Epsilón - *Revista de Educación Matemática* *2010, Vol. 27(2), nº 75, pp. 27-39*

El método de Monte Carlo para el estudio de la primalidad de números

Francisco Javier Martínez López
IES Celia Viñas
Sergio Martínez Puertas, Amalia Gallegos Ruiz y José Ramón Ferrón de Haro
Universidad de Almería

INTRODUCCIÓN HISTÓRICA

El naturalista francés Georges-Louis Leclerc de Buffon (1707-1786) fue el primer científico destacado que intentó discutir públicamente las leyes de la evolución tal como dejó reflejado en su voluminosa obra *"Historia natural"*, en 44 tomos. Dado que todo se estropea con el tiempo, Buffon, quien había tenido el gran acierto de postular el cambio de las especies a lo largo del tiempo, cometió el error de considerar que la evolución era un simple fenómeno degenerativo: los monos serían humanos degenerados, los asnos caballos degenerados y así sucesivamente.

Buffon ocupa un lugar destacado en la historia de las ciencias, más concretamente en la de las Matemáticas, ya que fue el precursor del método de Monte Carlo, con el que los matemáticos usan el azar para conseguir informaciones muy difíciles de obtener mediante otros procedimientos y que se aplican en muy diversos problemas, incluido el de diseño de nuevos y complejos fármacos. ***El método de Monte Carlo fue bautizado así por su clara analogía con los juegos de ruleta de los casinos, el más célebre de los cuales es el de Monte Carlo.***

Otros precursores del método fueron Kurt Otto Friedrichs, Richard Courant o Guillermo Thomson Kelvin.

INTRODUCCIÓN AL CONCEPTO MATEMÁTICO

Bajo el nombre de "Método de Monte Carlo" o "Simulación Monte Carlo" se agrupan una serie de procedimientos que analizan distribuciones de variables aleatorias usando simulación de números aleatorios.

El Método de Monte Carlo da solución a una gran variedad de problemas matemáticos, haciendo experimentos con muestreos estadísticos en una compu-

tadora. El método es aplicable a cualquier tipo de problema, ya sea estocástico o determinístico.

Generalmente, en estadística los modelos aleatorios se usan para simular fenómenos que poseen algún componente aleatorio. Pero en el método de Monte Carlo, por otro lado, el objeto de la investigación es el objeto en sí mismo, un suceso aleatorio o pseudo-aleatorio se usa para estudiar el modelo.

A veces, la aplicación del método de Monte Carlo se usa para analizar problemas que no tienen un componente aleatorio explícito; en estos casos, un parámetro determinista del problema se expresa como una distribución aleatoria y se simula dicha distribución.

El uso real de los métodos de Monte Carlo como una herramienta de investigación, viene del trabajo de la bomba atómica, durante la Segunda Guerra Mundial. Este trabajo involucraba la simulación directa de problemas probabilísticos de hidrodinámica, concernientes a la difusión de neutrones aleatorios en material de fusión.

COMPROBACIÓN DE PRIMALIDAD. ESTUDIO TEÓRICO

Los algoritmos de Monte Carlo cometen ocasionalmente un error, pero encuentran la solución correcta con una probabilidad alta sea cual sea el caso considerado. Según Campos (1999), "e*sto es mejor que decir que funciona bien la mayoría de las veces"*, fallando tan sólo de vez en cuando en algunos casos especiales: no debe haber ningún caso en el cual la probabilidad de error sea elevada. Sin embargo, no suele darse ningún aviso cuando el algoritmo comete un error.

Posiblemente el algoritmo más famoso de Monte Carlo sea el que decide si un entero impar dado es o no primo. No se conoce ningún algoritmo para resolver este problema con certeza en un tiempo razonable cuando el número que hay que estudiar no tiene más que unos pocos centenares de dígitos decimales.

La historia de la comprobación probabilista de *primalidad* tiene sus raíces en *Pierre de Fermat*, el padre de la teoría de los números moderna. En 1640 enunció el teorema siguiente, que a veces se denomina teorema menor de Fermat.

> ***Sea n un primo. Entonces,***
>
> $$a^{n-1} \bmod n = 1$$
>
> ***para cualquier entero tal que*** $1 \leq a \leq n-1$

Podemos ver un ejemplo muy didáctico (Campos, 1999):

"Sean n=7 y a=5. Se tiene que es:

$$a^{n-1} = 5^6 = 15625 = 2232 \times 7 + 1$$

y, por tanto, tenemos que $a^{n-1} \bmod n = 1$".

