Probabilidad, Números Primos y Factorización de Enteros. Implementaciones en Java y VBA para Excel.

Walter Mora F.
wmora2@yahoo.com.mx
Escuela de Matemática
Instituto Tecnológico de Costa Rica

Palabras claves: Teoría de números, probabilidad, densidad, primos, factorización, algoritmos.

1.1 Introducción

"God may not play dice with the universe, but something strange is going on with the prime numbers." Paul Erdös, 1913-1996.

"Interestingly, the error $O(\sqrt{n\ln^3 n})$ predicted by the Riemann hypothesis is essentially the same type of error one would have expected if the primes were distributed randomly. (The law of large numbers.) Thus the Riemann hypothesis asserts (in some sense) that the primes are pseudorandom - they behave randomly, even though they are actually deterministic. But there could be some sort of "conspiracy" between members of the sequence to secretly behave in a highly "biase" or "non-random" manner. How does one disprove a conspiracy?." Terence Tao, Field Medal 2006.

Este trabajo muestra cómo algunos métodos estadísticos y probabilísticos son usados en teoría de números. Aunque en este contexto no se puede definir una medida de probabilidad en el sentido del modelo axiomático, los métodos probabilísticos se han usado para orientar

investigaciones acerca del comportamiento en promedio, de los números primos. Algunos de estos resultados se usan para hacer estimaciones acerca de la eficiencia en promedio de algunos algoritmos para factorizar enteros. Consideramos dos métodos de factorización, "ensayo y error" y el método "rho" de Pollard. Los enteros grandes tienen generalmente factores primos pequeños. Es normal tratar de detectar factores menores que 10^8 con el método de ensayo y error y factores de hasta doce o trece dígitos con alguna variación eficiente del método rho de Pollard. Los números "duros" de factorizar requieren algoritmos más sofisticados (un número duro podría ser, a la fecha, un entero de unos trescientos dígitos con solo dos factores primos muy grandes). Aún así, estos algoritmos (a la fecha) han tenido algún éxito solo con números (duros) de hasta unas doscientos dígitos, tras un largo esfuerzo computacional.

Además de discutir algunos algoritmos, se presenta la implementación en Java. En el apéndice se presentan algunas implementaciones en VBA Excel.

1.2 A los números primos les gustan los juegos de azar.

1.2.1 ¿La probabilidad de que un número natural, tomado al azar, sea divisible por p es $1/p$?

¿Qué significa "tomar un número natural al azar"?. Los naturales son un conjunto infinito, así que no tiene sentido decir que vamos a tomar un número al azar. Lo que si podemos es tomar un número de manera aleatoria en un conjunto finito $\{1,2,...,n\}$ y luego (atendiendo a la noción frecuencista de probabilidad) ver que pasa si n se hace grande (i.e. $n \longrightarrow \infty$).

Hagamos un pequeño experimento: Fijamos un número p y seleccionamos de manera aleatoria un número en el conjunto $\{1,2,...,n\}$ y verificamos si es divisible por p. El experimento lo repetimos m veces y calculamos la frecuencia relativa.

En la tabla que sigue, hacemos este experimento varias veces para n, m y p.

n	m	p	Frecuencia relativa
100000	10000	5	0.1944
			0.2083
			0.2053
			0.1993
10000000	100000	5	0.20093
			0.19946
			0.1997
			0.20089
100000000	1000000	5	0.199574
			0.199647

Tabla 1.1

Y efectivamente, parece que "la probabilidad" de que un número tomado al azar en el conjunto $\{1,2,...,n\}$ sea divisible por p es $1/p$

De una manera sintética: Sea $E_p(n) =$ los múltiplos de p en el conjunto $\{1,2,...,n\}$. Podemos calcular la la proporción de estos múltiplos en este conjunto, es decir, podemos calcular $\frac{E_p(n)}{n}$ para varios valores de n

n	Múltiplos de $p = 5$	Proporción
100	20	0.2
10230	2046	0.2
100009	20001	0.199992
1000000	199999	0.199999

Tabla 1.2

Parece que en el conjunto $\{1,2,...,n\}$, la proporción de los múltiplos de $p = 5$ se aproxima a $1/5$, conforme n se hace grande. ¿Significa esto que la probabilidad de que un número natural, tomado al azar, sea divisible por 5 es $1/5$?. Por ahora, lo único que podemos decir

es que este experimento sugiere que la densidad (o la proporción) de los múltiplos de 5 en $\{1,2,...,n\}$ parece ser $1/5$ conforme n se hace grande. Generalizando,

Definición 1.1 Sea E un conjunto de enteros positivos con alguna propiedad especial y sea $E(N) = E \bigcap \{1,2,...,N\}$. La densidad (o medida relativa) de E se define como

$$D[E] = \lim_{n\to\infty} \frac{E(n)}{n}$$

siempre y cuando este límite exista.

¿Es esta densidad una medida de probabilidad?. Para establecer una medida de probabilidad P, en el modelo axiomático, necesitamos un conjunto Ω ("espacio muestral"). En Ω hay una colección E de subconjuntos E_i (una $\sigma-$álgebra sobre Ω), llamados "eventos", con medida de probabilidad $P(E_i)$ conocida. La idea es extender estas medidas a una colección más amplia de subconjuntos de Ω. P es una medida de probabilidad si cumple los axiomas

1. $P(E_i) \geq 0$ para todo $E_i \in \Omega$,
2. Si $\{E_j\}$ es una colección de conjuntos disjuntos dos a dos en F, entonces
$$P\left(\bigcup_j E_j\right) = \sum_j P(E_j),$$
3. $P(\Omega) = 1$

Cuando el espacio muestral Ω es finito y los posibles resultados tienen igual probabilidad entonces $P(E) = \dfrac{|E|}{|\Omega|}$ define una medida de probabilidad.

La densidad D no es una medida de probabilidad porque no cumple el axioma 2.

La idea de la demostración [21] usa el Teorema de Mertens (ver más adelante). Si denotamos con E_p los múltiplos positivos de p y si suponemos que hay una medida de probabilidad P en $\mathbb{Z}^+$ tal que $P(E_p) = 1/p$, entonces $P\left(E_p \cap E_q\right) = (1-1/p)(1-1/q)$ para p, q primos distintos. De manera

inductiva y utilizando el teorema de Mertens se llega a que $P(\{m\}) = 0$ para cada entero positivo m. Luego,

$$P\left(\bigcup_{m\in\mathbb{Z}^+}\{m\}\right) = 1 \neq \sum_m P(\{m\}) = 0.$$

Aunque en el esquema frecuencista se puede ver la densidad como la "probabilidad" de que un entero positivo, escogido aleatoriamente, pertenezca a E, aquí identificamos este término con *densidad o proporción*. Tenemos,

Teorema 1.1 *La densidad de los naturales divisibles por p es $\frac{1}{p}$, es decir, si E_p es el conjunto de enteros positivos divisibles por p, entonces*

$$D[E_p] = \lim_{n\to\infty}\frac{E_p(n)}{n} = \frac{1}{p}$$

Prueba: Para calcular el límite necesitamos una expresión analítica para $E_p(n)$. Como existen p, r tales que $n = pk + r$ con $0 \leq r < p$, entonces $kp \leq n < (k+1)p$, es decir, hay exactamente k múltiplos positivos de p que son menores o iguales a n. Luego $E_p(n) = k = \frac{n-r}{p}$.

Por lo tanto,

$$\begin{aligned} D[E_p] &= \lim_{n\to\infty}\frac{E_p(n)}{n} = \lim_{n\to\infty}\frac{\frac{n-r}{p}}{n} \\ &= \lim_{n\to\infty}\frac{n-r}{pn} = \lim_{n\to\infty}\frac{1}{p} - \frac{r}{pn} = \frac{1}{p} \end{aligned}$$

1.2.2 Teorema de los Números Primos

$\pi(x)$ denota la cantidad de primos que no exceden x. Por ejemplo, $\pi(2) = 1$, $\pi(10) = 4$ y $\pi(\sqrt{1000}) = 11$.

Para la función $\pi(x)$ no hay una fórmula sencilla. Algunas fórmulas actuales son variaciones un poco más eficientes que la fórmula recursiva de Legendre (1808).

Fórmula de Legendre.

Esta fórmula esta basada en el principio de inclusión-exclusión. Básicamente dice que el conjunto $\{1,2,...,[\![x]\!]\}$ es la unión del entero 1, los primos $\leq x$ y los enteros compuestos $\leq x$,

$$[\![x]\!] = 1 + \pi(x) + \#\{\text{ enteros compuestos} \leq x\}$$

Un entero compuesto en $A = \{1,2,...,[\![x]\!]\}$ tiene al menos un divisor primo menor o igual a $\sqrt{x}$[1]. Esto nos ayuda a detectar los números compuestos en A: solo tenemos que contar los elementos de A con un divisor primo $\leq \sqrt{x}$.

$\lfloor\!\lfloor x \rfloor\!\rfloor = n$ si $n \leq x < n+1$. Entonces si $\lfloor\!\lfloor x/p \rfloor\!\rfloor = k$ significa que $kp \leq x$, i.e. $p < 2p < ... < k \cdot p \leq x$. Luego, la cantidad de enteros $\leq x$ divisibles por p es $\lfloor\!\lfloor x/p \rfloor\!\rfloor$.

Ahora, ¿$\#\{$ enteros compuestos $\leq x\}$ es igual a al conteo total de los múltiplos de cada primo $p_i \leq \sqrt{x}$? No, pues este conteo incluye a los propios primos p_i, así que hay que reponer con $\pi(\sqrt{x})$ para hacer una corrección. Pero también habría que restar los compuestos que son divisibles por p_i y p_j pues fueron contados dos veces, pero esto haría que los números divisibles por p_i, p_j, p_k fueran descontados una vez más de lo necesario así que hay que agregar una corrección para estos números, y así sucesivamente.

EJEMPLO 1.1 Si $x = 30$, los primos menores que $\lfloor\!\lfloor \sqrt{30} \rfloor\!\rfloor = 5$ son $2, 3$ y 5.

$\lfloor\!\lfloor 30/2 \rfloor\!\rfloor = 15$ cuenta $\{2,4,6,8,10,12,14,16,18,20,22,24,26,28,30\}$
$\lfloor\!\lfloor 30/3 \rfloor\!\rfloor = 10$ cuenta $\{3,6,9,12,15,18,21,24,27,30\}$
$\lfloor\!\lfloor 30/5 \rfloor\!\rfloor = 6$ cuenta $\{5,10,15,20,25,30\}$

En el conteo $\lfloor\!\lfloor 30/2 \rfloor\!\rfloor + \lfloor\!\lfloor 30/3 \rfloor\!\rfloor + \lfloor\!\lfloor 30/5 \rfloor\!\rfloor$:

- se contaron los primos $2, 3$ y 5.

[1]Esto es así pues si $n \in A$ y si $n = ab$, no podría pasar que a y b sean ambos $\geq \sqrt{x}$ pues sería una contradicción pues $n \leq x$.

- $6,12,18,24,30$ fueron contados dos veces como múltiplos de $2,\ 3$
- $10,20,30$ fueron contados dos veces como múltiplos de $2,\ 5$
- $15,30$ fueron contados dos veces como múltiplos de $3,\ 5$
- 30 fue contado tres veces como múltiplo de $2,3$ y 5.

Finalmente,

$$\begin{aligned} \#\{\text{ enteros compuestos} \leq 30\} &= \lfloor\!\lfloor 30/2 \rfloor\!\rfloor + \lfloor\!\lfloor 30/3 \rfloor\!\rfloor + \lfloor\!\lfloor 30/5 \rfloor\!\rfloor \\ &- \lfloor\!\lfloor 30/(2\cdot 3) \rfloor\!\rfloor - \lfloor\!\lfloor 30/(2\cdot 5) \rfloor\!\rfloor - \lfloor\!\lfloor 30/(3\cdot 5) \rfloor\!\rfloor \\ &+ \lfloor\!\lfloor 30/(2\cdot 3 + \cdot 5) \rfloor\!\rfloor \\ &= 31-3-5-3-2+1=19 \end{aligned}$$

El último sumando se agrega pues el 30 fue contado tres veces pero también se resto tres veces.

Observe ahora que en $\{1,2,...,30\}$ hay 19 compuestos y el 1, así que quedan 10 primos.

Sea p_i el $i-$ésimo primo. La fórmula de Legendre es,

$$1+\pi(x)=\pi\left(\sqrt{x}\right)+\lfloor\!\lfloor x \rfloor\!\rfloor-\sum_{p_i\leq\sqrt{x}}\left[\frac{x}{p_i}\right]+\sum_{p_i<p_j\leq\sqrt{x}}\left[\frac{x}{p_ip_j}\right]-\sum_{p_i<p_j<p_k\leq\sqrt{x}}\left[\frac{x}{p_ip_jp_k}\right]+\cdots$$

Para efectos de implementación es mejor poner $\alpha=\pi(\sqrt{x})$ y entonces la fórmula queda

$$1+\pi(x)=\pi\left(\sqrt{x}\right)+\lfloor\!\lfloor x \rfloor\!\rfloor-\sum_{i\leq\alpha}\left[\frac{x}{p_i}\right]+\sum_{i<j\leq\alpha}\left[\frac{x}{p_ip_j}\right]-\sum_{i<j<k\leq\alpha}\left[\frac{x}{p_ip_jp_k}\right]+\cdots$$

EJEMPLO 1.2 Calcular $\pi(100)$
Solución: Como $\sqrt{100}=10,$ solo usamos los primos $\{2,3,5,7\}$.

$$\begin{aligned}
1+\pi(100) &= \pi(10)+\lfloor\!\lfloor 100\rfloor\!\rfloor \\
&\quad -(\lfloor\!\lfloor 100/2\rfloor\!\rfloor+\lfloor\!\lfloor 100/3\rfloor\!\rfloor+\lfloor\!\lfloor 100/5\rfloor\!\rfloor+\lfloor\!\lfloor 100/7\rfloor\!\rfloor) \\
&\quad +\lfloor\!\lfloor 100/2\cdot 3\rfloor\!\rfloor+\lfloor\!\lfloor 100/2\cdot 5\rfloor\!\rfloor+\lfloor\!\lfloor 100/2\cdot 7\rfloor\!\rfloor+\lfloor\!\lfloor 100/3\cdot 5\rfloor\!\rfloor+\lfloor\!\lfloor 100/3\cdot 7\rfloor\!\rfloor+\lfloor\!\lfloor 100/5\cdot 7\rfloor\!\rfloor \\
&\quad -(\lfloor\!\lfloor 100/2\cdot 3\cdot 5\rfloor\!\rfloor+\lfloor\!\lfloor 100/2\cdot 3\cdot 7\rfloor\!\rfloor+\lfloor\!\lfloor 100/2\cdot 3\cdot 7\rfloor\!\rfloor+\lfloor\!\lfloor 100/3\cdot 5\cdot 7\rfloor\!\rfloor) \\
&\quad +\lfloor\!\lfloor 100/3\cdot 3\cdot 5\cdot 7\rfloor\!\rfloor \\
&= 4+100-(50+33+20+14)+(16+10+7+6+4+2)-(3+2+0+1)+0 \;=\; 26
\end{aligned}$$

El problema con esta fórmula es la cantidad de cálculos que se necesita para calcular las correcciones.

Las cantidad de partes enteras $\lfloor\!\lfloor x/(p_{i_1}p_{i_2}\cdots p_{i_k})\rfloor\!\rfloor$ corresponde a la cantidad de subconjuntos no vacíos $\{i_1,i_2,\cdots,i_k\}$ de $\{1,2,...,\alpha\}$, es decir, hay que calcular $2^\alpha-1$ partes enteras.

Si quisieramos calcular $\pi(10^{33})$, entonces, puesto que $\sqrt{10^{33}}=10^{18}$, tendríamos que tener los primos $\leq 10^{18}$ y calcular las partes enteras $\left[x/(p_{k_1}p_{k_2}...p_{k_j})\right]$ que corresponden al cálculo de todos los subconjuntos de $\{1,2,...,\pi(10^{18})\}$. Como $\pi(10^{18})=24739954287740860$, tendríamos que calcular

$$2^{24739954287740860}-1 \quad \text{partes enteras.}$$

que constituye un número nada razonable de cálculos.

Fórmula de Meisel.

La fórmula de Meisel es un re-arreglo de la fórmula de Legendre. Pongamos

$$\text{Legendre}(x,\alpha)=\sum_{i\leq\alpha}\left[\frac{x}{p_i}\right]-\sum_{i<j\leq\alpha}\left[\frac{x}{p_ip_j}\right]+\sum_{i<j<k\leq\alpha}\left[\frac{x}{p_ip_jp_k}\right]+\cdots$$

Así $\pi(x)=[\![x]\!]-1+\alpha-\text{Legendre}(x,\alpha)$ donde $\alpha=\pi(\sqrt{x})$, es decir, $\text{Legendre}(x,\alpha)-\alpha$ cuenta la cantidad de números compuestos $\leq x$ o, en otras palabras, los números $\leq x$ con al menos un divisor primo inferior a $\alpha=\sqrt{x}$.

Ahora $\text{Legendre}(x,\alpha)$ va a tener un significado más amplio: Si $\alpha\in\mathbb{N}$,

$$\text{Legendre}(x,\alpha) = \sum_{i\le\alpha}\left[\frac{x}{p_i}\right] - \sum_{i<j\le\alpha}\left[\frac{x}{p_i p_j}\right] + \sum_{i<j<k\le\alpha}\left[\frac{x}{p_i p_j p_k}\right] + \cdots$$

es decir, $\text{Legendre}(x,\alpha)-\alpha$ cuenta los compuestos $\le x$ que son divisibles por primos $\le p_\alpha$. La resta es necesaria pues la manera de contar cuenta también los primos $p_1,\ p_2,\ \ldots, p_\alpha$

Ahora, dividamos los enteros en cuatro grupos: $\{1\}$, $\{\text{primos } \le x\}$, $C_3 \cup C_4 =$ los compuestos $\le x$.

$[\![x]\!] = 1 + \pi(x) + \#C_3 + \#C_4$

$\#C_3$: Es la cantidad de números compuestos $\le x$ con al menos un divisor primo $\le p_\alpha$, es decir $\text{Legendre}(x,\alpha)-\alpha$.

