Teoría de Números

Brian Morris Esquivel

Facultad de Ciencias Exactas, Ingeniería y Agrimensura
Universidad Nacional de Rosario

Training Camp Argentina 2019

Guía de la clase

Repaso

Conocimientos previos recomendados

Para aprovechar al máximo esta clase es muy recomendable tener claros los siguientes conceptos:

- **Criba de Eratóstenes**
- **Factorización en primos en $O(\sqrt{n})$ y en $O(\log(n))$ usando la Criba**
- **Aritmética modular**
- **Exponenciación Modular (ExpMod)**
- **Euclides Extendido e Inverso modular (InvMod)**
- **Pequeño Teorema de Fermat**

Criba de eratóstenes

La **criba de eratóstenes** es una tabla que te indica si un número es primo o no, y podemos modificarla para que, si un número es compuesto, te indique algún primo que lo divide.

La criba para los números desde $\mathbf{1}$ **hasta** $\boldsymbol{M}$ puede ser construida con complejidad $\boldsymbol{O(MlogM)}$

Se puede implementar eficientemente para que puedas calcularla con M hasta 10^7.

Factorización de un número

La factorización de un número es algo sencillo, pero tenemos que tener sabidos los siguientes métodos:

Factorización bruta: Buscar divisores primos del número recorriendo todos los enteros hasta $\sqrt{n}$
Complejidad: $O(\sqrt{n}\,)$

Factorización bruta podada: Buscar divisores primos del número recorriendo solamente los primos hasta $\sqrt{n}$
Complejidad: $O\left(\frac{\sqrt{n}}{\log\sqrt{n}}\right)$

Factorización con Criba: Usar la criba modificada para encontrar factores primos en $O(1)$ y factorizar rápido. Funciona solo si $\boldsymbol{n \leq M}$.
Complejidad: $O(\log n)$

Aritmética Modular

La aritmética modular es una rama de la matemática que estudia ecuaciones del tipo:

$$a \equiv b(m)$$

Que significa:
"Los enteros $\boldsymbol{a}$ y $\boldsymbol{b}$ tienen el mismo resto en la división por $\boldsymbol{m}$"
o bien:
"La diferencia $\boldsymbol{a - b}$ es múltiplo de $\boldsymbol{m}$"

Es importante tener experiencia con este tipo de ecuaciones y conocer las propiedades básicas.

Exponenciación Modular

Existe un algoritmo que nos permite hallar rápido el resto de una potencia en la división por m.

Es decir, nos permite hallar el resultado de:

$$x = a^e \ \% \ m$$

…con complejidad $O(log\ e)$

Euclides Extendido e Inverso Modular

Dados dos números coprimos $\boldsymbol{a}$ y $\boldsymbol{m}$. Existen distintos métodos para hallar rápido el inverso de $\boldsymbol{a}$ módulo $\boldsymbol{m}$.

Nuestro método de preferencia es usar el algoritmo de **Euclides Extendido**, ya que al ejecutarlo en el par (a, m) obtenemos:

$$a.\boldsymbol{x} + m.\boldsymbol{y} = 1$$

Donde x es efectivamente el inverso de a modulo m.

Euclides Extendido tiene complejidad $O(\log a + \log m)$

Pequeño Teorema de Fermat

Dados un número primo $\boldsymbol{p}$ y un número $\boldsymbol{a}$ coprimo a p
Se verifica la siguiente ecuación:

$$a^p \equiv a(p)$$

Introducción

¿Qué es la Teoría de Números?

Es el arte de trabajar con propiedades que se cumplen dentro del conjunto de los números enteros.

Los temas más populares tocados en Teoría de números son:
Primalidad. Divisibilidad. Aritmética Modular.
Todos temas lindos.

La Teoría de Números aparece frecuentemente en problemas de Grafos y de Combinatoria. Es el condimento algebraico al trabajar numericamente en estos temas.

De que va a tratar la clase.

Vamos a aprender algunos métodos nuevos para atacar problemas.

Vamos a dar un vistazo a la teoría detrás de los métodos y vamos a tratar de hacernos una idea de por qué funcionan.

No vamos a detenernos a analizar las demostraciones de los teoremas. Tampoco vamos a detenernos a analizar la implementación de los algoritmos.