Considérese ahora la versión contrapositiva del teorema de Fermat (enunciado contra recíproco del mismo teorema):

"*Si a y n son enteros tales que* $1 \leq a \leq n-1$, *y si* $a^{n-1} \bmod n \neq 1$, *entonces n no es primo*".

Una anécdota sobre Fermat y su teorema

"*En su búsqueda de una fórmula que sólo produjese números primos, formuló la hipótesis de que* $F_n = 2^{2^n} + 1$ *es primo para todo n. Lo comprobó para:* $F_0 = 3$, $F_1 = 5$, $F_2 = 17$, $F_3 = 257$, $F_4 = 65537$. *Pero no pudo comprobarlo para* $F_5 = 4294967297$. *Fue Euler, casi cien años después, quien factorizó ese número:*

$$F_5 = 641 \times 6700417$$" (Campos, 1999)

probando así la falsedad de la conjetura de Fermat.

Utilización del pequeño teorema de Fermat para comprobar la primalidad

En el caso de F_5, a Fermat le hubiera bastado con ver que

"$\exists\alpha$: $1 \leq \alpha \leq F_5 - 1$ *tal que* $a^{F_5-1} \bmod F_5 \neq 1$ $(\alpha=3)$" (Campos, 1999)

Esto sugiere el siguiente algoritmo probabilista para la comprobación de *primalidad*. Suponemos que $n \geq 2$:

```
public static boolean Fermat(long n)
{
        long a=rand.randomInt(1,(n-1));
        if( (a^(n-1) mod n) == 1)
                return true;
        else
                return false;
}
```
[1]

Visto lo anterior, sabemos que *n* es compuesto cuando una llamada a *Fermat(n)* devuelva el valor falso por el teorema menor de Fermat.

1. Implementación en pseudocódigo adaptado a JAVA (obtenido en Campos, 1999).

¿Qué se puede decir, sin embargo, si *Fermat(n)* devuelve el valor verdadero? Para concluir que *n* es primo, necesitaríamos el recíproco y el contrapositivo del teorema de Fermat. Desafortunadamente, esto no es cierto.

Un entero *a* tal que $2 \leq a \leq n-2$ y que $a^{n-1} \bmod n = 1$ se denomina *testigo falso de primalidad* para *n,* si, de hecho, *n* es compuesto. Por esto, la modificación en el algoritmo sería la de elegir *a* entre 2 y *n*–2, luego el algoritmo sólo fallaría para números no primos cuando se elija un testigo falso de *primalidad*. Pero hay una parte buena y otra mala, la buena es que el número de testigos falsos de *primalidad* son pocos. De hecho, la probabilidad media de error del algoritmo sobre los números impares no primos menores que 1000 es menor que 0.033 y es todavía menor para números mayores que 1000.

La mala noticia es que hay números no primos que admiten muchos falsos testigos de *primalidad*. Un caso convincente es el que plantea un numero de 15 dígitos: en Campos (1999) vemos que "*Fermat(651693055693681) devuelve verdadero con una probabilidad mayor que 99.9965% a pesar de que no es primo"*. Por tanto, puede demostrarse que el algoritmo de Fermat no es p-correcto para ningún $p>0$, luego la "*probabilidad de error no puede disminuirse mediante repeticiones independientes del algoritmo*".

Afortunadamente, una pequeña modificación de la comprobación de Fermat resuelve esta dificultad. "*Sea n un entero impar mayor que 4, y sean s y t enteros tales que* $n-1 = 2^s t$, *donde t es impar. Obsérvense que s > 0 puesto que n-1 es par*" (Campos, 1999).

Por tanto, siempre y cuando n sea impar y $2 \leq a \leq n-2$, una llamada a *pruebaB(a, n)* devuelve el valor verdadero si y solo si $a \in B(n)$.

Veamos cómo evaluar *B(n)*:

```
public static boolean funciónB(long n)
{
        long a=rand.randomInt(2,(n–2));
        return pruebaB(a, n);
}
```

Donde *pruebaB* sería:

```
public static boolean pruebaB(long a,long n)
{
        int s = 0;
        long t = n–1;

        do
        {
```

```
            s++;
            t = t/2;
        }
        while(t%2 != 1);

        long x = expomod(a, t, n);

        if(x==1 || x==n–1)
            return true;
        for(int i=0; i<(s–1); i++)
        {
            x = (x*x)%n;
            if(x == n–1)
                return true;

        }
        return false;
}
```
[2]

Veamos un ejemplo para que nos quede más claro:

