Trabajo Práctico

Problemas de congruencias y primalidad en la encriptación de datos

Taller de Álgebra I
Segundo cuatrimestre 2017

Fecha límite de entrega:
Martes **14 de Noviembre** hasta las **23:59** hs.
Coloquio: 22 y 24 de Noviembre (en su turno)

1. Introducción

En este Trabajo Práctico vamos a ver un protocolo simple de criptografía *asimétrica* que se basa en propiedades sobre números primos y congruencias vistas en la materia. Este protocolo se llama RSA. Fue diseñado por Rivest, Shamir y Adleman en 1977, y sigue usándose ampliamente hoy en día.

El problema que se nos plantea es el siguiente. Imaginemos que dos amigos, Alicia y Bob, quieren mandarse mensajes, pero lamentablemente tienen una enemiga: Miranda. Miranda puede leer todo lo que ellos se mandan y publicará en todas las redes sociales cualquier mensaje vergonzoso que encuentre. De más está decir que la diversión de Alicia y de Bob es directamente proporcional a la cantidad de mensajes vergonzosos que se manden.

Por suerte existe el protocolo RSA que nos permite, siguiendo ciertos pasos, encriptar mensajes y salvarse de mirones. Para esto imaginemos que la persona A quiere enviarle el mensaje M[1] a la persona B. Estos son los pasos que deberán seguir:

1. B eligirá dos primos p y q de alguna manera.
2. Luego B calculará los siguientes valores:
   - $n = p \times q$
   - $\phi = (p-1)(q-1)$ [2]
   - e y d tales que $0 \leq e < \phi$ y $e \times d \equiv 1 \mod \phi$

   y llamará (e, n) a su ***clave pública*** y (d, n) a su ***clave privada***.
3. B liberará al mundo su clave pública (e, n). [3]
4. A tomará el mensaje M y (e, n) la clave pública liberada por B y hará la siguiente operación:

[1] El mensaje que le enviará será un número módulo n. Cómo traducir texto plano en un número módulo n y viceversa es un problema aparte, del que *no* nos tendremos que ocupar.

[2] Notar que $\phi = \phi(n)$, donde esta última denota a la función de Euler. En otras palabras, el número ϕ es la cantidad de enteros módulo n que son coprimos con n.

[3] En particular Miranda tendrá acceso a esta clave.

- Si $\text{mcd}(M, n) = 1$ codificará (o encriptará) a M como $M^e \mod n$ [4].
- Si $\text{mcd}(M, n) \neq 1$, codificará a M como $-M$.

y le enviará el mensaje codificado a B. Llamaremos a este mensaje codificado C[5].

5. B, una vez recibido el mensaje C codificado por A, procederá de la siguiente manera:
   - Si $C \geq 0$, decodificará el mensaje como $C^d \mod n$ y obtendrá el mensaje original [6].
   - Si $C < 0$, decodificará C como $-C$, obteniendo el mensaje original.

Asumimos entonces que Miranda conoce los valores de e, n y C, por lo que puede recuperar el mensaje M si logra averiguar quién es d. Notar que d es el inverso de e módulo ϕ, por lo que a Miranda le basta con calcular ϕ para saber quién es d. A su vez, para conocer ϕ le alcanza con factorizar n. Vemos entonces que si A y B quieren que sus secretos estén a salvo, B tiene que elegir primos p y q de manera que su producto n resulte difícil de factorizar. En otras palabras, la fortaleza del protocolo RSA está basada en lo difícil que es factorizar enteros (grandes).

Para más información sobre el protocolo RSA se puede, por ejemplo, consultar [HPS14, Capítulo 3].

Aclaraciones generales

- El código debe ser declarativo (el algoritmo debe quedar claro al leer el código).
- Se deben utilizar las técnicas vistas en clase y no otras.
- Se recomienda el uso de Pattern Matching para mejorar la legibilidad de las funciones.
- Pueden utilizar funciones vistas en clase que les faciliten su trabajo sin problema.
- Pueden crear cualquier función auxiliar y todas **deben** estar acompañadas por su signatura (encabezado de la función con los tipos).
- Para las funciones que deben completar, no se puede cambiar la signatura, (si lo creen necesario, relean el punto anterior).
- Aunque no evaluamos eficiencia, recomendamos que piensen si no están haciendo cuentas de más ya que si lo hacen, no podrán ver el resultado de la sección 6

2. Archivos

En la página de la materia encontrarán dos archivos.

