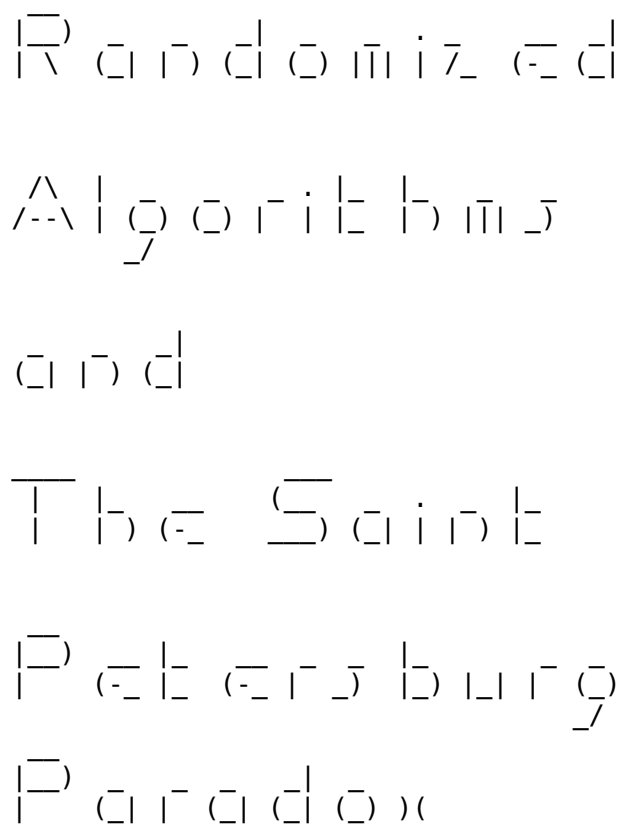
"Análisis de Algoritmos"
José Helo Guzmán



Contenido

▬▬▬▬ 1. Introducción.

“Randomized Algorithms”, conocidos como algoritmos aleatorios, son ciertos algoritmos que utilizan números aleatorios y probabilidades para decidir el siguiente paso o movimiento.

La necesidad de utilizar números aleatorios puede surgir de la descripción misma del problema, por ejemplo para simular un juego de Monopoly, o Gran Banco, se necesita poder programar un dado. El cual tiene que dar valores de forma aleatoria.

También se pueden utilizar números aleatorios en un problema determinísitico para hacer más rápida la búsqueda de una solución. Hasta el momento el único algoritmo que hemos visto de este tipo ha sido el “quick sort” aleatorio.

▬▬▬ 2. Generadores Pseudoaleatorios.

El algoritmo más utilizado para la generación de números pseudoaleatorios se conoce como el método congruencial mixto. A continuación un ejemplo de este método.

```
x(0)   : 4
x(n+1) : ( 5 * x(n) + 7 ) mod 8
```

```
x( 0)  :                        4 mod 8  :  4
x( 1)  :  (5*4 + 7) mod  8  =  27 mod 8  :  3
x( 2)  :  (5*3 + 7) mod  8  =  22 mod 8  :  6
x( 3)  :  (5*6 + 7) mod  8  =  37 mod 8  :  5
x( 4)  :  (5*5 + 7) mod  8  =  32 mod 8  :  0
x( 5)  :  (5*0 + 7) mod  8  =   7 mod 8  :  7
x( 6)  :  (5*7 + 7) mod  8  =  42 mod 8  :  2
x( 7)  :  (5*2 + 7) mod  8  =  17 mod 8  :  1
x( 8)  :  (5*1 + 7) mod  8  =  12 mod 8  :  4
x( 9)  :  (5*4 + 7) mod  8  =  27 mod 8  :  3
x(10)  :  (5*3 + 7) mod  8  =  22 mod 8  :  6
x(11)  :  (5*6 + 7) mod  8  =  37 mod 8  :  5
```

Este procedimiento se puede programar de la siguiente manera:

```
randu(n):=block
(
    x0: 4,
    a:  5,
    b:  7,
    m:  8,

    memo:[x0],

    for i:1 thru n do
    (
        xn: last(memo) ,
        xnn: mod(a*xn + b,m),
        memo:endcons(xnn,memo)
    ),
    memo
);
```

En la actualidad los parámetros más utilizados para este método están dados por el algoritmo de Mersenne Twister. A continuación se muestra una de las fórmulas de este algoritmo.