$\#C_4$ son los compuestos $\le x$ cuyos divisores primos son $> p_\alpha$: Aquí es donde entra en juego la escogencia de α para determinar la cantidad de factores primos de estos números.

Sea p_i el $i-$ésimo primo. Sean p_α y p_β tal que $p_\alpha^3 \le x < p_{\alpha+1}^3$ y $p_\beta^2 \le x < p_{\beta+1}^2$. En otras palabras: $\alpha = \pi(\sqrt[3]{x})$ y $\beta = \pi(\sqrt{x})$.

Consideremos la descomposición prima de $n \in C_4$, $n = p_{i_1}\cdot p_{i_2}\cdots p_{i_k}$ con $\alpha < p_{i_1} < p_{i_2} < \ldots < p_{i_k}$ y $k \ge 2$. Como $p_{\alpha+1}^k \le p_{i_1}\cdot p_{i_2}\cdots p_{i_k} \le x < p_{\alpha+1}^3 \implies k = 2$.

Así que estos números en C_4 son de la forma $p_{\alpha+k}p_j \le x$, $a+k \le j$, $k = 1,2,\ldots$
Pero la cantidad de números $p_{\alpha+k}p_j$ es igual a la cantidad de p_j's tal que $p_j \le x/p_{\alpha+k}$: $\pi(x/p_{\alpha+k}) - (\alpha+k)$.

Además $\alpha < \alpha+k \le \beta$ pues si $\alpha+k = \beta$, $p_\beta \cdot p_\beta = p_\beta^2 \le x$ pero $p_{\beta+1}p_j \ge p_{\beta+1}^2 > x$.

Así, usando la fórmula $\sum_{i=1}^{n-1} i = n(n-1)/2$,

$$\#C_4 = \sum_{\alpha<i\le\beta}\{\pi(x/p_i) - (i-1)\} = \frac{1}{2}\beta(\beta-1) - \frac{1}{2}\alpha(\alpha-1) + \sum_{\alpha<i\le\beta}\pi(x/p_i)$$

¿Cuál es la ganancia?

Mientras que con la fórmula de Legendre necesitamos conocer $\pi(\sqrt{x})$ y calcular con primos $\leq \sqrt{x}$, con la fórmula de Meisel solo necesitamos conocer hasta $\pi(\sqrt[3]{x})$ y calcular con primos $\leq \sqrt[3]{x} < \sqrt{x}$.

EJEMPLO 1.3 1.1 Calcule $\pi(100)$ usando la fórmula de Meisel.

Solución: **Meisel:** Como $\alpha = \pi(\sqrt[3]{100}) = 2$ y $\beta = \pi(\sqrt{100}) = 4$, solo vamos a usar los primos $p_1 = 2,\ p_2 = 3,\ p_3 = 5,\ p_4 = 7$.

$$\begin{aligned} \text{Legendre}(100,2) &= \lfloor\!\lfloor 100/2 \rfloor\!\rfloor + \lfloor\!\lfloor 100/3 \rfloor\!\rfloor + \lfloor\!\lfloor 100/2\cdot 3 \rfloor\!\rfloor \\ &= 50+33-16 = 67 \end{aligned}$$

$$\begin{aligned} \text{Meisel}(100,2,4) &= \pi(100/5)+\pi(100/7) \\ &= \pi(20)+\pi(4) = 8+6 = 14 \end{aligned}$$

Así, $\pi(100) = 100+6-0-67-14 = 25$

A la fecha (2007) se conoce $\pi(x)$ hasta $x = 10^{22}$. *Mathematica* (Wolfram Research Inc.) implementa $\pi(x)$ con el comando `PrimePi`$[x]$ hasta $x \approx 8\times 10^{13}$. En esta implementación, si x es pequeño, se calcula $\pi(x)$ usando colado y si x es grande se usa el algoritmo Lagarias-Miller-Odlyzko.

Estimación asintótica de $\pi(x)$. Teorema de los números primos.

La frecuencia relativa $\pi(n)/n$ calcula la proporción de primos en el conjunto $A = \{1,2,\ldots,n\}$. Aunque la distribución de los primos entre los enteros es muy irregular, el comportamiento promedio si parece ser agradable. Basado en un estudio empírico de tablas de números primos, Legendre y Gauss (en 1792, a la edad de 15 años) conjeturan que la ley que gobierna

el cociente $\pi(n)/n$ es aproximadamente igual a $\dfrac{1}{\ln(n)}$.

En [9] se indica que Gauss y Legendre llegaron a este resultado, de manera independiente, estudiando la densidad de primos en intervalos que difieren en potencias de diez: notaron que la proporción de primos en intervalos centrados en $x = 10^n$ decrece lentamente y disminuye aproximadamente a la mitad cada vez que pasamos de x a x^2. Este fenómeno es muy bien modelado por $1/\ln(x)$ pues $1/\ln(x^2) = 0.5/\ln(x)$.

EJEMPLO 1.4 Frecuencia relativa y estimación.

n	$\pi(n)$	$\pi(n)/n$	$1/\ln(n)$
10^7	664579	0.0664579	0.0620420
10^8	5761455	0.0576145	0.0542868
10^9	50847534	0.0508475	0.0482549
10^{10}	455052511	0.0455052	0.0434294
10^{11}	4118054813	0.0411805	0.0394813
10^{11}	37607912018	0.0376079	0.0361912

Tabla 1.3

Acerca de este descubrimiento, Gauss escribió a uno de sus ex-alumnos, Johann Franz Encke, en 1849

> *"Cuando era un muchacho considere el problema de cuántos primos había hasta un punto dado. Lo que encontré fue que la densidad de primos alrededor de x es aproximadamente $1/\ln(x)$."*

La manera de interpretar esto es que si n es un número "cercano" a x, entonces es primo con "probabilidad" $1/\ln(x)$. Claro, un número dado es o no es primo, pero esta manera de ver las cosas ayuda a entender de manera muy intuitiva muchas cosas acerca de los primos.

Lo que afirma Gauss es lo siguiente: Si Δx es "pequeño" comparado con x (en el mundillo asintótico esto quiere decir que $\Delta x/x \to 0$ conforme $x \to \infty$) entonces

$$\frac{\pi(x+\Delta x)-\pi(x)}{\Delta x} \approx \frac{1}{\ln(x)}$$

$\frac{\pi(x+\Delta x)-\pi(x)}{\Delta x}$ es la densidad de primos en le intervalo $[x,\, x+\Delta x]$ y $\frac{1}{\ln(x)}$ es el promedio estimado en este intervalo.

Por esto decimos: $1/\ln(x)$ es la "probabilidad" de que un número n, en las cercanías de x, sea primo.

Para hacer un experimento, podemos tomar $\Delta x = \sqrt{x}$ (que claramente es dominada por x),

x	$\pi(x+\Delta x)-\pi(x)$	$\frac{\pi(x+\Delta x)-\pi(x)}{\Delta x}$	$\frac{1}{\ln(x)}$
10	2	0.632	0.434
100	4	0.4	0.217
1000	5	0.158	0.144
10000	11	0.11	0.108
100000	24	0.075	0.086
1000000	75	0.075	0.072
10000000	197	0.0622	0.062
100000000	551	0.0551	0.054
1000000000	1510	0.0477	0.048
10000000000	4306	0.0430	0.043
100000000000	12491	0.0395	0.039
1000000000000	36249	0.0362	0.036

Hadamard y de la Vallée Poussin probaron en 1896, usando métodos basados en análisis complejo, el

Teorema 1.2 (Teorema de los Números Primos) *Sea* $\mathrm{li}(x) = \int_2^x \frac{dt}{\ln(t)}$. $\pi(x) \sim \mathrm{li}(x)$, *es decir,* $\lim_{x\to\infty} \frac{\pi(x)}{\mathrm{li}(x)} = 1$

La conjetura de Legendre era $\pi(x) \sim x/\ln(x)$. Esta expresión se usa mucho cuando se hacen estimaciones "gruesas":

Teorema 1.3 $\mathrm{li}(x) \sim x/\ln(x)$, *es decir* $\lim_{x\to\infty} \dfrac{\pi(x)}{x/\ln(x)} = 1$

Prueba: Para relacionar $\mathrm{li}(x)$ con $x/\ln(x)$, integramos por partes,

$$\begin{aligned}
\mathrm{li}(x) &= \int_2^x \frac{dt}{\ln(t)}, \quad \text{tomamos} \quad u = 1/\ln(t) \quad \text{y} \quad dv = dt, \\
&= \left.\frac{t}{\ln(t)}\right|_2^x - \int_2^x t \cdot \frac{-1/t\,dt}{\ln^2(t)} \\
&= \frac{x}{\ln(x)} + \int_2^x \frac{dt}{\ln^2(t)} + K_1, \quad \text{tomamos} \quad u = 1/\ln^2(t) \quad \text{y} \quad dv = dt, \\
&= \frac{x}{\ln(x)} + \frac{x}{\ln^2(x)} + 2\int_2^x \frac{dt}{\ln^3(t)} + K_2
\end{aligned}$$

Ahora vamos a mostrar que $2\displaystyle\int_2^x \frac{dt}{\ln^3(t)} + K_2 = O\left(\frac{x}{\ln^2 x}\right)$. Para esto, vamos a usar el hecho de que $\sqrt{x} = O\left(x/\ln^2 x\right)$.

Primero que todo observemos que solo necesitamos mostrar que $\displaystyle\int_2^x \frac{dt}{\ln^3(t)} = O\left(\frac{x}{\ln^2(x)}\right)$ pues como $\dfrac{x}{\ln^2(x)}$ tiende a infinito, podemos despreciar K_2. Además podemos ajustar la constante involucrada en la definición de la $O-$grande para "absorver" el coeficiente 2.

Como $\displaystyle\int_2^x = \int_2^e + \int_e^{\sqrt{x}} + \int_{\sqrt{x}}^x = K + \int_e^{\sqrt{x}} + \int_{\sqrt{x}}^x$, nos vamos a concentrar en estas dos últimas integrales.

Puesto que $e \le t \Longrightarrow \dfrac{1}{\ln^3 t} < 1$. Luego,

$$\int_e^{\sqrt{x}} \frac{dt}{\ln^3(t)} < \int_e^{\sqrt{x}} 1\,dt = \sqrt{x} - e < \sqrt{x}, \quad \text{es decir,} \quad \int_e^{\sqrt{x}} \frac{dt}{\ln^3(t)} = O\left(x/\ln^2 x\right).$$

Ahora, la otra integral. Puesto que $t < x$ entonces $\dfrac{x}{t} > 1$. Multiplicando la segunda integral por $\dfrac{x}{t}$ obtenemos,

$\displaystyle\int_{\sqrt{x}}^{x} \frac{dt}{\ln^3(t)} < x\int_{\sqrt{x}}^{x} \frac{dt}{t\ln^3(t)}.$

Usando la sustitución $u = \ln t$,

$$\begin{aligned} x\int_{\sqrt{x}}^{x} \frac{dt}{t\ln^3(t)} &= x\left(\frac{\ln^{-2} t}{-2}\right)\Bigg|_{\sqrt{x}}^{x} \\ &= x\left(\frac{1}{2\ln^2\sqrt{x}} - \frac{1}{2\ln^2 x}\right) \\ &< x\frac{1}{2\ln^2\sqrt{x}} = \frac{x}{\ln^2 x} = O\left(x/\ln^2 x\right) \end{aligned}$$

Finalmente, $\mathrm{li}(x) = \dfrac{x}{\ln x} + O\left(x/\ln^2 x\right)$.

La definición de $O-$grande nos permite dividir a ambos lados por $x/\ln x$, entonces

$$\frac{\mathrm{li}(x)}{x/\ln x} = 1 + O(1/\ln x)$$

y como $1/\ln x \longrightarrow 0$ conforme $x \longrightarrow \infty$,

$$\mathrm{li}(x) \sim x/\ln(x)$$

como queríamos.

$\lim\limits_{x\to\infty} \dfrac{\pi(x)}{x/\ln(x)} = 1$ debe entenderse en el sentido de que $x/\ln(x)$ aproxima $\pi(x)$ con un *error relativo* que se aproxima a cero conforme $x \to \infty$, aunque el error absoluto nos puede parecer muy grande.

Por ejemplo, si $n = 10^{13}$ (un número pequeño, de unos 13 dígitos solamente) entonces, una estimación de $\pi(10^{13}) = 346065536839$ sería $10^{13}/\ln(10^{13}) \approx 334072678387$. El error

relativo es $(\pi(n) - n/\ln(n))/\pi(n) = 0.034,$ es decir un 3.4%.

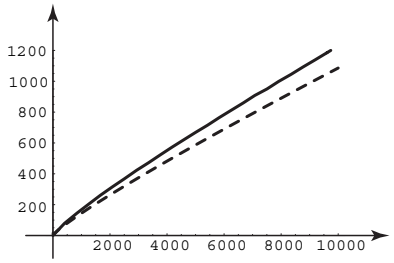


Figura 1.1 Comparando $x/\ln(x)$ con $\pi(x)$.

1.2.3 Teorema de Mertens.

¿Qué mide $\left(1-\frac{1}{2}\right)\left(1-\frac{1}{3}\right)\left(1-\frac{1}{5}\right)\cdots = \prod_{\substack{2\le p\le G,\\ p \text{ primo}}}\left(1-\frac{1}{p}\right)$?

$1/p$ es, a secas, la proporción de números en el conjunto $\{1,2,...,n\}$ que son divisibles por p. Luego $1-1/p$ sería la proporción de números en este conjunto que no son divisibles por p.

Aquí estamos asumiendo demasiado porque esta proporción no es exactamente $1/p$. Este número solo es una aproximación.

Si "ser divisible por p" es un evento independiente de "ser divisible por q", $\left(1-\frac{1}{p}\right)\left(1-\frac{1}{q}\right)$ sería la proporción de números en el conjunto $\{1,2,...,n\},$ que *no* son divisibles por p ni por q.

En general, $\prod_{\substack{2\le p\le G,\\ p \text{ primo},}}\left(1-\frac{1}{p}\right)$ sería una estimación de la proporción de números en el conjunto $\{1,2,...,n\},$ que son divisibles por ninguno de los primos menores o iguales a G : Esto si tiene utilidad práctica, como veremos más adelante.

Hay que tener algunos cuidados con esta fórmula. Si la "probabilidad" de que un que un número n sea primo es la probabilidad de que no sea divisible por un primo $p \leq \sqrt{x}$, entonces podríamos concluir erróneamente que $\Pr[X es primo] = \prod_{\substack{2 \leq p \leq \sqrt{x}, \\ p \text{ primo}}} \left(1 - \frac{1}{p}\right)$

Esta conclusión no es correcta pues $\prod_{\substack{2 \leq p \leq \sqrt{x}, \\ p \text{ primo}}} \left(1 - \frac{1}{p}\right) \not\sim 1/\ln(x)$ como estable el Teorema de Mertens (Teorema 1.4).

EJEMPLO 1.5 Hagamos un experimento. Sea $d_n = \#\{m \leq n : m \text{ es divisible por } 2, 3, 5, \text{o } 7\}$.

n	d_n	d_n/n
103790	80066	0.7714230658059543
949971	732835	0.7714288120374201
400044	308605	0.7714276429592745
117131	90359	0.7714354013881893
124679	96181	0.7714290297483939

Tabla 1.4

La proporción de números naturales $\leq n$ divisibles por $2, 3, 5$ es ≈ 0.7714. Así, $1 - 0.7714 = 0.2286$ es la proporción de números en $\{1, 2, ..., n\}$ que *no* son divisibles por los primos $2, 3, 5$ y 7.

Y efectivamente, $\left(1 - \frac{1}{2}\right)\left(1 - \frac{1}{3}\right)\left(1 - \frac{1}{5}\right)\left(1 - \frac{1}{7}\right) = 0.228571.$

Si intentamos calcular el producto para cantidades cada vez grandes de primos, rápidamente empezaremos a tener problemas con el computador. En vez de esto, podemos usar el

Teorema 1.4 (Fórmula de Mertens)

$$\prod_{\substack{2\le p\le x,\\ p\ primo}}\left(1-\frac{1}{p}\right) \;=\; \frac{e^{-\gamma}}{\ln(x)}+O(1/\ln(x)^2),\quad \gamma \text{ es la constante de Euler}$$

Para efectos prácticos consideramos la expresión

$$\prod_{\substack{2\le p\le x,\\ p\ \text{primo}}}\left(1-\frac{1}{p}\right) \;\sim\; \frac{e^{-\gamma}}{\ln(x)}\approx\frac{0.5615}{\ln(x)} \quad \text{si } x\longrightarrow\infty \tag{1.1}$$

Sustituyendo en (1.1), x por $x^{0.5}$ encontramos que

$$\prod_{\substack{2\le p\le \sqrt{x},\\ p\ \text{primo}}}\left(1-\frac{1}{p}\right) \;\sim\; \frac{2e^{-\gamma}}{\ln(x)}\approx\frac{1.12292}{\ln(x)},\quad \text{si } x\longrightarrow\infty$$

EJEMPLO 1.6 Usando la fórmula de Mertens.

x	$\prod_{\text{primos } p\le\sqrt{x}}(1-1/p)$	$\frac{2e^{-\gamma}}{\ln(x)}$
100000	0.0965	0.0975
100000000000000	0.034833774529614024	0.03483410793219253

Tabla 1.5

También, multiplicando (1.1) por 2, la fórmula

$$\prod_{\substack{3\le p,\\ p\ \text{primo}}}^{G}\left(1-\frac{1}{p}\right) \;\sim\; \frac{2e^{-\gamma}}{\ln(G)}\approx\frac{1.12292}{\ln(G)}$$

nos daría la proporción aproximada de números impares que no tienen un factor primo $\leq G$.

EJEMPLO 1.7 Calculando la proporción aproximada de impares sin factores primos $\leq G$.

G	Proporción approx de impares sin factores primos $\leq G$.
100	0.243839
1000	0.162559
10000	0.121919
100000	0.0975355
1000000	0.0812796
10000000	0.0696682
100000000	0.0609597
1000000000	0.0541864
10000000000	0.0487678

Tabla 1.6

Esta tabla nos informa que "típicamente", los números grandes tienen factores primos pequeños.

