

**UNIVERSIDAD DE CHILE**  
**FAC. DE CS. FIS. Y MAT.**

# NUMEROS ALEATORIOS.

(Random Numbers)

Prof. Patricio Cordero  
Aux. Juan Pineda

Marcelo Robles C.  
Jueves, 13 de diciembre 2001

*“Cualquiera que considere métodos aritméticos para producir dígitos aleatoriamente está, por supuesto, en estado de pecado”*

John Von Neumann (1951)

## 1.1 Introducción.

En un sentido, no existe tal cosa como un número aleatorio. Por ejemplo, ¿es 2 un número aleatorio?, Más bien hablamos de una secuencia de *números aleatorios independientes* y con una *distribución* específica. Esto significa aproximadamente que cada número se obtuvo simplemente por azar, sin tener ninguna relación con otros números de la secuencia y cada número tiene una probabilidad especificada de caer en cualquier rango dado de valores.

Aparentemente es una imposibilidad conceptual el decir que un computador, totalmente determinista, sea capaz de generar “números aleatorios”<sup>1</sup>. Después de todo, cualquier programa producirá una salida que es enteramente predecible y por tanto, no verdaderamente “aleatoria”<sup>2</sup>.

Sin embargo, programas de computador generadores de números aleatorios se usan comúnmente. A menudo – y considerando esta dificultad conceptual – uno llama a estas secuencias generadas por computador *pseudo-aleatorias*<sup>3</sup>, mientras que la palabra *aleatoria* se reserva para la salida de un proceso físico intrínsecamente aleatorio, tal como el tiempo transcurrido entre clicks de un contador Geiger puesto cerca de una muestra de algún elemento radioactivo.

Una definición operatoria, aunque imprecisa, de aleatoriedad<sup>4</sup> en el contexto de las secuencias generadas por computador es decir que el programa determinista que genera que produce una secuencia aleatoria debe ser distinta de, en todos los aspectos medibles, y estadísticamente no correlacionada con el programa de computador que usa su salida. Es decir, dos diferentes generadores de números aleatorios deben producir estadísticamente los mismos resultados cuando se aplican al mismo programa particular de aplicaciones. Si ello no ocurre, entonces al menos uno de ellos no es (desde ese particular punto de vista) un buen generador.

---

<sup>1</sup> Random Numbers.

<sup>2</sup> Random.

<sup>3</sup> Pseudo-Random

<sup>4</sup> Randomness.

La definición anterior puede parecer circular al comparar, como lo hace, un generador con el otro. Sin embargo, existe un conjunto de generadores de Números Aleatorios que mutuamente satisfacen dicha definición sobre una muy amplia clase de programas de aplicación. De gran importancia también es destacar que se halla empíricamente resultados idénticos de Números Aleatorios producidos por procesos físicos. Luego, debido a que dichos generadores existen, se puede dejar a los filósofos el problema de definirlos (Ver por ejemplo Knuth, Donald [1]).

Un punto de vista pragmático es decir que la Aleatoriedad se halla en el ojo del observador (o programador). Aquello que es suficientemente aleatorio para una aplicación puede no ser lo suficiente para otra aplicación. A pesar de ello, uno no se halla totalmente a la deriva en un mar de inconmensurables programas de aplicaciones. Existe una lista de pruebas (Tests) estadísticas, algunas sensibles y otras veneradas por motivos históricos, las cuales como un todo hacen un muy buen trabajo en descubrir cualquier correlación que podría ser develada por un programa de aplicaciones. Los buenos generadores de Números Aleatorios deben pasar todas estas pruebas, o al menos, el usuario debería estar al tanto de cualquiera de ellas que estos no superen, de manera que pueda tener la opción de juzgar si esto es relevante al caso en análisis.

### 1.1.1 Algo de Historia.

Al principio la gente que necesitó números aleatorios en sus trabajos científicos, debió extraer bolas de una urna bien batida o debió rodar dados o tirar cartas. Una tabla de cerca de 40 000 dígitos aleatorios “tomados al azar de reportes del censo” fue publicada en 1927 por L. H. Tippet. Desde entonces se ha construido una cantidad de artilugios para generar números aleatorios mecánicamente. La primera de estas máquinas fue usada en 1939 por M.G.Kendall y B.Babington-Smith para producir una tabla de 100 000 dígitos aleatorios, y en 1955 la Corporación RAND publicó una tabla ampliamente utilizada de 1 000 000 de dígitos aleatorios obtenidos con la ayuda de otro aparato especial. Una famosa máquina llamada ERNIE ha sido utilizada para sacar los números ganadores en la Lotería Británica.

Poco tiempo después que los computadores fueron introducidos, la gente comenzó a buscar maneras eficientes de obtener números aleatorios con programas computacionales.

John Von Neumann sugirió por primera vez este enfoque (uso de programas computacionales) alrededor de 1946 usando el método del “Centro del Cuadrado”. Su idea fue tomar el cuadrado del número aleatorio previo y extraer los dígitos centrales. Por ejemplo, si estuviéramos generando números de 10 dígitos y el valor previo fuese 5 772 156 649, lo elevamos al cuadrado para obtener 33 317 792 380 594 909 201. El siguiente número será:

33317792380594909201  
----- 7923805949-----

Existe en justicia una obvia objeción a esta técnica: ¿Cómo puede una secuencia generada de tal forma ser aleatoria si cada número es determinado por su predecesor? La respuesta es que la secuencia *no es* aleatoria pero *parece* serlo. En aplicaciones típicas, la relación entre un número y su sucesor no tiene significado físico, por lo que el carácter no aleatorio no es realmente indeseado. Intuitivamente el “Centro del Cuadrado” parece ser una muy buena manera de “desordenar” el número previo.

El método original de Von Neumann ha sido calificado como una fuente comparativamente pobre de números aleatorios. El peligro es que la secuencia tiende a caer en un ciclo repetitivo con pocos elementos que se repiten. Por ejemplo si el cero aparece como un número de la secuencia, continuará perpetuándose por sí mismo.

## 1.2 Distribuciones Uniformes.

Las Distribuciones Uniformes son números aleatorios que se hallan dentro de un determinado rango (Típicamente 0 a 1), con cualquier número en el rango con la misma probabilidad que otro cualquiera. Son aquello que normalmente asimilaríamos como “Números Aleatorios”. Sin embargo, queremos distinguir Distribuciones Uniformes de otros tipos de Números Aleatorios, como por ejemplo Números obtenidos de una distribución Normal (Gaussiana) de media y desviación estandar especificadas. Estos otros tipos de distribuciones son casi siempre generadas mediante la realización de operaciones apropiadas sobre una o más Distribuciones Uniformes, como ya se viera en el curso.

