Criptografía, MAPLE y RSA

Autor: Carlos B.

Escuela Técnica Superior de Ingenieros de Telecomunicación.
Universidad Politécnica de Madrid.
Marzo de 1998

"Un analfabeto digital es un analfabeto"

Nicholas Negroponte

Resumen

En este trabajo se utiliza el sistema de computación matemática *MAPLE V R4* como soporte didáctico para introducir la criptografía y especialmente el algoritmo de clave pública RSA. Primeramente se introducen los conceptos fundamentales de criptografía y criptosistemas, haciendo especial hincapié en el esquema RSA, para más tarde tratar cómo sería la implementación de RSA con ordenador utilizando *MAPLE*.
Los algoritmos que se presentan en la tercera parte no están optimizados para una ejecución veloz, y por lo tanto sólo deben ser utilizados para codificar pequeñas cantidades de datos. Sin embargo, con ligeras modificaciones se puede mejorar su velocidad de ejecución de manera notable.

Agradecimientos

Querría expresar mi agradecimiento a Francisco Ballesteros Olmo, por haberme dirigido esta monografía, tanto en el enfoque como en la discusión de los pormenores de la implementación de los algoritmos. Deseo además agradecer a David Rodríguez Ibeas sus innumerables comentarios que suscitaron correcciones en los puntos de vista adoptados para la implementación del código de los algoritmos. También querría hacerle patente mi agradecimiento expreso por construir y mostrarme, de un día para otro, una función de conversión de texto a número entero y por explicarme la codificación en base 256 de los caracteres ISO-Latin 1 con un algoritmo de su invención. Su interés y dedicación sobrepasaron los límites de la amistad. La función "val" por él diseñada no está incluida debido a que MAPLE dispone de la función parse, que es idéntica a la por él realizada, pero los algoritmos "file2declist" y "dec2word" están basados en sus ideas. A los dos, muchas gracias por el tiempo y el esfuerzo que me han prestado.

Conceptos de criptografía

Introducción

La *criptografía* es la ciencia que se ocupa de la escritura secreta. Desde la Antigüedad los

hombres han necesitado cifrar sus mensajes para que pudieran ser enviados por canales inseguros sin ser interceptados por terceras personas. Hay a lo largo de la Historia muchos ejemplos de sistemas criptográficos utilizados por personas relevantes: *Cesar, Vigenère, Beaufort*,... pero todos ellos son fáciles de "romper" aun sin saber la clave (mediante, por ejemplo, sustituciones estadísticas o simple "fuerza bruta").
La criptografía moderna, sin embargo, se encarga de la construcción de esquemas eficientes para los cuales es inviable (no imposible) recuperar el texto original a partir del texto cifrado.
El lector puede preguntarse ahora qué diferencia hay entre *inviable* e *imposible*. Para explicar la diferencia entre ambos vocablos necesitamos primero definir qué es una función de un sólo sentido.

Definición. *Una función* $f; [0, 1] \to [0, 1]$ *se llama unidireccional si y sólo si:*
(i) existe un algoritmo eficiente que para una entrada x produzca una salida f(x).
(ii) dada f(x), donde x se ha seleccionado uniformemente, no es viable encontrar, con probabilidad apreciable, una preimagen de f(x), es decir, si un algoritmo intenta encontrar una preimagen de f(x) en tiempo finito la probabilidad de que lo encuentre es despreciable.

A la luz de la anterior definición vemos la diferencia entre inviable e imposible. Lo primero significa que hay una probabilidad muy pequeña pero existente de encontrar una preimagen sin conocer la función, mientras que lo segundo quiere decir que no existe probabilidad alguna. Los algoritmos utilizados para cifrar siempre tienen función preimagen (si no sería imposible obtener de vuelta el texto original), lo que sucede es que encontrarla por azar es muy difícil.
Hemos utilizado el término "eficiente", pero no nos ocuparemos de definirlo formalmente. Podríamos, simplemente, decir que es un algoritmo que funciona de manera correcta.
Para codificar un texto se utilizan los sistemas criptográficos, como veremos a continuación. Con ellos se establece una aplicación entre un espacio inicial, donde se encuentra el mensaje sin codificar, y otro final donde está el mensaje codificado.

Sistemas criptográficos

Los componentes de un sistema criptográfico o criptosistema son cinco:
1. El espacio de caracteres del texto original M.
2. El espacio de caracteres del texto cifrado C.
3. El espacio de la clave de cifrado K.
4. Una familia de transformaciones de cifrado $E_k; M \to C$.
5. Una familia de transformaciones de descifrado $D_k; C \to M$.