Comprobamos que 158 pertenece a $B(289)$. Hacemos $s = 5$ y $t = 9$ porque $n - 1 = 288 = 2^5 \times 9$. Entonces calculamos

$$x = a^t \bmod n = 158^9 \bmod 289 = 131$$

No concluye aquí la prueba, porque 131 no es 1 ni n–1. A continuación, elevamos x al cuadrado (modulo n) hasta 4 veces (s–1 = 4) para ver si obtenemos 288.

$$x = a^{2t} \bmod n = 131^2 \bmod 289 = 110$$
$$x = a^{2t} \bmod n = 110^2 \bmod 289 = 251$$
$$x = a^{2t} \bmod n = 251^2 \bmod 289 = 288$$

Llegados aquí, nos debemos detener, porque hemos encontrado el resultado que estábamos esperando n-1 y, en conclusión, podemos decir que 158 existe en $B(289)$.

Una extensión del teorema de Fermat demuestra que $a \in B(n)$ para todo $2 \leq a \leq n-2$ cuando n es primo. Se dice que n es un *pseudoprimo* fuerte con base a y que

2. Implementación en pseudocódigo adaptado a JAVA (obtenido en Campos, 1999).

a es un testigo falso fuerte de *primalidad* para *n* siempre que $n > 4$ es un número impar compuesto y $a \in B(n)$.

Por suerte, el número de falsos testigos de *primalidad* en el sentido fuerte es mucho menor que el de falsos testigos de *primalidad*. Podremos concluir con el siguiente teorema (Campos, 1999):

"*Consideremos un número impar arbitrario n > 4:*

- ✓ *Si n es primo, entonces* $B(n) = \{a \mid 2 \le a \le n-2\}$
- ✓ *Si n es compuesto, entonces* $|B(n)| \le (n-9)/4$"

Por tanto, *pruebaB* devolverá verdadero en las condiciones anteriores y falso con probabilidad de más de ¾ cuando *n* es un entero impar compuesto mayor que 4 y *a* se selecciona aleatoriamente entre 2 y *n*–2. En otras palabras, el algoritmo de Monte Carlo es ¾ correcto para la comprobación de *primalidad*; se conoce con el nombre de comprobación de Miller-Rabin.

```
public static boolean MillRab(long n)
{
        int a = rand.randomInt(2,(n-2));
        return pruebaB(a, n);
}
```
[3]

En el siguiente algoritmo siempre devuelve la respuesta correcta cuando $n > 4$ es primo. Cuando $n > 4$ es un impar compuesto, cada llamada a *MillRab* tiene como mucho una probabilidad de ¼ de llegar a un testigo falso fuerte, devolviendo verdadero de manera errónea.

```
public static String MillerRabin(long n, int s)
{
        int cont =0;
        for(int j=0;j<s;j++)
                if(MillRab(n)==true)
                        cont++;
                if(cont==s)
                        return "N PUEDE SER PRIMO";
                else
                        return "N ES COMPUESTO";
}
```
[4]

3. Implementación en pseudocódigo adaptado a JAVA (obtenido en Campos, 1999).
4. Implementación en pseudocódigo adaptado a JAVA (obtenido en Campos, 1999).

Dado que la única manera de que *MillerRabin* devuelva el valor verdadero en este caso es encontrar aleatoriamente k testigos falsos fuertes seguidos. Por tanto, *MillerRabin* es un algoritmo de Monte Carlo *"(1–4k) correcto"* para la *primalidad*.

El coste de la probabilidad de error ε es

$$"O(\log^3 n \log 1/\varepsilon)"$$

Por consiguiente, deducimos que tendrá menos error cuando el número es muy alto y, por el contrario, tendremos más probabilidades de error cuando el número es muy bajo.

Así pues, resumiendo, podemos decir, que el tiempo total necesario para decidir acerca de la *primalidad* de n con una probabilidad de error acotada por ε *es* de $O(\log^3 n \log 1/\varepsilon)$. Esto es, según Campos (1999), *"enteramente razonable"* en la práctica para números de mil dígitos y una probabilidad de error menor que 10^{-100}.

¿Puede un número ser probablemente primo?

Si se aplica diez veces la comprobación de Miller Rabin a algún entero impar n y se obtiene la respuesta verdadero en las diez ocasiones, se puede deducir que n será primo, salvo por la probabilidad de error dada. Pero esta afirmación carece de sentido pues un número es primo o no lo es, por tanto es mejor decir *"creo que es primo"*.