x(0) : semilla,

x(n+1): (7^5)*x(n) mod (2^31-1)

O bien:

x(0): semilla,

x(n+1): (2^16+3)*x(n) mod (2^31)

Veamos un ejemplo de cómo funciona este generador aleatorio [0,1[.

```
randu(xn):=block
(
    a: 69069,
    c: 1,
    m: 2^32,

    xn:mod((xn*a)+c,m),
    print(xn),
    xf:float(xn/m),
    printf(true,"~7f",xf)
);
```

```
(%i) randu(7);
483484
0.000113

(%i) randu(483484);
3328985325
0.77509

(%i) randu(3328985325);
2908188362
0.677115

(%i) randu(2908188362);
2926442947
0.681366

(%i) randu(2926442947);
1031989288
0.240279
```

▬▬▬ 3. La Función Random(n).

Muchos lenguajes de programación incluyen funciones que pueden generar números de forma aleatoria. Por ejemplo la función “random(n)” genera con la misma probablidad un número que se encuentra entre [0, n-1].

```
(%i) random(6);
(%o) 2

(%i) random(6);
(%o) 0

(%i) random(6);
(%o) 5
```

Podemos ver como se generan números en [0,5]. De esta forma, si quisiéramos generar un dado podríamos usar la siguiente función:

```
dado():=random(6)+1;
```

Veamos como funciona:

```
(%i) dado();
(%o) 5

(%i) dado();
(%o) 2


(%i) dado();
(%o) 6
```

Veamos este programa que lanza muchas veces un dado:

```
dado():=random(6)+1;


tirarDado(n):=block
(
   memo:[0,0,0,0,0,0],
   for i:1 thru n do
   (
      index:dado(),
      memo[index]:memo[index]+1
   ),
   memo
);
```

Veamos como funciona:

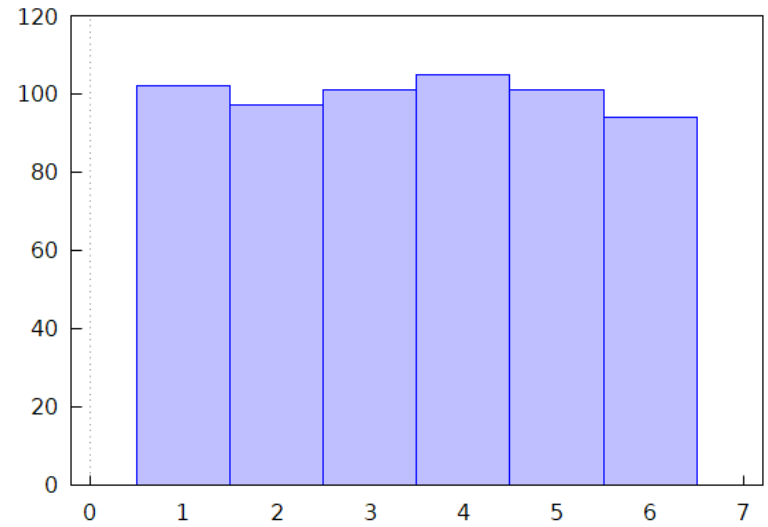
```
(%i) tirarDado(600);
(%o) [100,98,101,90,111,100]
```

En teoría, un dado perfecto debería dar un valor de 100 en cada uno de los elementos del arreglo. En la vida real, al igual que en nuestro programa, eso no es posible. Por lo tanto obtenemos pequeñas variaciones.

» Gráfico.

» Comportamiento del Dado.

```
load(draw)$

dado():=random(6)+1;

tirarDado(n):=block
(
   dado:[0,0,0,0,0,0],
   for i:1 thru n do
   (
      index:dado(),
      dado[index]:dado[index]+1
   ),
   dado
);

altura: tirarDado(600);
ini:0;
fin:7;

barras:
bars( [1,altura[1],1],
      [2,altura[2],1],
      [3,altura[3],1],
      [4,altura[4],1],
      [5,altura[5],1],
      [6,altura[6],1]
);

wxdraw2d
(
   xrange = [ini-0.20,fin+0.20],
   yrange = [ini-0.20,120],
   grid=false,
   xaxis=true,
   yaxis=true,

   color=blue,
   fill_color=blue,
   fill_density=0.25,
   barras

)$
```



▬▬▬ 4. La Función Random(1.0).