Cada transformación de cifrado E_k está definida por un algoritmo de cifrado E, que es común al tipo de transformación de la familia, y una clave K que la distingue del resto de transformaciones. Lo mismo sucede para las transformaciones de descifrado D_k. Para una K dada debe cumplirse que D_k sea inversa de E_k, y por tanto que $D_k(E_k(M)) = M$ para cualquier texto original M.
Todo criptosistema ha de cumplir tres requisitos generales:
1. Las transformaciones de cifrado y descifrado han de ser eficientes para cualquier clave.
2. El sistema tiene que ser fácil de usar.
3. La seguridad del sistema sólo debe depender de lo bien guardada que esté la clave, y no de

lo bien preservado que se encuentre el algoritmo, que, antes bien, debe ser público.

A continuación presentaremos las características que han de tener los criptosistemas para garantizar tanto la invulnerabilidad como la autenticidad del emisor.

Invulnerabilidad del criptosistema.

Las características de invulnerabilidad consiguien que no sea posible determinar el contenido de un mensaje cifrado. Para ello:
1. Debe ser inviable computacionalmente para un criptoanalista (persona no autorizada que intenta averiguar el contenido del mensaje cifrado) determinar sistemáticamente la transformación de descifrado a partir de un texto cifrado interceptado C, aun si el texto original M correspondiente se conoce.
2. Debe ser inviable computacionalmente para un criptoanalista determinar sistemáticamente el texto original M a partir del texto cifrado C.

Autenticidad del emisor.

El criptoanalista no debe ser capaz de sustituir falso texto cifrado C' por texto cifrado C sin que el receptor autorizado lo detecte.
1. Debe ser computacionalmente inviable para un criptoanalista determinar sistemáticamente la transformación de cifrado E_k a partir de C, incluso si se conoce M.
2. Debe ser computacionalmente inviable para un criptoanalista hallar sistemáticamente C' tal que ? sea válido en M.

Simmons clasifica los criptosistemas en dos tipos: de una clave y de dos claves.
En los sistemas de una sola clave (simétricos) las claves de cifrado y descifrado son iguales (o se determinan fácilmente una a partir de la otra). Por lo tanto ambas han de ser conservadas en secreto.
En los sistemas asimétricos o de dos claves D_k y E_k son claves diferentes que no se pueden deducir la una de la otra, y por lo tanto se puede hacer pública una sin comprometer la seguridad de la otra.

Sistemas de clave pública

El concepto de criptosistemas de clave pública fue introducido en 1976 por *Diffie* y *Hellman* . Con este nuevo método cada usuario tiene un par de claves. Una la mantiene secreta (privada), que es la que se corresponde con D_k mientras que la otra la publica (la E_k).
Veamos a continuación cómo se puede transmitir un mensaje cifrado sin comprometer su integridad por un canal inseguro. Supongamos, por ejemplo, que Nuria quiere enviar un mensaje a su amigo David y no desea que nadie más lo pueda leer. Nuria posee una clave pública E_{Nuria} y una privada D_{Nuria}, y David tiene las correspondientes E_{David} y D_{David}. Para enviar un mensaje M, Nuria lo encripta con la clave pública de David, E_{David} , y se lo envía por el canal inseguro. De esta manera sólo David lo podrá leer con su clave privada D_{David}, ya que sólo él conoce esa clave. Se cumplirá la relación $D_{David}(E_{David}(M)) = M$, y el texto quedará inteligible.

No obstante este esquema no garantiza que el emisor del mensaje M sea Nuria, pues cualquiera podría encriptar el mensaje con la clave pública de David y enviárselo bajo un nombre falso. De hecho, Laura, que se lleva muy mal con Nuria y quiere destruir su amistad con David tiene intención de hacer pasar un falso mensaje M´ como si fuera de Nuria. ¿Es posible evitar esta situación con nuestro sistema de clave pública? La respuesta es afirmativa, pero sólo con un tipo de criptosistema de clave pública que estudiaremos en breve, el RSA. Veamos, pues, cuál sería el esquema que se debería seguir. Si prescindimos por un momento de la necesidad de privacidad, Nuria podría mandar el mensaje M a David encriptado con su clave privada D_{Nuria}. Cuando éste le llegue a David, lo podrá desencriptar con la clave pública E_{Nuria} y comprobar por tanto su autenticidad. Esto es posible porque existe conmutatividad entre las claves D y E, es decir, se cumple la relación $D(E(M)) = E(D(M))$ (donde los subíndices se han omitido por motivos de claridad). Pero esto no garantiza la privacidad del mensaje. Sin embargo, para satisfacer esta nueva condición no hay más que encriptarlo posteriormente con la clave pública de David, enviando $E_{David}(D_{Nuria}(M))$. Sin embargo, habrá que tener la precaución de que los espacios de cifrado y de texto original sean iguales entre sí. Por ejemplo, los dos espacios podrían ser el alfabeto castellano.