En resumen: El teorema de los números primos establece que $\pi(x)$ es aproximadamente igual a $x/\ln x$ en el sentido de que $\pi(x)/(x/\ln x)$ converge a 1 conforme $x \to \infty$. Se cree que la densidad de primos en las cercanías de x es aproximadamente $1/\ln x$, es decir, un entero tomado aleatoriamente en las cercanías de x tiene una probabilidad $1/\ln x$ de ser primo.

El producto $\displaystyle\prod_{\substack{3 \leq p, \\ p \text{ primo}}}^{G} \left(1 - \frac{1}{p}\right)$ se puede usar para obtener la proporción aproximada de números impares que no tienen un factor primo $\leq G$.

También podemos estimar otras cosas. Para contar la cantidad de primos que hay entre $\sqrt{x}$ y x: Escribimos todos los números desde 2 hasta x, luego quitamos el 2 y todos sus múltiplos, luego quitamos el 3 y todos sus múltiplos, luego quitamos el 5 y todos sus múltiplos, seguimos así hasta llegar al último primo $p_k \leq \sqrt{x}$ el cual quitamos al igual que sus múltiplos. Como cualquier entero compuesto n, entre $\sqrt{x}$ y x, tiene que tener un factor primo $\leq \sqrt{n}$ entonces este entero fue quitado a esta altura del proceso. Lo que nos queda son solo los primos entre $\sqrt{x}$ y x. Este proceso de colado es una variación de la Criba ("colador") de Eratóstenes.

Por ejemplo, podemos colar $\{2,...12\}$ para dejar solo los primos entre $[\sqrt{12}] = 3$ y 12 :

$$\not{2},\ \not{3},\ \not{4},\ \not{5},\ \not{6},\ 7,\ \not{8},\ \not{9},\ \not{10},\ 11,\ \not{12}$$

así que solo quedan $7, 11$.

Para hacer un "colado aleatorio" para estimar la cantidad de primos entre $\sqrt{x}$ y x, podemos ver $\prod_{p\leq\sqrt{x}} \left(1-\frac{1}{p}\right)$ como una estimación de la proporción de números sin factores primos inferiores a $\sqrt{x}$. Entonces $\prod_{p\leq\sqrt{x}} \left(1-\frac{1}{p}\right) x$ es aproximadamente la cantidad de primos entre $\sqrt{x}$ y x.

La interpretación probabilística en muy delicada: Si vemos $\prod_{p\leq\sqrt{x}} \left(1-\frac{1}{p}\right)$ como la "probabilidad" de que un número no tenga divisores primos inferiores a x, entonces, por el teorema de los números primos, $\prod_{p\leq\sqrt{x}} \left(1-\frac{1}{p}\right) x \sim x/\ln(x)$. Pero esto no es correcto: El teorema de Mertens dice que

$$\prod_{p\leq\sqrt{x}} \left(1-\frac{1}{p}\right) x \ \sim\ \frac{2xe^{-\gamma}}{\ln(x)} \not\sim x/\ln(x)$$

La discrepancia la produce el hecho de que la divisibilidad por diferentes primos no constituyen " eventos suficientemente independientes" y tal vez el factor $e^{-\gamma}$ cuantifica esto en algún sentido. Otra manera de verlo es observar que el colado de Eratóstenes que usamos deja una fracción $1/\ln(x)$ de primos sin tocar, mientras que el colado aleatorio deja una

fracción más grande, $1.123/\ln(x)$.

EJEMPLO 1.8 Si $x = 100000000000000$, $\pi(x) = 3204941750802$ y $\pi(\sqrt{x}) = 664579$. Los primos entre $\sqrt{x}$ y x son 3204941086223 mientras que

$$\prod_{p \leq \sqrt{x}} \left(1 - \frac{1}{p}\right) x \sim \frac{2xe^{-\gamma}}{\ln(x)} = 3483410793219.25$$

1.2.4 El número de primos en una progresión aritmética.

Sean a y b números naturales y consideremos los enteros de la progresión aritmética $an+b$, $n = 0,1,2,\ldots$. De los números inferiores x en esta progresión, ¿cuántos son primos?. Si $\pi_{a,b}(x)$ denota la cantidad de primos $\leq x$ en esta sucesión, tenemos

Teorema 1.5 (Teorema de Dirichlet) *Si a y b son primos relativos entonces*

$$\lim_{x \to \infty} \frac{\pi_{a,b}(x)}{li(x)} = \frac{1}{\varphi(a)}$$

φ es la función de Euler,

$$\varphi(m) = \text{número de enteros positivos} \leq m \text{ y coprimos con } m$$

En particular $\varphi(6) = 2$ y $\varphi(10) = 4$. De acuerdo con el teorema de Dirichlet, "en el infinito" (es decir tomando el límite), un 50% de primos están en cada una de las sucesiones $6n+1$ y $6n-1$ mientras que un 25% de primos se encuentra en cada una de las cuatro sucesiones $10n \pm 1$ y $10n \pm 3$. Se puede probar también que si $\text{mcd}(a,b) = 1$ entonces la sucesión $an+b$ tiene un número infinito de primos.

En realidad, las sucesiones $6n+1$ y $6n-1$ contienen todos los primos pues todo primo es de la forma $6k+1$ o $6k-1$. En efecto, cualquier natural es de la forma $6k+m$ con $m \in \{0,1,2,3,4,5\}$ (por ser "$\equiv \pmod 6$" una relación de equivalencia en $\mathbb{N}$, es decir parte $\mathbb{N}$ en seis clases). Ahora, todos los enteros $6k+m$ son claramente compuestos excepto para $m = 1$ y $m = 5$ por lo que si p es primo, entonces $p = 6k+1$ o $p = 6k+5 = 6q-1$ (si $q = k+1$), es decir p es de la forma $6k \pm 1$.

1.2.5 Cantidad de factores primos de un número grande.

El teorema del límite central dice que si una población (continua o discreta) tiene media μ y varianza finita σ^2, la media muestral $\overline{X}$ tendrá una distribución que se aproxima a la normal.

Teorema 1.6 (Limite Central) *Si tenemos $X_1, X_2, \ldots, X_n$ variables aleatorias independientes, idénticamente distribuidas, con media μ y varianza σ^2, entonces, si n es suficientemente grande, la probabilidad de que $S_n = X_1 + X_2 + \ldots + X_n$ esté entre $n\mu + \alpha\sigma\sqrt{n}$ y $n\mu + \beta\sigma\sqrt{n}$ es*

$$\frac{1}{\sqrt{2\pi}} \int_{\alpha}^{\beta} e^{-t^2/2}\, dt$$

EJEMPLO 1.9 Si lanzamos una moneda limpia unas 10000 veces, uno esperaría que aproximadamente 5000 veces salga "cara". Si denotamos con $X_i = 1$ el evento "en el lanzamiento i sale cara", como la probabilidad que asumimos para el evento "sale cara" es $1/2$, entonces $n\mu = n \cdot 0.5 = 5000$ y $\sigma = \sqrt{n \cdot 0.25} = 5$. Luego, para calcular la probabilidad de que el número de caras esté entre 4850 y 5150, debemos calcular los límites α y β. Por razones de ajuste del caso discreto al caso continuo, se usa un factor de corrección de $1/2$. Resolviendo, $5000 + (\alpha)\sqrt{50} = 4850 - 0.5 \Longrightarrow \alpha = -3.01$ $5000 + (\alpha)\sqrt{50} = 5150 + 0.5 \Longrightarrow \beta = 3.01$

$$\frac{1}{\sqrt{2\pi}} \int_{-3.01}^{3.01} e^{-t^2/2}\, dt = 0.997388$$

Así, la probabilidad de que el número de caras esté entre 4850 y 5150 es de 0.997388

Si $\omega(n)$ denota la cantidad de factores primos de n, esta función se puede denotar como una suma de funciones $\rho_p(n)$, estadísticamente independientes, definidas por

$$\rho_p(n) = \begin{cases} 1 & \text{si} \quad p|n \\ 0 & \text{si} \quad p \nmid n \end{cases}$$

Esto sugiere que la distribución de los valores de $\omega(n)$ puede ser dada por la ley normal (con media $\ln\ln n$ y desviación estándar $\sqrt{\ln\ln n}$).

Mark Kac y Paul Erdös probaron que la densidad del conjunto de enteros n para el cual el número de divisores primos $\omega(n)$ está comprendido entre $\ln\ln n+\alpha\sqrt{\ln\ln n}$ y $\ln\ln n+\beta\sqrt{\ln\ln n}$, es

$$\frac{1}{\sqrt{2\pi}}\int_{\alpha}^{\beta} e^{-t^2/2}\,dt$$

es decir, el número de divisores primos está distribuido de acuerdo a la ley normal.

Teorema 1.7 *Denotamos con $N(x,a,b)$ la cantidad de enteros n en $\{3,4,\ldots,x\}$ para los cuales*

$$\alpha \leq \frac{\omega(n)-\ln\ln n}{\sqrt{\ln\ln n}} \leq \beta$$

Entonces, conforme $x\to\infty$,

$$N(x,a,b) = (x+o(x))\frac{1}{\sqrt{2\pi}}\int_{\alpha}^{\beta} e^{-t^2/2}\,dt$$

Para efectos prácticos, hacemos referencia a este teorema en estos términos

> Típicamente, el número de factores primos, inferiores a x, de un número n suficientemente grande es aproximadamente $\ln\ln x$.

1.3 Criba de Eratóstenes: Cómo colar números primos.

La criba[2] de Eratóstenes es un algoritmo que permite "colar" todos los números primos menores que un número natural dado n, eliminando los números compuestos de la lista $\{2,\ldots,n\}$. Es simple y razonablemente eficiente.

Primero tomamos una lista de números $\{2,3,\ldots,n\}$ y eliminamos de la lista los múltiplos de 2. Luego tomamos el primer entero después de 2 que no fue borrado (el 3) y eliminamos de la lista sus múltiplos, y así sucesivamente. Los números que permanecen en la lista son

[2]Criba, tamiz y zaranda son sinónimos. Una criba es un herramienta que consiste de un cedazo usada para limpiar el trigo u otras semillas, de impurezas. Esta acción de limpiar se le dice cribar o tamizar.

los primos $\{2,3,5,7,...\}$.

EJEMPLO 1.10 Primos menores que $n = 10$

Lista inicial	2	3	4	5	6	7	8	9	10
Eliminar múltiplos de 2	2	3	~~4~~	5	~~6~~	7	~~8~~	9	~~10~~
Resultado	2	3	5	7	9				
Eliminar múltiplos de 3	2	3	5	7	~~9~~				
Resultado	2	3	5	7					

Primer refinamiento: Tachar solo los impares

Excepto el 2, los pares no son primos, así que podríamos "tachar" solo sobre la lista de impares $\leq n$:

$$\{3,5,9,...,\} = \left\{ 2i+3 \ : \quad i = 0,1,... \left[\!\left[\frac{n-3}{2} \right]\!\right] \right\}$$

El último impar es n o $n-1$. En cualquier caso, el último impar es $2 \cdot \left[\!\left[\frac{n-3}{2} \right]\!\right] + 3$ pues,

Si n es impar, $n = 2k+1$ y $\left[\!\left[\frac{n-3}{2} \right]\!\right] = k-1 \implies 2(k-1)+3 = n$.

Si n es par, $n = 2k$ y $\left[\!\left[\frac{n-3}{2} \right]\!\right] = k-2 \implies 2(k-2)+3 = 2k-1 = n-1$.

Segundo refinamiento: Tachar de p_k^2 en adelante

En el paso $k-$ésimo hay que tachar los múltiplos del primo p_k desde p_k^2 en adelante.

Esto es así pues en los pasos anteriores se ya se tacharon $3 \cdot p_k$, $5 \cdot p_k, ..., p_{k-1} \cdot p_k$.

Por ejemplo, cuando nos toca tachar los múltiplos del primo 7, ya se han eliminado los múltiplos de $2,3$ y 5, es decir, ya se han eliminado $2 \cdot 7$, $3 \cdot 7$, $4 \cdot 7$, $5 \cdot 7$ y $6 \cdot 7$. Por eso iniciamos en 7^2.

Tercer refinamiento: Tachar mientras $p_k^2 \leq n$

En el paso $k-$ésimo hay que tachar los múltiplos del primo p_k solo si $p_k^2 \leq n$. En otro caso, nos detenemos ahí.

¿Porque?. En el paso $k-$ésimo tachamos los múltiplos del primo p_k desde p_k^2 en adelante, así que si $p_k^2 > n$ ya no hay nada que tachar.

EJEMPLO 1.11 Encontrar los primos menores que 20. El proceso termina cuando el cuadrado del mayor número confirmado como primo es < 20.

1. La lista inicial es $\{2,3,5,7,9,11,13,15,17,19\}$
2. Como $3^2 \leq 20$, tachamos los múltiplos de 3 desde $3^2 = 9$ en adelante:

$$\{2,3,5,7,\cancel{9},11,13,\cancel{15},17,19\}$$

3. Como $5^2 > 20$ el proceso termina aquí.
4. Primos < 20 : $\{2,3,5,7,11,13,17,19\}$

1.3.1 Algoritmo e implementación.

1. Como ya vimos, para colar los primos en el conjunto $\{2,3,...,n\}$ solo consideramos los impares:

$$\left\{2i+3 \ : \ \ i = 0,1,... \left[\!\left[\frac{n-3}{2}\right]\!\right]\right\} = \{3,5,7,9,...\}$$

2. Por cada primo $p = 2i+3$ (tal que $p^2 < n$), debemos eliminar los *múltiplos impares* de p menores que n, a saber

$$(2k+1)p = (2k+1)(2i+3), \quad k = i+1, i+2, ...$$

Observe que si $k = i+1$ entonces el primer múltiplo en ser eliminado es $p^2 = (2i+3)(2i+3)$, como debe ser.

Esto nos dice que para implementar el algoritmo solo necesitamos un arreglo (booleano) de tamaño "`quo(n-3,2)`". En Java se pone "`(n-3)/2`" y en VBA se pone "`(n-3)\2`".

El arreglo lo llamamos `EsPrimo[i], i=0,1,...,(n-3)/2.`

Cada entrada del arreglo "`EsPrimo[i]`" indica si el número $2i+3$ es primo o no.

Por ejemplo

`EsPrimo[0] = true` pues $n = 2\cdot 0+3 = 3$ es primo,

`EsPrimo[1] = true` pues $n = 2\cdot 1+3 = 5$ es primo,

`EsPrimo[2] = true` pues $n = 2\cdot 2+3 = 7$ es primo,

`EsPrimo[3] = false` pues $n = 2\cdot 3+3 = 9$ no es primo.

Si el número $p = 2i+3$ es primo entonces $i = (p-3)/2$ y

`EsPrimo[(p-3)/2] = true.`

Si sabemos que $p = 2i+3$ es primo, debemos poner

`EsPrimo[((2k+1)(2i+3) - 3)/2] = false`

pues estas entradas representan a los múltiplos $(2k+1)(2i+3)$ de p. Observe que cuando $i = 0,1,2$ tachamos los múltiplos de $3,5$ y 7; cuando $i = 3$ entonces $2i+3 = 9$ pero en este momento `esPrimo[3]=false` así que proseguimos con $i = 4$, es decir, proseguimos tachando los múltiplos de 11.

En resumen: Antes de empezar a tachar los múltiplos de $p = 2i+3$ debemos preguntar si `esPrimo[i]=true`.

Algoritmo 1.1: Criba de Eratóstenes

Entrada: $n \in \mathbb{N}$
Resultado: Primos entre 2 y n

```
1  máx = (n−3)/2;
2  boolean esPrimo[i],  i = 1,2,...,máx;
3  for i = 1,2,...,máx do
4      esPrimo[i] =True;
5  i = 0;
6  while (2i+3)(2i+3) ≤ n do
7      k = i+1;
8      if esPrimo(i) then
9          while (2k+1)(2i+3) ≤ n do
10             esPrimo[((2k+1)(2i+3)−3)/2] =False;
11             k = k+1;
12     i = i+1;
13 Imprimir;
14 for j = 1,2,...,máx do
15     if esPrimo[j] =True then
16         Imprima j
```

1.3.1.1 Implementación en Java. Vamos a agregar un método a nuestra clase "`Teoria_Numeros`". El método recibe el número natural $n > 2$ y devuelve un vector con los números primos $\leq n$. Para colar los números compuestos usamos un arreglo

```
boolean [] esPrimo = new boolean[(n-3)/2].
```

Al final llenamos un vector con los primos que quedan.

```
import java.math.BigInteger;
public class Teoria_Numeros
{   ...
 public static Vector HacerlistaPrimos(int n)
```

```
{
          Vector        salida = new Vector(1);
          int k   = 1;
          int max = (n-3)/2;
          boolean[]   esPrimo  = new boolean[max+1];

          for(int i = 0; i <= max; i++)
              esPrimo[i]=true;

          for(int i = 0; (2*i+3)*(2*i+3) <= n; i++)
          {
            k = i+1;
            if(esPrimo[i])
            {
              while( ((2*k+1)*(2*i+3)) <= n)
              {
                  esPrimo[((2*k+1)*(2*i+3)-3)/2]=false;
                  k++;
              }
            }
           }
           salida.addElement(new Integer(2));
           for(int i = 0; i <=max; i++)
           { if(esPrimo[i])
             salida.addElement(new Integer(2*i+3));
           }
           salida.trimToSize();
           return salida;
  }
  public static void main(String[] args)
  {
    System.out.println("\n\n");
   //---------------------------------------------------------------------
    int    n = 100;
    Vector primos;
           primos = HacerlistaPrimos(n);
    //Cantidad de primos <= n
    System.out.println("Primos <="+ n+": "+primos.size()+"\n");
    //imprimir vector (lista de primos)
```

```
  for(int p = 1; p < primos.size(); p++)
  {
      Integer num = (Integer)primos.elementAt(p);
      System.out.println(""+(int)num.intValue());
  }
//--------------------------------------------------------------
  System.out.println("\n\n");
}}
```

1.3.1.2 Uso de la memoria En teoría, los arreglos pueden tener tamaño máximo `Integer.MAX_INT` $= 2^{31} - 1 = 2\,147\,483\,647$ (pensemos también en la posibilidad de un arreglo multidimensional!). Pero en la práctica, el máximo tamaño del array depende del hardware de la computadora. El sistema le asigna una cantidad de memoria a cada aplicación; para valores grandes de n puede pasar que se nos agote la memoria (veremos el mensaje "`OutOfMemory Error`"). Podemos asignar una cantidad de memoria apropiada para el programa "cribaEratostenes.java" desde la línea de comandos, si n es muy grande. Por ejemplo, para calcular los primos menores que $n = 100\,000\,000$, se puede usar la instrucción

```
C:\usrdir> java -Xmx1000m -Xms1000m Teoria_Numeros
```

suponiendo que el archivo "`Teoria_Numeros.java`" se encuentra en `C:\usrdir`.