1. En el archivo `Catedra.hs` se definen nuevos renombres de tipos y otros datos que veremos luego:

[4] Para que el proceso de codificación y decodificación funcione correctamente, es necesario que la clave pública n sea lo suficientemente grande como para que dos mensajes distintos M y M' no puedan satisfacer que $M^e \equiv M'^e \mod n$. Dado el tamaño de los mensajes que consideraremos, para evitar esto bastará con tomar primos p y q mayores a 300.

[5] A priori el mensaje C no es público. Pero asumiremos que nos encontramos en el peor escenario posible, esto es: Miranda interceptó la comunicación entre A y B, gracias a lo cual conoce el valor de C.

[6] No es magia, es una consecuencia del Teorema de Fermat generalizado.

```haskell
type Set a      = [a]  -- el tipo conjunto (visto en clase)
type ClPub      = (Integer, Integer) -- (e, n) la clave publica
type ClPri      = (Integer, Integer) -- (d, n) la clave privada
type Mensaje    = [Char]
type Cifrado    = [Integer]
```

Además, están definidas dos funciones que se explican en la próxima sección.

Importante: Este archivo no deben modificarlo.

2. El archivo `main.hs` es el encargado de unificar todo, cuando quieran probar que lo que hayan hecho funcione deberán cargarlo en GHCI con `ghci main.hs`, esto incluirá todos los archivos que hayan modificado.

   Luego, podrán utilizar la función `main`, que codifica, decodifica y rompe algunos mensajes dados por nosotros. O también la función `test`, a la cual le pueden pasar una lista de mensajes que codificará, decodificará y romperá.

3. En los archivos `Aritmetica.hs`, `RSA.h` y `Crack.hs` encontrarán las funciones que deben implementar y, en estos archivos, podrán agregar posibles funciones auxiliares que les parezcan útiles.

   Importante: No se admite cambios en el nombre de funciones ni en las signaturas.

3. Tres palabras sobre módulos

Cuando uno se encuentra con proyectos grandes, suele ser una mala idea tener todo en un mismo archivo. Seguro que durante la materia les pasó alguna de estas dos cosas:

- Todos los ejercicios quedaron en un super archivo, imposible de leer y de entender, o
- cada tanto les hubiese gustado tener acceso a funciones que resolvieron durante otras clases, pero que se encontraban en otros archivos.

La forma de resolver ese problema dentro de Haskell es a través de la utilización de módulos, estos se encuentran implícitos en el esqueleto del TP y no es necesario que los entiendan, pero básicamente permiten encapsular muchas funciones en un grupo y luego poder traer grupos de funciones cuando se requiera.

Importante: para que todo esto funcione no pueden cambiar los nombres de los archivos y todos deben estar en la misma carpeta.

4. Funciones dadas

Las siguientes funciones se daran ya implementadas.

1. ```haskell
   elegidor :: Set Integer -> (Integer, Integer)
   ```

   Que dado un conjunto de al menos dos elementos, extrae dos de ellos y los devuelve en una tupla. No es necesario que ustedes utilicen esta función para resolver el TP.

2. ```haskell
   modExp :: Integer -> Integer -> Integer -> Integer
   ```

   Que dados tres enteros, x, y y z devuelve $x^y \bmod z$. Es muy importante usar esta función y no volver a implementarla por cuestiones de tiempo de ejecución.

3. `aEnteros :: [Char] -> [Integer]`

   Esta función toma una lista de caracteres y la transforma en una lista de enteros, donde cada caracter lo corresponde con su código ASCII.

4. `aChars :: [Integer] -> [Char]`

   Esta función toma una lista de enteros y la transforma en una lista de caracteres, donde cada caracter lo corresponde con su código ASCII. Es la inversa de la anterior. Notar que esta función asume que los enteros pasados pueden ser convertidos a un caracter, es decir, que no son excesivamente grandes.

Ejercicios

1. `mcdExt :: Integer -> Integer -> (Integer, (Integer, Integer))`

   Que dados dos enteros a y b calcula, a través del algoritmo de Euclides extendido, una tripla $(m, (x, y))$ tal que $\text{mcd}(a, b) = m$ y $ax + by = m$.

   Notar que esto es necesario para el segundo paso del protocolo ya que, dados e y ϕ, nos permite buscar un d que cumple con lo siguiente $ed + \phi k = \text{mcd}(e, \phi)$, es decir, que si conseguimos un e coprimo con ϕ, tendremos $ed + \phi k = 1 \Leftrightarrow ed - 1 = \phi k \Leftrightarrow ed = 1(\phi)$ lo que necesitamos.