Complejidad

La complejidad matemática nos da una base para analizar los requisitos que han de satisfacer las técnicas criptoanalíticas, y para estudiar las dificultades inherentes a los sistemas de cifrado.
La bondad de un criptosistema está determinada por la complejidad de los algoritmos usados para descifrarlo. Dicha complejidad viene determinada por el tiempo que gastan y el espacio que ocupan. Es muy frecuente expresar la complejidad como orden de magnitud, en la forma O(g(n)), donde "O" es la "O mayúscula" de *Landau* del Cálculo, que tiene un significado matemático preciso: si f(n)=O(g(n)) entonces $f(n) \le c\,|g(n)|$ para cualquier $n_0 \le n$, y cualquier constante "c". Se dice que un algoritmo es de "orden polinómico" si su tiempo de ejecución es $T = O(n^t)$. Se dice que es "constante" si t=0, "lineal" si t=1, "cuadrático" si t=2, etcétera. Es "exponencial" cuando $T = O(t^{h(n)})$ para alguna constante "t" y un polinomio "h(n)".

En su trabajo de 1976, Diffie y Hellman sugirieron aplicar la complejidad matemática para diseñar los algoritmos de encriptación. Señalaron que aquellos problemas *NP-completos* (solubles en tiempo polinómico de manera no determinista y completos, es decir, aquellos problemas en los que se puede suponer una solución y determinar si es correcta o no en tiempo de orden polinómico, pero en los cuales suponer todas las posibles soluciones de manera exhaustiva requiere un tiempo de orden exponencial) eran excelentes candidatos para un algoritmo de encriptación.
Para ello se introduciría un problema de difícil tratamiento computacional que conllevase una función unidireccional. La información necesaria para utilizar la "puerta trasera" estaría en la clave de descifrado D_k. Sin ella, habría que resolver el problema de la manera tradicional, por "fuerza bruta" en un tiempo de orden exponencial.

Hay que señalar que la dificultad de resolver los problemas de complejidad en los que se basan todos los algoritmos de cifrado no está demostrada matemáticamente. Cabría la

posibilidad de que el día de mañana se descubriera un algoritmo para simplificar dicha complejidad, y se pasara de un problema de orden exponencial a otro de orden polinómico, con lo que toda la información codificada con cualquier algoritmo de los actualmente utilizados se vería comprometida. ¡Eso supondría que hasta la mismísima CIA tendría los trapos sucios al descubierto! Piensen en lo divertido (y peligroso) que podría llegar a ser...

RSA

RSA es un sistema de clave pública que sirve tanto para cifrado como para autentificación. Fue inventado en 1977 por *Ron Rivest, Adi Shamir* y *Leonard Adleman*, del *MIT* y funciona de la siguiente manera: se eligen dos números primos grandes "p" y "q" y se multiplican formando el producto $n = p\,q$. Como "p" y "q" son primos, sabemos que la función de Euler $\phi(n) = (p-1)\,(q-1)$. Se escoge un número "e" que sea menor que "n" y tal que "e" y $\phi(n)$ sean primos entre sí. Se elige otro número "d" tal que cumpla que $e\,d - 1$ sea divisible por $\phi(n)$. A los números "e" y "d" se les llama exponentes público y privado respectivamente. La clave pública será el par (e,n) y la privada el par (d,n).
Si se pudiera factorizar "n" en "p" y "q" se podría obtener la clave privada y la seguridad se vería comprometida. Sin embargo la eficiencia del esquema RSA se basa en la presunción de que la factorización es un problema complejo por tratarse de un problema NP-completo. Si no fuera así, se podría "romper" una clave RSA de manera muy sencilla.
Para encriptar con RSA procederemos de la siguiente manera:

Obtendremos el criptotexto $C = M^e \bmod n$, donde "M" es el mensaje (codificado numéricamente de manera apropiada), "e" es el exponente público de encriptación, "n" es el producto de "p" y "q" y "mod" denota la operación exponentación módulo "n".