Esta instrucción asigna al programa una memoria inicial (Xmx) de 1000 MB y una memoria máxima (Xms) de 1000 MB (siempre y cuando existan tales recursos de memoria en nuestro sistema).

En todo caso hay que tener en cuenta los siguientes datos

n	Primos $\leq n$
10	4
100	25
1 000	168
10 000	1 229
100 000	9 592
1 000 000	78 498
10 000 000	664 579
100 000 000	5 761 455
1 000 000 000	50 847 534

10 000 000 000	455 052 511
100 000 000 000	4 118 054 813
1 000 000 000 000	37 607 912 018
10 000 000 000 000	346 065 536 839

1.3.1.3 Implementación en Excel. Para la implementación en Excel usamos un cuaderno como el de la figura (1.2).

El número n lo leemos en la celda $(4,1)$. El código VBA incluye una subrutina para imprimir en formato de tabla, con `ncols` columnas. Este último parámetro es opcional y tiene valor default 10. También incluye otra subrutina para limpiar las celdas para que no haya confusión entre los datos de uno y otro cálculo.

	A	B	C	D	E	F	G	H	I	J	K
1						COLAR PRIMOS (ERATOSTENES)					
2	*Primos ≤ n*		Imprimir en tabla.								
3	*n*		Número de columnas								
4	1000		25								
5											
6	2	3	5	7	11	13	17	19	23	29	31
7	101	103	107	109	113	127	131	137	139	149	151
8	233	239	241	251	257	263	269	271	277	281	283
9	383	389	397	401	409	419	421	431	433	439	443
10	547	557	563	569	571	577	587	593	599	601	607

Figura 1.2 Primos $\leq n$.

Imprimir en formato de tabla

Para esto usamos la subrutina

```
Imprimir(ByRef Arr() As Long, fi, co, Optional ncols As Variant).
```

La impresión inicia en la celda "`(fi,co)`". Para imprimir en formato de tabla usamos `Cells(fi + k, co + j)` con el número de columnas `j` variando de 0 a `ncols-1`. Para reiniciar `j` en cero actualizamos `j` con `j = j Mod ncols`. Para cambiar la fila usamos `k`. Esta variable aumenta en 1 cada vez que `j` llega a `ncols-1`. Esto se hace con división entera: `k = k + j \ (ncols - 1)`

Subrutina para borrar celdas

Para esto usamos la subrutina

`LimpiaCeldas(fi, co, ncols).`

Cuando hacemos cálculos de distinto tamaño es conveniente borrar las celdas de los cálculos anteriores para evitar confusiones. La subrutina inicia en la celda `(fi,co)` y borra `ncols` columnas a la derecha. Luego pasa a la siguiente fila y hace lo mismo. Prosigue de igual forma hasta que encuentre la celda `(fi+k,co)` vacía.

```
Option Explicit
    Private Sub CommandButton1_Click()
    Dim n, ncols
    n = Cells(4, 1)
    ncols = Cells(4, 3)
    Call Imprimir(ERATOSTENES(n), 6, 1, ncols)
End Sub

' Imprime arreglo en formato de tabla con "ncols" columnas,
' iniciando en la celda (fi,co)
Sub Imprimir(ByRef Arr() As Long, fi, co, Optional ncols As Variant)
Dim i, j, k
    ' Limpia celdas
    ' f        = fila en que inicia la limpieza
    ' co       = columna en q inicia la limpieza
    ' ncols    = nmero de columnas a borrar
Call LimpiaCeldas(fi, co, ncols)
If IsMissing(ncols) = True Then
            ncols = 10
End If
'Imprimir
j = 0
k = 0
For i = 0 To UBound(Arr)
    Cells(fi + k, co + j) = Arr(i)
    k = k + j \ (ncols - 1) 'k aumenta 1 cada vez que j llegue a ncols-1
    j = j + 1
    j = j Mod ncols         'j=0,1,2,...,ncols-1
```

```
Next i

End Sub


Function ERATOSTENES(n) As Long()
Dim i, j, k, pos, contaPrimos
Dim max As Long
Dim esPrimo() As Boolean
Dim Primos() As Long
max = (n - 3) \ 2 ' Divisin entera
ReDim esPrimo(max + 1)
ReDim Primos(max + 1)
For i = 0 To max
    esPrimo(i) = True
Next i
contaPrimos = 0
Primos(0) = 2 'contado el 2
j = 0
While (2 * j + 3) * (2 * j + 3) <= n
    k = j + 1
    If esPrimo(j) Then
        While (2 * k + 1) * (2 * j + 3) <= n
            pos = ((2 * k + 1) * (2 * j + 3) - 3) \ 2
            esPrimo(pos) = False
            k = k + 1
        Wend
    End If
j = j + 1
Wend

For i = 0 To max
   If esPrimo(i) Then
        contaPrimos = contaPrimos + 1 '3,5,...
        Primos(contaPrimos) = 2 * i + 3
    End If
Next i

ReDim Preserve Primos(contaPrimos) 'Cortamos el vector
ERATOSTENES = Primos()
End Function
```

```
Private Sub LimpiaCeldas(fi, co, nc)
Dim k, j
k = 0
While LenB(Cells(fi + k, co)) <> 0   ' celda no vac\'ia
      For j = 0 To nc
          Cells(fi + k, co + j) = ""  ' borra la fila hasta nc columnas
      Next j
      k = k + 1
Wend
End Sub
```

1.3.2 Primos entre m y n.

Para encontrar todos los primos entre m y n (con $m < n$) procedemos como si estuviéramos colando primos en la lista $\{2,3,...,n\}$, solo que eliminamos los múltiplos que están entre m y n : Eliminamos los múltiplos de los primos p para los cuales $p^2 \leq n$ (o también $p \leq \sqrt{n}$), que están entre m y n.

Múltiplos de p entre m y n

Para los primos p inferiores a $\sqrt{n}$, buscamos el primer múltiplo de p entre m y n.

Si $m-1 = pq+r,\ 0 \leq r < p \implies p(q+1) \geq m$

Así, los múltiplos de p mayores o iguales a m son

$$p(q+1),\ p(q+2),\ p(q+3),\ \ldots \quad \text{con} \quad q = \text{quo}(m-1,p)$$

EJEMPLO 1.12 Para encontrar los primos entre $m = 10$ y $n = 30$, debemos eliminar los múltiplos de los primos $\leq \sqrt{30} \approx 5$. Es decir, los múltiplos de los primos $p = 2,3,5$.

Como $10-1 = 2\cdot 4+1$, el 2 elimina los números $2(4+k) = 8+2k,\ k \geq 1$; es decir $\{10,12,...,30\}$

Como $10-1 = 3\cdot 3+0$, el 3 elimina los números $3(3+k) = 9+3k,\ k \geq 1$; es decir $\{12,15,18,21,24,27,30\}$

Como $10-1 = 5\cdot 1+4$, el 5 elimina los números $5(1+k) = 5+5k,\ k \geq 1$; es decir $\{10,15,20,25.\}$

Finalmente nos quedan los primos $11, 13, 17, 19, 23, 29$.

1.3.2.1 Algoritmo. Como antes, solo consideramos los impares entre m y n. Si ponemos

$$min = \text{quo}(m+1-3, 2) \text{ y } max = \text{quo}(n-3, 2)$$

entonces $2 \cdot min + 3$ es el primer impar $\geq m$ y $2 \cdot max + 3$ es el primer impar $\leq n$. Así, los impares entre m y n son los elementos del conjunto $\{2 \cdot i + 3 : i = min, ..., max\}$

Como antes, usamos un arreglo booleano `esPrimo(i)` con $i = min, ..., max$. `esPrimo(i)` representa al número $2 \cdot i + 3$.

EJEMPLO 1.13 Si $m = 11$ y 20, $[\![(m+1-3)/2]\!] = 4$ y $[\![(n-3)/2]\!] = 8$. Luego $2 \cdot 4 + 3 = 11$ y $2 \cdot 8 + 3 = 19$.

Para aplicar el colado necesitamos los primos $\leq \sqrt{n}$. Esta lista de primos la obtenemos con la función `Eratostenes(isqrt(n))`. Aquí hacemos uso de la función `isqrt(n)` (algoritmo **??**).

Para cada primo p_i en la lista,

1. si $m \leq p_i^2$, tachamos los múltiplos impares de p_i como antes,

```
1 if m ≤ p_i^2 then
2     k = (p_i − 1)/2;
3     while (2k+1)p_i ≤ n do
4         esPrimo[((2k+1)p_i − 3)/2] =False;
5         k = k+1;
```

Note que si $k = (p_i - 1)/2$ entonces $(2k+1)p_i = p_i^2$

2. si $p_i^2 < m$, tachamos desde el primer múltiplo impar de p_i que supere m :

```
1 if p_i^2 < m then
2     q = (m-1)/p;
3     q_2 = rem(q,2);
4     k = q_2;
5     mp = (2k+1-q_2+q)·p_i;
6     while mp ≤ n do
7         esPrimo[(mp-3)/2] =False;
8         k = k+1;
9         mp = (2k+1-q_2+q)·p_i
```

Los múltiplos de p_i que superan m son $p_i(q+k)$ con $q = \text{quo}(m-1,p)$. De esta lista solo nos interesan los múltiplos impares. Esto requiere un pequeño análisis aritmético.

Como p_i es impar, $p_i(q+k)$ es impar solo si $q+k$ es impar. Poniendo $q_2 = \text{rem}(q,2)$ entonces $(2k+1-q_2+q)$ es impar si $k = q_2,\ q_2+1,\ldots$. En efecto,

$$2k+1-q_2+q = \begin{cases} 2k+1+q & \text{si} \quad q \text{ es par. Aquí} \quad k = q_2 = 0,1,\ldots \\ 2k+q & \text{si} \quad q \text{ es impar. Aquí} \quad k = q_2 = 1,2,\ldots \end{cases}$$

Luego, los múltiplos impares de p_i son los elementos del conjunto

$\{(2k+1-q_2+q)\cdot p \ : \ q_2 = \text{rem}(q,2) \ \text{ y } \ k = q2, q2+1,\ldots\}$

La manera de tachar los múltiplos impares de p_i aparece arriba.

Ahora podemos armar el algoritmo completo.

Algoritmo 1.2: Colado de primos entre m y n.

Entrada: $n, m \in \mathbb{N}$ con $m < n$.
Resultado: Primos entre m y n

```
1  Primo() = una lista de primos ≤ √n;
2  min = (m+1−3)/2; max = (n−3)/2;
3  esPrimo[i], i = min,...,max ;
4  for j = min,...,max do
5      esPrimo[j] =True;
6  np = cantidad de primos en la lista Primos;
7  Suponemos Primo(0) = 2;
8  for i = 1,2,...,np do
9      if m ≤ p_i^2 then
10         k = (p_i − 1)/2;
11         while (2k+1)p_i ≤ n do
12             esPrimo[((2k+1)p_i − 3)/2] =False;
13             k = k+1;
14     if p_i^2 < m then
15         q = (m−1)/p;
16         q_2 = rem(q,2);
17         k = q_2;
18         mp = (2k+1−q_2+q)·p_i;
19         while mp ≤ n do
20             esPrimo[(mp−3)/2] =False;
21             k = k+1;
22             mp = (2k+1−q_2+q)·p_i
23 Imprimir;
24 for j = min,...,max do
25     if esPrimo[j] =True then
26         Imprima 2∗i+3
```

1.3.2.2 Implementación en Excel. Para la implementación en Excel usamos un cuaderno como el de la figura (1.3).

m y n los leemos en la celdas $(4,1)$, $(4,2)$. Como antes, el código VBA hace referencia a las subrutinas para imprimir en formato de tabla y limpiar las celdas (sección 1.3.1.3).

	A	B	C	D	E	F	G	H	I
1							Primos m y n		
2	*Primos entre m y n*			Imprimir en tabla.					
3	*m*	n		Número de columnas					
4	900	1100		5					
5									
6	907	911	919	929	937				
7	941	947	953	967	971				

Figura 1.3 Primos $\leq n$.

En VBA Excel podemos declarar un arreglo que inicie en *min* y finalice en *max*, como el algoritmo. Por eso, la implementación es muy directa.

```
Option Explicit
Private Sub CommandButton1_Click()
Dim n, m, ncols
m = Cells(4, 1)
n = Cells(4, 2)
ncols = Cells(4, 4)
Call Imprimir(PrimosMN(m, n), 6, 1, ncols)
End Sub

Sub Imprimir(ByRef Arr() As Long, fi, co, Optional ncols As Variant)
...
End Sub

Function ERATOSTENES(n) As Long()
...
End Sub

Function isqrt(n) As Long
Dim xk, xkm1
If n = 1 Then
   xkm1 = 1
End If
```

```
If n > 1 Then
    xk = n
    xkm1 = n \ 2
    While xkm1 < xk
         xk = xkm1
         xkm1 = (xk + n \ xk) \ 2
    Wend
End If
isqrt = xkm1
End Function
' m < n
Function PrimosMN(m, n) As Long()
Dim i, j, k, pos, contaPrimos, mp, q, q2
Dim min, max
Dim esPrimo() As Boolean
Dim primo() As Long
Dim PrimosMaN() As Long

min = Int((m + 1 - 3) \ 2)
max = Int((n - 3) \ 2)

ReDim esPrimo(min To max)
ReDim PrimosMaN((n - m + 2) \ 2)
For i = min To max
    esPrimo(i) = True
Next i

primo = ERATOSTENES(isqrt(n))

For i = 1 To UBound(primo)          'primo(1)=3
    If m <= primo(i) * primo(i) Then
       k = (primo(i) - 1) \ 2
       While (2 * k + 1) * primo(i) <= n
       esPrimo(((2 * k + 1) * primo(i) - 3) \ 2) = False
       k = k + 1
       Wend
    End If
    If primo(i) * primo(i) < m Then
       q = (m - 1) \ primo(i)  'p(q+k)-> p*k
       q2 = q Mod 2
       k = q2
       mp = (2 * k + 1 - q2 + q) * primo(i)  'm\'ultiplos impares
```

```
        While mp <= n
            esPrimo((mp - 3) \ 2) = False
            k = k + 1
            mp = (2 * k + 1 - q2 + q) * primo(i)
       Wend
    End If
Next i

If m > 2 Then
    contaPrimos = 0
Else
    contaPrimos = 1
    PrimosMaN(0) = 2
End If

For i = min To max
   If esPrimo(i) Then
        PrimosMaN(contaPrimos) = 2 * i + 3
        contaPrimos = contaPrimos + 1 '3,5,...
    End If
Next i

If 1 <= contaPrimos Then
ReDim Preserve PrimosMaN(contaPrimos - 1)
Else
ReDim PrimosMaN(0)
End If

PrimosMN = PrimosMaN()
End Function
```

1.4 Factorización por ensayo y error.

1.4.1 Introducción

El método más sencillo de factorización (y muy útil) es el método de *factorización por ensayo y error* (FEE). Este método va probando con los posibles divisores de n hasta encontrar una factorización de este número.

En vez de probar con todos los posibles divisores de n (es decir, en vez de usar *fuerza bruta*) podemos hacer algunos refinamientos para lograr un algoritmo más eficiente en el sentido de reducir las pruebas a un conjunto de números más pequeño, en el que se encuentren los divisores pequeños de n.

1.4.2 Probando con una progresión aritmética.

Como estamos buscando factores pequeños de $n,$ podemos usar el siguiente teorema,

Teorema 1.8

Si $n \in \mathbb{Z}^+$ admite la factorización $n = ab,$ con $a,b \in \mathbb{Z}^+$ entonces $a \leq \sqrt{n}$ o $b \leq \sqrt{n}.$

Prueba. Procedemos por contradicción, si $a > \sqrt{n}$ y $b > \sqrt{n}$ entonces $ab > \sqrt{n}\sqrt{n} = n$ lo cual, por hipótesis, no es cierto.

Del teorema anterior se puede deducir que

- Si n no tiene factores d con $1 < d \leq \sqrt{n},$ entonces n es primo.

- Al menos uno de los factores de n es menor que $\sqrt{n}$ (no necesariamente todos). Por ejemplo $14 = 2 \cdot 7$ solo tiene un factor menor que $\sqrt{14} \approx 3.74166).$

De acuerdo al teorema fundamental de la aritmética, Cualquier número natural > 1 factoriza, de manera única (excepto por el orden) como producto de primos. Esto nos dice que la estrategia óptima de factorización sería probar con los primos menores que $\sqrt{n}.$ El problema es que si n es muy grande el primer problema sería que el cálculo de los primos de prueba duraría siglos (sin considerar los problemas de almacenar estos números).

Recientemente (2005) se factorizó un número de 200 cifras[3] (RSA-200). Se tardó cerca de 18 meses en completar la factorización con un esfuerzo computacional equivalente a 53 años de trabajo de un CPU 2.2 GHz Opteron.

1.4.3 Algoritmo.

Identificar si un número es primo es generalmente fácil, pero factorizar un número (grande) arbitrario no es sencillo. El método de factorización de un número N probando con divisores primos ("trial division") consiste en probar dividir N con primos pequeños. Para esto se debe previamente almacenar una tabla suficientemente grande de números primos o generar la tabla cada vez. Como ya vimos en la criba de Eratóstenes, esta manera de proceder trae consigo problemas de tiempo y de memoria. En realidad es más ventajoso proceder de otra manera.

- Para hacer la pruebas de divisibilidad usamos los enteros $2, 3$ y la sucesión $6k \pm 1,\ k = 1,2,\ldots$.

  Esta elección cubre todos los primos e incluye divisiones por algunos números compuestos (25,35,...) pero la implementación es sencilla y el programa suficientemente rápido (para números no muy grandes) que vale la pena permitirse estas divisiones inútiles.