Amplificación de la ventaja estocástica

Los dos algoritmos de Monte Carlo estudiados hasta el momento tienen una propiedad útil: una de las respuestas posibles siempre es correcta cuando se obtiene. Uno garantiza que un número dado es compuesto, el otro que un producto matricial dado es incorrecto. Decimos que estos algoritmos están *sesgados*. Gracias a esta propiedad, es sencillo reducir arbitrariamente la probabilidad de error repitiendo el algoritmo un número apropiado de veces. La aparición de una sola respuesta "garantizada" basta para producir una certeza; un gran número de respuestas probabilistas idénticas, por otra parte, incrementa la confianza en que se haya obtenido la respuesta correcta. Esto se conoce con el nombre de *amplificación de la ventaja estocástica.*

Supongamos que disponemos de un algoritmo de Monte Carlo *no sesgado* cuya probabilidad de error sea no nula independientemente del caso que haya que resolver, y de la respuesta proporcionada. ¿Sigue siendo posible reducir arbitrariamente la probabilidad de error repitiendo el algoritmo? La respuesta es que depende de la probabilidad de error original. Por sencillez, vamos a concentrarnos en algoritmos que conciernen a la toma de decisiones. Considérese un algoritmo de Monte Carlo del cual lo único que se sabe es que es p-correcto. El primer co-

mentario evidente es que la amplificación de la ventaja estocástica es imposible a no ser que p > ½ porque siempre existe el (inútil) algoritmo ½-correcto:

```
función estúpida (x)
    Si tiramoneda = cara entonces devolver verdadero
        Sino devolver falso
```

cuya "ventaja" estocástica no se puede amplificar. Siempre y cuando p ≥ ½, se define la *ventaja* de un algoritmo de Monte Carlo p-correcto como p – ½. Todo algoritmo de Monte Carlo cuya ventaja sea positiva se puede transformar en otro cuya probabilidad de error sea tan pequeña como deseemos. Comenzaremos por un ejemplo.

Sea MC un algoritmo de Monte Carlo no sesgado ¾-correcto para resolver algún problema de decisión. Considérese el siguiente algoritmo, que llama tres veces a *MC(x)* y devuelve la respuesta más frecuente:

```
función MC3(x)
    t ← MC(x); u←MC(x); v←MC(x);
    si t = u o t = v entonces devolver t
        sino devolver u
```

¿Cuál es la probabilidad de error de MC3? Sean R y W las respuestas correcta e incorrecta respectivamente. Sabemos que t, u y v tienen una probabilidad mínima de ¾ de ser R, independientemente unas de otras. Supongamos por sencillez que esta probabilidad es exactamente de ¾, puesto que claramente el algoritmo MC3 sería todavía mejor si la probabilidad de error de MC fuera menor que ¼. Hay ocho resultados posibles para las tres llamadas a MC, cuyas probabilidades se resumen en la tabla siguiente:

t	*u*	*v*	*prob*	*MC3*
R	R	R	27/64	R
R	R	W	9/64	R
R	W	R	9/64	R
R	W	W	3/64	W
W	R	R	9/64	R
W	R	W	3/64	W
W	W	R	3/64	W
W	W	W	1/64	W

Sumamos las probabilidades asociadas a las filas 1, 2, 3 y 5 concluimos que MC3 es correcto con una probabilidad de 27/32, que es mejor que 84%.

Conclusión del estudio teórico

En esta primera parte hemos estado comentando ampliamente la evolución histórica del algoritmo de Monte Carlo que busca la *primalidad,* pasando por Fermat y terminando el Miller-Rabin, con este último "añadido" del MC. Por consiguiente, sólo nos resta mostrar el algoritmo más óptimo que hemos encontrado para la búsqueda de la *primalidad:*

```
public static boolean primoMR(int n, int t)
{
    boolean compuesto = true;
    int m=n-1; int s=0;
    int r,a,b;

    while ((m % 2) == 0)
    {
        s=s+1; m=m/2;// s = veces que 2 divide a n-1
    }

    r=m;
    int i=1;

    while (compuesto && (i<t))
    {
        a=(int)Math.floor(2+(Math.random()*(n-2)));
        //a=aleatorio 2..n-1
    if (mcd(a,n)!=1)
        i=t; //acabamos -> es compuesto

    else
    {
        b=(int)expRapida(a,r,n); // b = a^r mod n
        if ((b!=1) && (b!=-1) )
            for (int j=0; j<=s-1; j++)
                b=(int)expRapida(b,2,n);//b=b^2 mod n
        compuesto = (b!=1) && (b!=-1);
        // sigue sin ser primo
```

```
            }
            i++;
        }

        return !compuesto;
    }
```

ESTUDIO EXPERIMENTAL

Una vez llevada a cabo la implementación de los algoritmos de Miller-Rabin y de Fermat, nos dispusimos a realizar las pruebas con 50 números primos. Comenzamos desde números primos pequeños como el 13, 23 o 29, hasta acabar con números superiores a los 1000 millones. Dichos resultados se muestran en la tabla anexa.