Para desencriptar haremos $M := C^d \bmod n$.

Rivest, Shamir y *Adleman* recomendaron escoger "d" dentro del intervalo $[\max(p, q) + 1, n - 1]$. Si $\text{inv}(d, \phi(n))$ da "e" tal que $e < \log_2(n)$, entonces se ha de elegir un nuevo valor para "d" de forma que se asegure que todo el mensaje queda sometido, al menos, a una vuelta entera (reducción módulo "n").
RSA no se suele utilizar en la vida real tal como se ha descrito aquí, debido a su notoria lentitud (comprobar en la siguiente sección). Para ver qué es lo que sucede imaginemos que Nuria le quiere enviar un nuevo mensaje a David. Primeramente, lo que hará será encriptar el mensaje con DES, utilizando una clave aleatoria. Entonces buscará la clave pública RSA de David y la usará para encriptar la clave DES y enviársela junto con el mensaje encriptado. Al recibir el mensaje, David desencriptará la clave DES con su clave privada RSA y entonces usará esa clave para desencriptar el mensaje. Esto combina la velocidad de DES con la facilidad de manejo de claves de RSA.
¿Qué es la clave *DES*? *DES* significa *Data Enryption Standard* (estándar de encriptación de datos) y es un criptosistema simétrico, por lo que para ser usado en una comunicación tanto el emisor como el receptor han de conocer la clave de antemano. Utiliza bloques de 64 bits, y una clave de 56 bit durante la encriptación. Es muchísimo más rápido que RSA, razón por la cual se utiliza en protocolos de Internet y de comercio electrónico.

RSA: un ejemplo práctico

A continuación ejemplificaremos todo lo que anteriormente hemos dicho. Para nuestra demostración práctica utilizaremos un hipotético fichero que se encuentre en el mismo directorio

que este archivo, y que se llame EJEMPLO.TXT.

Primeramente, reiniciemos MAPLE.

```
> restart:
```

Definamos el procedimiento "getprime" que acepta un entero y devuelve un número primo aleatoriamente generado con el número de cifras indicado por el entero que se ha introducido. Utiliza la función de MAPLE "nextprime" que devuelve el número primo más cercano por encima al número que se le introduce. Hay que hacer notar que para saber si el número es primo se utiliza el test de Lucas. Este test no garantiza la primalidad, por lo que podría ser que dicho número no fuera primo. Sin embargo, esto sólo debilitaría la clave, no la invalidaría completamente.

```
> getprime:=proc(a) local pnumero,seed_alea,num_alea,Seed;
    if (op(0,a)<>integer) then
      ERROR(`Invalid parameter type. Must be an integer!`);
    fi;
    Seed:=readlib(randomize)():
    seed_alea:=rand(10^a):
    num_alea:=seed_alea():
    pnumero:=nextprime(num_alea):
    RETURN(pnumero):
  end:
```

A continuación se definen los algoritmos para obtener los exponentes de encriptación y desencriptacion "e" y "d".

La primera de las funciones, "decryptkey" acepta un entero que ha de ser la función de Euler $\phi(n)$ y devuelve "d". Como para calcular "e" necesitamos conocer "d" -o viceversa-, la función "encryptkey" requiere tanto $\phi(n)$ como "d".

```
> decryptkey:=proc(f) local mcd,r,d,semilla;
    if (op(0,f)<>integer) then
      ERROR(`Invalid parameter type. Must be an integer!`);
    fi;
    mcd:=0:
    r:=0:
    while (mcd<>1) do
      semilla:=rand(f):
      d:=semilla();
      mcd:=gcd(d,f);
    od:
    RETURN(d);
  end:
> encryptkey:=proc(d,f) local aux,enc,e;
    if (op(0,d)<>integer) or (op(0,f)<>integer) then
      ERROR(`Invalid parameter type. Must be an integer!`);
    fi;
    enc:=msolve(aux*d=1,f):
    e:=rhs(enc[1]);
    RETURN(e);
```

```
    end:
```

La exponentación necesaria para el cálculo del texto cifrado y para su posterior desciframiento requiere un algoritmo que acepte números grandes tanto de base y exponente como para el módulo. MAPLE provee la función "power" que será la que utilizaremos por ser más rápida que la que aquí presentamos. No obstante, por razones didácticas mostramos el algoritmo de exponentación rápida "fastexp", que acepta la base, el exponente y el módulo.