### 1.2.1 Generadores de Números Aleatorios Proporcionados por Sistemas.

La mayoría de las implementaciones de C tienen un par de subrutinas para inicializar y entonces generar, Números Aleatorios. En ANSI<sup>5</sup> C, la sintaxis es:

```
# include <stdlib.h>
# define RAND_MAX...

void srand(semilla sin signo);
int rand(void);
```

El Manual de Referencia<sup>6</sup> de C comenta respecto de estas rutinas:

**srand<sup>7</sup>**

---

---

<sup>5</sup> American National Standards Institute.

<sup>6</sup> Manual de Referencia C; Herbert Schildt; 2001; Osborne Mc Graw-Hill.

<sup>7</sup> Ibid. Pag.411

Establece el punto de partida para la secuencia generada por `rand()`. (La función **rand()** devuelve Números Pseudo Aleatorios). La función **srand()** se utiliza a menudo para permitir que las distintas ejecuciones de un programa utilicen diferentes secuencias de Números Pseudo Aleatorios. Sin embargo, se puede generar la misma secuencia de Números Pseudo Aleatorios una y otra vez llamando a **srand()** con la misma semilla cada vez que vaya a comenzar la secuencia.

## **rand**<sup>8</sup>

---

La función **rand()** genera una secuencia de Números Pseudo Aleatorios. Cada vez que se llama, devuelve un entero entre cero y `RAND_MAX`. `RAND_MAX` debe ser, al menos, 32 767.

Se inicializa el generador de Números Aleatorios invocando **srand(semilla)** con alguna *semilla* arbitraria. Cada valor de inicio dará lugar a una distinta secuencia aleatoria, o al menos, un distinto punto de partida en alguna secuencia enormemente larga. El mismo valor de inicio de la semilla siempre nos devolverá a la misma secuencia Aleatoria.

Es posible obtener Números Aleatorios sucesivos en la secuencia mediante sucesivas llamadas a **rand()**. Esta función entrega un entero que está típicamente en el rango 0 al mayor valor positivo representable del tipo **int** (inclusive). Usualmente, tal como es el caso en ANSI C, este valor más grande se obtiene como `RAND_MAX`, pero a veces uno tiene que dárselo a si mismo. Si se desea un valor aleatorio del tipo **float**, entre 0.0 (incluso) y 1.0 (no incluido), se puede obtener por una expresión como la siguiente (usada en los programas desarrollados en el curso:

```
x = rand()/(RAND_MAX+1.0);
```

Ahora bien, uno debe ser muy cuidadoso de una expresión como la anterior, basada en una rutina **rand()** proporcionada por un sistema. Si todos los artículos científicos cuyos resultados se hallan en duda debido de la errada aplicación de un `rand()` debieran desaparecer de las bibliotecas, habría un gran espacio adicional en los anaqueles.

Las rutinas **rand()** son casi siempre generadores del tipo congruencia lineal<sup>9</sup> (LCG<sup>10</sup>), los cuales generan una secuencia de enteros  $I_1, I_2, I_3, \dots$ , cada uno de ellos entre 0 y  $m-1$  (e.g. `RAND_MAX`) mediante la siguiente relación de recurrencia:

$$I_{j+1} = a I_j + c \pmod{m} \quad (*)$$

---

<sup>8</sup> Ibid. Pag. 409

<sup>9</sup> Linear Congruencial Generator.

<sup>10</sup> Introducidos por D.H.Lehmer, en 1949.

En dicha expresión a  $m$  se le denomina módulo, y  $a$  y  $c$  son enteros positivos llamados, respectivamente, el multiplicador y el incremento. La recurrencia eventualmente se repetirá a sí misma con un período que obviamente no es mayor que  $m$ . Si  $m$ ,  $a$  y  $c$  se escogen apropiadamente, entonces el período será de largo máximo, es decir, de largo  $m$ . En tal caso, todo entero posible entre  $0$  y  $m-1$  será tomado en algún punto de forma tal que cualquier elección inicial de la semilla de  $I_0$  es tan buena como cualquier otra; la secuencia sólo arranca desde ese punto.

Aunque esta estructura general es suficientemente poderosa como para generar algunos decentes Números Aleatorios, su implementación en muchos, sino en la mayoría, de las bibliotecas de ANSI C es defectuosa; con un número de implementaciones en la categoría de “totalmente echado a perder”. La vergüenza debe ser dividida aproximadamente en partes iguales entre el comité ANSI C y quienes las implementaron.

Los problemas típicos son: *primero*, dado que el estandar ANSI C especifica que **rand()** devuelve un valor del tipo **int**, el cual es solamente una cantidad de dos bytes sobre muchas máquinas, RAND\_MAX es a menudo no demasiado grande. El estandar ANSI C sólo que este sea *al menos* 32 767. Esto puede ser desastroso en muchas circunstancias, por ejemplo en una integración de Monte Carlo se puede muy bien evaluar  $10^6$  puntos diferentes, pero en realidad estar evaluando los mismos 32 767 puntos, 30 veces cada uno lo cual, definitivamente, ¡no es lo mismo!.

Debe categoricamente desecharse cualquier rutina de biblioteca para Números Aleatorios con un valor devuelto de dos bytes.

*Segundo*, la publicación del comité de ANSI incluye el siguiente malévolo pasaje: “el comité ha decidido que una implementación debe ser permitida para proporcionar una función **rand** que genere la mejor secuencia aleatoria posible en aquella implementación y por tanto, “mandated” ningún algoritmo estandar. Sin embargo reconoce el valor de ser capaz de generar la misma secuencia pseudo – aleatoria en diferentes implementaciones, y por ello *ha publicado un ejemplo* [énfasis adicionado]. El “ejemplo” es:

```
unsigned long next =1;

int rand(void) /*No recomendado*(ver texto)/
{
    next = next*1103515245 + 12345;
    return (unsigned int)(next/65536) % 32768;
}

void srand(unsigned int semilla)
{
    next=semilla;
}
```

Esto corresponde a la ecuación (\*) con  $a = 1\ 103\ 515\ 245$ ,  $c = 12\ 345$  y  $m = 2^{32}$  (dado que la aritmética hecha sobre cantidades **unsigned long** garantiza devolver correctamente los bits de bajo orden). Estas no son elecciones particularmente buenas para  $a$  y  $c$ , aunque no son errores gruesos por sí mismas. El equívoco mayor acontece cuando quienes la implementan, tomando la afirmación anterior del comité como una licencia tratan de mejorar el ejemplo publicado. Por ejemplo, un compilador popular PC compatible de 32 bit, proporciona un generador **long** que usa la congruencia anterior pero intercambia los 16 bits de alto y bajo orden del valor retornado. Alguien probablemente pensó que esta innovación extra aumentaría la aleatoriedad cuando de hecho, arruina el generador. Mientras este tipo de errores puede, por supuesto, fijarse, todavía permanece un defecto fundamental en los generadores congruentes simplemente lineales, el cual procederemos a discutir a continuación.

