Generación de números pseudoaleatorios

Sebastián Santisi

2021-11-24

En numerosas aplicaciones de la informática se hace necesaria la generación de secuencias de números aleatorios, que cumplan con determinada distribución. Ahora bien, los programas se implementan en base a algoritmos y es imposible que un algoritmo determinístico genere una secuencia aleatoria, por eso se hace necesario abordar este tema con una explicación detallada.

Aleatoreidad

Para generar un valor aleatorio es necesario contar con un sistema donde la obtención de dicho valor dependa de características de azar no manipulables, predecibles ni repetibles, idea la cual va absolutamente en contramano con lo que solemos exigirle a un procesador que es control, consistencia y previsibilidad.

Por lo general para generar valores aleatorios se requiere generar de algún modo algún tipo de entropía y acumular valores de eventos azarozos, cosa que es posible y existen circuitos electrónicos que generan ruido y utilizar eso para generar valores. De hecho muchos procesadores modernos implementan esto vía hardware, como por ejemplo la arquitectura Intel 64 a través de la instrucción `RDRAND`, aunque hay que mencionar que dependiendo de la versión del procesador puede llevar hasta 2500 ciclos de máquina obtener un único valor[1].

Ahora bien, más allá de la performance, en la biblioteca estándar de C no contamos con acceso a estas funciones de procesador sino que los números "aleatorios" se generan mediante cómputos determinísticos, lo que equivale a decir que no son aleatorios.

Pseudoaleatorios

Ante la imposibilidad de generar números aleatorios vía software lo que se diseñan son funciones que partiendo de algún valor particular aplican una serie de operaciones que permiten derivar un número nuevo muy diferente del original. Por ejemplo, este fragmento puede encontrarse en la implementación de GNU de la `libc`[2]:

```
val = ((val * 1103515245U) + 12345U) & 0x7fffffff;
```

Con esta idea se puede generar una secuencia de números determinísticos dispares entre sí dependiendo la misma del valor inicial.

A la pregunta de si esto emula a la aleatoreidad la respuesta es que dependerá del contexto. Por ejemplo, si una aplicación mostrara el valor tal cual como salió de la cuenta, y si además la implementación de la función pseudoaleatoria fuera conocida, entonces sería muy fácil saber cuál será el siguiente valor a generar, por lo que alguien podría predecir lo que hará nuestra aplicación a continuación. Ahora bien, generalmente los números computados se operan y derivan para utilizar en nuestras aplicaciones por lo que no necesariamente se expondrá el valor crudo, lo cual generalmente dificultará predecir el siguiente valor. Esto asumiendo que el valor inicial cambie entre ejecuciones, si no nuestra aplicación generará siempre la misma secuencia de números.

[1] Por ejemplo en un procesador Intel Core i7 lleva 463 ciclos mientras que en un AMD Ryzen 1200 ciclos para generar un aleatorio de 32 bits.

[2] Vale mencionar que el estándar de C pone como ejemplo una función que implementa la misma cuenta, pero que internamente trabaja con más precisión que el valor que devuelve, el cual es truncado.

Implementación en C

En C la biblioteca provee dos funciones y una macro para operar con números aleatorios todas definidas en el encabezado `<stdlib.h>`:

- `int rand(void)`: Devuelve el siguiente número aleatorio de la secuencia.
- `void srand(unsigned int seed)`: Setea la semilla, es decir, el valor inicial de la secuencia.
- `RAND_MAX`: Indica el valor máximo de los aleatorios, la función `rand()` genera aleatorios entre `0` y `RAND_MAX` inclusive.

Entonces, para obtener valores aleatorios hay que llamar a `rand()`, cada vez que la llamemos nos devolverá un valor nuevo de la secuencia. Si nunca inicializáramos una semilla la misma iniciará en 1, si quisiéramos inicializarla deberíamos llamar a `srand()` una única vez al comienzo con el valor de inicialización. El rango de los números generados estará determinado por la implementación del algoritmo y estará informado por `RAND_MAX`, vale decir que el estándar garantiza que la misma valdrá al menos $2^{15} - 1$.

Variando la semilla

Si bien para muchas aplicaciones es importante generar comportamientos repetibles[3] en otras aplicaciones es necesario generar secuencias diferentes cada vez que se ejecuta. Luego de lo ya expuesto debería resultar evidente que no hay manera de generar dentro de la misma aplicación un valor no predecible para alimentar la semilla.

Para resolver esto lo que se suele hacer no es inicializar la semilla en un valor impredecible sino que nos contentamos con inicialiarla en un valor que cambie, de ser posible, cada vez que ejecutemos nuestro programa.

Lo más usual suele ser hacer uso del encabezado `<time.h>` que nos provee funciones para manejar el tiempo, e inicializar de esta forma:

```
srand(time(NULL));
```

Esta línea inicializa la semilla en el valor resultante de la fecha y hora actual.

Cabe destacar que este no es un valor aleatorio sino absolutamente predecible y que si ejecutamos dos instancias de nuestra aplicación en un corto intervalo de tiempo ambas van a generar la misma secuencia porque, por lo general, `time()` devuelve el tiempo con una precisión de segundos. Pero al menos esto logra el objetivo de no repetir siempre la misma secuencia en nuestra aplicación.

Versión no portable

> Lo siguiente se incluye a título informativo, en este curso hacemos especial hincapié en la portabilidad por lo tanto no se permite el uso de características que dependen de una plataforma o procesador.

Una alternativa para obtener un valor realmente aleatorio para la inicialización de la semilla consiste en invocar a la instrucción de procesador que genera uno utilizando entropía. Esto va a funcionar sólamente en procesadores que implementen el juego de instrucciones de Intel 64, PCs generalmente.