- Las divisiones útiles son las divisiones por números primos, pues detectamos los factores que buscamos. Por el teorema de los números primos, hay $\pi(G) \approx G/\ln G$ números primos inferiores a G, ésta sería la cantidad aproximada de divisiones útiles.

  Los números 2, 3 y $6k \pm 1$, $k \in \mathbb{N}$ constituyen, hablando en grueso, una tercera parte de los naturales (note que $\mathbb{Z} = \bigcup \mathbb{Z}_6 = \bigcup \{\overline{0}, \overline{1}, \overline{2}, \overline{3}, \overline{4}, \overline{5}\}$, $\{\overline{1}, \overline{-1} = \overline{5}\}$ es una tercera parte). En $\{1,2,\ldots,G\}$ hay $\approx G/3$ de estos números. En estos $G/3$ números están los $\pi(G) \approx G/\ln G$ primos inferiores a G, es decir, haríamos $\approx \dfrac{G/\ln G}{G/3} = 3/\ln G$ divisiones útiles.

[3]Se trata del caso más complicado, un número que factoriza como producto de dos primos (casi) del mismo tamaño.

Si probamos con todos los números, tendríamos que hacer $1/0.22 = 4.6$ más cálculos para obtener un 22% de divisiones útiles.

Cuando se juzga la rapidez de un programa se toma en cuenta el tiempo de corrida en el *peor caso* o se toma en cuenta el *tiempo promedio de corrida* (costo de corrida del programa si se aplica a muchos números). Como ya sabemos (por el Teorema de Mertens) hay un porcentaje muy pequeño de números impares sin divisores $\leq G$, así que en promedio, nuestra implementación terminará bastante antes de alcanzar el límite G (el "peor caso" no es muy frecuente) por lo que tendremos un programa con un comportamiento deseable.

Detalles de la implementación.

- Para la implementación necesitamos saber cómo generar los enteros de la forma $6k \pm 1$. Alternando el -1 y el 1 obtenemos la sucesión

$$5,\ 7,\ 11,\ 13,\ 17,\ 19, ...$$

que iniciando en 5, se obtiene alternando los sumandos 2 y 4. Formalmente, si $m_k = 6k-1$ y si $s_k = 6k+1$ entonces, podemos poner la sucesión como

$$7,\ 11,\ 13, ..., m_k,\ s_k,\ m_{k+1},\ s_{k+1}, ...$$

Ahora, notemos que $s_k = m_k + 2$ y que $m_{k+1} = s_k + 4 = m_k + 6$. La sucesión es

$$7,\ 11,\ 13, ..., m_k,\ m_k+2,\ m_k+6,\ m_{k+1}+2,\ m_{k+1}+6, ...$$

En el programa debemos probar si el número es divisible por 2, por 3 y ejecutamos el ciclo

$p = 5$;
While $p \leq G$ Do $\{$
Probar divisibilidad por p
Probar divisibilidad por $p+2$
$p = p+6$ $\}$

- En cada paso debemos verificar si el divisor de prueba p alcanzó el límite Mín $\{G, \sqrt{N}\}$. Si se quiere evitar el cálculo de la raíz, se puede usar el hecho de que si $p > \sqrt{N}$ entonces $p > N/p$.

Algoritmo 1.3: Factorización por Ensayo y Error.

Entrada: $N \in \mathbb{N},\ G \leq \sqrt{N}$
Resultado: Un factor $p \leq G$ de N si hubiera.

```
1  p = 5;
2  if N es divisible por 2 o 3 then
3  |  Imprimir factor;
4  else
5  |  while p ≤ G do
6  |  |  if N es divisible por p o p+2 then
7  |  |  |  Imprimir factor;
8  |  |  |  break;
9  |  |  end
10 |  |  p = p+6
11 |  end
12 end
```

1.4.4 Implementación en Java.

Creamos una clase que busca factores primos de un número N hasta un límite G. En el programa, $G =$ Mín $\{\sqrt{N}, G\}$.

Usamos un método `reducir(N,p)` que verifica si p es factor, si es así, continua dividiendo por p hasta que el residuo no sea cero. Retorna la parte de N que no ha sido factorizada.

El método `Factzar_Ensayo_Error(N, G)` llama al método `reducir(N,p)` para cada $p = 2,\ 3,\ 7,\ 11,\ 13, \ldots$. hasta que se alcanza el límite G.

```
import java.util.Vector;
import java.math.BigInteger;

public class Ensayo_Error
{
    private Vector    salida = new Vector(1);
```

```
static BigInteger Ge     = new BigInteger("10000000");//10^7
BigInteger        UNO    = new BigInteger("1");
BigInteger        DOS    = new BigInteger("2");
BigInteger        TRES   = new BigInteger("3");
BigInteger        SEIS   = new BigInteger("4");
BigInteger        Nf;
int               pos    = 1; //posicin del exponente del factor

public Ensayo_Error(){}

public BigInteger reducir(BigInteger Ne, BigInteger p)
{
    int exp = 0, posAct = pos;
    BigInteger residuo;
    residuo =  Ne.mod(p);

    if(residuo.compareTo(BigInteger.ZERO)==0)
    {
      salida.addElement(p); //p es objeto BigInteger
      salida.addElement(BigInteger.ONE); //exponente
      pos = pos+2; //posicin del siguiente exponente (si hubiera)
    }

    while(residuo.compareTo(BigInteger.ZERO)==0)
    {
     Ne      =  Ne.divide(p); // Ne = Ne/p
     residuo =  Ne.mod(p);
     exp=exp+1;
     salida.set(posAct, new BigInteger(""+exp)); //p es objeto BigInteger
    }

    return Ne;
}//

public Vector Factzar_Ensayo_Error(BigInteger Ne, BigInteger limG)
{
    BigInteger p    = new BigInteger("5");

    Nf = Ne;
```

```
    Nf = reducir(Nf, DOS);
    Nf = reducir(Nf, TRES);

    while(p.compareTo(limG)<=0)
    {
      Nf= reducir(Nf, p);          //dividir por p
      Nf= reducir(Nf, p.add(DOS)); //dividir por p+2
      p = p.add(SEIS); //p=p+6
    }

    if(Nf.compareTo(BigInteger.ONE)>0)
    {
      salida.addElement(Nf); //p es objeto BigInteger
      salida.addElement(BigInteger.ONE); //exponente
    }
    return salida;
}
// Solo un argumento.
public Vector Factzar_Ensayo_Error(BigInteger Ne)
{
    BigInteger limG = Ge.min(raiz(Ne));
    return Factzar_Ensayo_Error(Ne, limG);
}

//raz cuadrada
public BigInteger raiz(BigInteger n)
{
                BigInteger xkm1 = n.divide(DOS);
                BigInteger xk   = n;

                  if(n.compareTo(BigInteger.ONE)< 0)
                    return xkm1=n;
                while(xk.add(xkm1.negate()).compareTo(BigInteger.ONE)>0)
                {
                    xk=xkm1;
                    xkm1=xkm1.add(n.divide(xkm1));
                    xkm1=xkm1.divide(DOS);
                }
                return xkm1;
```

```
     }
     //Imprimir
     public String print(Vector lista)
     {
        String tira="";
        for(int p = 0; p < lista.size(); p++)
        {
            if(p%2==0)     tira= tira+lista.elementAt(p);
            else           tira= tira+"^"+lista.elementAt(p)+" * ";
        }
        return tira.substring(0,tira.length()-3);
     }


    public static void main(String[] args)
    {
       BigInteger limG;
       BigInteger Nduro   = new BigInteger("2388005888439481");
       BigInteger N       = new BigInteger("27633027771706698949");
       Ensayo_Error Obj   = new Ensayo_Error();
       Vector   factores;

       factores = Obj.Factzar_Ensayo_Error(N); //factoriza

       //Imprimir vector de factores primos
       System.out.println("\n\n");
       System.out.println("N = "+N+"\n\n");
       System.out.println("Hay " +factores.size()/2+" factores primos <= " + Ge+"\n\n");
       System.out.println("N = "+Obj.print(factores)+"\n\n");
       System.out.println("\n\n");
    }
}
```

Al ejecutar este programa con $N = 367367653565289976655797,$ después de varios segundos la salida es

```
N = 27633027771706698949
Hay 3 factores primos <= 100000
N = 7^2 * 3671^3 * 408011^1
```

1.5 Método de factorización "rho" de Pollard.

1.5.1 Introducción.

En el método de factorización por ensayo y error, en su versión más cruda, probamos con todos los números entre 2 y $\sqrt{N}$ para hallar un factor de N. Si no lo hallamos, N es primo.

En vez de hacer estos $\approx \sqrt{N}$ pasos (en el peor caso), vamos a escoger una lista aleatoria de números, más pequeña que $\sqrt{N}$, y probar con ellos.

A menudo se construyen sucesiones *seudo-aleatorias* $x_0, x_1, x_2, \ldots$ usando una iteración de la forma $x_{i+1} = f(x_i) \pmod N$, con $x_0 = \text{random}(0, N-1)$. Entonces $\{x_0, x_1, \ldots\} \subseteq \mathbb{Z}_N$. Por lo tanto los x_i's se empiezan a repetir en algún momento.

La idea es esta: Supongamos que ya calculamos la sucesión $x_0, x_1, x_2, \ldots$ y que es "suficientemente aleatoria". Si p es un factor primo de N y si

$$\begin{cases} x_i & \equiv & x_j \pmod p \\ x_i & \not\equiv & x_j \pmod N \end{cases}$$

entonces, como $x_i - x_j = kp$, resulta que $\text{MCD}(x_i - x_j, N)$ es un factor no trivial de N.

Claro, no conocemos p, pero conocemos los x_i's, así que podemos revelar la existencia de p con el cálculo del MCD: En la práctica se requiere comparar, de manera eficiente, los x_i con los x_j hasta revelar la presencia del factor p vía el cálculo del $\text{MCD}(x_i - x_j, N)$.

$$\begin{cases} x_i & \equiv & x_j \pmod p \\ x_i & \not\equiv & x_j \pmod N \end{cases} \Longrightarrow \text{MCD}(x_i - x_j, N) \quad \text{es factor no trivial de} \quad N$$

Si $x_0, x_1, x_2, \ldots$ es "suficientemente aleatoria", hay una probabilidad muy alta de que encontremos pronto una "repetición" del tipo $x_i \equiv x_j \pmod p$ antes de que esta repetición ocurra (mod N).

Antes de entrar en los detalles del algoritmo y su eficiencia, veamos un ejemplo.

EJEMPLO 1.14 Sea $N = 1387$. Para crear una sucesión "seudoaleatoria" usamos $f(x) = x^2 - 1$ y $x_1 = 2$. Luego,

$$\begin{aligned} x_0 &= 2 \\ x_{i+1} &= x_i^2 - 1 \pmod{N} \end{aligned}$$

es decir,

$$\begin{aligned} \{x_0, x_1, x_2, \ldots\} = \{&2, 3, 8, 63, 1194, \\ &1186, 177, 814, 996, 310, 396, 84, 120, 529, 1053, 595, 339, \\ &1186, 177, 814, 996, 310, 396, 84, 120, 529, 1053, 595, 339, \ldots\} \end{aligned}$$

Luego, "por inspección" logramos ver que $1186 \not\equiv 8 \pmod{N}$ y luego usamos el detector de factores: $\text{MCD}(1186 - 8, N) = 19$. Y efectivamente, 19 es un factor de 1387. En este caso detectamos directamente un factor primo de N.

Por supuesto, no se trata de comparar todos los x_i's con los x_j's para $j < i$. El método de factorización "rho" de Pollard, en la variante de R. Brent, usa un algoritmo para detectar rápidamente un ciclo en una sucesión ([4]) y hacer solo unas cuantas comparaciones. Es decir, queremos detectar rápidamente $x_i \equiv x_j \pmod{p}$ usando la sucesión $x_{i+1} = f(x_i) \pmod{N}$ (que alcanza un ciclo un poco más tarde) y el test $\text{MCD}(x_i - x_j, N)$.

Típicamente necesitamos unas $O(\sqrt{p})$ operaciones. El argumento es heurístico y se expone más adelante. Básicamente lo que se muestra es que, como en el problema del cumpleaños, dos números x_i y x_j, tomados de manera aleatoria, son congruentes módulo p con probabilidad mayor que $1/2$, después de que hayan sido seleccionados unos $1.177\sqrt{p}$ números.

Aunque la sucesión $x_{i+1} = f(x_i) \pmod{N}$ cae en un ciclo en unas $O(\sqrt{N})$ operaciones, es muy probable que detectemos $x_i \equiv x_j \pmod{p}$ en unos $O(\sqrt{p})$ pasos. Si $p \approx \sqrt{N}$ entonces encontraríamos un factor de N en unos $O(N^{1/4})$ pasos. Esto nos dice que el algoritmo "rho" de Pollard factoriza N^2 con el mismo esfuerzo computacional con el que el método de ensayo y error factoriza N.

1.5.2 Algoritmo.

La algoritmo original de R. Brent compara $x_{2^{k}-1}$ con x_j, donde $2^{k+1}-2^{k-1} \leq j \leq 2^{k+1}-1$. Los detalles de cómo esta manera de proceder detectan rápidamente un ciclo en una sucesión no se ven aquí pero pueden encontrarse en [4] y [22].

EJEMPLO 1.15 Sean $N = 3968039$, $f(x) = x^2 - 1$ y $x_0 = 2$. Luego,

$$\begin{array}{lcl}
\text{MCD}(x_1 - x_3, N) & = & 1 \\
\text{MCD}(x_3 - x_6, N) & = & 1 \\
\text{MCD}(x_3 - x_7, N) & = & 1 \\
\text{MCD}(x_7 - x_{12}, N) & = & 1 \\
\text{MCD}(x_7 - x_{13}, N) & = & 1 \\
\text{MCD}(x_7 - x_{14}, N) & = & 1 \\
\text{MCD}(x_7 - x_{15}, N) & = & 1 \\
\vdots & \vdots & \vdots \\
\text{MCD}(x_{63} - x_{96}, N) & = & 1 \\
\text{MCD}(x_{63} - x_{97}, N) & = & 1 \\
\text{MCD}(x_{63} - x_{98}, N) & = & 1 \\
\text{MCD}(x_{63} - x_{99}, N) & = & 1 \\
\text{MCD}(x_{63} - x_{100}, N) & = & 1 \\
\text{MCD}(x_{63} - x_{101}, N) & = & 1 \\
\text{MCD}(x_{63} - x_{102}, N) & = & 1 \\
\text{MCD}(x_{63} - x_{103}, N) & = & 1987
\end{array}$$

$N = 1987 \cdot 1997$.

En general, hay muchos casos en los que $\text{MCD}(x_i - x_j, N) = 1$. En vez de calcular todos estos $\text{MCD}(z_1, N)$, $\text{MCD}(z_2, N), \ldots$, calculamos unos pocos $\text{MCD}(Q_k, N)$, donde $Q_k = \prod_{j=1}^{k} z_j \pmod{N}$. Brent sugiere escoger k entre $\ln N$ y $N^{1/4}$ pero lejos de cualquiera de los dos extremos ([4]). Riesel ([9]) sugiere tomar k como un múltiplo de 100.

EJEMPLO 1.16 Sean $N = 3968039$, $f(x) = x^2 - 1$ y $x_0 = 2$. Luego, tomando $k = 30$

$$Q_{30} = \prod_{j=1}^{30} z_j \pmod{N} = 3105033, \qquad \text{MCD}(Q_{30}, N) = 1$$
$$Q_{60} = \prod_{j=31}^{60} z_j \pmod{N} = 782878, \qquad \text{MCD}(Q_{60}, N) = 1987$$

El algoritmo que vamos a describir aquí es otra variante del algoritmo de Brent ([5]) que es más sencillo de implementar.

Se calcula $\text{MCD}(x_i - x_j, N)$ para $i = 0,1,3,7,15,...$ y $j = i+1,...,2i+1$ hasta que, o $x_i = x_j \pmod{N}$ (en este caso se debe escoger una f diferente o un x_0 diferente) o que un factor no trivial de N sea encontrado.

Observe que si $i = 2^k - 1$ entonces $j = 2i+1 = 2^{k+1} - 1$, es decir el último j será el 'nuevo' i. Por tanto, en el algoritmo actualizamos x_i al final del `For`, haciendo la asig-

nación $x_i = x_{2i+1} = x_j$.

Algoritmo 1.4: Método rho de Pollard (variante de R. Brent)

Entrada: $N \in \mathbb{N}$, f, x_0

Resultado: Un factor p de N o mensaje de falla.

```
1  salir=false;
2  k = 0;
3  x_i = x_0;
4  while salir=false do
5      i = 2^k − 1;
6      for j = i+1, i+2, ..., 2i+1 do
7          x_j = f(x_0) (mod N);
8          if x_i = x_j then
9              salir=true;
10             Imprimir "El método falló. Reintentar cambiando f o x_0";
11             Exit For;
12         g = MCD(x_i − x_j, N);
13         if 1 < g < N then
14             salir=true;
15             Imprimir N = N/g · g;
16             Exit For;
17         x_0 = x_j;
18     x_i = x_j;
19     k++;
```

1.5.3 Implementación en Java.

La implementación sigue paso a paso el algoritmo.

```
import java.math.BigInteger;
public class rhoPollard
{
rhoPollard(){}

public BigInteger f(BigInteger x)
{
```

```
    return x.multiply(x).add(BigInteger.ONE);//x^2+1
}

 public  void FactorPollard(BigInteger N)
 {
     int i, k;
     BigInteger xi,xj;
     BigInteger g   = BigInteger.ONE;
     BigInteger x0  = new BigInteger(""+2);;
     boolean salir  = false;

     k = 0;
     xi= x0;
     xj= x0;
     while(salir==false)
     { i=(int)(Math.pow(2,k)-1);
       for(int j=i+1; j<=2*i+1; j++)
       {
           xj=f(x0).mod(N);
           if(xi.compareTo(xj)==0)//si son iguales
           {salir=true;
            System.out.print("Fallo"+"\n\n");
            break;
           }
           g= N.gcd(xi.subtract(xj));
           if(g.compareTo(BigInteger.ONE)==1 && g.compareTo(N)==-1)//1<g<N
           {salir=true;
            System.out.print("Factor = "+g+"\n\n");
            break;
           }
           x0=xj;
       }
       xi=xj;
       k++;
     }
     System.out.print(N+" =  "+g+" . "+N.divide(g)+"\n\n");
  }

  public static void main(String[] args)
```

```
  {
   System.out.print("\n\n");
      rhoPollard obj = new rhoPollard();
      BigInteger N = new BigInteger("39680399966886876527");
      obj.FactorPollard(N);
   System.out.print("\n\n");
 }
}//
```

Sería bueno implementar una variante con el producto $Q_k = \prod_{j=1}^{k} z_j \pmod{N}$.

