Sistemas Lineales de Congruencia

Notas de la décima clase – Taller de Álgebra

Joan Gonzalez

8 de Junio de 2022

I. Objetivo

El objetivo principal de esta clase es obtener un programa en el lenguaje Haskell que permita resolver sistemas lineales de ecuaciones de congruencia sobre los enteros, del siguiente tipo:

$$\begin{cases} a_1 \cdot X & \equiv & b_1 & (\text{mod } m_1) \\ a_2 \cdot X & \equiv & b_2 & (\text{mod } m_2) \\ \vdots & \vdots & \vdots & \vdots \\ a_n \cdot X & \equiv & b_n & (\text{mod } m_n) \end{cases}$$

Si recordamos las clases de Álgebra, nos daremos cuenta que de alguna forma vamos a querer aplicar el Teorema Chino del Resto (TCR) para resolver simultáneamente todas las ecuaciones del sistema. Sin embargo, para ello primero necesitamos que las ecuaciones estén dadas de manera simplificada y que los módulos del sistema de ecuaciones de congruencia sean coprimos dos a dos.

II. Coprimizando ecuaciones

Empecemos primero estableciendo qué significa que queramos resolver las ecuaciones y darlas de forma *simplificada*. Consideremos la ecuación:

$$a \cdot X \equiv b \,(\text{mod } m) \tag{1}$$

Sabemos que tiene solución $\iff d = (a : m) \mid b$. Si esto se cumple, entonces podemos reducir la ecuación a través de:

$$\frac{a}{d} \cdot X \equiv \frac{b}{d} \,(\text{mod } \frac{m}{d}) \;=\; a' \cdot X \equiv b' \,(\text{mod } m') \tag{2}$$

Como dividimos por el máximo común divisor entre a y b, ahora hemos *coprimizado* la ecuación. Por lo que ésta, tiene una propiedad adicional respecto a la primera: $(a' : m') = 1$. Usando ésto a nuestro favor, ahora la ecuación tendrá una solución, y será una combinación lineal de a' y m', de forma tal que:

$$s \cdot a' + t \cdot m' = 1 \tag{3}$$

Si ahora tomamos congruencia módulo m', entonces la ecuación es equivalente a:

$$X \equiv s \cdot b \,(\text{mod } m') \tag{4}$$

Donde $b = a' \,(\text{mod } m')$.

Teniendo esto en cuenta, escribamos algunas funciones en Haskell que nos permitan dar una solución a una ecuación de congruencia.

Primero, haremos uso extensivo del Algoritmo de Euclides y dado que lo programamos en la clase 9, simplemente reutilizaremos parcialmente ese código:

```haskell
emcd :: Int -> Int -> (Int, Int, Int)
emcd a 0 = (a,1,0)
emcd a b = (g,s,t)
        where
           (g,s',t') = emcd b (mod a b)
           q = div a b
           s = t'
           t = s' - t' * q

mcd :: Int -> Int -> Int
mcd a 0 = a
mcd a b = mcd b (mod a b)
```

Donde **emcd** es el algoritmo *extendido* de Euclides, que devuelve una terna (g, s, t) dados $a, b \in \mathbb{Z}$ donde $s \cdot a + t \cdot b = g$, y **mcd** da el máximo común divisor entre a y b.

Además, representaremos a las ecuaciones de la forma (1) a través del tipo de dato **(Int, Int, Int)**, en general:

$$a \cdot X \equiv b \pmod{m} \rightsquigarrow \texttt{(a,b,m)}$$

Análogamente, representaremos las soluciones (clases de congruencia) de la forma (4) a través del tipo de dato **(Int, Int)**, en general:

$$X \equiv s \cdot b \pmod{m'} \rightsquigarrow \texttt{(b,m)}$$

Así, definimos:

```haskell
type Ecuacion = (Int, Int, Int)
type Solucion = (Int, Int)
```

Luego, queremos:

1. Una función que tome una **Ecuacion** y la convierta en una **Ecuacion** equivalente, como se muestra en (2).
2. Una función que tome una **Ecuacion** y la convierta en **Solucion** de forma tal que la nueva ecuación coprimizada tenga la propiedad de $(a' : m') = 1$ como se muestra en (4).
3. Una *composición* de funciones, que tome una **Ecuacion** y de directamente la **Solucion** correspondiente a la misma.