El objetivo de la clase es que al final de la misma tengan noción de la existencia de más herramientas con las que pelear en la prueba.

Pollard´s Rho

De que va a tratar Pollard's Rho

Vamos a ver un nuevo algoritmo de factorización. En muchos casos resulta más rápido que los que conocemos.

Nos va a permitir factorizar números del orden de hasta 10^{18} (lo que entra en un entero de 64-bits), casos en los cuales los algoritmos que conocíamos hasta ahora son muy lentos.

Vamos a ver de que trata...

El test de primalidad de Fermat

Lo primero que vamos a ver es un método rápido de decidir si un número es o no un número primo.

Un primer acercamiento que vamos a ver es el test de **primalidad de Fermat.**

Por el **Pequeño Teorema de Fermat**, sabemos que si un número $\boldsymbol{p}$ es primo, para todo entero $\boldsymbol{a}$ entre 2 y $p-1$ se verifica:

$$a^{p-1} \equiv 1(p)$$

Probaremos la veracidad de la ecuación para distintos valores de a.

Si para algun valor de a la ecuación es inválida sabemos que el número **no es primo**. Si en todos los casos probados es válida decimos que p es un **probable primo**.

El método de Rabin-Miller

El test anterior tiene una versión mejorada.

Partiendo de la ecuación anterior: $a^{p-1} - 1 \equiv 0(p)$

Si "factorizamos" el número $\boldsymbol{p-1}$ de forma $p - 1 = 2^t k$ donde k es impar. Podemos factorizar el lado izquierdo de la ecuación de arriba como:

$$(a^{2^t k} - 1) \equiv 0(p)$$

$$\left(a^{2^{t-1} k} + 1\right)\left(a^{2^{t-2} k} + 1\right) \ldots (a^{2^0 k} + 1)(a^{2^0 k} - 1) \equiv 0(p)$$

Esto quiere decir que p es divisor de al menos uno de los factores del lado izquierdo.

El método de Rabin-Miller (cont.)

Es decir, se tiene que cumplir alguna de las ecuaciones siguientes:

$$a^k \equiv 1(p)$$
$$a^k \equiv -1(p)$$
$$a^{2k} \equiv -1(p)$$
$$\dots$$
$$a^{2^t k} \equiv -1(p)$$

Esta condición es más fuerte que la de la propuesta anterior, es decir, hay menos números compuestos que darán un **falso positivo** al test.

Por qué anda el método

Este test por supuesto que puede fallar.
Cuando obtenemos un **falso** (resulta n compuesto) estamos seguros de que el resultado es correcto.
Pero cuando tenemos un **verdadero** solo sabemos que "puede que sea verdad".

Resulta que cada vez que ejecutamos un test y da **verdadero** decimos que tiene cierta probabilidad ρ de fallar.

Si ejecutamos el test k veces, la probabilidad de que falle es $\mathrm{P} = \rho^k$
Podemos hacer que ese número sea muy pequeño.

Una práctica común es ejecutar el test sobre los números $\boldsymbol{a}$:

$$\{2, 3, 5, 7, 11, 13, 17, 19, 23\}$$

Implementación

Una implementación del método de Rabin-Miller es la siguiente:

```
typedef long long ll;
typedef unsigned long long ull;
bool is_prime_prob(ll n, int a){
    if(n == a)return true;
    ll s = 0, d = n-1;
    while(d%2 == 0) s++, d /= 2;
    ll x = expmod(a, d, n);
    if((x == 1) || (x+1 == n))return true;
    for(int i = 0; i < s-1; i++){
        x = mulmod(x, x, n);
        if(x == 1) return false;
        if(x+1 == n) return true;
    }
    return false;
}
bool rabin(ll n){
    if(n == 1)return false;
    int ar[]={2,3,5,7,11,13,17,19,23};
    for(int i = 0; i < 9; i++)if(!is_prime_prob(n, ar[i]))return false;
    return true;
}
```

Nota: Pueden suponer que el algoritmo anda para todos los números hasta 10^{18} y, en fin, son todos los números que nos importan.

El algoritmo Rho

Lo siguiente que vamos a mirar es el método Rho descrito por John Pollard.