1.5.4 Complejidad.

En el método de Pollard-Brent $x_{i+1} = f(x_i) \pmod{N}$ entonces, si f entra en un ciclo, se mantiene en este ciclo. Esto es así pues si $f(x_i) = x_j \implies f(x_{i+1}) = f(f(x_i)) = f(x_j) = x_{j+1}$.

Si $a \neq 0, -2,$ se ha establecido de manera empírica (aunque no ha sido probado todavía), que esta es una sucesión de números suficientemente aleatoria que se vuelve periódica después de unos $O(\sqrt{p})$ pasos.

Lo de que se vuelva periódica, en algo parecido a $O(\sqrt{p})$ pasos, es fácil de entender si consideramos el *problema del cumpleaños* ("birthday problem"): ¿Cuántas personas se necesita seleccionar, de manera aleatoria, de tal manera que la probabilidad de que al menos dos de ellas cumplan años el mismo día, exceda $1/2$?

Si tomamos $n+1$ objetos (con reemplazo) de $N,$ la probabilidad de que *sean diferentes* es

$$Pr(n) = \left(1 - \frac{1}{N}\right)\left(1 - \frac{2}{N}\right)\cdots\left(1 - \frac{n}{N}\right)$$

Puesto que $\ln(1-x) > -x,$ si x es suficientemente pequeño,

$$\ln Pr(n) \; > \; -\frac{1}{N}\sum_{k=1}^{n} k \; \sim \; -\frac{\frac{1}{2}n^2}{N}$$

Por tanto, para estar seguros que $Pr(n) \leq 1/2$ necesitamos

$$n > \sqrt{N}\cdot\sqrt{2\ln 2} \; \approx \; 1.1774\sqrt{N}.$$

Ahora, si no tomamos en cuenta que los años tienen distinto número de días (digamos que son de $N = 365$ días) y que hay meses en que hay más nacimientos que otros, el razonamiento anterior dice que si en un cuarto hay $n = 23$ personas, la probabilidad de que al menos dos coincidan en su fecha de nacimiento es más grande que $1/2$.

Ahora bien, ¿qué tan grande debe ser k de tal manera que al menos dos enteros, escogidos de manera aleatoria, sean congruentes (mod p) con probabilidad mayor que $1/2$?

Bueno, en este caso $N = p$ y k debe cumplir

$$Pr(k) = \left(1 - \frac{1}{p}\right)\left(1 - \frac{2}{p}\right)\cdots\left(1 - \frac{k-1}{p}\right) < \frac{1}{2}$$

Así, $\ln Pr(k) = -\frac{1}{p}\sum_{j=1}^{k-1} j \implies Pr(k) \approx e^{-k(k-1)/2p}$. Luego, $Pr(k) \approx 1/2$ si $k \approx \sqrt{2p\ln 2} \approx 1.18\sqrt{p}$.
Bien, si $N = p \cdot m$ con p primo y $p \leq \sqrt{N}$ y seleccionamos de manera aleatoria algo más de $\sqrt{p}$ enteros x_i entonces la probabilidad de que $x_i = x_j \pmod p$ con $i \neq j$, es mayor que $1/2$.

1.6 Pruebas de Primalidad.

1.6.1 Introducción.

Para decidir si un número n pequeño es primo, podemos usar el método de ensayo y error para verificar que no tiene divisores primos inferiores a $\sqrt{n}$.

Para un número un poco más grande, la estrategia usual es primero verificar si tiene divisores primos pequeños, sino se usa el test para seudoprimos fuertes de Miller-Rabin con unas pocas bases p_i (con p_i primo) y usualmente se combina con el test de Lucas. Esta manera de proceder decide de manera correcta si un número es primo o no, hasta cierta cota 10^M. Es decir, la combinación de algoritmos decide de manera correcta si $n < 10^M$ Sino, decide de manera correcta solamente con una alta probabilidad y cabe la (remota)

posibilidad de declarar un número compuesto como primo.

Aquí solo vamos a tratar rápidamente la prueba de Miller-Rabin.

1.6.2 Prueba de primalidad de Miller Rabin.

Iniciamos con test de primalidad de Fermat, por razones históricas. Esta prueba se basa el el teorema,

Teorema 1.9 (Fermat)
Sea p primo. Si $MCD(a,p)=1$ entonces $a^{p-1} \equiv 1 \pmod{p}$.

Este teorema nos dice que si n es primo y a es un entero tal que $1 \leq a \leq n-1$, entonces $a^{n-1} \equiv 1 \pmod{n}$.

Por tanto, para probar que n es *compuesto* bastaría encontrar $1 \leq a \leq n-1$ tal que $a^{n-1} \not\equiv 1 \pmod{n}$.

Definición 1.2 *Sea n compuesto. Un entero $1 \leq a \leq n-1$ para el que $a^{n-1} \not\equiv 1 \pmod{n}$, se llama "testigo de Fermat" para n.*

Un testigo de Fermat para n sería un testigo de no-primalidad. De manera similar, un número $1 \leq a \leq n-1$ para el que $a^{n-1} \equiv 1 \pmod{n}$, apoya la posibilidad de que n sea primo,

Definición 1.3 *Sea n un entero compuesto y sea a un entero para el cual $1 \leq a \leq n-1$ y $a^{n-1} \equiv 1 \pmod{n}$. Entonces se dice que n es un seudoprimo respecto a la base a. Al entero a se le llama un "embaucador de Fermat" para n.*

Por ejemplo, $n = 645 = 3 \cdot 5 \cdot 43$ es un seudoprimo en base 2 pues $2^{n-1} \equiv 1 \pmod{n}$.

Es curioso que los seudoprimos en base 2 sean muy escasos. Por ejemplo, hay 882 206 716 primos inferiores a 2×10^{10} y solo hay 19685 seudoprimos en base 2 inferiores a 2×10^{10}. Esto nos dice que la base 2 parece ser muy poco "embaucadora" en el sentido de que si tomamos un número grande n de manera aleatoria y si verificamos que $2^{n-1}\equiv 1\,(\mathrm{mod}\,n)$, entonces es muy probable que n sea primo. También los seudoprimos en base 3 son muy escasos y es altamente improbable que si tomamos un número grande n de manera aleatoria, este sea compuesto y que a la vez sea simultáneamente seudoprimo en base 2 y base 3.

Es decir, si un número n pasa los dos test $2^{n-1}\equiv 1\,(\mathrm{mod}\,n)$ y $3^{n-1}\equiv 1\,(\mathrm{mod}\,n)$; es muy probable que sea primo.

Sin embargo, hay enteros n compuestos para los cuales $a^{n-1}\equiv 1\,(\mathrm{mod}\,n)$ para todo a que cumpla $\mathrm{MCD}(a,n)=1$. A estos enteros se les llama números de Carmichael.

Por ejemplo, $n=561=3\cdot11\cdot17$ es número de Carmichael. Aunque este conjunto de números es infinito, son más bien raros (poco densos). En los primeros 100 000 000 números naturales hay 2051 seudoprimos en base 2 y solo 252 números de Carmichael.

Nuestra situación es esta: Es poco probable que un número compuesto pase varios test de "primalidad" $a^{n-1}\equiv 1\,(\mathrm{mod}\,n)$ excepto los números de Carmichael, que son compuestos y pasan todos estos test.

Hay otro test, llamado "test fuerte de seudo-primalidad en base a" el cual los números de Carmichael no pasan. Además, si tomamos k números de manera aleatoria $a_1,a_2,\ldots,a_k$ y si n pasa este test en cada una de las bases a_i, podemos decir que la probabilidad de que nos equivoquemos al declarar n como primo es menor que $1/4^k$. Por ejemplo, si $k=200$ la probabilidad de que nos equivoquemos es $<10^{-120}$

Teorema 1.10 *Sea n un primo impar y sea $n-1=2^s r$ con r impar. Sea a un entero tal que $MCD(a,n)=1$. Entonces, o $a^r\equiv 1(\mathrm{mod}\,n)$ o $a^{2^j r}\equiv -1(\mathrm{mod}\,n)$ para algún j, $0\leq js-1$.*

Con base en el teorema anterior, tenemos

Definición 1.4 *Sea n impar y compuesto y sea $n-1=2^s r$ con r impar. Sea $1 \leq a \leq n-1$.*

(i) Si $a^r \not\equiv 1 (\mod n)$ y si $a^{2^j r} \not\equiv -1 (\mod n)$ para $0 \leq j \leq s-1$, entonces a es llamado un testigo fuerte (de no-primalidad) de n.

(ii) Si $a^r \equiv 1 (\mod n)$ y si $a^{2^j r} \equiv -1 (\mod n)$ para $0 \leq j \leq s-1$, entonces n se dice un seudoprimo fuerte en la base a. Al entero a se le llama "embaucador fuerte".

Así, un seudoprimo fuerte n en base a es un número que actúa como un primo en el sentido del teorema 1.10.

Teorema 1.11 *(***Rabin)**

Si n es un entero compuesto, a lo sumo $\frac{1}{4}$ de todos los números a, $1 \leq a \leq n-1$, son embaucadores fuertes de n.

Supongamos que tenemos un número compuesto n. Tomamos k números $\{a_1, a_2, ..., a_k\}$ de manera aleatoria y aplicamos el test fuerte de seudo-primalidad a n con cada uno de de estas bases a_i. Entonces, hay menos que un chance en cuatro de que a_1 no sea testigo de no-primalidad de n, y menos que un chance en cuatro de que a_2 no sea testigo de no-primalidad de n, etc. Si n es primo, pasa el test para cualquier $a < n$. Si cada a_i falla en probar que n es compuesto, entonces la probabilidad de equivocarnos al decir que n es primo es inferior a $\frac{1}{4^k}$.

1.6.3 Algoritmo.

Algoritmo 1.5: Miller-Rabin

Entrada: $n \geq 3$ y un parámetro de seguridad $t \geq 1$.
Resultado: "n es primo" o "n es compuesto".

```
1  Calcule r y s tal que n − 1 = 2^s r, r impar;
2  for i = 1,2,...,t do
3      a = Random(2, n − 2);
4      y = a^r (mod n);
5      if y ≠ 1 y y ≠ n − 1 then
6          j = 1;
7          while j ≤ s − 1 y y ≠ n − 1 do
8              y = y^2 (mod n);
9              if y = 1 then
10                 return "Compuesto";
11             j = j + 1;
12         if y ≠ n − 1 then
13             return "Compuesto";
14 return "Primo";
```

El algoritmo 1.5 verifica si en cada base a se satisface la definición 1.4. En la línea 9, si $y = 1$, entonces $a^{2^j r} \equiv 1 \pmod{n}$. Puesto que este es el caso cuando $a^{2^{j-1} r} \not\equiv \pm 1 \pmod{n}$ entonces n es compuesto. Esto es así pues si $x^2 \equiv y^2 \pmod{n}$ pero si $x \not\equiv \pm y \pmod{n}$, entonces $\text{MCD}(x-y, n)$ es un factor no trivial de n. En la línea 12, si $y \neq n-1$, entonces a es un testigo fuerte de n.

Si el algoritmo 1.5 declara compuesto a n entonces n es definitivamente compuesto, por el teorema 1.10. Si n es primo, es declarado primo. Si n es compuesto, la probabilidad de que el algoritmo lo declare primo es inferior a $1/4^t$.

El algoritmo 1.5 requiere, para $n-1 = 2^j r$ con r impar, $t(2+j)\ln n$ pasos. t es el número de bases.

Una estrategia que se usa a veces es fijar las bases. Se toman como base algunos de los primeros primos en vez de tomarlas de manera aleatoria. El resultado importante aquí es este: Si $p_1, p_2, ..., p_t$ son los primeros t primos y si ψ_t es el más pequeño entero compuesto el cual es seudoprimo para todas las bases $p_1, p_2, ..., p_t$, entonces el algoritmo de

Miller-Rabin, con las bases $p_1, p_2, ..., p_t$, siempre responde de manera correcta si $n < \psi_t$. Para $1 \leq t \leq 8$ tenemos

t	ψ_t
1	2047
2	1373653
3	25326001
4	3215031751
5	2152302898747
6	3474749660383
7	341550071728321
8	341550071728321

1.6.4 Implementación en Java.

En la clase BigInteger de Java ya viene implementado el método `this.modPow(BigInteger r, BigInteger N)` para calcular $y = a^r \pmod{N}$. Para calcular r y s solo se divide $N-1$ por dos hasta que el residuo sea diferente de cero.

En esta implementación usamos los primeros ocho primos como bases. Así el algoritmo responde de manera totalmente correcta si $N < 341550071728321$.

```
import java.math.BigInteger;
import java.util.*;

public class Miller_Rabin
{
    public Miller_Rabin(){}

    public boolean esPrimoMR(BigInteger N)
    {
        //n>3 e impar. Respuesta 100% segura si N <341 550 071 728 321
        BigInteger  N1      = N.subtract(BigInteger.ONE);//N-1
        BigInteger  DOS     = new BigInteger("2");
        int[]       primo   = {2,3,5,7,11,13,17,19};
        int         s       = 0;
        boolean     esPrimo = true;
        BigInteger  a,r,y;
        int         j;
```

```
    //n-1 = 2^s r
    while(N1.remainder(DOS).compareTo(BigInteger.ZERO)==0)
    {    N1=N1.divide(DOS);
         s=s+1;
    }
    r  = N1;
    N1 = N.subtract(BigInteger.ONE);

    for(int i=0; i<=7; i++)
    {
        a = new BigInteger(""+primo[i]);
        y = a.modPow(r, N);
        if( y.compareTo(BigInteger.ONE)!=0 && y.compareTo(N1)!=0)
        {
           j=1;
           while(j<= s-1 && y.compareTo(N1)!=0 )
           {
              y = y.modPow(DOS, N);
              if(y.compareTo(BigInteger.ONE)==0) esPrimo=false;
              j++;
           }
           if(y.compareTo(N1)!=0) esPrimo = false;
        }

    }
   return esPrimo;
}

public static void main(String[] args)
{
  System.out.println("\n\n");
  BigInteger  N     = new BigInteger("6658378974");
  Miller_Rabin  obj = new Miller_Rabin();

  System.out.println(N+" es primo = "+obj.esPrimoMR(N)+"\n\n");

  System.out.println("\n\n");
}
```

```
}
```

Apéndice A

Notación O grande y algoritmos.

En esta primera parte vamos a establecer algunos resultados que nos servirán más adelante para hacer estimaciones que nos ayuden a analizar los algoritmos que nos ocupan. Como vamos a hablar de algunos tópicos de la teoría de números en términos de comportamiento promedio, necesitamos establecer algunas ideas y teoremas antes de entrar a la parte algorítmica.

A.1 Notación O grande

La notación O grande se usa para comparar funciones "complicadas" con funciones más familiares.

Por ejemplo, en teoría analítica de números, es frecuente ver el producto

$$\left(1-\frac{1}{2}\right)\left(1-\frac{1}{3}\right)\left(1-\frac{1}{5}\right)\cdots = \prod_{\substack{p\leq x,\\ p \text{ primo}}}\left(1-\frac{1}{p}\right)$$

*

Este producto nos da una estimación de la proporción (o densidad) de enteros positivos sin factores primos inferiores a x (en un conjunto $\{1,2,...,n\}$). Si x es muy grande, se vuelve complicado calcular este producto porque no es fácil obtener todos los primos que uno quiera en un tiempo razonable. En vez de eso, tenemos la fórmula de Mertens (1874)

$$\prod_{\substack{p\leq x,\\ p \text{ primo}}} \left(1-\frac{1}{p}\right) = \frac{e^{-\gamma}}{\log x} + O\left(1/\log^2(x)\right)$$

Esta fórmula dice que el producto es comparable a la función $e^{-\gamma}/\log x$, con la cual nos podemos sentir más cómodos. Aquí $e^{-\gamma} \approx 0.561459$ y $O\left(1/\log^2(x)\right)$, la estimación del "error" en esta comparación, indica una función inferior a un múltiplo de $1/\log^2(x)$ si x es suficientemente grande.

Definición A.1 *Si $h(x) > 0$ para toda $x \geq a$, escribimos*

$$f(x) = O(h(x)) \quad \textit{si existe} \quad C \quad \textit{tal que} \quad |f(x)| \leq C h(x) \quad \textit{para toda} \quad x \geq a. \tag{A.1}$$

Escribimos

$$f(x) = g(x) + O(h(x))$$

si existe C tal que $|f(x) - g(x)| \leq C h(x)$ siempre que $x \geq a$

También podemos pensar en "$O(h(x))$" como una función que es dominada por $h(x)$ a partir de algún $x \geq a$.

Definición A.2 *Si*

$$\lim_{x\to\infty} \frac{f(x)}{g(x)} = 1$$

decimos que f es asintóticamente igual a g conforme $x \to \infty$ y escribimos,

$$f(x) \sim g(x) \quad \textit{conforme} \quad x \to \infty$$

En los siguientes ejemplos se exponen algunos resultados y algunos métodos que vamos a usar más adelante con la intención de que las pruebas queden de tamaño aceptable.

EJEMPLO A.1 Si $h(x) > 0$ para toda $x \geq a$ y si h es acotada, entonces h es $O(1)$. En particular $\operatorname{sen}(x) = O(1)$

EJEMPLO A.2 $\sqrt{x} = O\left(\dfrac{x}{\ln^2(x)}\right)$

Lo que hacemos es relacionar $\sqrt{x}$ con $\ln(x)$ usando (la expansión en serie de) e^x. Sea $y = \ln(x)$. Luego $e^y = \sum_{k=0}^{\infty} y^k/k! \geq y^4/4!$, así que

$$\begin{aligned} 24e^y \geq y^4 &\implies \ln^4(x) \leq 24x \\ &\implies \ln^2(x)\sqrt{x} \leq \sqrt{24}x \\ &\implies \sqrt{x} \leq \sqrt{24}\,\frac{x}{\ln^2(x)} \end{aligned}$$

EJEMPLO A.3 Si $n, d \neq 0 \in \mathbb{N}$ entonces $[n/d] = n/d + O(1)$. Aquí, $[x]$ denota la parte entera de x.