La forma más habitual de calcular la exponencial de $b = a^z$ como *a a a a .. z veces* no es adecuada con grandes números debido al enorme número de pasos de cálculo que habría que realizar. Así pues utilizamos este algoritmo que se basa en la descomposición en base 2.

Si ?, con z_i perteneciente a { 0, 1 }. Partiendo de que b=1, si $z_0 = 1$ hacemos $b = a$ 1, en caso contrario no modificamos "b". Calculamos $a_1 = a_0^2$. De nuevo, si $z_1 = 1$ multiplicamos "b" por a_1, y si no dejamos "b" inalterada. Calculamos $a_2 = a_1^2$ y repetimos el proceso. De esta manera, en la etapa k-1 tendremos $b = a^z$. Utilizando este procedimiento podemos tomar el módulo de la división de a_i por "n" y de "b" por "n" en cada paso, y reemplazar "a_i" y "b" respectivamente por sus restos. Con esta técnica evitaremos trabajar con números excesivamente grandes.

```
> fastexp := proc (a,z,n)  local a1,z1,x;
    if (op(0,a)<>integer) or (op(0,z)<>integer)
       or (op(0,n)<>integer) then
          ERROR(`Invalid parameter type. Must be an integer!`);
    fi;
    # return x=a^z mod n
    a1:=a;z1:=z;
    x:=1;
    while (z1<>0) do
      while ((z1 mod 2)=0) do
        z1:=iquo(z1,2);
        a1:=(a1*a1) mod n;
      od;
      z1:=z1-1;
      x:=(x*a1) mod n
    od;
    RETURN(x);
  end:
```

Cuando leemos un archivo con la función "readbytes" nos devuelve una lista de enteros. El problema está en que nosotros necesitamos un sólo entero cuya longitud no exceda la del número "n" producto de los dos primos. MAPLE no tiene muchas funciones de tratamiento de cadenas y por tanto resulta bastante farragoso implementar el código para obtener dicho número. Así pues, no se recomienda al lector poco interesado en los entresijos de la programación en MAPLE revisar el código de la función "file2declist". Esta función lee un archivo y devuelve, a través de la variable global LISTA, un array de enteros. Cada entero tiene de longitud la especificada por el segundo argumento. Además, de manera directa devuelve el número de elementos que conforma el array de enteros. Para codificar los bytes

dentro del número hemos optado por utilizar una codificación de cada valor de la tabla ASCII/ISO Latin-1 a la base 256. Con esto creamos una función biunívoca que nos permite relacionar los bytes individuales con el número final que se usa para la encriptación y viceversa.
Hay también que hacer notar la razón por la cual estamos constreñidos a escoger bloques fijos cuya longitud no supere el número de cifras de $n = p\,q$. Cuando se utiliza el esquema RSA se realiza una exponentación módulo "n" con el exponente "e" para obtener el texto encriptado, y otra exponentación con el exponente "d" para desencriptar. Pues bien: no sería posible obtener de vuelta el texto original $M = C^d \bmod n$ a partir del criptotoexto si al calcular este último los bloques numéricos que hemos elegido poseen mayor longitud que el número de cifras de "n". La razón es que nuestra función dejaría de ser biunívoca. Veamos ahora el procedimiento:

```
> file2declist :=proc(archivo,n)
    local fd, numero, cont, EOF,i,dimension:
    global LISTA;
    numero := 0:
    fd := fopen(archivo,READ):
    EOF := filepos(fd,infinity):
    filepos(fd,0):
    i:=1;numero[i]:=0;
    for cont from 1 to EOF do
      numero[i] := numero[i] * 256 + readbytes(fd,1):
      if length(numero[i])>=(n-1) then
        i:=i+1;
        numero[i]:=0;
      fi;
    od:
    fclose(fd):
    LISTA:=numero;
    dimension:=i;
    RETURN(dimension):
  end:
```