No vamos a explicar el método ni el algoritmo porque hacerlo correctamente requiere conocimientos más avanzados a esta clase.

Nos será suficiente saber que el algoritmo consigue “rápido” un divisor no trivial de $\boldsymbol{n}$ si es que $\boldsymbol{n}$ es compuesto.

Es decir, dado $\boldsymbol{n}$, conseguimos $\boldsymbol{a}$ tal que: $\boldsymbol{2 \leq a \leq n-1,\ a|n}$

La **complejidad** del algoritmo está aún en estudio, una buena cota del tiempo para la mayoría de los casos es $\sqrt[4]{n}$

La factorización Rho de Pollard

Ahora juntaremos los dos métodos que acabamos de aprender para factorizar un número.

Para factorizar un número $\boldsymbol{n}$ hacemos lo siguiente:

.Chequeamos si $\boldsymbol{n}$ es primo usando **Rabin-Miller**.

.Si da cierto lo agregamos a la factorización y terminamos.

.Si no es primo, usamos **Rho** para encontrar un divisor $\boldsymbol{d}$ no trivial de $\boldsymbol{n}$. Una vez hallado factorizamos $\boldsymbol{d}$ y $\frac{\boldsymbol{n}}{\boldsymbol{d}}$ por separado y unimos las soluciones.

Implementación

Esta es una implementación de la factorización de Rho, nótese que en la factorización se llama a rabin().

```
ll rho(ll n){
    if(!(n&1))return 2;
    ll x = 2, y = 2, d = 1;
    ll c = rand()%n + 1;
    while(d==1){
        x = (mulmod(x,x, n)+c)%n;
        y = (mulmod(y,y, n)+c)%n;
        y = (mulmod(y,y, n)+c)%n;
        if(x >= y)d = gcd(x-y, n);
        else d = gcd(y-x, n);
    }
    return d == n ? rho(n) : d;
}
void fact(ll n, map<ll, int> &f){
    if(n == 1)return;
    if(rabin(n)){ f[n]++; return; }
    ll q = rho(n); fact(q, f); fact(n/q, f);
}
```

Usando factorizaciones

¿Qué método usar para factorizar en cada caso?

Podemos decir que conocemos hasta el momento 4 métodos de factorización, los 3 mencionados al principio y Pollard's Rho ahora.

Siempre debemos intentar usar el más fácil y rápido de codear, siempre y cuando ese método entre en tiempo, el orden obvio de prioridad es:

- Factorización Bruta
- Factorización Bruta Podada
- Factorización con Criba
- Factorización Pollard´s Rho

Si la Criba ya está implementada en el código la factorización con Criba es preferible.

Para números chicos, $x < 10^8$, el método con Rho puede ser más lento que la factorización bruta incluso.

Función cantidad de divisores de un número

Un teorema conocido en Teoría de Números es el cálculo de **cantidad de divisores** de un número a partir de su factorización.

El teorema nos dice que dado un número y su factorización:

$$n = p_1^{\alpha_1} p_2^{\alpha_2} \dots p_k^{\alpha_k}$$

La cantidad de divisores del número se puede calcular:

$$\tau(n) = (\alpha_1 + 1)(\alpha_2 + 1) \dots (\alpha_k + 1)$$

Esto surge de que cada divisor de un número puede factorizarse como:

$$d = p_1^{\beta_1} p_2^{\beta_2} \dots p_k^{\beta_k},\ 0 \leq \beta_i \leq \alpha_i$$

Implementación

Es muy corta la implementación de la función:

```
ll CantDiv(ll n){
    fact(n);
    ll ans = 1;
    for(auto f : F)ans *= (f.second + 1);
    return ans;
}
```

Notemos que para usar la función CantDiv() necesitamos de una función factorización.

Función suma de divisores de un número

A partir de conocer la factorización de un número n, también podemos hallar la suma de divisores de n.