El método congruente lineal tiene la ventaja de ser muy rápido, requiriendo de sólo unas cuantas operaciones por llamada, de donde procede su uso universal. Tiene la desventaja de no ser completamente libre de correlaciones secuenciales en las llamadas sucesivas. Si usamos  $k$  Números Aleatorios en un instante para graficar puntos en un espacio  $k$ -dimensional (con cada coordenada con valores entre 0 y 1) entonces los puntos no tenderán a llenar el espacio  $k$ -dimensional, sino más bien pertenecerán a planos  $(k-1)$ -dimensionales. Habrá *como máximo*  $m^{1/k}$  de tales planos. Si las constantes  $m$ ,  $a$  y  $c$  no se escogen cuidadosamente, entonces habrá muchos menos planos  $(k-1)$ -dimensional que tal valor. Si  $m$  es tan malo como 32 768, entonces el número de planos sobre los cuales triadas de puntos pertenecientes a un espacio 3-dimensional no será mayor que la raíz cúbica de 32 768, esto es, 32. Incluso si  $m$  es cercano al entero más grande representable en la máquina, es decir  $2^{32} = 4\ 294\ 967\ 296$ , el número de planos sobre los cuales triadas de puntos pertenecientes a un espacio 3-dimensional usualmente no será mayor que alrededor de la raíz cúbica de  $2^{32}$ , es decir, aproximadamente 1 625. Ud. podría estar enfocando su atención sobre un proceso físico que ocurre en una pequeña fracción del volumen total de modo que la discretitud de los planos puede ser muy pronunciada.

Incluso peor, Ud. podría estar utilizando un generador cuyas elecciones de  $m$ ,  $a$  y  $c$  no han sido las apropiadas. Un ejemplo de tal rutina, RANDU, con  $a = 65\ 539$  y  $m = 2^{31}$ , fue utilizada ampliamente en computadores mainframe IBM por muchos años y muy copiada hacia otros sistemas.<sup>11</sup>

La correlación en el espacio  $k$ -dimensional no es la única debilidad de los generadores Congruentes Lineales (LCG). Tales generadores a menudo tienen sus bits de bajo orden (menos significantes) mucho menos aleatorios que sus bits de alto orden. Si se desea generar un entero aleatorio entre 1 y 10. debe siempre utilizarse bits de alto orden como:

```
j = 1 +(int)(10.*rand()/(RAND_MAX +1.0));
```

y nunca mediante nada como lo siguiente:

---

<sup>11</sup> *Anécdota*: Uno de los autores de N.R. recuerda haber graficado Números Aleatorios con RANDU, obteniendo sólo 11 planos a lo cual el encargado del Centro de Computación habría respondido: "Garantizamos que cada número es aleatorio individualmente, pero no garantizamos que más de uno de ellos sea aleatorio". De Ripley.

$j = 1 + (\text{rand}() \% 10);$

El cual utiliza bits de bajo orden.

Similarmente nunca debe tratar de separar un número “rand()” en variadas piezas supuestamente aleatorias. Antes bien, prefiera llamadas separadas para cada pieza.

## 1.2.2 Generadores Portables de Números Aleatorios

### 1.2.2.1 Generador Mínimo de Park y Miller: ran0.

Park y Miller [1] han revisado un gran número de generadores de números aleatorios que han sido utilizados en los últimos 30 años o más. Junto con una buena revisión teórica, ellos presentan una muestra anecdótica de un número de generadores inadecuados que han llegado a ser de amplio uso.

Existe una buena evidencia, teórica y empírica, que el algoritmo congruente simplemente lineal:

$$I_{j+1} = a I_j \pmod{m}$$

puede ser tan bueno como cualquiera de los generadores congruentes lineales más generales que tienen  $c \neq 0$ , si el multiplicador  $a$  y el módulo  $m$  se escogen con un cuidado exquisito. Park y Miller proponen un generador de “Estandar Mínimo”, basado en las elecciones siguientes:

$$a = 7^5 = 16\,807 \qquad m = 2^{31} - 1 = 2\,147\,483\,647$$

Primeramente propuesto por Lewis, Goodman y Miller en 1969, este generador ha logrado, en los años siguientes, pasar todos los exámenes teóricos nuevos, y (quizas lo más importante) ha acumulado una gran cantidad de usos exitosos. Park y Miller no aseguran que el generador sea “perfecto” (veremos que no lo es), sino que sólo es un buen estandar mínimo contra el cual los otros generadores deben ser juzgados.

No es posible implementar las ecuaciones anteriores directamente en un lenguaje de alto nivel porque el producto de  $a$  y  $m-1$  excede el valor máximo para un entero de 32 bits. La implementación en lenguaje Assembler (de máquina) usando un registro de producto de 64 bits es directo, pero no portable de máquina a máquina. Un truco debido a Schrage [2,3] para multiplicar dos enteros de 32 bits modulo una constante de 32 bits, sin usar intermediarios mayores que 32 bits (incluyendo un bit para el signo) es por lo tanto en extremo interesante pues permite implementar el generador de Estandar Mínimo en esencialmente cualquier lenguaje de programación o máquina.



El algoritmo de Schrage se basa en una *factorización aproximada* de  $m$ ,

$$m = aq + r, \quad \text{es decir,} \quad q = \lfloor m/a \rfloor, \quad r = m \bmod a$$

Donde los paréntesis cuadrados denotan parte entera. Si  $r$  es pequeño, específicamente  $r < q$ , y  $0 < z < m-1$ , puede demostrarse que ambos,  $a(z \bmod q)$  y  $r[z/q]$  se hallan en el rango 0, .....,  $m-1$ , y que:

$$az \bmod m = \begin{cases} a(z \bmod q) - r[z/q] & \text{si es mayor a cero} \\ a(z \bmod q) - r[z/q] + m & \text{en otro caso} \end{cases}$$

La aplicación del algoritmo de Schrage a las constantes propuestas por Park y Miller usa los valores  $q = 127\,773$  y  $r = 2\,836$ .

A continuación se halla una implementación del generador Estandar Mínimo:

```
# define IA 16807
# define IM 2147483647
# define AM (1.0/IM)
# define IQ 127773
# define IR 2836
# define MASK 123459876
```

```
float ran0(long*idum)
```

```
/* "Generador de Números Aleatorios "Mínimo" de Park y Miller. Devuelve una distribución aleatoria
uniforme entre 0.0 y 1.0. Setear o resetear idum a cualquier valor entero (excepto el valor distinto
MASK) para inicializar la secuencia; idum no debe ser alterado entre llamadas para valores
sucesivos en una secuencia. */
```

```
{
    long k;
    float ans;

    *idum ^= MASK;           // XORing con MASK permite el uso de cero y otros patrones
    k=(*idum)/IQ             // simples de bit para idum.
    idum = IA*(*idum-k*IQ)-IR*k; // Calcula idum=(IA*idum) % IM sin desborde mediante el
    if (*idum < 0) *idum += IM; // método de Schrage.
    Ans = AM*(*idum);        // Convierte idum a un resultado flotante.
    *idum ^= Mask;          // desenmascara antes de retornar.
    return ans;
}
```

El periodo de `ran0` es  $2^{31} - 2 \approx 2.1 \times 10^9$ . Una característica de los generadores de este tipo es que nunca debe permitirse como la semilla inicial pues se perpetúa a sí mismo, y nunca ocurre si por otro lado la semilla inicial es distinta de cero. La experiencia ha demostrado que los usuarios siempre utilizan como semilla para un generador de Números Aleatorios  $idum = 0$ .