En efecto, por el algoritmo de la división, existe $k, r \in \mathbb{Z}$ tal que $n = k \cdot d + r$ con $0 \leq r < d$ o también $\dfrac{n}{d} = k + \dfrac{r}{d}$. Luego, $\left[\dfrac{n}{d}\right] = k = \dfrac{n-r}{d}$.

Ahora, $\left|\left[\dfrac{n}{d}\right] - \dfrac{n}{d}\right| = \dfrac{r}{d} < 1$ para cada $n \geq 0$. Así, tenemos $[n/d] = n/d + O(1)$, tomando $C = 1$.

EJEMPLO A.4 Aunque la serie armónica $\displaystyle\sum_{k=1}^{\infty} \frac{1}{k}$ es divergente, la función $H_n = \displaystyle\sum_{k=1}^{n} \frac{1}{k}$ es muy útil en teoría analítica de números. La función $\tau(n)$ cuenta cuántos divisores tiene n. Vamos a mostrar que $\displaystyle\sum_{k=1}^{n} \tau(k) = nH(n) + O(n)$ y entonces, $\displaystyle\sum_{k=1}^{n} \tau(k) = n\ln(n) + O(n)$.

Primero, vamos a mostrar, usando argumentos geométricos, que existe una número real γ, llamada *constante de Euler*, tal que

$$H_n = \ln(n) + \gamma + O(1/n).$$

Prueba. Hay que mostrar que $\exists C$ tal que $0 < H_n - \ln(n) - \gamma < C \cdot 1/n$ para $n > n_0$.

Usando integral de Riemann,

$$\sum_{k=1}^{n-1} \frac{1}{k} = \int_1^n \frac{1}{x}\,dx + E_n \quad \text{i.e.} \quad H_{n-1} = \ln(n) + E_n$$

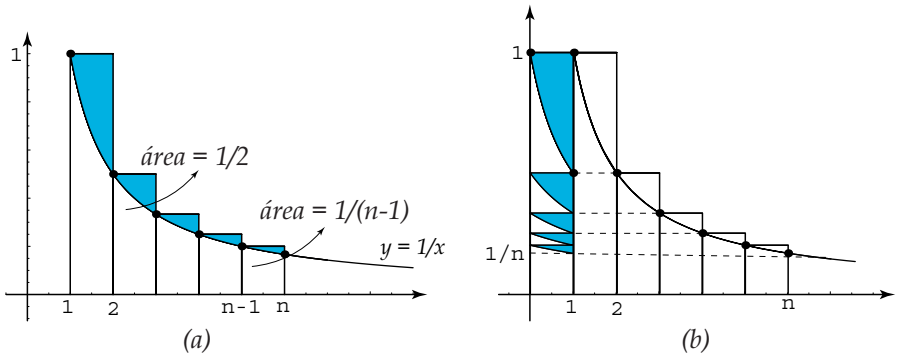


Figura A.1 Comparando el área $\ln(n)$ con la suma H_n.

Geométricamente, H_{n-1} corresponde a la suma de las áreas de los rectángulos desde 1 hasta n y E_n la suma de las áreas de las porciones de los rectángulos sobre la curva $y = 1/x$.

En el gráfico (b) de la figura A.1 vemos que $E_n \leq 1$ para toda $n \geq 1$, así que E_n es una función de n, que se mantiene acotada y es creciente, por lo tanto esta función tiene un límite, el cual vamos a denotar con γ. Así, $\lim_{n\to\infty} E_n = \gamma$. En particular, para cada n fijo, $\gamma > E_n$.

Como $\gamma - E_n$ corresponde a la suma (infinita) de las áreas de las regiones sombreadas en la figura A.2, se establece la desigualdad

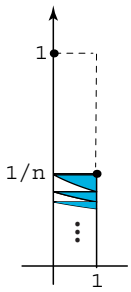


Figura A.2 $\gamma - E_n$.

$$\gamma - E_n < 1/n$$

de donde

$$0 < \gamma - (H_{n-1} - \ln(n)) < 1/n.$$

Ahora restamos $1/n$ a ambos lados para hacer que aparezca H_n, tenemos

$$\frac{1}{n} > H_n - \ln(n) - \gamma > 0$$

que era lo que queríamos demostrar.

Aunque en la demostración se establece $H_n - \ln(n) - \gamma < 1/n$, la estimación del error $O(1/n)$ corresponde a una función dominada por un múltiplo de $1/n$. Veamos ahora algunos cálculos que pretender evidenciar el significado de $O(1/n)$.

n	H_n	$\ln(n)$	$\lvert H_n - \ln(n) - \gamma \rvert$	$1/n$
170000	12.62077232	12.62076938	$2.94117358 \times 10^{-6}$	$5.88235294 \times 10^{-6}$
180000	12.67793057	12.67792779	$2.77777520 \times 10^{-6}$	$5.55555555 \times 10^{-6}$
190000	12.73199764	12.73199501	$2.63157663 \times 10^{-6}$	$5.26315789 \times 10^{-6}$
200000	12.78329081	12.78328831	$2.49999791 \times 10^{-6}$	$5. \times 10^{-6}$

Observando las dos últimas columnas se puede establecer una mejor estimación del error con $\frac{1}{2n}$ y todavía mejor con $\frac{1}{2n} - \frac{1}{12n^2}$!

n	H_n	$\ln(n)+\gamma+\frac{1}{2n}-\frac{1}{12n^2}$
100000	12.090146129863427	12.090146129863427
150000	12.495609571309556	12.495609571309554
200000	12.783290810429621	12.783290810429623

También, de estas tablas se puede obtener la aproximación $\gamma \approx 0.577216$

Segundo, vamos a mostrar que $\sum_{k=1}^{n} \tau(k) = nH(n)+O(n)$ y que $\sum_{k=1}^{n} \tau(k) = n\ln(n)+O(n)$. Podemos poner $\tau(k)$ como una suma que corre sobre los divisores de k, $\tau(k) = \sum_{d|k} 1$. Luego,

$$\sum_{k=1}^{n} \tau(k) = \sum_{k=1}^{n} \sum_{d|k} 1$$

La idea es usar argumentos de divisibilidad para usar la expansión del ejemplo A.3. Si $d|k$ entonces $k = d \cdot c \leq n$. Esto nos dice que el conjunto de todos los divisores positivos de los números k inferiores o iguales a n, se puede describir como el conjunto de todos los pares (c,d) con la propiedad $cd \leq n$ (por supuesto, se puede hacer una demostración formal probando la doble implicación "$\Longleftrightarrow$").

Ahora, $cd \leq n \iff d \leq n \wedge c \leq n/d$. Entonces podemos escribir,

$$\sum_{k=1}^{n} \tau(k) = \sum_{\substack{c,d \\ cd \leq n}} 1 = \sum_{d \leq n} \sum_{c \leq n/d} 1$$

La suma $\sum_{c \leq n/d} 1$ corre sobre los enteros positivos menores o iguales que n/d. Esto nos da $[n/d]$ sumandos, i.e. $\sum_{c \leq n/d} 1 = [n/d]$. Finalmente, usando el ejemplo A.3,

$$\begin{aligned}\sum_{k=1}^{n}\tau(k) &= \sum_{d\le n}[n/d]\\ &= \sum_{d\le n}\{n/d+O(1)\}\\ &= \sum_{d\le n}n/d \;+\; \sum_{d\le n}O(1)\\ &= n\sum_{d\le n}1/d \;+\; \sum_{d\le n}O(1)\\ &= nH_n+O(n)\end{aligned}$$

En los ejercicios se pide mostrar, usando la figura A.1, que $H_n = \log(n)+O(1)$. Usando este hecho,

$$\sum_{k=1}^{n}\tau(k) = nH_n+O(n) = n\{\ln(n)+O(1)\}+O(n) = n\ln(n)+O(n).$$

(Los pequeños detalles que faltan se completan en los ejercicios)

EJERCICIOS

A.1 Probar que $\sum_{d\le n}O(1) = O(n)$

A.2 Probar que $nO(1)+O(n) = O(n)$

A.3 Usar la figura A.1 para probar que $H_n = \log(n)+O(1)$.

A.4 Probar que $\dfrac{H_n-\log(n)}{\gamma} = 1+O(1/n)$

A.5 Probar que $\sqrt{n}H_n = \sqrt{n}\log(n)+O(\sqrt{n})$

A.6 Probar que $H_n-\log(n) \sim \gamma$

A.7 Probar que $H_n \sim \log(n)$

A.2 Estimación de la complejidad computacional de un algoritmo.

La teoría de la complejidad estudia la cantidad de pasos necesarios para resolver un problema dado. Lo que nos interesa aquí es cómo el número de pasos crece en función del tamaño de la entrada ("input"), despreciando detalles de hardware.

Tiempo de corrida.

El tiempo de corrida de un algoritmo es, simplificando, el número de pasos que se ejecutan al recibir una entrada de tamaño n. Cada paso i tiene un costo c_i distinto, pero hacemos caso omiso de este hecho y solo contamos el número de pasos.

La complejidad $T(n)$ de un algoritmo es el número de pasos necesario para resolver un problema de tamaño (input) n. Un algoritmo es de orden a lo sumo $g(n)$ si existen constantes c y M tales que,

$$T(n) < cg(n), \quad \forall n > M$$

En este caso escribimos $T(n) = O(g(n))$. Esta definición dice que esencialmente T no crece más rápido que g. El interés, por supuesto, funciones g que crecimiento "lento".

Clases de complejidad.

Las funciones g usadas para medir complejidad computacional se dividen en diferentes *clases de complejidad.*

Si un algoritmo es de orden $O(\ln n)$ tiene *complejidad logarítmica.* Esto indica que es un algoritmo muy eficiente pues si pasamos de un input de tamaño n a otro de tamaño $2n$, el algoritmo hace pocos pasos adicionales pues $T(2n) \leq c\ln 2n = c\ln 2 + c\ln n \leq c(1+\ln n)$.

Otra importante clase de algoritmos, son los de orden $O(n\ln n)$. Muchos algoritmos de ordenamiento son de esta clase.

Los algoritmos de orden $O(n^k)$ se dicen de *complejidad polinomial.* Hay muchos algoritmos importantes de orden $O(n^2)$, $O(n^3)$. Si el exponente es muy grande, el algoritmo usualmente se vuelve ineficiente.

Si g es exponencial, decimos que el algoritmo tiene *complejidad exponencial*. La mayoría de estos algoritmos no son prácticos.

EJEMPLO A.5

- Si $T(n) = n^3 + 4$ entonces $T(n) = O(n^3)$ pues $n^3 + 4 \leq cn^3$ si $c > 1$.

- Si $T(n) = n^5 + 4n + \ln(n) + 5$ entonces $T(n) = O(n^5)$.

  Para ver esto, solo es necesario observar que n^5 *domina* a los otros sumandos para n suficientemente grande.

  En particular, $n^5 \geq \ln(n)$ si $n \geq 1$: Sea $h(n) = n^5 - \ln(n)$. Entonces $h'(n) = 5n^4 - 1/n \geq 0$ si $n \geq 1$ entonces h es creciente. Luego $h(n) \geq h(1) \geq 0$ si $n \geq 1$.

EJEMPLO A.6 Consideremos el siguiente fragmento de código,

```
while (N > 1)
{
    N = N / 2;
}
```

En este ejemplo, cada iteración requiere una comparación `'N>1'` y una división `'N/2'`. Obviamos el costo de estas operaciones y contamos el número de pasos.

Si la entrada es un número real n entonces la variable N almacenará los valores $\{n, n/2, n/2^2, n/2^3, ...\}$. En el $k-$ésimo paso, la variable N almacena el número $n/2^{k-1}$ y el ciclo se detiene si $n/2^{k-1} < 1$, es decir, el ciclo se detiene en el momento que $k - 1 > \lg(n)$. Así que se ejecutan aproximadamente $2\lg(n) + 2$ pasos para un input de "tamaño" n. Por lo tanto es el tiempo de corrida es $T(n) = O(\lg(n))$

EJEMPLO A.7 En el siguiente código, contamos los ceros en un arreglo `a` de tamaño N.

```
int count = 0;
for (int i = 0; i < N; i++)
    if (a[i] == 0) count++;\\
```

Los pasos son

Declaración de variables	2
Asignación de variables	2
Comparaciones `i<N`	$N+1$
Comparaciones `a[i] == 0`	N
Accesos al array `a[]`	N
Incrementos	$\leq 2N$
Total	$4N+5 \leq T(N) \leq 5N+5$

Observe que el incremento `i++` se realiza N veces mientras que el incremento `count++` se realiza como máximo N veces (solo si el arreglo `a[]` tiene todas las entradas nulas).

El tiempo de corrida es $T(n) = O(n)$.

EJEMPLO A.8 En el siguiente código, contamos las entradas `(i,j)` tal que $a[i]+a[j]=0$. `a[]` es un arreglo de tamaño N.

```
int count = 0;
    for (int i = 0; i < N; i++)
        for (int j = i+1; j < N; j++)
            if (a[i] + a[j] == 0) count++;
```

Para el conteo, usamos la identidad $\sum_{i=1}^{N} i = 1/2N(N+1)$

Declaración de variables	$N+2$
Asignación de variables	$N+2$
Comparaciones `i<N, J<N`	$1/2(N+1)(N+2)$
Comparaciones `a[i] + a[j] == 0`	$1/2N(N-1)$
Accesos al array `a[]`	$N(N-1)$
Incrementos (máximo N^2, mínimo N^2-N)	$\leq N^2$

$5+N+3N^2 \leq T(N) \leq 3N^2+2N+5$

El tiempo de corrida es $T(n) = O(n^2)$

EJEMPLO A.9 1. Si $f,g:\mathbb{N}\longrightarrow\mathbb{R}^+$, muestre que $\text{Máx}\{f(n),g(n)\}=O(f(n)+g(n))$.

Solución.

Hay que demostrar que existe c y M tal que $\text{Máx}\{f(n),g(n)\}\leq c\cdot(f(n)+g(n)),\ \forall n\geq M$.

Por definición de máximo, $\text{Máx}\{f(n),g(n)\}\leq f(n)+g(n)$. Así, la definición de O-grande se cumple para $c=1$ y $M=1$,

$$\text{Máx}\{f(n),g(n)\}\leq 1\cdot(f(n)+g(n))\ \ \forall n\geq 1.$$

2. Muestre que $2^{n+1}=O(2^n)$ pero $2^{2n}\neq O(2^n)$

Solución.

A.) Hay que demostrar que existe c y M tal que $2^{n+1}\leq c\cdot 2^n\ \ \forall n\geq M$.

$2^{n+1}=O(2^n)$ pues $2^{n+1}\leq 2\cdot 2^n\ \ \forall n$.

B.) Por contradicción. $2^{2n}\neq O(2^n)$ pues si $2^{2n}\leq c2^n\implies 2^n\leq c$ y esto no puede ser pues 2^n no es acotada superiormente (2^n es creciente).

3. Muestre que $f\in O(g)$ no implica necesariamente que $g\in O(f)$

Solución.

Un ejemplo es suficiente.

Por ejemplo, $n=O(n^2)$ pero $n^2\neq O(n)$ pues $f(n)=n$ no es acotada.

4. Si $f(n)\geq 1$ y $\lg[g(n)]\geq 1$ entonces muestre que si $f\in O(g)\implies \lg[f]\in O(\lg[g])$

Solución.

Hay que demostrar que existe c y M tal que $\lg f(n) \leq c_1 \cdot \lg g(n),\ \forall\, n \geq M$.

$f \in O(g) \implies f(n) \leq c\,g(n),\ \forall n \geq M$. Luego, $\lg f(n) \leq \lg(c\,g(n)) = \lg c + \lg g(n)$ por ser lg una función creciente.

Como $\lg[g(n)] \geq 1 \implies \lg c \ \leq\ \lg c \cdot \lg g(n)$, entonces

$$\lg f(n) \leq \lg c + \lg g(n) \leq (\lg c + 1) \lg g(n), \quad \forall n \geq M.$$

Así que basta tomar $c_1 = \lg c + 1$ para que se cumpla la definición de O-grande.

5. Calcule el tiempo de corrida del siguiente programa

```
public class ThreeSum {
    // retorna el n\'umero de distintos (i, j, k)
    // tal que (a[i] + a[j] + a[k] == 0)
public static int count(int[] a) {
int N = a.length;
int cnt = 0;
    for (int i = 0; i < N; i++)
        for (int j = i+1; j < N; j++)
            for (int k = j+1; k < N; k++)
                if (a[i] + a[j] + a[k] == 0) cnt++;
return cnt;
}}
```

Solución.

Debemos contar los pasos que hace el programa.

El término dominante en el tiempo de corrida es la cantidad de accesos a la instrucción ‘`if (a[i] + a[j] + a[k] == 0) cnt++;`’. Contar este número de accesos

será suficiente para establecer el tiempo de corrida.

En el conteo que sigue usamos las fórmulas

$$\sum_{i=1}^{M} i = \frac{1}{2}M(M+1) \tag{A.2}$$

$$\sum_{i=1}^{M} i^2 = \frac{1}{6}M(M+1)(2M+1). \tag{A.3}$$

Para contar cuántas veces se ejecuta el `if` del tercer `for`, observemos que este `if` solo se ejecuta (pasa el test `k < N`) si $j \leq N-2$, es decir, si $i \leq N-3$.

i	j	Veces que se ejecuta el `if` del 3er `for`
0	1 to N	$\sum_{k=2}^{N-1}(N-k) = 1+2+\ldots+N-2$
1	2 to N	$\sum_{k=3}^{N-1}(N-k) = 1+2+\ldots+N-3$
$\vdots$	$\vdots$	$\vdots$
$N-3$	$N-2$ to N	1
$N-2$	$N-1$ to N	No se ejecuta

Así, el `if` del tercer `for` se ejecuta

$$\begin{aligned}
\sum_{j=2}^{N}(1+2+\ldots+N-j) &= \sum_{j=2}^{N}(1/2(N-j)(N-j+1)) \quad \text{por (A.2)} \\
&= \sum_{j=2}^{N}\left(\frac{j^2}{2} - (N+1/2)j + \frac{N^2}{2} + \frac{N}{2}\right) \\
&= \frac{N^3}{6} - \frac{N^2}{2} + \frac{N}{3} \quad \text{usando (A.2) y (A.3)}
\end{aligned}$$

Finalmente, $T(n) = O(n^3)$.