Sea un número n por su factorización:

$$n = {p_1}^{\alpha_1} {p_2}^{\alpha_2} \dots {p_k}^{\alpha_k}$$

Podemos hallar la suma de sus divisores como:

$$\sigma(n) = \frac{\left({p_1}^{\alpha_1+1} - 1\right)}{p_1 - 1} \cdot \frac{\left({p_2}^{\alpha_2+1} - 1\right)}{p_2 - 1} \dots \frac{\left({p_k}^{\alpha_k+1} - 1\right)}{p_k - 1}$$

$$\sigma(n) = \prod_{i=1}^{k} \frac{{p_i}^{\alpha_i+1} - 1}{p_i - 1}$$

¿De dónde sale esto…?

Función suma de divisores de un número (cont.)

Recordemos que:

$$\frac{p_i^{a_i+1}-1}{p-1} = {p_i}^0 + p_i^1 + \cdots + {p_i}^{\alpha_i}$$

Por lo tanto:

$$\sigma(n) = \prod_{i=1}^{k} ({p_i}^0 + p_i^1 + \cdots + {p_i}^{\alpha_i})$$

Y si expandimos esa expresión obtenemos todos los sumandos de la forma:

$$d = {p_1}^{\beta_1} {p_2}^{\beta_2} \ldots {p_k}^{\beta_k},\ 0 \leq \beta_i \leq \alpha_i$$

…que representan todos los divisores de n.

Implementación

```
ll SumDiv(ll n){
    fact(n);
    ll ans = 1;
    for(auto f : F){
        ans *= (exp(f.first, f.second+1) - 1)/(f.first-1);
    }
    return ans;
}
```

Notemos otra vez que para usar la función SumDiv() necesitamos de una función factorización.
También necesitamos una función de exponenciación, la podríamos implementar algo parecido a expmod.

No necesitamos que la exponenciación sea rápida ya que si es lenta tarda a lo sumo lo mismo que la factorización.

La función Phi de Euler

Por último vamos a ver una función especial.
Dado un entero n, esta función cuenta la cantidad de números menores o iguales que n que son coprimos con n.

La llamaremos "función phi".
Euler demostró que ésta función se puede calcular a partir de la factorización del número n como:

$$n = {p_1}^{\alpha_1} {p_2}^{\alpha_2} \dots {p_k}^{\alpha_k}$$

$$\phi(n) = n \prod_{i=1}^{k} \left(1 - \frac{1}{p_i}\right) = n \prod_{i=1}^{k} \left(\frac{p_i - 1}{p_i}\right)$$

Implementación de Euler's Phi

```
ll Phi(ll n){
    fact(n);
    for(auto t : F) n -= n/t.first;
    return n;
}
```

Asi cerramos el capítulo de factorización.
Vamos a ver un poco los usos de la función Phi.

Usando Euler's Phi

Usos directos de la función Phi

Hay problemas que requerirán saber usar Phi por si sola.

Tomemos por ejemplo el siguiente problema de UVa:

A *fraction* $\frac{m}{n}$ is *basic* if $0 \leq m < n$ and it is *irreducible* if $\gcd(m, n) = 1$. Given a positive integer n, in this problem you are required to find out the number of *irreducible basic fractions* with denominator n.

For example, the set of all *basic fractions* with denominator 12, before reduction to lowest terms, is

$$\frac{0}{12}, \frac{1}{12}, \frac{2}{12}, \frac{3}{12}, \frac{4}{12}, \frac{5}{12}, \frac{6}{12}, \frac{7}{12}, \frac{8}{12}, \frac{9}{12}, \frac{10}{12}, \frac{11}{12}$$

Reduction yields

$$\frac{0}{1}\ \cancel{\frac{0}{12}}, \frac{1}{12}, \frac{1}{6}, \frac{1}{4}, \frac{1}{3}, \frac{5}{12}, \frac{1}{2}, \frac{7}{12}, \frac{2}{3}, \frac{3}{4}, \frac{5}{6}, \frac{11}{12}$$

Hence there are only the following 4 *irreducible basic fractions* with denominator 12

$$\frac{1}{12}\ \cancel{\frac{0}{12}}, \frac{5}{12}, \frac{7}{12}, \frac{11}{12}$$

https://uva.onlinejudge.org/index.php?option=onlinejudge&page=show_problem&problem=1120

Usos directos de la función Phi

Ya están bastante spoileados. Es claro que la respuesta al problema va a ser $\phi(n)$. ¿Es claro por qué es así?

Este problema es relativamente sencillo pero en general la dificultad en los problemas recae en darte cuenta de que la respuesta es equivalente a la función conocida.