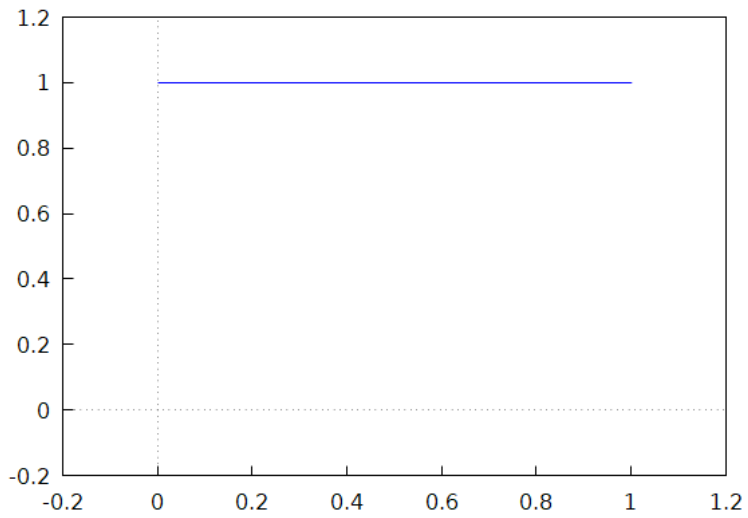
Existe otra función que genera números aleatorios en el intervalo [0,1]. En realidad es una gran cantidad de números discretos que nos da la ilusión de ser números continuos.

```
(%i) random(1.0);
(%o) 0.1951846177977887


(%i) random(1.0);
(%o) 0.4823905248516196


(%i) random(1.0);
(%o) 0.3554400571592862
```

Veamos a continuación el gráfico teórico.

» Gráfico.
» La Función Uniforme.

```
load(draw)$
f(x):=1;
ini:0;
fin:1;
wxdraw2d
(
   xrange = [ini-0.20,fin+0.20],
   yrange = [ini-0.20,1.20],
   grid=false,
   xaxis=true,
   yaxis=true,

   color=blue,
   explicit(f(x),x,0,1)

)$
```



▬▬▬▬ 5. Generando una Moneda.

Usemos ahora este generador aleatorio de números para generar una moneda perfecta. Esta moneda tiene la misma probabilidad de producir un escudo o una corona. Denotaremos el escudo por un valor de “0” y la corona con un valor de “1”. Es usual que las probabilidades se midan en el intervalo [0,1[. Y además todos los posibles resultados deben tener la suma de 1.

Por lo tanto deseamos:

p   = probabilidad(0) = p(0) = 0.50

1-p = probabilidad(1) = p(1) = 0.50

Para construir una moneda realizamos el siguiente proceso:

Generamos un número aleatorio “r” entre [0,1[.

Si (r <= 0.50) es un escudo, devolvemos un “0”.

Si (r > 0.50) es una corona, devolvemos un “1”.

```
flip(prob):=block
(  [r],
   r:random(1.0),
   if (r <= prob) then
      1
   else
      0
);
```

▬▬▬▬ 6. Hacer un Flip hasta obtener un “1”.

Tenemos el siguiente problema.

• Se tira una moneda.

• Si sale un “0” se vuelve a lanzar.

• Si sale un “1” se detiene el proceso.

En promedio cuántos lanzamientos se necesitan?

Preguntar el número promedio de lanzamientos que se necesitan es equivalente a preguntar por la complejidad del problema. Pues el problema se detiene cuando se encuentra un valor de “1”.

```
flip(prob):=block
(  [r],
   r:random(1.0),
   if (r <= prob) then
      1
```

```
   else
      0
);


simular(prob):=block
(  [acum,moneda],
   acum:0,
   moneda:0,
   while is(moneda=0) do
   (
      acum:acum+1,
      moneda:flip(prob),
      print(moneda)
   ),
   print(""),
   acum
);
```

Veamos algunas corridas:

```
(%i) simular(0.50);
0
0
0
0
0
1
(%o) 6


(%i) simular(0.50);
1
(%o) 1


(%i) simular(0.50);
0
1
(%o) 2


(%i) simular(0.50);
1
(%o) 1



(%i) simular(0.50);
0
1
(%o) 2
```

Estas son muy pocas corridas o simulaciones del juego. Sin embargo si suponemos que son suficientes, podemos obtener el número promedio de corridas que se se necesitan.

$$\text{promedio} = \left(\frac{6 + 1 + 2 + 1 + 2}{5} \right)$$

promedio = 2.40

Para asegurarnos que este valor es el correcto deberíamos de realizar muchas más corridas, o en su defecto encontrar una fórmula general.