Tras realizar estas pruebas con los 4 algoritmos, sacamos las siguientes conclusiones: Los métodos "primitivos" de Fermat y Miller-Rabin resultan ser bastante ineficientes. Como se refleja en la tabla, el método de Fermat ya comienza a fallar con el número 23. Mientras el de Miller-Rabin sólo da resultados correctos hasta el número 31.

Estos resultados explican porque fue necesaria la mejora de ambos algoritmos. Dicha mejora, explicada en los apartados anteriores, supuso un avance extraordinario en cuanto a eficiencia y veracidad en los resultados sobre la *primalidad* de los números primos. De pasar a ser métodos poco fiables y con gran porcentaje de error, se pasó a ser, especialmente el test de Miller-Rabin, el algoritmo con mayor porcentaje de acierto para números muy grandes.

Las pruebas llevadas a cabo con la implementación de los algoritmos detallada anteriormente nos muestran, que, tanto el método de 'Fermat bueno', como el método de 'Miller-Rabin bueno' aciertan en el 100% de los casos para números primos menores que 100.000.000.

Una vez sobrepasado ese límite, comienzan a apreciarse diferencias entre los métodos de 'Fermat bueno' y 'Miller-Rabin bueno'. Fermat comienza a perder eficiencia y se dan algunos fallos, dando como resultado números compuestos, aquellos que son primos (véase en la tabla el caso de 100.000.007, 100.000.037, 100.000.039). Por el contrario, Miller-Rabin sigue mostrando que estos números son primos. Tan sólo es capaz de "fallar" cuando el factor de seguridad es igual a 2. No obstante, lo normal para obtener resultados fiables, es que el factor de seguridad sea mayor, siendo en este caso Miller-Rabin muy eficiente.

Para darnos cuenta de cuándo empieza a dar resultados "falsos" el algoritmo de Miller-Rabin tenemos que llegar hasta el número primo 180.000.017. A partir de este número, deja de ser fiable el resultado de dicho algoritmo, siendo indiferente el valor que tome el factor de seguridad.

No obstante, se puede observar como el algoritmo de Miller-Rabin mejoró considerablemente los resultados obtenidos por Fermat, ya que casi dobló el número de primos que es capaz de "acertar" (de 100.000.000 a 180.000.000). También se explica la necesidad de que se llevara a cabo la mejora de los algoritmos primitivos, puesto que éstos no ofrecían resultados fiables.

Llevamos a cabo también la implementación de la *clase validacion* que mostrará los tiempos de ejecución de los 4 algoritmos. Prácticamente, en todos los casos dichos tiempos fueron nulos. Tan sólo cuando llegamos a números cercanos a 1000.000.000 y el factor de seguridad se elevaba hasta 1000, hallamos tiempos de ejecución apreciables. Pero incluso en estos casos, dichos tiempos no llegaban a los 400 milisegundos (330 ms. con el número 1000000447 y factor de seguridad 1000).

CONCLUSIONES

Como conclusión podemos extraer, que Monte Carlo resulta ser un método muy válido para hallar la *primalidad* de números grandes. La probabilidad de error de dicho algoritmo es muy baja, y en base a las pruebas experimentales que hemos realizado, para números de 8 dígitos acierta en el 100% de las ocasiones y únicamente comienza a ser menos fiable con los números primos de 180 millones en adelante.

REFERENCIAS BIBLIOGRÁFICAS

Allen, M. (2000). *Estructuras de datos en java.* Madrid: Addison Wesley.

Brassard, G. y Bratley, P. (1997). *Fundamentos de Algoritmia.* Madrid: Prentice Hall.

Campos, J. (1999). Algoritmos probabilistas. Archivo web, en http://www.lsi.upc.es/~iea/transpas/8_probabilistas/

De Berg, M., Van Kreveld, M., Overmaks, M. y Schwarzkopf, O. (2000). *Computacional Geometry. Algorithms and Applications.* 2nd edition. New York: Springer-Verlag.