   Tomamos la tripla de m, x e y, como la tupla de tuplas que contiene a $(m, (x, y))$

2. `criba :: Integer -> Set Integer`

   Que, dado un entero n, devuelve un conjunto de enteros con todos los primos menores a n. Por ejemplo, los siguientes son correctos:

   - `criba 4` $\rightsquigarrow$ `[2, 3]`
   - `criba 7` $\rightsquigarrow$ `[2, 3, 5]`

   Pero los siguientes no lo son:

   - `criba 4` $\rightsquigarrow$ `[2, 19, 5]`
   - `criba 7` $\rightsquigarrow$ `[11, 11, 19, 7, 23, 5]`
   - `criba 3` $\rightsquigarrow$ `[2, 3, 5, 7, 9, 11]`
   - `criba 3` $\rightsquigarrow$ `[2, 3, 8]`

3. `coprimoCon:: Integer -> Integer`

   Dado un número n, encuentra otro número x, con $1 < x < n - 1$, de tal forma que n y x sean coprimos.

4. `inversoMultiplicativo:: Integer -> Integer -> Integer`

   Dados dos números n y m, otorga el inversoMultiplicativo de n módulo m. Notar que esto sólo se puede hacer si dichos números son coprimos, y se puede utilizar el algoritmo de Euclides extendido.

5. `claves :: (Integer, Integer) -> (Integer, Integer, Integer)`

   Que dado dos números p y q, que se asumen primos, calcula los valores e, d y n utilizados por RSA y los devuelve en una tupla en ese orden.

6. `codificador :: ClPub -> Mensaje -> Cifrado`

   Que dada la clave pública y una secuencia de caracteres, la codifica caracter a caracter y devuelve la secuencia obtenida.

7. `decodificador :: ClPri -> Cifrado -> Mensaje`

   Que dada la clave privada y una secuencia de enteros, la decodifica y devuelve la secuencia original.

8. `romper :: ClPub -> ClPri`

   Que dada la clave pública intenta factorizar n y devolver la clave privada.

9. `espia :: ClPub -> Cifrado -> Mensaje`

   Que dada la clave pública y una secuencia de enteros, intenta decodificarla y devuelve la secuencia original.

5. Pautas de Entrega

Observaciones generales:

- El trabajo se debe realizar en grupos de tres personas.
- El código fuente debe enviarse por mail a la lista de docentes de la materia: `algebra1-doc@dc.uba.ar`, indicando nombre, apellido y libreta (o DNI) de cada integrante.
- El programa debe correr usando `ghci` que está instalado en los laboratorios del DC.
- Se evaluará la correctitud, claridad y prolijidad del código entregado.
- La fecha límite de entrega es el Martes **14 de Noviembre** hasta las **23:59** hs

6. YAPA (hacer si tienen tiempo y ganas ☺)

1. La función `modExp` referida en 4 permite calcular potencias módulo z rápidamente[7]. Está basada en el llamado *algoritmo de exponenciación rápida*, del cual se puede encontrar una descripción en [HPS14, Sección 1.3.2].

   Implementar este algoritmo en Haskell.

2. Al intentar factorizar el entero n que es parte de la clave pública, el atacante puede explotar el hecho de que sabe *a priori* que es un producto de dos primos. Factorizar un entero cualquiera es ligeramente más complicado.

   Implementar una función

   ```
   factorizarEnteros :: Integer -> Set (Integer, Integer)
   ```

   que, dado un número entero $n > 1$, devuelva el conjunto de pares de enteros (p, e_p) con p primo y $e_p \geq 1$ tales que $n = \prod_p p^{e_p}$. O más pomposamente, materializar el Teorema Fundamental de la Aritmética.

   Notar que si bien esta función no es obligatoria, les puede ayudar para hacer la que sí lo es.

[7] Por ejemplo, si $y = 2^{1000}$ calcular $x^y \mod z$ con el algorimo *naif* llevaría más tiempo que la edad del universo, mientras que con `modExp` llevaría solo alrededor de 2000 multiplicaciones.

Referencias

[HPS14] Jeffrey Hoffstein, Jill Pipher, and Joseph H. Silverman. *An introduction to mathematical cryptography*. Undergraduate Texts in Mathematics. Springer, New York, second edition, 2014.