En este caso por definición de ϕ lo podemos demostrar.

El Pequeño Teorema de Fermat

Recordemos el Pequeño Teorema de Fermat:

Sean $\boldsymbol{p}$ un número primo, y $\boldsymbol{a}$ un entero coprimo con p, se satisface la ecuación:

$$a^p \equiv a\ (p)$$

Como a es invertible en p, por ser coprimo a p, podemos escribir:

$$a^{p-1} \equiv 1\ (p)$$

El Pequeño Teorema de Fermat (cont.)

Una utilidad interesante del teorema es hallar el inverso modular de $\boldsymbol{a}$ modulo $\boldsymbol{p}$, ya que también se cumple la igualdad:

$$a^{p-2} \equiv a^{-1}(p)$$

Pero eso no es todo. La mayor utilidad está cuando tengo que calcular $a^e \% p$ cuando e es un número grande. Observemos que:

$$\boldsymbol{a^e \equiv a^{e\%(p-1)}\ (p)}$$

Esto es de suma utilidad cuando tenemos que manejar exponentes muy grandes que no son representables por un entero de 64-bits.

El Teorema de Fermat-Euler

Vamos a ponernos como objetivo resolver el problema de calcular $a^e \ \%m$ cuando el exponente e es muy grande.

El Teorema anterior nos limita a los casos en los que m es primo. Por eso introducimos un nuevo teorema.

Teorema de Fermat-Euler

Sean a y m dos enteros coprimos, se verifica: $a^{\phi(m)} \equiv 1 \ (m)$

Notemos que el Pequeño Teorema de Fermat es un caso particular de éste, cuando m es primo.

Podemos trabajar algebraicamente y llegar a:

$$a^e \equiv a^{e\%\phi(m)} \ (m)$$

Generalización

Existen algunos casos especiales en los que esta última ecuación se puede generalizar para cualquier terna de números $\{a, e, m\}$. Sin necesidad de que a y m sean coprimos.

Formalmente, para toda terna $\{a, e, m\}$ de enteros, en la que $\boldsymbol{e \geq \log_2 m}$ se cumple la siguiente ecuación.

$$\boldsymbol{a^e \equiv a^{\phi(m) + e\%\phi(m)} \ (m)}$$

Recordemos que los casos que nos interesan resolver son aquellos en los que el exponente **e** es demasiado grande y no así el modulo **m**.

Esta última ecuación resulta válida para todos estos casos.

Sistemas de Ecuaciones Modulares

¿A qué llamamos ecuación modular?

Una “ecuación modular” es una ecuación de la forma:

$$a.x \equiv b(m)$$

La **solución** al sistema es el conjunto de valores de x para los cuales la ecuación se verifica.

La solución a una ecuación modular puede ser **vacía** o tener infinitos valores.

Si la solución es vacía decimos que la ecuación es **inconsistente**.

Caso contrario decimos que es consistente. Je.

Ecuaciones inconsistentes

Tenemos dada la ecuación: $a.x \equiv b\ (m)$

Sea d el máximo común divisor entre a y m: $d = mcd(a, m)$

Podemos demotrar que la ecuación es consistente si y solo si $d|b$

Es decir, si sucede que $b\%d \neq 0$ entonces la ecuación es inconsistente y terminamos.

En el caso contrario, podemos demostrar que la ecuación original es equivalente a la ecuación:

$$\frac{a}{d}.x \equiv \frac{b}{d}\left(\frac{m}{d}\right)$$

Ecuaciones normalizadas

Tenemos dada la ecuación consistente: $a.x = b\ (m)$

Dado $d = mcd(a, m)$ definimos: $a' = \frac{a}{d}$ $b' = \frac{b}{d}$ $m' = \frac{m}{d}$

Sabemos que se verifica la ecuación: $a'.x = b'\ (m')$

Notemos que en esta nueva ecuación a' y m' son coprimas, por lo que existe el inverso modular de a' modulo m'.

Luego podemos reescribir: $x = b'(a')^{-1} = a''\ (m')$

A esta nueva ecuación la llamamos **ecuación normalizada.**
Podemos ver que la solución de esta ecuación es **única módulo m'**.