Nota: Otra forma de contar es observando que el programa recorre todos los tripletes, con componentes distintas, (i, j, k) de un conjunto de N elementos (el arreglo `a[]`), es decir,

$$\binom{N}{3} = 1/6N(N-1)(N-2) = \frac{N^3}{6} - \frac{N^2}{2} + \frac{N}{3}$$

Tamaño del input en teoría de números.

El tamaño del "input" depende de la clase de problema que estemos analizando. En los algoritmos de búsqueda y ordenamiento el tamaño de la entrada es el número de datos. En teoría algorítmica de números, el tamaño de la entrada es el número de dígitos del número de entrada y la complejidad se mide en términos de cantidad de operaciones de bits.

Número de dígitos.

La representación en base $b = 2$ de un entero n es $(d_{k-1}d_{k-2}\cdots d_0)_2$ donde

$$n = d_{k-1}2^{k-1} + d_{k-2}2^{k-2} + \cdots + d_0 2^0, \ d_i \in \{0,1\}$$

EJEMPLO A.10 $2^{10} = 1024 = (10000000000)_2$

Si $2^{k-1} \leq n \leq 2^k$ entonces n tiene k dígitos en base $b = 2$.

EJEMPLO A.11 $2^{10} \leq 2^{10} \leq 2^{11}$, así $n = 2^{10}$ tiene 11 dígitos en base 2.

$2^7 = 128 \leq 201 \leq 256 = 2^8$, así $n = 201$ tiene 8 dígitos en base 2. En efecto, $201 = (11001001)_2$

El número k de dígitos, en base $b = 2$, de un número $n \neq 0$ se puede calcular con la fórmula

$$k = [\log_2 |n|] + 1 = \left[\frac{\ln |n|}{\ln(2)}\right] + 1$$

EJEMPLO A.12 Si $n = 2^{10}$ entonces $k = \lceil \log_2(2^{10}) \rceil + 1 = 10 + 1 = 11$

Si $n = 201$ entonces $k = [\log_2(201)] + 1 = [7.65105...] + 1 = 8$

Recordemos que acostumbra usar "$\lg n$" en vez de $\log_2(n)$. En términos de "$O-$grande", el número de bits de n es $[\lg(n)] + 1 = O(\lg n) = O(\ln n)$. Se acostumbra decir "el número de bits de n es $O(\ln n)$".

La sumar dos números de tamaño $\ln n$ requiere $O(\ln n)$ operaciones de bits y la multiplicación y la división $O(\ln n)^2$.

Complejidad polinomial en teoría de números

EJEMPLO A.13

- Supongamos que un número n se representa con β bits, es decir, $\beta = \lfloor \lg n \rfloor + 1$ o $n = O(2^{\log n})$.

  Si un algoritmo que recibe un entero n, tiene complejidad $O(n)$ *en términos del número de operaciones aritméticas*, entonces, si por cada operación aritmética se hacen $O(\ln n)^2$ operaciones de bits, el algoritmo tiene complejidad

$$\begin{aligned} O(n) &= O(n(\ln n)^2) \\ &= O\left(2^{\ln n}(\ln n)^2\right) \end{aligned}$$

  en términos de operaciones de bits.

  Es decir, un algoritmo con complejidad polinomial en términos del número de operaciones aritméticas, tiene complejidad exponencial en términos de operaciones de bit.

- El algoritmo de Euclides para calcular MCD(a,b) con $a < b$ requiere $O(\ln a)$ operaciones aritméticas (pues un teorema de Lamé (1844) establece que el número de

divisiones necesarias para calcular el MCD(a,b) es a lo sumo cinco veces el número de dígitos decimales de a, es decir $O(\log_{10} a)$). Esto corresponde a $O(\ln a)^3$ en términos de operaciones de bits (asumiendo como antes que divisiones y multiplicaciones necesitan $O(\ln n)^2$ operaciones de bit).

Apéndice B
Implementaciones en VBA para Excel.

B.1 Introducción.

Para hacer las implementaciones en VBA para Excel necesitamos Xnumbers, un complemento (gratuito) para VBA y VB. Xnumbers nos permite manejar números grandes de hasta 200 dígitos. XNumbers se puede obtener en

`http://digilander.libero.it/foxes/SoftwareDownload.htm`

Aquí usamos un "dll", `Xnumbers.dll`. Para usarlo y hacer el cuaderno Excel portable, debemos proceder como sigue,

1. Ponemos `Xnumbers.dll` en al misma carpeta del cuaderno Excel,

2. En el editor de Visual Basic, hacemos una referencia (Herramientas-Referencias-Examinar). Con esto ya podemos usar las funciones de este paquete.

*

3. Para hacer el cuaderno portable (para que funcione en cualquier PC) insertamos un módulo en `ThisWorkBook` y pegamos el código

```
Option Explicit

Sub Installation_Procedure()
' Installation Procedure
' v.  6/10/2004
Dim myTitle As String
    myTitle = "XNUMBERS"
    'Activate the error handler
    On Error Resume Next

    'Check if Excel allows to make changes to VBA project
    Excel_Security_Check
    If Err <> 0 Then GoTo Error_handler

    'Remove old reference to this VBA project (if any)
    VBA_Link_Remove "DLLXnumbers"

    'Add the reference to this VBA project
    VBA_Link_Add "xnumbers.dll"
    If Err <> 0 Then GoTo Error_handler

    'Check if the ActiveX works
    ActiveX_test "xnumbers.dll"
    If Err <> 0 Then
        Err.Clear
        'The activeX may be to register. Do it.
        DLL_Register "xnumbers.dll"
        If Err <> 0 Then GoTo Error_handler
        MsgBox "Xnumbers.dll registering...", vbInformation
        'Repeat the check
        ActiveX_test "xnumbers.dll"
        If Err <> 0 Then GoTo Error_handler
    End If

Exit Sub
```

```
Error_handler:
    'Something has gone wrong. Show a message
    MsgBox Err.Description, vbCritical, myTitle
    Exit Sub
End Sub

Sub VBA_Link_Add(DLLname)
'Links a DLL library to an XLA project
Dim LibFile, Msg
    LibFile = ThisWorkbook.Path & "\" & DLLname
    If Dir(LibFile) <> "" Then
        ThisWorkbook.VBProject.References.AddFromFile LibFile
    Else
        Msg = "Unable to find " & LibFile
        Err.Raise vbObjectError + 513, , Msg
    End If
End Sub

Sub VBA_Link_Remove(FileLinked)
'Removes the Links of a DLL library from a XLA project
On Error Resume Next
    With ThisWorkbook.VBProject
        .References.Remove .References(FileLinked)
    End With
End Sub

Sub Excel_Security_Check()
'Shows a warning if Excel not allowes to change a XLA project
Dim Msg As String, myTitle As String
myTitle = "XNUMBERS addin"
If Excel_VBA_Protection Then
    Msg = "Your Excel security restriction does not allow to install this addin."
    Err.Raise vbObjectError + 513, , Msg
End If
End Sub

Private Function Excel_VBA_Protection() As Boolean
' Checks if Excel has the VBA protection. Returns true/false
Dim tmp
```

```
On Error Resume Next
tmp = ThisWorkbook.VBProject.Name
If Err <> 0 Then
    Excel_VBA_Protection = True
Else
    Excel_VBA_Protection = False
End If
End Function

Sub ActiveX_test(ActiveXname)
' Checks if the Xnumbers ActiveX works
Dim Msg, Xnum As New Xnumbers
On Error GoTo Error_handler
    With Xnum

    End With
Exit Sub
Error_handler:
    Msg = Err.Description & " <" & ActiveXname & ">"
    Err.Raise vbObjectError + 513, , Msg
End Sub

Private Sub DLL_Register(DLLname, Optional UnReg = False)
'Tries to register the Xnumbers ActiveX
Dim DLLfile, Msg, Save_path, cmd_line
    Save_path = CurDir
    Path_Change ThisWorkbook.Path
    If Dir(DLLname) <> "" Then
    If UnReg = False Then
        cmd_line = "REGSVR32 /s " + DLLname     'register silent
    Else
        cmd_line = "REGSVR32 /u /s " + DLLname  'unregister silent
    End If
        Shell cmd_line
    Else
        Msg = "Unable to find " & DLLfile
        Err.Raise vbObjectError + 513, , Msg
    End If
    Path_Change Save_path
```

```
End Sub

Sub Path_Change(myPath)
'change global path (drive+path)
Dim myDrive
    myDrive = Left(myPath, 1)
    ChDrive myDrive
    ChDir myPath
End Sub

'-------- test routines  ----------------
Sub DLL_Register_test()
    DLL_Register "xnumbers.dll"
End Sub

Sub DLL_UnRegister_test()
    DLL_Register "xnumbers.dll", True
End Sub

Sub VBA_Link_Remove_test()
    VBA_Link_Remove "DLLXnumbers"
End Sub
'----------------------------------------
```

Adicionalmente, deberá permitir macros (nivel de protección bajo) y en el menú “Herramientas - Opciones - Seguridad - Seguridad de Macros - Fuentes de Confianza” deberá habilitar la opción “Confiar en el acceso a proyectos Visual Basic”.

Si usamos Excel, debemos tener el cuidado de leer e imprimir estos números en celdas con formato de texto. Esta propiedad puede ser establecida desde el mismo código VBA.

B.2 Algoritmos Básicos.

Para los algoritmos de este trabajo, necesitamos el MCD, cálculo de residuos, potencias módulo m e inversos multiplicativos en $\mathbb{Z}_m$. El complemento XNumbers para VB y para VBA para Excel, viene con `gcd` y `xPowMod`. El cálculo de residuos módulo m se puede

hacer con `xPowMod`. Los inversos requieren implementar el algoritmo extendido de Euclides pues `xPowMod` no permite potencias negativas.

B.2.1 Cálculo de inversos multiplicativos en $\mathbb{Z}_m$

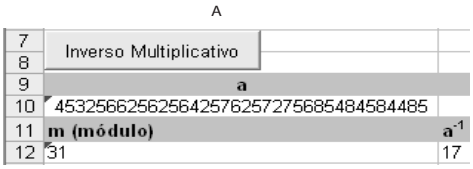
Sea $a \in \mathbb{Z}_m$. Si a y m son primos relativos, a^{-1} existe. En este caso, existen s,t tal que $sa+tm=1$. Usualmente s,t se calculam usando el algoritmo extendido de Euclides. Luego $a^{-1} = s \mod m$. El algoritmo es como sigue,

Algoritmo B.1: Inverso Multiplicativo mod m.

Entrada: $a \in \mathbb{Z}_m$
Resultado: $a^{-1} \mod m$, si existe.

```
1 Calcular x,t tal que xa+tm = MCD(a,m);
2 if MCD(a,m) > 1 then
3 |   a^{-1} mod m no existe
4 else
5 |   return s mod m
```

Para hacer una implementación en Excel, hacemos un cuaderno con el número a en la celda `A10`. Imprimos en la celda `B12`. El módulo lo leemos en la celda `A12`.



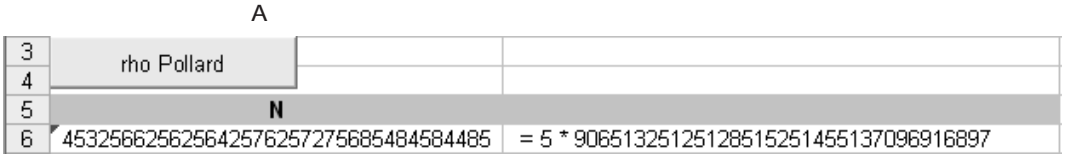

```
'BOTON
Private Sub CommandButton2_Click()
Call invMult
End Sub

'Necesitamos una funci\'on signo
Function MPSgn(x)
Dim MP As New Xnumbers
Dim salida
salida = 1
```

```
 If MP.xComp(x) = 1 Then
 salida = 1
  Else: salida = -1
 End If
 MPSgn = salida
End Function


Sub invMult()
Dim MP As New Xnumbers
Dim a, m, c, c1, d, d1, d2, q, r, x, t
MP.DigitsMax = 100
'Entran a y m, sale a^-1 mod m
a  = Cells(10, 1)
m  = Cells(12, 1)
'algoritmo extendido de Euclides
c  = MP.xAbs(a)
d  = MP.xAbs(m)
c1 = "1"
d1 = "0"
c2 = "0"
d2 = "1"

While MP.xComp(d) <> 0
q  = MP.xDivInt(c, d)
r  = MP.xSub(c, MP.xMult(q, d))
r1 = MP.xSub(c1, MP.xMult(q, d1))
r2 = MP.xSub(c2, MP.xMult(q, d2))
c  = d
c1 = d1
c2 = d2
d  = r
d1 = r1
d2 = r2
Wend
x = MP.xDivInt(c1, MPSgn(a) * MPSgn(c))
Cells(12, 2).NumberFormat = "@" 'pasar a formato texto
If mcd > 1 Then
```

```
      Cells(12, 2) = "Inverso no existe"
Else: Cells(12, 2) = MP.xPowMod(x, 1, m) 'Escribe el n\'umero
End If
Set MP = Nothing 'destruir el objeto.
End Sub
```

B.2.2 Algoritmo rho de Pollard.

El número N queremos factorizar, lo leemos en la celda (en formato texto) `A6`. La factorización la imprimimos en la celda `B6`



La subrutina VBA es

```
'BOTON
Private Sub CommandButton1_Click()
Call rhoPollard
End Sub



Sub rhoPollard()
Dim MP As New Xnumbers
Dim salir As Boolean
Dim n, nc, i, k, xj, x0, g
MP.DigitsMax = 100
n = Cells(6, 1)
k = 0
x0 = 2
xi = x0
salir = False
While salir = False
```

```
    i = 2 ^ k - 1
    For j = i + 1 To 2 * i + 1
        xj = MP.xPowMod(MP.xSub(MP.xPow(x0, 2), 1), 1, n) 'x^2-1
        If MP.xComp(xi, xj) = 0 Then
            salir = True
            Cells(6, 2) = "  El m\'etodo fall\'o, cambie f o x0"
        Exit For
        End If
        g = MP.xAbs(MP.xGCD(MP.xSub(xi, xj), n)) 'MCD(xi-xj,n)
        If MP.xComp(1, g) = -1 And MP.xComp(g, n) = -1 Then
            salir = True
            Cells(6, 2).NumberFormat = "@"  'los XNumbers son texto
            Cells(6, 2) = "  = " + g + " * " + MP.xDivInt(n, g)
            Exit For
        End If
        x0 = xj
    Next j
    xi = xj
    k = k + 1
Wend
Set MP = Nothing
End Sub
```

Bibliografía

[1] M. Kac. *Statistical Independence in Probability, Analysis and Number Theory.* Wiley, New York, 1959.

[2] N. Koblitz *A course in number theory and cryptography.* 2ed., Springer,1994.

[3] G.H. Hardy, J.E. Littlewood. *An Introduction to Theory of Numbers.* Oxford Univ. Press. 1938.

[4] R. Brent. "An Improved Monte Carlo Factorization Algorithm." BIT 20 (1980), 176-184. (`http://wwwmaths.anu.edu.au/~brent/pub/pubsall.html`).

[5] R. Brent, J. M. Pollard. "Factorization of the Eighth Fermat Number." Mathematics of Computation, vol 36, n 154 (1981), 627-630. (`http://wwwmaths.anu.edu.au/~brent/pub/pubsall.html`).

[6] Harold M. Edwards. *Riemann's Zeta Function.* Dover Publications Inc. 2001.

[7] P.L. Chebyshev. “The Theory of Probability”. Translated by Oscar Sheynin (www.sheynin.de) 2004. Versión en internet: `http://www.sheynin.de/download/4_Chebyschev.pdf`. Consultada Diciembre 16, 2006.

[8] Lindsay N. Childs. *A Concrete Introduction to Higher Algebra*. Springer-Verlag New York, 1995.

[9] Hans Riesel. *Prime Numbers and Computer Methods for Factorization.* Springer; 2 edition. 1994.

[10] K.O. Geddes, S.R. Czapor, G.Labahn. *Algorithms for Computer Algebra.* Kluwer Academic Publishers. 1992.

[11] J. Stopple. *A Primer of Analytic Number Theory. From Pythagoras to Riemann.* Cambridge. 2003.

[12] RSA, Inc. `http://www.rsasecurity.com/`. Consultada Noviembre 11, 2006.

[13] Raymond Séroul, *Programming for Mathematicians.* Springer, 2000.

[14] ArjenK. Lenstra. “Integer Factoring”. `http://modular.fas.harvard.edu/edu/Fall2001/124/misc/` Consultada: Octubre, 2006.

[15] P. Montgomery. “Speeding the Pollard and Elliptic Curve Method”. Mathematics of Computation. Vol 48, Issue 177. Jan 1987. 243-264. `http://modular.fas.harvard.edu/edu/Fall2001/124/misc/` Consultada: Octubre, 2006.

[16] Joachim von zur Gathen, Jürgen Gerhard. “*Modern Computer Algebra*”. Cambridge University Press, 2003.

[17] Maurice Mignotte. “*Mathematics for Computer Algebra*”. Springer,1992.

[18] A. Menezes, P. van Oorschot, S. Vanstone. *Handbook of Applied Cryptography.* Vanstone, CRC Press, 1996. `(www.cacr.math.uwaterloo.ca/hac)`

[19] W.Gautschi. *Numerical Analysis. An Introduction.* Birkhäuser, 1997.

[20] J.Stopple. *A primer of Analytic Number Theory.* Cambridge, 2003.

[21] G. Tenenbaum. *Introduction to Analytic and Probabilistic Number Theory*. Cambridge Studies in Advanced Mathematics.1995.

[22] S. Y. Yan. *Number Theory for Computing.* 2nd edition. Springer. 2001.