▬▬▬ 7. Hagamos Muchas Corridas.

Vamos a hacer un programa que repita el proceso anterior muchas veces y encuentre el promedio de todas las simulaciones.

```
simularN(n,prob):=block
(   [suma,promedio],
    suma:0,
    for i:1 thru n do
    (
        res:simular(prob),
        /*
        print(res),
        */
        suma:suma+res
    ),
    promedio: float(suma/n)
);
```

Veamos como funciona este programa, con un valor de p=0.50.

```
(%i) simularN(1000,0.50);
(%o) 2.111

(%i) simularN(1000,0.50);
(%o) 1.969

(%i) simularN(1000,0.50);
(%o) 1.977
```

Se ve que el número de corridas es cercano a 2.

●● Hagamos la simulación con p=0.20.

```
(%i) simularN(1000,0.20);
(%o) 5.042

(%i) simularN(1000,0.20);
(%o) 5.073

(%i) simularN(1000,0.20);
(%o) 4.905
```

Se ve que el número de corridas es cercano a 5.

●● Hagamos la simulación con p=0.10.

```
(%i) simularN(1000,0.10);
(%o) 10.442

(%i) simularN(1000,0.10);
(%o) 9.559

(%i) simularN(1000,0.10);
(%o) 10.203
```

Se ve que el número de corridas es cercano a 10.

●● Fórmula general para el número de corridas.

Se puede demostrar (pero está fuera del alcance de este curso), que el promedio de corridas está dado por la siguiente fórmula.

Sea “p” la probabilidad de obtener el valor “1”.

Sea “1 - p” la probabilidad de obtener el valor de “0”.

Entonces el promedio del número de corridas estará dado por:

$$\text{promedio} = \left(\frac{1}{p} \right)$$

promedio = 1/p

En nuestro caso tenemos:

p	1/p
0.50	2
0.20	5
0.10	10

▬▬▬ 8. La Paradoja de San Petersburgo.

En 1725 Daniel y Nicolás Bernoulli enseñaban matemáticas en la academia rusa de San Petersburgo.

Nicolás había propuesto el siguiente problema, conocido como la “Paradoja de San Petersburgo”.

Se utiliza una moneda con dos lados. Un lado lo denominaremos con el valor de 0 y el otro lado con el valor de 1. Utilizando esta moneda, lanzar la moneda hasta que salga un valor de 1.

Si sale 1, se paga $1

Si sale 0,1 se pagan $2

Si sale 0,0,1 se pagan $4

Si sale 0,0,0,1 se pagan $8

Si sale 0,0,0,0,1 se pagan $16

En general se paga 2^n con “n” el número de ceros que se obtienen.

De esta forma se tiene:

		Prob
1	$1	1/2
0,1	$2	1/4
0,0,1	$4	1/8
0,0,0,1	$8	1/16
0,0,0,0,1	$16	1/32

¿Cuánto se debe pagar en promedio al jugar este juego?

Pare resolver este problema Georges Lous Letrec, conocido como El Conde Buffon (1707 – 1788), realiza la primera prueba empírica. Ordena a uno de sus sirvientes a repetir el juego 2000 veces y anotar cuanto se paga.

Esto por supuesto, es una aplicación inmediata del problema anterior.

▬▬▬ 9. Resumen.

• Los algoritmos aleatorios son aquellos que utilizan probabilidades para resolver un problema.

• Calcular el tiempo de ejecución de estos algoritmos es bastante complejo, pues dependerán de los valores de probabilidad que reciban.

▬▬▬ 10. Ejercicios.

●● Ejercicio 1.

Construya un programa que resuelva el problema de la “Paradoja de San Peterbursgo”. Ejecute este programa muchas veces, obtenga el promedio e indique cuánto es en promedio lo que se debe pagar al jugar este juego.