La rutina `ran0` corresponde a un Estandar Mínimo, satisfactorio para la mayoría de las aplicaciones, pero no es recomendable como la última palabra en generadores de Números Aleatorios. Nuestro razonamiento se debe precisamente a la simplicidad del Estandar Mínimo. No es difícil pensar en situaciones donde Números Aleatorios sucesivos puedan ser usados en una forma que accidentalmente entre en conflicto con el algoritmo de generación. Por ejemplo, dado que números sucesivos difieren por un múltiplo de sólo  $1.6 \times 10^4$  respecto de un módulo de más de  $2 \times 10^9$ , muy pequeños Números Aleatorios tenderán a ser seguidos por valores menores que los números promedios. Una vez en  $10^6$ , por ejemplo, habrá un valor retornado  $< 10^{-6}$  (como debe ser) pero este valor será *siempre* seguido por un valor menor que 0.0168. Uno puede pensar fácilmente en aplicaciones que involucran eventos raros donde esta propiedad puede conducir a resultados equivocados.

#### 1.2.2.2 Generador Mínimo de Park y Miller – Bays y Durham: ran1.

Diagrama de flujo de la función de cifrado de la primera iteración de la red de Feistel:

- Se genera un valor aleatorio **RAN**.
- Se toma el valor de entrada **iv** (índice de inicio) y se genera un índice **iy** (índice de salida).
- Se genera un índice **3** (índice de la clave).
- Se genera un índice **2** (índice de la clave).
- Se genera un índice **1** (índice de la clave).
- Se genera un índice **Iv<sub>31</sub>** (índice de la clave).
- Se genera un índice **Salida** (índice de la clave).

9

### Rutina ran1:

```
# define IA 16807
# define IM 2147483647
# define AM (1.0/IM)
# define IQ 127773
# define IR 2836
# define NTAB 32
# define NDIV (1+(IM-1)/NTAB)
# define EPS 1.2 e-7
# define RNMIX (1.0-EPS)
```

```
float ran1(long*idum)
```

*/\* Generador de Números Aleatorios “Mínimo” de Park y Miller, con el desordenamiento de Bays –Durham y precauciones adicionales. Entrega una distribución<sup>12</sup> aleatoria uniforme entre 0.0 y 1.0 (excluyendo los puntos extremos) Llama mediante idum un número entero negativo para inicializar. De allí en adelante, no altera idum entre sucesivos valores en una secuencia. RNMIX debe aproximarse al mayor valor de punto flotante que es menor que 1*

```
{
  int j;
  long k;
  static long iy=0;
  static long iv[NTAB];
  float temp;

  if (*idum <= 0 || !iy) {      // Inicializa
    if (-(*idum) < 1) *idum = 1;  // Nos aseguramos de prevenir idum=0
    else *idum = -(*idum);
    for (j=NTAB+7; j>=0; j--) {  // Carga la tabla de desordenamiento despues de 8 “warm-ups”
      k=(*idum)/IQ;
      *idum = IA*( *idum-k*IQ)-IR*k;
      if(*idum<0) *idum += IM;
      if (j<NTAB) iv[j]= *idum;
    }
    iy=iv[0];
  }

  k= (*idum)/IQ;                // comenzar aqui cuando no se está inicializando.
  *idum=IA*( *idum-k*IQ)-IR*k;  // Calcular idum=(IA*idum) % IM sin rebalse,
  if (*idum <0) *idum += IM     // mediante el Método de Schrage.
  j = iy/NDIV;                  // Estará en el rango 0 ... NTAB-1
  iy = iv[j];                   // Entrega el valor previamente almacenado y rellena
  iv[j] = *idum;                // la tabla de desordenamiento
  if((temp=AM*iy) > RNMIX) return RNMIX; // Debido a que los usuarios no esperan valores extremos.
  else return temp.
}
```

La rutina **ran1** aprueba todas aquellas pruebas (tests) estadísticos que se sabe **ran0** no aprueba. De hecho, no se conoce de ninguna prueba estadística que **ran1** no logre aprobar, excepto cuando el número de llamadas comienza a ser del orden del período  $m$ , digamos  $> 10^8 \approx m/20$ .

---

<sup>12</sup> deviate

### 1.2.2.3 Generador de L'Ecuyer: ran2.

Para situaciones en las cuales se precisa de secuencias aleatorias aún mayores, L'Ecuyer [6] ha dado una buena forma de combinar dos diferentes secuencias con diferentes períodos de forma de obtener una nueva secuencia, cuyo período es el mínimo común múltiplo de los dos períodos. La idea básica es sencillamente sumar las dos secuencias modulo el modulo de cualquiera de ellos. Un truco para evitar un valor intermedio que desborde el tamaño de la palabra entera es restar más bien que sumar y luego sumar de retorno la constante  $m-1$  si el resultado es  $\leq 0$ , de manera de pertenecer dentro del intervalo deseado  $0, \dots, m-1$ .

Notese que no es necesario que esta resta sea capaz de alcanzar todos los valores  $0, \dots, m-1$  desde cada valor de la primera secuencia. Considere el absurdo caso extremo cuando el valor sustraído esta sólo entre 1 y 10: la secuencia resultante no será aún menos aleatoria que la primera secuencia por sí misma. Desde un punto de vista práctico, es solo necesario que la segunda secuencia tenga un rango que cubra sustancialmente todo el rango de la primera. L'Ecuyer recomienda el uso de dos generadores  $m_1 = 2147483563$  (con  $a_1 = 40014$ ,  $q_1 = 53668$ ,  $r_1 = 12211$ ) y  $m_2 = 2147483399$  (con  $a_2 = 40692$ ,  $q_2 = 52774$ ,  $r_2 = 3791$ ). Ambos módulos son ligeramente menores a  $2^{31}$ . Los períodos  $m_1 - 1 = 2 \times 3 \times 7 \times 631 \times 81031$  y  $m_2 - 1 = 2 \times 19 \times 31 \times 1019 \times 1789$ , comparten sólo el factor 2, luego el período del generador combinado es  $\approx 2.3 \times 10^{18}$ . Para los computadores actuales, el exceder el período es una imposibilidad práctica.