La función "dec2word" es la inversa de la anterior. Acepta un entero y devuelve la cadena de caracteres ASCII equivalente. Al igual que "file2declist" el código es farragoso y no es necesario leerla con detenimiento para comprender correctamente el funcionamiento del algoritmo RSA. Notemos, sin embargo, que la función "file2declist" genera bloques de longitud variable. Como nosotros los almacenamos en un "array" no hay ningún problema para luego calcular cuál es la longitud de cada bloque por separado. Pero si quisiéramos utilizar este algoritmo en un entorno no controlado, sería necesario convertir cada bloque a una longitud fija. Eso se consigue sin más que, por ejemplo, rellenar con "0" cada bloque hasta obtener la longitud deseada.

```
> dec2word:=proc(l) local a,r,s,t,num,p,q,i;
    r:=0;
    num:=l;
    while (num<>0) do
      r:=r+1;
      s[r]:=(num mod 256);
```

```
      num:=iquo(num,256);
    od;
    for a from 1 to r do
      t[r-a+1]:=s[a];
    od;
    p:=[seq(t[i],i=1..r)];
    q:=convert(p,bytes);
    RETURN(q);
  end:
```

Ahora introducimos las funciones "encrypt" y "decrypt" que elevan el número obtenido anteriormente al exponente respectivo ("e" o "d") módulo "n". Para ello no hemos usado nuestro algoritmo "fastexp" sino la función "power".

```
> encrypt:=proc(l,e,n) local a,enc;
    for a from 1 to depth do
      enc:=power(l,e)mod n;
    od;
    RETURN(enc);
  end:
> decrypt:=proc(l,d,n) local a,dec;
    for a from 1 to depth do
      dec:=power(l,d) mod n;
    od;
    RETURN(dec);
  end:
```

Con esto hemos concluido la introducción de funciones definidas. Apliquemos a continuación lo explicado anteriormente bajo el epígrafe RSA.

Obtengamos dos primos grandes, de 100 y 103 cifras respectivamente, para asegurarnos de que será muy difícil la factorización de su producto. Además, los elegimos con esa diferencia de tres cifras para que no estén cerca el uno del otro, ya que ello podría debilitar la complejidad del problema de factorización asociado, ya que si están cerca el uno del otro la probabilidad de encontrarlos mediante búsqueda exhaustiva deja de ser despreciable.

```
> p:=getprime(100);
  q:=getprime(103);
```

$p := 695179869376613152477737031573815836875247439088118263322744340541\backslash$
$6082742687064818306798788600692 39$

$q := 648420576202584595255567659413399658426757592281524505750358423597 66\backslash$
$3779494052387540419600569177691119 9$

Ahora obtenemos el producto y la función $\phi(n)$. Aunque esta última se puede obtener con MAPLE teóricamente, en la práctica no es factible precisamente porque para ello necesita descomponer "n" en sus factores, y eso es un problema NP-completo, por lo tanto, difícil. Asi pues, simplemente la definiremos como $ffi := (p-1)(q-1)$.

```
> n:=p*q;
  ffi:=(p-1)*(q-1);
```

$n := 450768931465620993726269819344265576495782577726651975555845622199 54\backslash$
$5732299291866477019081806531936632632698259358848445564732998255484719 9\backslash$

792038059930963786127974001214163447683006939168390294694507561

ffi := 450768931465620993726269819344265576495782577726651975555845622199 5\
4573229929186647701908180653193657735169421873976400375193362816984447095\
5340867805597788019220993684034760232890425053141704724057527124

Obtengamos ahora los exponentes de encriptación y decriptación mediante las funciones previamente definidas.

```
> d:=decryptkey(ffi);
  e:=encryptkey(d,ffi);
```

d := 313345855696855998218321990315626922065872380676517427508536148820 51\
6948089556115008762323170002509118033278689388960735776339410583231900 88\
1390274365811536878602441935763698234403672482481510703754371 65

e := 356204668475425500130791339038697881107501440978205558494716096712 51\
9386020833127841703663890132408054376579978454343061852261053937531247 79\
1285052981685430629891214464985389118201001926272089633187313 29

Comprobamos que el exponente "d" pertenece al rango [máx(p, q) + 1, n – 1]:

```
> is(d>max(p,q)+1);
  is(d<n-1);
```

true

true

También hemos de cerciorarnos de que $\log_2(n) < e$.

```
> is(e>log[2](n));
```

true

Como ha superado las dos pruebas recomendadas por Rivest, Shamir y Adleman sigamos adelante. Definamos una variable con el nombre del archivo de texto.

```
> fichero:=`EJEMPLO.TXT`;
```

fichero := *EJEMPLO.TXT*

Saquemos la información del fichero y convirtámosla en lista de enteros almacenada en la variable global LISTA. La longitud de cada entero será igual a la de "n" menos una unidad. Almacenaremos en la variable "depth" el tamaño del array generado.

```
> depth:=file2declist(fichero,length(n)-1);
```

depth := 6

Comprobemos que la variable LISTA existe y tiene los niveles que indica *depth*.

```
> seq(LISTA[i],i=1..depth);
```