Esta ecuación se verifica si y solo si se verifica la ecuación original.

Estructura Modular-Equation

Para trabajar con ecuaciones modulares vamos a definirnos una estructura en C++, en la cual agregaremos una función “normalize()”.

```
#define mod(x, m) ((x%m+m)%m)
struct MEq{ /// a*x = b (m)
    ll a, b, m;
    MEq(ll a = 0, ll b = 0, ll m = 0): a(a), b(b), m(m){}
    bool normalize(){
        a = mod(a, m); b = mod(b, m);
        ll d = gcd(a, m); if(b%d)return false;
        a/=d; b/=d; m/=d;
        b = b*inv(a, m)%m; a = 1;
        return true;
    }
};
```

Destaquemos que toda ecuación modular consistente se puede normalizar.

Sistema de ecuaciones modulares

Llamamos un sistema de ecuaciones modulares a un conjunto de ecuaciones modulares que deben cumplirse en simultáneo:

$$\begin{cases} a_1.x = b_1(m_1) \\ a_2.x = b_2(m_2) \\ \quad\quad \dots \\ a_k.x = b_k(m_k) \end{cases}$$

La solución a este sistema es el conjunto de valores de x para los que se verifican todas las ecuaciones.

Nos ponemos como objetivo hallar las soluciones a un sistema.

Sistema de ecuaciones modulares normalizadas

Si **no existe** ninguna x que verifique el sistema decimos que el sistema es inconsistente.

Si alguna **ecuación** dentro del sistema es **inconsistente**, resulta que no existe una x que cumpla esa ecuación.
Por lo tanto el **sistema es inconsistente**.

Cuando llegamos a que el sistema es inconsistente terminamos.

Si no es así, podemos normalizar todas las ecuaciones y conseguir un **sistema de ecuaciones modulares normalizadas**.

Sistema de ecuaciones modulares normalizadas (cont.)

Un sistema de ecuaciones modulares normalizadas se ve:

$$\begin{cases} x = a_1 (m_1) \\ x = a_2 (m_2) \\ \quad \dots \\ x = a_k (m_k) \end{cases}$$

Lo siguiente que veremos es como hallar las soluciones a estos sistemas.

El Teorema Chino del Resto

Supongan que tenemos un sistema de ecuaciones modulares normalizadas.

$$\begin{cases} x = a_1 (m_1) \\ x = a_2 (m_2) \\ \quad \dots \\ x = a_k (m_k) \end{cases}$$

Tal que para cada par (m_i, m_j) con i $\neq j$: $\quad \boldsymbol{mcd(m_i, m_j) = 1}$

El **Teorema Chino del Resto** nos dice que este sistema tiene **exactamente** una solución en el intervalo $[0, M)$, donde $\boldsymbol{M}$ es la multiplicación de todos los $\boldsymbol{m_i}$.

Sistema de 2 ecuaciones de modulos coprimos

Supongamos que tenemos un sistema de 2 ecuaciones con módulos coprimos.

$$\begin{cases} x = a_1 (m_1) \\ x = a_2 (m_2) \end{cases}$$

El Teorema Chino del Resto nos dice que hay exactamente una solución al sistema en el intervalo $[0, m_1 m_2)$, llamemos **A** a esta solución.

Sistema de 2 ecuaciones de modulos coprimos (cont.)

Existe un método para hallar A.

Con el método de **Euclides Extendido** podemos hallar dos enteros $\boldsymbol{n_1}$ y $\boldsymbol{n_2}$ que verifiquen:

$$m_1 n_1 + m_2 n_2 = 1$$

Definimos: $$\boldsymbol{x = a_1 m_2 n_2 + a_2 m_1 n_1}$$

Podemos demostrar algebraicamente que $\boldsymbol{x}$ verifica ambas ecuaciones y por lo tanto es solución al sistema.

Entonces podemos hallar el valor de A.

$$\boldsymbol{M = m_1 m_2,} \qquad \boldsymbol{A = (a_1 m_2 n_2 + a_2 m_1 n_1)\%M}$$

Sistema de 2 ecuaciones de modulos coprimos (cont.)