Un ejemplo mínimo de invocación a `RDRAND`[4] para obtener un aleatorio de 32 bits puede ser el siguiente:

```
uint32_t rnd;
uint8_t ok;
__asm__ volatile ("rdrand %0; setc %1": "=r" (rnd), "=qm" (ok));
```

donde la variable `rnd` va almacenar el valor aleatorio mientras que `ok` valdrá `1` si es que había un valor aleatorio disponible. Si hubiera un valor disponible el mismo podría utilizarse o bien como semilla para `srand()` o bien como valor aleatorio en sí, sabiendo que llamar a esta función cada vez es extremadamente costoso.

[3] Y eso no implica ninguna contradicción con la idea de que los valores se comporten como si fueran azarozos.

[4] Existe también una instrucción `RDSEED` que es similar a `RDRAND` y que sería más apropiada si quisiéramos generar una semilla, pero que tiene más chances de falla. La invocación a esta es análoga, pero hay que poner especial atención a la validación.

Esta implementación depende pura y exclusivamente de que exista esta instrucción en el procesador. Este código no va a compilar en arquitecturas donde no esté disponible, y va a resultar en una señal `SIGILL` (instrucción ilegal) en procesadores donde no esté implementada. Se recomienda consultar la guía de Intel[5] donde explica cómo realizar el chequeo de si está disponible o no, en caso de necesitar implementar una solución basada en esto.

Distribuciones

Los valores devueltos por la función `rand()` se corresponden a una función de distribución uniforme discreta en el intervalo $[0, \texttt{RAND\_MAX}]$, ahora bien, generalmente en las aplicaciones es necesario generar funciones de distribución particulares según la necesidad.

Partiendo de la base de que los valores devueltos por `rand()` tienen una distribución uniforme se pueden fabricar funciones con otras distribuciones.

Distribución $\mathrm{U}[0, 1)$

Para obtener un número random flotante uniforme entre cero y uno puede simplemente ajustarse el intervalo aritméticamente:

```
float rand_01(void) {
    return rand() / (RAND_MAX + 1.0);
}
```

Notar que adicionamos uno, dado que queremos intervalo abierto en 1.

Teniendo implementada esta función es fácil generar transformaciones para obtener uniformes en cualquier rango, simplemente transladando y escalando estos valores.

Distribución $\mathrm{U}[0, a)$

Para obtener un número random entero uniforme entre cero y un valor a puede aplicarse un enfoque similar al aplicado para obtener flotantes, pero hay un método más sencillo y eficiente. Como los valores devueltos por `rand()` se encuentran distribuídos uniformemente, los mismos pueden periodicizarse con frecuencia a y esta transformación también dará valores uniformes, siempre y cuando $a << \texttt{RAND\_MAX}$. Para esto utilizamos el operador módulo:

```
int rand_a(int a) {
    return rand() % a;
}
```

que generará siempre números entre 0 y $a - 1$.

Distribución $\mathrm{U}[a, b)$

Como último caso particular de uniforme, si quisiéramos generar valores uniformes dentro de un intervalo $[a, b)$ podemos operar:

```
float rand_ab(float a, float b) {
    return rand_01() * (b - a) + a;
}
```

Esta idea puede ser generalizada para obtener cualquier otro intervalo o tipo.

[5] https://www.intel.com/content/www/us/en/developer/articles/guide/intel-digital-random-number-generator-drng-software-implementation-guide.html

Ensayo de Bernoulli

Para obtener un valor con determinada probabilidad de éxito p ($0 \leq p \leq 1$) podemos simplemente operar:

```
bool bernoulli(float p) {
    return rand_01() < p;
}
```

Otras distribuciones

> Hace falta un mínimo conocimiento de Probabilidad y Estadística para entender lo que sigue a continuación, lo cual no es un problema dado que sólo vas a necesitar generar otras distribuciones si ya sabés Probabilidad y Estadística.

Ya vimos cómo generar números uniformes, pero muchas veces hace falta generar ensayos de variables aleatorias con otra función de distribución de probabilidad $f(x)$. La operatoria en estos casos utiliza la función de distribución acumulada $F(x) = \int_{-\infty}^{x} f(u)\,du$. Notar que como la probabilidad total de $f(x)$ vale 1 y $f(x)$ es siempre positiva, entonces $F(x)$ será monótona creciente entre 0 y 1. Como $F(x) = P[X \leq x]$, incrementos en el valor de $F(x)$ se corresponden con densidades de probabilidad en x.

La estrategia para generar realizaciones entonces es generar valores de $p = F(x)$ distribuídos de forma uniforme y despejar luego los valores de x, es decir, hay que obtener la inversa de la función de distribución acumulada $x = F^{-1}(p)$ y alimentarla con valores de $P \sim \mathrm{U}[0, 1)$. Los valores resultantes tendrán una función de distribución de probabilidad $f(x)$.

Por ejemplo, si se tuviera una distribución exponencial $X \sim \mathrm{Exp}(\lambda)$, con función de densidad de probabilidad

$$f_X(x) = \lambda e^{-\lambda x}, \quad x \geq 0,$$

entonces podemos calcular su función de distribución acumulada como

$$F_X(x) = \int_{-\infty}^{x} f_X(u)\,du = 1 - e^{-\lambda x}, \quad x \geq 0.$$

De ahí despejamos para obtener la inversa

$$F_X^{-1}(p) = -\frac{\ln(1-p)}{\lambda}, \quad 0 \leq p < 1.$$

Finalmente podemos implementar nuestra función:

```
float rand_exp(float lambda) {
    return - log(1 - rand_01()) / lambda;
}
```