Facultad de Informática. U.P.M. (2003). *Algorítmica.* Madrid: U.P.M.

O´Rourke, J. (1994). *Computacional Geometry in C.* New York: Cambridge University Press.

http://leebyte.iespana.es/leebyte/Programacion/Metodologia/Esq%20Algoritmicos/4_dinamica/tsld034.htm

ANEXO: RESULTADOS OBTENIDOS EN LA FASE EXPERIMENTAL

Número	Factor	M-R bueno	Fermat bueno	Fermat	M-R
13	2	PRIMO	PRIMO	COMPUESTO	PRIMO
23	200	PRIMO	PRIMO	COMPUESTO	PRIMO
29	100	PRIMO	PRIMO	COMPUESTO	PRIMO
31	100	PRIMO	PRIMO	COMPUESTO	**COMPUESTO**
101	100	PRIMO	PRIMO	COMPUESTO	COMPUESTO
10007	700	PRIMO	PRIMO	COMPUESTO	COMPUESTO
100103	10	PRIMO	PRIMO	COMPUESTO	COMPUESTO
1001041	2	PRIMO	PRIMO	COMPUESTO	COMPUESTO
10010411	2	PRIMO	PRIMO	COMPUESTO	COMPUESTO
90010421	2	PRIMO	PRIMO	COMPUESTO	COMPUESTO
90010421	20	PRIMO	PRIMO	COMPUESTO	COMPUESTO
95010439	2	PRIMO	PRIMO	COMPUESTO	COMPUESTO
95010439	20	PRIMO	PRIMO	COMPUESTO	COMPUESTO
96010459	20	PRIMO	PRIMO	COMPUESTO	COMPUESTO
99010433	20	PRIMO	COMPUESTO	COMPUESTO	COMPUESTO
99010433	2	PRIMO	PRIMO	COMPUESTO	COMPUESTO
100000007	10	PRIMO	**COMPUESTO**	COMPUESTO	COMPUESTO
100000007	2	COMPUESTO	PRIMO	COMPUESTO	COMPUESTO
100000007	10	PRIMO	COMPUESTO	COMPUESTO	COMPUESTO
100000037	10	PRIMO	COMPUESTO	COMPUESTO	COMPUESTO
100000037	2	PRIMO	PRIMO	COMPUESTO	COMPUESTO
100000039	2	COMPUESTO	COMPUESTO	COMPUESTO	COMPUESTO
100000039	20	PRIMO	COMPUESTO	COMPUESTO	COMPUESTO

Número	Factor	M-R bueno	Fermat bueno	Fermat	M-R
120000031	10	PRIMO	COMPUESTO	COMPUESTO	COMPUESTO
130000061	1000	PRIMO	COMPUESTO	COMPUESTO	COMPUESTO
140000089	1000	PRIMO	COMPUESTO	COMPUESTO	COMPUESTO
160000103	1000	PRIMO	COMPUESTO	COMPUESTO	COMPUESTO
170000177	1000	PRIMO	COMPUESTO	COMPUESTO	COMPUESTO
180000017	**100**	**COMPUESTO**	COMPUESTO	COMPUESTO	COMPUESTO
199990421	100	COMPUESTO	COMPUESTO	COMPUESTO	COMPUESTO
399990391	100	COMPUESTO	COMPUESTO	COMPUESTO	COMPUESTO
599990383	100	COMPUESTO	COMPUESTO	COMPUESTO	COMPUESTO
799990351	100	COMPUESTO	COMPUESTO	COMPUESTO	COMPUESTO
999990347	100	COMPUESTO	COMPUESTO	COMPUESTO	COMPUESTO
999999323	100	COMPUESTO	COMPUESTO	COMPUESTO	COMPUESTO
999999929	10	COMPUESTO	COMPUESTO	COMPUESTO	COMPUESTO
999999929	1000	COMPUESTO	COMPUESTO	COMPUESTO	COMPUESTO
1000000007	100	COMPUESTO	COMPUESTO	COMPUESTO	COMPUESTO
1000000007	1000	COMPUESTO	COMPUESTO	COMPUESTO	COMPUESTO
1000000447	20	COMPUESTO	COMPUESTO	COMPUESTO	COMPUESTO
1000000447	90	COMPUESTO	COMPUESTO	COMPUESTO	COMPUESTO
1000000447	900	COMPUESTO	COMPUESTO	COMPUESTO	COMPUESTO
1000000447	2000	COMPUESTO	COMPUESTO	COMPUESTO	COMPUESTO