La solución a este sistema de 2 ecuaciones modulares, es efectivamente una ecuación modular.

$$M = m_1 m_2$$

$$A = (a_1 m_2 n_2 + a_2 m_1 n_1) \% M$$

$$x \equiv A(M)$$

Le ecuación resultado es equivalente al sistema original.

Sistema de 2 ecuaciones

¿Qué pasa cuando las ecuaciones no tienen modulos coprimos?

En este caso no tenemos un teorema que demuestre la existencia de una solución y el sistema podría ser inconsistente.

Un ejemplo de sistema inconsistente:

$$\begin{cases} x \equiv 3(4) \\ x \equiv 4(6) \end{cases}$$

Notamos por un lado que los modulos no son coprimos. Por lo tanto no se cumplen las hipótesis del Teorema Chino del Resto.

La primer indicación implica que x es impar, mientras que la segunda indica que x es par. Es claro que no existen soluciones.

Sistema de 2 ecuaciones inconsistente

Dado un sistema de 2 ecuaciones:

$$\begin{cases} x \equiv a_1 (m_1) \\ x \equiv a_2 (m_2) \end{cases}$$

Hallamos $d = mcd(m_1, m_2)$ y vemos que se verifican:

$$x \equiv a_1 (d)$$
$$x \equiv a_2 (d)$$

Por lo tanto, debe cumplirse:

$$a_1 \equiv a_2 (d)$$

Se puede demostrar que el sistema es consistente **si y solo si** se verifica esa ecuación.

Sistema de 2 ecuaciones consistente

Si ya sabemos que el sistema anterior es consistente, nos queda hallar la solución al sistema.
Las solución en este caso es una ecuación modular con módulo $M = mcm(m_1, m_2)$
No nos adentraremos en el desarrollo. Pero veremos la solución.

Existe una construcción muy similar a la del caso en que los módulos son coprimos.

Usamos el método de Euclides Extendido para hallar $\boldsymbol{n_1}$ y $\boldsymbol{n_2}$ que verifican:

$$\boldsymbol{m_1 n_1 + m_2 n_2 = d = mcd(m_1, m_2)}$$

Sistema de 2 ecuaciones consistente

Luego definimos el valor $\boldsymbol{A}$ como:

$$A = \left(\frac{m_1 n_1 (a_2 - a_1)}{d} + a_1 \right) \%M$$

Se puede demostrar algebraicamente que verifica ambas ecuaciones.
No lo vamos a hacer aquí.

Implementación

Recuerden que usamos la estructura vista previamente para Ecuaciones Modulares.

```
MEq Euge(MEq S, MEq T){
    ll x, y, d = euclid(S.m, -T.m, x, y);
    if((S.b - T.b)%d)return MEq();
    ll M = S.m * (T.m / d), r = (T.b - S.b)/d;
    x = mulmod(x, r, M);
    ll A = mulmod(S.m, x, M) + S.b%M; A = mod(A, M);
    return MEq(1, A, M);
}
```

Usamos mulmod para evitar overflow, ya que puede ser que trabajemos con modulos mayores a 10^9.

Solución de un sistema de ecuaciones

El método que acabamos de ver convierte un sistema de dos ecuaciones modulares en una ecuación modular equivalente.

Podemos usar esto para diseñar un método que reduzca un sistema de n ecuaciones, para cualquier $n \geq 2$, a una única ecuación.

Dado un sistema de n ecuaciones tomamos 2 de ellas y las reemplazamos por su equivalente.
Realizamos el proceso $n - 1$ veces y terminamos.

Decimos que la ecuación equivalente final es la solución al sistema de ecuaciones. El conjunto solución es el de todos los enteros x que la verifiquen.

Implementación

Una posible implementación de un algoritmo que resuelva un sistema de ecuaciones es la siguiente.

```
MEq Euge(vector<MEq> &V){
    queue<MEq> Q; forn(i, V.size())Q.push(V[i]);
    while(Q.size() > 1){
        MEq A = Q.front(); Q.pop();
        MEq B = Q.front(); Q.pop();
        MEq X = Euge(A, B); if(!X.m) return MEq();
        Q.push(X);
    }
    return Q.front();
}
```

Sin embargo hay implementaciones “mejores”, que se espera que sean más rápidas.

Fin
¡Gracias por escuchar!