellenar el vector desde `a[0]` hasta `a[n-1]` del siguiente modo: para rellenar `a[i]` se gene-

ran números aleatorios hasta que aparezca uno que no haya sido usado previamente. Para realizar esta comprobación puede emplearse un vector de booleanos. Evalue el tiempo esperado de ejecución (¡ojo! tiene truco) y escriba un programa que compare el tiempo real de ejecución de la rutina, el tiempo obtenido por la estimación y la rutina de la Figura 9.7.

9.12. Supongamos que se quiere generar una variación aleatoria de N elementos distintos dentro del rango 1, 2, ..., M. (El caso $M = N$ corresponde a la generación de permutaciones.) El algoritmo de Floyd para conseguirlo es el siguiente: vamos generando de forma recursiva una variación de $N - 1$ elementos distintos dentro del rango 1, 2, ..., $M - 1$. Tras hacerlo generamos un entero aleatorio entre 1 y M. Si el número no está todavía en la permutación se añade el mismo, y si ya está se añade M. Debe hacer lo siguiente:

a) Pruebe que este algoritmo no genera elementos repetidos.
b) Pruebe que cada variación posible aparece de forma equiprobable.
c) Dé una implementación recursiva del algoritmo.
d) Dé una implementación iterativa del algoritmo.

9.13. Un *paseo aleatorio* en dos dimensiones es el siguiente juego, que se desarrolla en el sistema de coordenadas x, y. Se comienza en el origen (esto es, en (0, 0)). Cada iteración consiste en dar un paso, aleatoriamente elegido entre los siguientes: una unidad hacia arriba, hacia abajo, hacia la derecha o hacia la izquierda. El paseo termina cuando el caminante regresa al origen. Puede probarse que esto acaba ocurriendo más tarde o más temprano con probabilidad 1 en dos dimensiones. Sin embargo, si generalizamos el problema al espacio de tres dimensiones, la probabilidad pasa a ser menor que 1. Escriba un programa que realice 100 paseos independientes y calcule el número de pasos dados en cada dirección.

9.14. Un test estadístico muy sencillo, a la vez que efectivo, es el *test de la χ-cuadrado*. Supóngase que generamos, de forma supuestamente uniforme, N valores entre M posibles (por ejemplo, generamos N números entre 1 y M, ambos inclusive). El número de apariciones de cada valor es una variable aleatoria de media $\mu = N/M$. Para que el test, que comprueba la uniformidad, sea significativo debe verificarse $\mu > 10$. Sea f_i el número de veces que i es generado. Entonces se calcula el valor χ-cuadrado $V = \Sigma(f_i - \mu)^2/u$. El resultado debe estar próximo a M. Si el resultado está lejos de M demasiadas veces (en concreto, si lo supera en más de $2\sqrt{M}$, en uno de cada diez intentos), entonces el generador no ha pasado el test. Implemente este test y ejecútelo sobre su implementación del método `randomInt` (con `linf` = 1 y `lsup` = 100).

Bibliografía

En [3] puede encontrarse amplia información sobre los generadores de números aleatorios. El algoritmo de las permutaciones se debe a R. Floyd y puede encontrarse en [1]. El test de primalidad aleatorio se ha tomado de [2] y [4]. En cual-

quier buen libro de probabilidad y estadística se puede encontrar información adicional sobre los números aleatorios.

1. J. Bentley, «Programming Pearls», *Communications of the ACM* **30** (1987), 754-757.
2. G. L. Miller, «Riemann's Hypothesis and Tests for Primality», *Journal of Computer and System Science* **13** (1976), 300-317.
3. S. K. Park y K. W. Miller, «Random Number Generators: Good Ones Are Hard to Find», *Communications of the ACM* **31** (1988) 1192-1201. (See also *Technical Correspondence* in **36** (1993) 105-110.)
4. M. O. Rabin, «Probabilistic Algorithms for Testing Primality», *Journal of Number Theory* **12** (1980), 128-138.