La combinación de los dos generadores quiebra la correlación serial en forma considerable. A pesar de ello, se recomienda el desordenamiento adicional que se implementa en la siguiente rutina, **ran2**. Pensamos<sup>13</sup> que, dentro de los límites de la precisión de punto flotante, **ran2** proporciona Números Aleatorios “perfectos”<sup>14</sup>.

```
# define IM1 2147483563
# define IM2 2147483399
# define AM (1.0/IM1)
# define IMM1 (IM1-1)
# define IA1 40014
# define IA2 40692
# define IQ1 53668
# define IQ2 52774
# define IR1 12211
# define IR2 3791
# define NTAB 32
# define NDIV (1+IMM1/NTAB)
# define EPS 1.2E-7
# define RNMIX (1.0-EPS)
```

```
float ran2(long*idum)
```

---

<sup>13</sup> Autores de N.R.

<sup>14</sup> Se pagará US\$1000 al primer lector que demuestre lo contrario.

*/\* Generador de Números Aleatorios de período largo de L'Ecuyer, con desordenamiento de Bays-Durham y salvaguardas adicionales. Entrega una distribución aleatoria uniforme entre 0.0 y 1.0 (excluyendo los extremos). Llama un número entero negativo idum para inicializar; de allí en adelante no se altera idum entre valores sucesivos de una secuencia. RNMX debe aproximarse al más grande valor en punto flotante que es menor que 1.0. \*/*

```
{
int j;
long k;
static long idum2=123456789;
static long iy=0;
static long iv[NTAB];
float temp;

if (*idum <= 0) { // Inicializa.
    if (-(*idum) < 1) *idum=1; // Prevención para no tener idum=0.
    else *idum = -(*idum);
    idum2=(*idum);
    for (j=NTAB+7;j>0;j--) { //n Carga la tabla para desordenar.
        k=(*idum)/IQ1;
        *idum=IA1*(*idum-k*IQ1)-k*IR1;
        if(*idum < 0) *idum +=IM1;
        if (j < NTAB) iv[j] = *idum;
    }
    iy = iv[0];
}
k=(*idum)/IQ1; // Comenzar aqui cuando no se inicializa
*idum=IA1*(*idum-k*IQ1)-k*IR1; // Calcular idum=(IA1*idum) % IM1 sin
if (*idum < 0) *idum += IM1; // desbordamiento mediante el Método de Schrage.
k=idum2/IQ2;
idum2=IA2*(idum2-k*IQ2)-k*IR2; // Calcular idum2=(IA2*idum) % IM2 de la misma forma
if (idum2 < 0) idum2 += IM2;
j=iy/NDIV; // Estará en el rango 0.. NTAB-1
iy=iv[j]-idum2; // Aqui idum es desordenado, idum e idum2 se
iv[j] = *idum; // combinan para generar la salida.
if (iy < 1) iy += IMM1;
if ((temp=AM*iy)> RNMX) return RNMX; // Debido a que los usuarios no esperan valores extremos.
else return temp;
}
```

L'Ecuyer entrega una lista de otros generadores cortos que pueden ser combinados en mayores, incluyendo generadores que pueden ser implementados en una aritmética de enteros de 16 bits.

#### 1.2.2.4 Tiempos relativos y recomendaciones:

Los tiempos de ejecución son inevitablemente dependientes de la máquina empleada. A pesar de ello, se presenta en la siguiente tabla una comparación de los tiempos relativos, para máquinas típicas de los generadores uniformes de Números Aleatorios discutidos hasta aquí. Valores pequeños de tiempo de ejecución indican generadores más rápidos.

Generador	Tiempo relativo de ejecución
Ran0	$\equiv 1.0$
Ran1	$\approx 1.3$
Ran2	$\approx 2.0$

Haciendo un balance, se recomienda **ran1** para uso general. Es portable, basado en el generador del Estandar Mínimo de Park y Miller, con un desordenamiento adicional, y no se le conoce otro defecto que no sea el sobrepasamiento del período.

Si se genera más de 100 000 000 de Números Aleatorios en un sólo cálculo (esto es, más de un 5% del período de **ran1**), se recomienda el uso de **ran2**, el cual tiene un período más largo.

### 1.3 Métodos para generar Números Aleatorios.

#### 1.3.1 Generadores de Congruencia Lineal (LCG).

Ya se comentó en el punto 1.2.1 que el generador de Números Aleatorios que más se utiliza en la actualidad en las implementaciones de los lenguajes computacionales son los Generadores de Congruencia Lineal. Como ya se mencionó anteriormente la filosofía tras su implementación, sólo mencionaremos aquí el siguiente teorema que proporciona elementos teóricos para optimizar la longitud del período de un  $LCG(a,c,m,I_0)$ <sup>15</sup>

#### **Teorema**<sup>16</sup> :

La secuencia Congruente Lineal definida por  $a, c, m$  e  $I_0$

$$I_{n+1} = (a I_n + c) \mod m$$

tiene una longitud de período  $m$ , si y sólo si:

---

<sup>15</sup> Notación de Anderson.

<sup>16</sup> Knuth [4], Hull y Dobell demostraron el caso general en 1962.

- i)  $c$  es primo relativo con  $m$ ,
- ii)  $b = a - 1$  es un múltiplo de  $p$ , para cada primo  $p$  que divide a  $m$ ,
- iii)  $b$  es múltiplo de 4, si  $m$  es múltiplo de 4.

Ejemplo 1 LCG(5, 1, 16, 1).

Obviamente este no es el caso de un generador de Números Aleatorios que usaríamos, por ejemplo, en una integral Monte Carlo. Sin embargo, en su simpleza ilustra las características esenciales de los LCG.

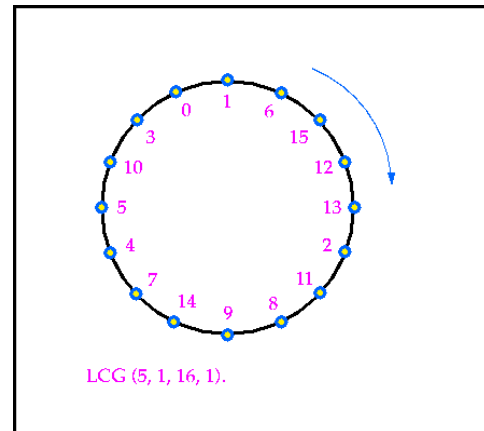
La secuencia de Números al azar generada por este algoritmo es:

1, 6, 15, 12, 13, 2, 11, 8, 9, 14, 7, 4, 5, 10, 3, 0, 1, 6, 15, 12, 13, 2, 11, 8, 9, 14, ...