Formalizando y dándole cuerpo a dichas funciones, podemos escribir:

```
equivalente :: Ecuacion -> Ecuacion
equivalente (a,b,m) | mod b d /= 0 = undefined
                    | otherwise = (div a d, div b d, div m d)
                    where d = mcd a m

coprimizar :: Ecuacion -> Solucion
coprimizar (a,b,m) = (c, m)
                   where
                    (_,s,_) = emcd a m
                    c = mod (s*b) m

solucion :: Ecuacion -> Solucion
solucion e = coprimizar (equivalente e)
```

III. Sistemas de ecuaciones de congruencia

Si volvemos a nuestro problema original, queríamos resolver sistemas que tengan la forma:

$$\begin{cases} a_1 \cdot X & \equiv & b_1 & (\text{mod } m_1) \\ a_2 \cdot X & \equiv & b_2 & (\text{mod } m_2) \\ \vdots & \vdots & \vdots & \vdots \\ a_n \cdot X & \equiv & b_n & (\text{mod } m_n) \end{cases} \tag{5}$$

Ahora bien, a través de nuestra capacidad de simplificar las ecuaciones de este sistema a sus clases de congruencia equivalentes, podemos reescribirlo como:

$$\begin{cases} X & \equiv & r_1 & (\text{mod } m'_1) \\ X & \equiv & r_2 & (\text{mod } m'_2) \\ & \vdots & \vdots & \vdots \\ X & \equiv & r_n & (\text{mod } m'_n) \end{cases} \tag{6}$$

Sin embargo, no podemos aún usar el Teorema Chino del Resto, ya que cabe la posibilidad de que los m_i sean coprimos dos a dos, como lo requerimos. Esto quiere decir que puede existir un primo $p \in \mathbb{Z}$ tal que p divide al menos a dos de los módulos $\{m'_1, m'_2, \cdots, m'_n\}$. Luego, el sistema no tiene módulos coprimos dos a dos, ya que p divide al menos a dos de dichos módulos. A p lo consideramos un primo *malo* del sistema.

Encontremos todos los primos malos para un sistema de ecuaciones de congruencia simplificado. Si encontramos un primo malo, necesitaremos *extraerlo* de todas las ecuaciones para encontrar un sistema equivalente. A modo de ejemplo, tomemos la siguiente ecuación de congruencia:

$$X \equiv r \ (\text{mod } m)$$

Sabiendo que $p \in \mathbb{Z} \ \wedge \ p^k \mid m$. Luego la ecuación es equivalente al sistema:

$$\begin{cases} X & \equiv & r' & (\text{mod } p^k) \\ X & \equiv & r & (\text{mod } \frac{m}{p^k}) \end{cases}$$

Donde r' es la congruencia módulo p^k de r, ya que puede ocurrir que $r \geqslant p^k$.

En este sistema, si m es una potencia de p, entonces $m = p^k$ para algún k. Por ende, la última ecuación de congruencia del sistema, queda:

$$X \equiv 0 \ (\text{mod } 1)$$

Y como no nos aporta información respecto de nuestro problema inicial, podemos eliminarla.

Apliquemos ahora lo ejemplificado a un caso más cercano a lo que nos interesa resolver. Tomemos el siguiente sistema:

$$S = \begin{cases} X & \equiv & r_1 & (\text{mod } m_1) \\ X & \equiv & r_2 & (\text{mod } m_2) \\ X & \equiv & r_3 & (\text{mod } m_3) \\ X & \equiv & r_4 & (\text{mod } m_4) \\ X & \equiv & r_5 & (\text{mod } m_5) \end{cases}$$