[10556295519708128647437758387292812936251376538999219892054432579132494\
7047061485239194115224081268017568314423310656168118443804707735073712 46\
2699956087139714823527758609302323106200426181721125742], [37833559884515\
7581757056382903176975745407770942910010384919785079200607033173222599 27\
7304348428288849604544396748661864849275993140931431365569194953941606 13\
6408860755653229675475059590208736770 88], [1137938350592129857220997648 24\
8243865656021868493075987465394711390338357626611631157198390308276496 04\

5439966369696550045414087512189578852156489962754195454533366370851234377\
3271577425457196919299557], [12903692604838368389445168302277801744289956 1\
5580916521335404310210252497857895931422782262948450098508791812757958 44\
2765544750283032627411956418615937519608722448006246757908274991000243 87\
010524777], [126623595529167981002714982850010841320072491544539310221924\
50569277292161238389160430890755779426553232691940742317657982711734920 7\
6899111845807714182139850210211008590018902874477772247644754494 13], [950\
10387048809346206350496711422744781969834331678458111840823763970409572 4\
4017647979638387939305894778576837966345362858997300242264808211260934 26\
06977427046400]

Encriptemos a continuación el contenido de LISTA[i]. Este proceso nos puede llevar un rato, debido a que se trata de elevar bases de 200 cifras a exponentes de 100 cifras bajo módulos de 200 cifras. Podemos esperar un rato...

```
> for i from 1 to depth do
    DataEncrypt[i]:=encrypt(LISTA[i][1],e,n);
  od;
```

$DataEncrypt_1$:= 532989799830362223502752404038551951879750637175057241574 0\
998703150388762649197120532313220225002957586810663754496014171466703013\
520977115693214507686335622314325811227814580473872368524010439033605848

$DataEncrypt_2$:= 100878807917842594865319659825406300483166267183122020304 3\
980073430284609716835704135521575039364086564818319408233759102843417133\
071987393299215071399238122677190671513401285165665272833616998396561184\
2

$DataEncrypt_3$:= 510365596230708890704194880710548375946983143534901600914 3\
536910485472304494425686154085961472273872364922639908699626913879165947\
171055339449973429074611882679777931241069944087390437670809102275140897

$DataEncrypt_4$:= 308998526006415889847761532125323480558253649138666503340 1\
638644688053088859322362246050072967830281953397760459198230189240725248\
580496960662723176778551564689680732262935148475233780488113994473288080\
5

$DataEncrypt_5$:= 117569079132787295449076460275588398696907276501712743409 3\
494523037808534317313629247777010446795027562875806830598780511267378295\
564298109744258614761367422222158500130270956471492774194137947093078751\
5

$DataEncrypt_6$:= 403695376920450261954179565455038828902128908523971908605 0\
190986102398728056711470124502880993919795391308412328937779643674941010\
009816601006804411289828699681381503616383645294091028993861440101650512

Como podemos ver, la información no tiene mucho parecido con la original. ¿Será posible

recuperarla? Comprobémoslo aplicando la función decrypt.

```
> for i from 1 to depth do
    DataDecrypt[i]:=decrypt(DataEncrypt[i],d,n);
  od;
```

$DataDecrypt_1$:= 10556295519708128647437758387292812936251376538999219892059\
44325791324947047061485239194115224081268017568314423310656168118443804\
7\
07735073712462699956087139714823527758609302323106200426181721125742

$DataDecrypt_2$:= 37833559884515758175705638290317697574540777094291001038499\
19785079200607033173222599277304348428288849604544396748661864849275993\
1\
40931431365569194953941606136408860755653229675475059590208736770889

$DataDecrypt_3$:= 11379383505921298572209976482482438656560218684930759874659\
39471139033835762661163115719839030827649604543996636969655004541408751\
2\
18957885215648996275419545453336637085123437327157742545719691929957

$DataDecrypt_4$:= 12903692604838368389445168302277801744289956155809165213359\
40431021025249785789593142278226294845009850879181275795844276554475028\
3\
03262741195641861593751960872244800624675790827499100024387010524777

$DataDecrypt_5$:= 12662359552916798100271498285001084132007249154453931022199\
24505692772921612383891604308907557794265532326919407423176579827117349\
2\
07689911184580771418213985021021100859001890287447777224764475449413