El ciclo de Números Aleatorios es:

Podemos observar las siguientes características:

- i) El período, esto es el número de enteros pares e impares. Es muy instructivo mirar la secuencia en binario donde puede observarse que la secuencia está fuertemente correlacionada. Puede observarse la alternancia “1” y “0” del último bit en la tercera columna, indicando la gran correlación de este bit de bajo orden. La cuarta columna se obtiene tomando el valor de la segunda columna y dividiendo por  $m$ , en este caso, 16.



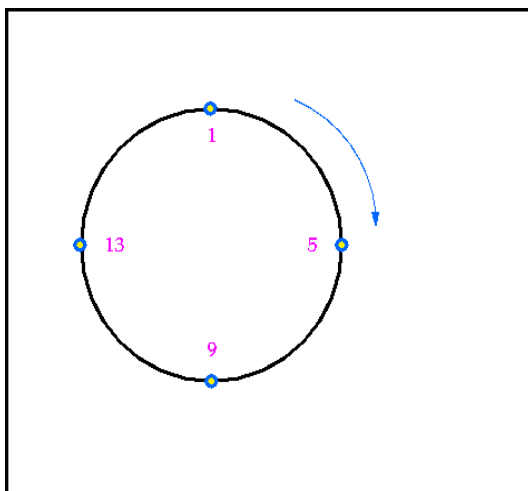
n	"Random" Integer - $X_n$	"Random" Binary - $X_n$	"Random" Real - $X_n$
0	1	0001	0.0625
1	6	0110	0.3750
2	15	1111	0.9375
3	12	1100	0.7500
4	13	1101	0.8125
5	2	0010	0.1250
6	11	1011	0.6875
7	8	1000	0.5000
8	9	1001	0.5625
9	14	1110	0.8750
10	7	0111	0.4375
11	4	0100	0.2500
12	5	0101	0.3125
13	10	1010	0.6250
14	3	0011	0.1875
15	0	0000	0.0000
Average			0.4688
Variance			0.0830

- iii) Debido a que cada entero se obtiene del anterior solamente, seleccionando cualquier semilla inicial desde 0 a 15 produce sólo un corrimiento cíclico de la secuencia. Por tanto, todo lo que una diferente selección de la semilla inicial hace, es desplazar el punto de partida en la secuencia ya determinada por los parámetros de LCG.
- iv) Finalmente, se hace notar que el promedio de los números reales es 0,4688 y la varianza 0,0830. El alejamiento de estos valores de los ideales de 0,5 y 0,0833 se debe al corto período de la secuencia y a la mala resolución de los números reales generados. Estas condiciones de promedio y varianza son condiciones necesarias pero no suficientes para un buen generador de Números Aleatorios.

### Ejemplo 2. LCG(5,0,16,1)

La secuencia de Números al azar generada por este algoritmo es:

1, 5, 9, 13, 1, 5, 9, 13, ...



- i) Nótese que ahora se tiene un período 4, esto es,  $m/4$ . De hecho, cuando  $m$  es una potencia de 2 y  $c = 0$ , el máximo período es  $2^{m-2}$ . Aquí nuevamente observamos que los bits de bajo orden no son aleatorios. De hecho, los dos bits menos significantes son constantes e iguales a 01.

n	"Random" Integer - $X_n$	"Random" Binary - $X_n$	"Random" Real - $X_n$
0	1	0001	0.0625
1	5	0101	0.3125
2	9	1001	0.5625
3	13	1101	0.8125
Average			0.4375
Variance			0.0781

- ii) También la secuencia está correlacionada pues todos los enteros sucesivos difieren en 4 de sus predecesores.



### 1.3.2 Otros Generadores de Números Aleatorios.

El hecho de utilizar ampliamente los generadores de Secuencia Congruente lineal (LCG) no indica que estos sean los únicos generadores de Números Aleatorios existentes. A continuación se mencionará algunos de estos métodos, aunque no se analizará en detalle su funcionamiento. Algunos de estos métodos son muy importantes, mientras que otros son interesantes puesto que no son tan buenos como una persona podría esperar.

Uno de las falacias más comunes encontradas en conexión con la generación de Números Aleatorios es la idea de que podemos tomar un buen generador y modificarlo un poco de manera de tener un generador “aún más aleatorio”. Esto a menudo es falso. Por ejemplo, se sabe que :

$$I_{n+1} = (a I_n + c) \mod m$$

conduce a razonablemente buenos Números Aleatorios. Entonces ¿No sería acaso la secuencia:

$$I_{n+1} = ((a I_n) \mod (m+1) + c) \mod m$$

incluso más aleatoria?

La respuesta es que la nueva secuencia es, probablemente, mucho menos aleatoria.

#### 1.3.2.1 Método Cuadrático Congruente

Teniendo lo anterior en mente, estudiemos un enfoque distinto: probemos de generalizar el Método Lineal Congruente a un Método Cuadrático Congruente:

$$I_{n+1} = (d I_n^2 + a I_n + c) \mod m$$

Esta secuencia tiene un período de máxima longitud y con restricciones no mucho más severas que aquellas del Método Lineal Congruente.

Un Método cuadrático interesante es el propuesto por R.R. Conveyou cuando  $m$  es una potencia de 2 por hallarse relacionado con el Método original de von Neumann del cuadrado central. En el Método de Conveyou,

$$I_0 \mod 4 = 2 \quad I_{n+1} = I_n (I_n + 1) \mod 2^e$$

### 1.3.2.2 Secuencias de Fibonacci

Es la más simple secuencia en que  $I_{n+1}$  depende de más de uno de los valores precedentes:

$$I_{n+1} = (I_n + I_{n-1}) \bmod m$$

Este generador fue considerado al inicio de los 50 y entrega usualmente una longitud de período mayor que  $m$ , pero las pruebas estadísticas han demostrado que los números producidos por la secuencia de Fibonacci son definitivamente no satisfactoriamente aleatorios. El interés actual en estas secuencias es que constituyen un hermoso “mal ejemplo” de generador de Números Aleatorios.

## 1.4 Pruebas Estadísticas.

Si se le pidiera a una persona que escriba 100 dígitos decimales al azar la probabilidad de que lo haga satisfactoriamente es muy pequeña. Tendemos a evitar cosas que parecen no aleatorias tales como pares de dígitos iguales y adyacentes (aunque uno de cada 10 dígitos debe ser igual a su predecesor). Si por otro lado se le pidiera a esa persona que observe una Tabla de verdaderos Números Aleatorios, es probable que diga que dicha Tabla no es aleatoria; su ojo encontraría ciertas regularidades.

Encontramos regularidades en nuestros números telefónicos, en nuestros números de rut, de patentes, etc. como ayuda memoria. El punto de lo anterior es que no podemos confiar en nosotros mismos para decidir si una secuencia de números es aleatoria o no. Debemos tener algunas pruebas mecánicas para aplicar. Mencionaremos algunas de ellas.

### 1.4.1 Pruebas $\chi^2$ (Chi - cuadrado)

La prueba  $\chi^2$  es probablemente el más conocido de todas las pruebas estadísticas, y es el método básico que es usado en conexión con muchas otras pruebas.