Ahora, si existe un primo malo p que divide a m_1, m_3 y a m_5, donde $m_3 = p^k$ para algún k, entonces:

$$S \rightsquigarrow S' = \begin{cases} X & \equiv & r_1 & (\text{mod} & p^{k_1}) \\ X & \equiv & r_1 & (\text{mod} & m_1/p^{k_1}) \\ X & \equiv & r_2 & (\text{mod} & m_2) \\ X & \equiv & r_3 & (\text{mod} & p^{k_3}) \\ X & \equiv & r_4 & (\text{mod} & m_4) \\ X & \equiv & r_5 & (\text{mod} & p^{k_5}) \\ X & \equiv & r_5 & (\text{mod} & m_5/p^{k_5}) \end{cases}$$

En el sistema equivalente a S, S' ahora hay ecuaciones que tienen sus módulos dados por potencias de p, y ecuaciones a las que les *extrajimos* las potencias de p. Reordenemos el sistema para que sea más fácil de verlas visualmente, tal que:

$$S' = \begin{cases} X & \equiv & r_1 & (\text{mod} & p^{k_1}) \\ X & \equiv & r_3 & (\text{mod} & p^{k_3}) \\ X & \equiv & r_5 & (\text{mod} & p^{k_5}) \\ X & \equiv & r_4 & (\text{mod} & m_4) \\ X & \equiv & r_1 & (\text{mod} & m_1/p^{k_1}) \\ X & \equiv & r_2 & (\text{mod} & m_2) \\ X & \equiv & r_5 & (\text{mod} & m_5/p^{k_5}) \end{cases}$$

Ahora las ecuaciones que tienen como módulo algún p^k pueden empezar a simplificarse. Veamos que si $k_\alpha \leqslant k_\beta$ entonces veamos que el siguiente sistema puede simplificarse:

$$\begin{cases} X & \equiv & r_\alpha & (\text{mod } p^{k_\alpha}) \\ X & \equiv & r_\beta & (\text{mod } p^{k_\beta}) \end{cases}$$

Si $r_\alpha \equiv r_\beta (\text{mod } p^{k_\alpha}) \implies$ Las soluciones de la segunda ecuación, están contenidas en la primera. Luego, la primer ecuación puede eliminarse.

Si $r_\alpha \not\equiv r_\beta (\text{mod } p^{k_\alpha}) \implies$ Las ecuaciones no comparten soluciones en común.

Juntando todo, dado un sistema simplificado de ecuaciones de congruencia S, podemos *desdoblarlo* en un sistema equivalente, extrayendo los primos malos p_i que pueden estar presentes en los módulos de S. Luego, para cada primo malo p, desdoblamos las ecuaciones que lo tengan como factor de su módulo en distintas ecuaciones, y resumimos los subsistemas hasta librarnos de todas las ecuaciones que no nos aporten información.

Programemos ésto en Haskell. Primero, representemos a un sistema de ecuaciones de congruencia general (5) con el tipo de dato **`[Ecuacion]`**, tal que:

$$\texttt{[(a1, b1, m1), ... , (an, bn, mn)]} \leftrightsquigarrow \begin{cases} a_1 \cdot X & \equiv & b_1 & (\text{mod } m_1) \\ \vdots & \vdots & \vdots & \vdots \\ a_n \cdot X & \equiv & b_n & (\text{mod } m_n) \end{cases}$$

Además, representemos al sistema simplificado (6) a través de **`[Solucion]`**:

$$\texttt{[(r1, m1), ... , (rn, mn)]} \leftrightsquigarrow \begin{cases} X & \equiv & r_1 & (\text{mod } m_1) \\ \vdots & \vdots & \vdots & \vdots \\ X & \equiv & r_n & (\text{mod } m_n) \end{cases}$$

Queremos,

1. Una función que tome un sistema general **`[Ecuacion]`** y de otro sistema **`[Solucion]`** coprimizado.
2. Una función que tome un sistema simplificado **`[Solucion]`** y devuelva un sistema equivalente **`[Solucion]`** donde se hayan extraido todos los primos malos del original.
3. Una función que tome un sistema simplificado sin primos malos con módulos coprimos dos a dos **`[Solucion]`** y lo resuelva, dando la **`Solucion`**.
4. Una composición de funciones, que tome un sistema general **`[Ecuacion]`** y devuelva la **`Solucion`** del sistema.