$DataDecrypt_6$:= 95010387048809346206350496711422744781969834331678458111849\
08237639704095724401764797963838793930589477857683796634536285899730024\
2\
26480821126093426069774270464 00

Apliquemos ahora "dec2word" a cada bloque individual para ver si sale un texto legible...

```
> for i from 1 to depth do
    texto[i]:=dec2word(DataDecrypt[i]);
  od:
```

Antes de ver el resultado, "peguemos" los trozos de texto.

```
> cat(seq(texto[i],i=1..depth));
```

"Zac Briant aspiró el aire frío de la mañana. Su aliento se condensaba y un \nmontón de n\
ubes de color blanco se desintegraban lentamente, apiñándose en la \ngélida y amplia calle\
. \nA pesar de la helada, Briant estaba sudando. Era un esforzado comienzo del \ndía. Con\
suerte, casi había acabado su trabajo. El coche estaba ya inmovilizado: \nBriant había dis\
parado a una de las ruedas, dejándola fuera de control. El vehículo se \nestrelló contra uno\
s contenedores de basura.\n"

¡Perfecto! Aquí tenemos el texto original. Nuestro algoritmo ha funcionado. Por si a alguien le queda la duda, comprobemos que es idéntico al contenido del fichero EJEMPLO.TXT, dado que existe una posibilidad contra 10^10000000000000000000 de que el ordenador se haya "inventado" un texto con sentido.

```
> readbytes(fichero,infinity,TEXT);
```

"Zac Briant aspiró el aire frío de la mañana. Su aliento se condensaba y un \nmontón de n\

ubes de color blanco se desintegraban lentamente, apiñándose en la \ngélida y amplia calle\
. \nA pesar de la helada, Briant estaba sudando. Era un esforzado comienzo del \ndía. Con\
suerte, casi había acabado su trabajo. El coche estaba ya inmovilizado: \nBriant había dis\
parado a una de las ruedas, dejándola fuera de control. El vehículo se \nestrelló contra uno\
s contenedores de basura.\n"

Esta es la prueba definitiva. No ha existido una confabulación contra nosotros. El texto desencriptado y el original son idénticos.

Comentarios finales

La criptografía, y en especial los algoritmos que utiliza, está sujeta a grandes avances . En 1917 el algoritmo de Vigenère fue descrito como "irrompible" por la prestigiosa revista *Scientific American*. Hoy día un mensaje con él codificado no resistiría más de dos minutos de tiempo de computación. El mundo avanza rápido, y con él la matemática y los ordenadores. Lo que ayer parecía imposible hoy es de simplicidad casi trivial. Los computadores cuánticos amenanzan con ser capaces de romper cualquier clave en un tiempo muy pequeño. Y los matemáticos no han dicho la última palabra en lo que a algoritmos de factorización se refiere. A pesar de que se están estudiando nuevas técnicas como son las basadas en el estudio de las curvas elípticas y las ya conocidas de los logaritmos discretos para hacer más difícil la labor del criptoanalista, el triunfo puede ser efímero. Todos los algoritmos se basan en la dificultad asociada al "problema de la mochila", que consiste en encontrar, a partir de un conjunto de "n" números $A = \{ a_1, a_2, ? .. ?, a_n \}$ y un entero "S", si existe un subconjunto de A que sume S. Si los matemáticos consiguiesen resolverlo, todos nuestros datos, hasta ahora protegidos por claves, quedarían comprometidos a merced de quién deseara comerciar con ellos. ¿Hasta qué punto podemos fiarnos de la seguridad de los criptosistemas? ¿Qué sucederá el día que fallen? ¿Merece la pena la ciega confianza que depositamos en ellos? ¿Estamos verdaderamente seguros de que ninguna agencia internacional tiene códigos de "puerta atrás" para los algoritmos que habitualmente utilizamos?

Bibliografía

[1] Denning, Dorothy. *Cryptography and Data Security*. 1982. Ed. Addison-Wesley.

[2] Goldreich, Oded. *The Foundations of Modern Cryptography*. Notas de una conferencia.
Dirección WWW: http://www.wisdom.weizmann.ac/people/homepages/oded/ln89.html.

[3] Mañas, Jose A. *Identidad Digital*. Transparencias de Satelec '98.
Dirección WWW: http://selva.dit.upm.es/~pepe/catalogo/satelec-98/satelec.htm.

[4] Salomaa, Arto. *Public key cryptosystems*. EATCS Vol. 23. 1990. Ed. Springer-Verlag.