Discutiremos la prueba de  $\chi^2$  aplicada a un caso particular de Números Aleatorios. Imaginemos que se tiene un Generador de Números Aleatorios que se desea evaluar mediante esta prueba. Entonces procedemos de la siguiente forma: primero generamos una cantidad de Números Aleatorios, por ejemplo, entre 0 y 1. Para fijar ideas pensemos en unos 10 000 Números, los que deberán ser contabilizados en una de las diez clases en que dividiremos el intervalo  $[0,1]$ . Estas clases serán  $[0,0.1)$ ,  $[0.1,0.2)$ , ...,  $[0.9,1.0)$ . El objeto de la prueba es determinar si las frecuencias observadas en cada clase son significativamente diferentes de aquellas que se esperaría si la hipótesis fuera verdadera (en este caso, que la distribución es uniforme).

En nuestro caso, esperamos que las frecuencias en cada clase sean las mismas e iguales a 1000 ( $10\,000/10$ ). Estas son las frecuencias esperadas,  $E_i$ . El siguiente paso es contar el número de Números Aleatorios que pertenecen a cada clase. Estas son las frecuencias observadas,  $O_i$ .

Obviamente debe cumplirse:

$$\sum_{i=1}^{10} E_i = \sum_{i=1}^{10} O_i = 10000 \quad (= n, \text{número de observaciones})$$

La prueba estadística  $\chi^2$  se calcula entonces utilizando:

$$\chi^2 = \sum_{i=1}^{10} \frac{(O_i - E_i)^2}{E_i}$$

El valor crítico para la prueba puede hallarse en tablas estadísticas (ver anexo 1) o usando la distribución apropiada. Si la prueba estadística excede el valor crítico entonces la hipótesis nula (esto es, la hipótesis que debería suceder, en nuestro caso, que los números si son Números Aleatorios) no puede ser aceptada y es rechazada. El valor crítico para esta prueba, por convención es un nivel de significancia de un 5%. Como puede verse en la tabla del anexo 1, ello significa  $\alpha=0.05$ , lo que con 9 ( $=10 - 1$ ) grados de libertad da el valor crítico:

$$\chi_{0.05,9}^2 = 16.92$$

Los grados de libertad se calculan restando al número de clases (en este ejemplo, 10) el número de constreñimientos lineales de la prueba (en este caso, 1). Cuando la prueba estadística es menor que el valor crítico, la hipótesis nula es aceptada y la muestra original de datos es aceptada como aleatoria.

El nivel convencional de significancia adoptado para probar la hipótesis es 5%. Este resultado significa que el 95% de los valores para la prueba estadística se hallan bajo el valor crítico equivalente. Por lo tanto, la probabilidad de observar un valor igual o mayor al actual es 0.05.

En el ejemplo desarrollado, si  $\chi_{0.05,9}^2 < 16.92$ , entonces se considerará al generador de Números Aleatorios como verdaderamente aleatorio, y viceversa.

Así, por ejemplo si los datos de nuestro ejemplo fueran los de la tabla siguiente,

Tabla.

Categoría	Frecuencia Observada	Frecuencia Esperada
0.100	1002	1000
0.200	1053	1000
0.300	963	1000
0.400	991	1000
0.500	982	1000
0.600	1034	1000
0.700	1020	1000
0.800	989	1000
0.900	973	1000
1.000	993	1000
Total	10000	10000

Entonces

$$\chi^2_{0.05,9} = 7.042$$

Y los datos correspondería a una secuencia Aleatoria.

#### 1.4.2 Otras Pruebas Estadísticas.

Existen otras pruebas estadísticas que permiten determinar la “aleatoriedad” de un generador de Números Aleatorios. Dado que no es el tema a desarrollar en el presente documento, se hará solamente mención de los más utilizados en la actualidad.

- 1) Prueba de Kolmogorov- Smirnov.
- 2) Prueba de Anderson-Darling.
- 3) Prueba de permutaciones.
- 4) Prueba de ordenamiento invertido.
- 5) Prueba sobre y bajo la mediana.
- 6) Prueba Espectral.

Entre otros.

En particular, la Prueba Espectral es muy importante pues no sólo los buenos generadores de Números Aleatorios pasan esta prueba, sino que todos los generadores que se sabe actualmente son malos generadores aleatorios *no pasan* esta prueba. La Prueba de Kolmogorov-Smirnov, a diferencia de la Prueba de  $\chi^2$  que se aplica a un número finito de categorías, se aplica al caso en que las cantidades aleatorias asuman infinitos valores, aún

cuando sólo un finito número de éstos pueda representarse en el computador. Por ejemplo, permite analizar el intervalo  $[0,1]$  como si se tratara de números reales aleatorios.

## 1.5 A manera de resumen.

Siguiendo a Donald Knuth [4], podemos preguntarnos en este momento: “¿Y cuál es el resultado de toda esta teoría? ¿Cuál es el generador de Números Aleatorios, simple, virtuoso que puedo utilizar en mis programas de manera de tener una fuente confiable de Números Aleatorios?.

Según Knuth, los siguientes procedimientos entregan los generadores de Números Aleatorios “más bonitos” y “más simples”: al comienzo del programa, asignar a alguna variable entera  $I$  el valor  $I_0$ . Esta variable  $I$  se usará sólo con el propósito de generar Números Aleatorios. Cuando se requiera un nuevo Número Aleatorio, utilice:

$$I \leftarrow (aI + c) \bmod m$$

y use el nuevo valor de  $I$  como un valor aleatorio. Es necesario escoger  $I_0$ ,  $a$ ,  $c$  y  $m$  apropiadamente y usar sabiamente los Números Aleatorios, según los siguientes principios:

- i)  $I_0$  puede ser escogida arbitrariamente. Si el programa se corre varias veces y se desea una fuente distinta de Números abitrarios, cada vez, escoja  $I_0$  como el último valor alcanzado por  $I$  en la corrida precedente, o, mejor, asigne  $I_0$  a la fecha y hora de su computador.
- ii) El número  $m$  debe ser grande, al menos  $2^{30}$ . Puede ser conveniente que sea tomado del tamaño de la palabra del computador. No debe haber error de roundoff.
- iii) Si  $m$  es potencia de 2, escoja  $a$  de forma que  $a \bmod 8 = 5$ . Si  $m$  es una potencia de 10, escoja  $a$  de modo que  $a \bmod 200 = 21$ .
- iv) El multiplicador  $a$  debe elegirse preferiblemente entre  $0.01m$  y  $0.99m$  sus dígitos binarios o decimales no deben tener una patrón simple y regular. El multiplicador debe pasar la Prueba Espectral.
- v) El valor de  $c$  no es importante cuando  $a$  es un buen multiplicador, excepto que  $c$  no debe tener factor común con  $m$ .
- vi) Los dígitos menos significativos de  $I$  no son muy aleatorios por lo tanto, decisiones basdas en el número  $I$  deben siempre ser influenciadas primariamente por los dígitosd más significantes. Genralmente es mejor pensar en  $I$  como una fracción aleatoria,  $I/m$  entre 0 y 1 antes que considerar a  $I$  como un entero

aleatorio entre 0 y  $m-1$ . Para calcular un entero aleatorio entre 0 y  $k-1$ , uno debe multiplicar por  $k$  y truncar el resultado.