Resolviendo en orden, podemos directamente resolver el primero de nuestros objetivos. Teniendo en cuenta que un sistema está representado como una lista de **Ecuaciones**, podemos recursivamente crear una lista de **Soluciones**, aplicando la función **solucion** que creamos previamente.

```
soluciones :: [Ecuacion] -> [Solucion]
soluciones [] = []
soluciones (e:es) = (solucion e) : (soluciones es)
```

IV. Primos malos

Ahora bien, para extraer todos los primos malos de un sistema simplificado, primero debemos conocer *cuales* son esos primos malos. Pero para conocer quienes son, primero necesitamos una lista de los módulos del sistema.

Extraer los módulos no es particularmente difícil, podemos adjuntar cada segundo elemento de cada tupla de las **Soluciones** a una nueva lista de enteros, tal que:

```
modulos :: [Solucion] -> [Int]
modulos [] = []
modulos ((r,m):es) = m:(modulos es)
```

Ahora, teniendo la lista de los módulos de un sistema simplificado S, queda preguntar: Dado un $n \in \mathbb{Z}$, ¿Es n un primo malo de S? Luego, es lo mismo que hacer dos preguntas separadas:

1. ¿Es n primo?
2. Usando la definición de *primo malo*,
   ¿Divide n al menos a dos módulos de S?

La primera parte de este subproblema también la podemos solucionar rápidamente utilizando parcialmente lo que vimos en la clase 5

```haskell
esPrimo :: Int -> Bool
esPrimo 1 = False
esPrimo n = (menorDivisor n) == n
          where
            menorDivisor n = menorDivisorDesde n 2
            menorDivisorDesde n k | mod n k == 0 = k
                                  | otherwise = menorDivisorDesde n (k+1)
```

Luego, usando **esPrimo** podemos determinar si un número es un primo o no.

Queda determinar si n es un primo malo. Ahora bien, preguntar si un número divide al menos a otros dos de una lista, es un caso particular de preguntar: Dada una lista de números $[m_1, m_2, \cdots, m_n]$, ¿Cuántos son múltiplos de n?

Es decir, si podemos contar la cantidad de múltiplos respecto de n en la lista de módulos, luego sólo queda comparar que esa cantidad sea mayor o igual a 2.

```haskell
cantidadMultiplos :: [Int] -> Int -> Int
cantidadMultiplos [] _ = 0
cantidadMultiplos (m:ms) n | mod m n == 0 = 1 + cantidadMultiplos ms n
                           | otherwise = cantidadMultiplos ms n

esPrimoMalo :: [Solucion] -> Int -> Bool
esPrimoMalo sist n = (esPrimo n) && sist n >= 2
                   where multiplos = cantidadMultiplos (modulos sist)
```

Teniendo **esPrimoMalo** ahora por cada módulo $m \in \{m_1, m_2, \cdots, m_n\}$ queda ver si m es un primo malo.

Para ello, primero hagamos una funcion auxiliar **todosLosPrimosMalosHasta** que, dada una lista y un máximo, nos de una lista de todos los números que sean primos malos (de esa lista) por debajo de ese máximo. Así, podemos escribir:

```haskell
todosLosPrimosMalosHasta :: [Solucion] -> Int -> [Int]
todosLosPrimosMalosHasta _ 0 = []
todosLosPrimosMalosHasta sist n | esPrimoMalo sist n = n:malos
                                | otherwise = malos
                                where
                                malos = todosLosPrimosMalosHasta sist (n-1)
```

Luego, hallar todos los primos malos de una lista, es cuestión de llamar a **todosLosPrimosMalosHasta** dando la lista y su máximo. Por lo que podemos escribir lo siguiente:

```haskell
todosLosPrimosMalos [] = []
todosLosPrimosMalos sist = todosLosPrimosMalosHasta sist limite
                         where limite = maximum (modulos sist)
```

Donde **maximum** es una función del propio **Prelude** de Haskell que devuelve el número más grande en una lista.

Con ésto resuelto, queda resolver el sistema equivalente sin primos malos, dando una **Solucion**.