- vii) Una limitación importante en la aleatoriedad de la secuencia es el hecho que la “exactitud” en dimensión- $t$  será del orden de  $m^{1/t}$ . Para Monte Carlo por ejemplo se necesitarán otras técnicas.

Finalmente, la política más prudente es correr cada programa al menos dos veces usando distintas fuentes de Números al azar antes de tomar las respuestas del programa seriamente.

## Bibliografía.

- [1] Knuth, D.E. 1981, Seminumerical Algorithms, 2nd ed., vol.2 of *The Art of Computer Programming*, Addison-Wesley, Chapter 3.
- [2] Press W.H., Teukolsky S.A., Vetterling W.T., Flannery B.P., *Numerical Recipes in C: The Art of Scientific Computing*. Cap. 7, Cambridge University Press, 2<sup>nd</sup> ed., 1992.

## Sitios Relevantes en Internet:

<http://www-cs-faculty.stanford.edu/~knuth>

Página Web de Knuth

<http://www.npac.syr.edu/projects/random/other-info.html>

Random Numbers on the Net

<http://www.itl.nist.gov/div898/handbook>

Engineering Statistics Handbook online

<http://csep1.phy.ornl.gov>

Computational Science Education Project

Anexo 1.  
 Tabla de distribución  $\chi^2$ . (Ver ejemplo página 18)

## I.2 Chi-Squared Distribution Tables

v df	Area in upper tail, $\alpha$									
	0.995	0.99	0.975	0.95	0.9	0.1	0.05	0.025	0.01	0.005
1	0.0000	0.0002	0.0010	0.0039	0.0158	2.7055	3.8415	5.0239	6.6349	7.8794
2	0.0100	0.0201	0.0506	0.1026	0.2107	4.6052	5.9915	7.3778	9.2103	10.5966
3	0.0717	0.1148	0.2158	0.3518	0.5844	6.2514	7.8147	9.3484	11.3449	12.8381
4	0.2070	0.2971	0.4844	0.7107	1.0636	7.7794	9.4877	11.1433	13.2767	14.8602
5	0.4117	0.5543	0.8312	1.1455	1.6103	9.2364	11.0705	12.8325	15.0863	16.7496
6	0.6757	0.8721	1.2373	1.6354	2.2041	10.6446	12.5916	14.4494	16.8119	18.5476
7	0.9893	1.2390	1.6899	2.1674	2.8331	12.0170	14.0671	16.0128	18.4753	20.2777
8	1.3444	1.6465	2.1797	2.7326	3.4895	13.3616	15.5073	17.5346	20.0902	21.9550
9	1.7349	2.0879	2.7004	3.3251	4.1682	14.6837	16.9190	19.0228	21.6660	23.5893
10	2.1559	2.5582	3.2470	3.9403	4.8652	15.9871	18.3070	20.4831	23.2093	25.1882
11	2.6032	3.0535	3.8158	4.5748	5.5778	17.2750	19.6751	21.9200	24.7250	26.7569
12	3.0738	3.5706	4.4038	5.2260	6.3038	18.5494	21.0261	23.3367	26.2170	28.2995
13	3.5650	4.1069	5.0087	5.8919	7.0415	19.8119	22.3621	24.7356	27.6883	29.8194
14	4.0747	4.6604	5.6287	6.5706	7.7895	21.0642	23.6848	26.1190	29.1413	31.3193
15	4.6009	5.2294	6.2621	7.2609	8.5468	22.3072	24.9958	27.4884	30.5779	32.8013
16	5.1422	5.8122	6.9077	7.9616	9.3122	23.5418	26.2962	28.8454	31.9999	34.2672
17	5.6972	6.4078	7.5642	8.6718	10.0852	24.7690	27.5871	30.1910	33.4087	35.7185
18	6.2648	7.0149	8.2308	9.3905	10.8649	25.9894	28.8693	31.5264	34.8053	37.1564
19	6.8440	7.6327	8.9066	10.1170	11.6509	27.2036	30.1435	32.8523	36.1908	38.5822
20	7.4339	8.2604	9.5908	10.8508	12.4426	28.4120	31.4104	34.1696	37.5662	39.9968
21	8.0337	8.8972	10.2829	11.5913	13.2396	29.6151	32.6705	35.4789	38.9321	41.4010
22	8.6427	9.5425	10.9823	12.3380	14.0415	30.8133	33.9244	36.7807	40.2894	42.7958
23	9.2604	10.1957	11.6885	13.0905	14.8479	32.0069	35.1725	38.0757	41.6384	44.1813
24	9.8862	10.8564	12.4011	13.8484	15.6587	33.1963	36.4151	39.3641	42.9798	45.5585
25	10.5197	11.5240	13.1197	14.6114	16.4734	34.3816	37.6525	40.6465	44.3141	46.9278
26	11.1603	12.1981	13.8439	15.3791	17.2919	35.5631	38.8852	41.9232	45.6417	48.2899
27	11.8076	12.8786	14.5733	16.1513	18.1138	36.7412	40.1133	43.1944	46.9630	49.6449
28	12.4613	13.5648	15.3079	16.9279	18.9392	37.9159	41.3372	44.4607	48.2782	50.9933
29	13.1211	14.2565	16.0471	17.7083	19.7677	39.0875	42.5569	45.7222	49.5879	52.3356
30	13.7867	14.9535	16.7908	18.4926	20.5992	40.2560	43.7729	46.9792	50.8922	53.6720
40	20.7065	22.1643	24.4331	26.5093	29.0505	51.8050	55.7585	59.3417	63.6907	66.7659
50	27.9907	29.7067	32.3574	34.7642	37.6886	63.1671	67.5048	71.4202	76.1539	79.4900
60	35.5346	37.4848	40.4817	43.1879	46.4589	74.3970	79.0819	83.2976	88.3794	91.9517
70	43.2752	45.4418	48.7576	51.7393	55.3290	85.5271	90.5312	95.0231	100.4250	104.2150
80	51.1720	53.5400	57.1532	60.3915	64.2778	96.5782	101.8790	106.6290	112.3290	116.3210
90	59.1963	61.7541	65.6466	69.1260	73.2912	107.5650	113.1450	118.1360	124.1160	128.2990
100	67.3276	70.0648	74.2219	77.9295	82.3581	118.4980	124.3420	129.5610	135.8070	140.1690