Capítulo 1

Algoritmos básicos en Teoría de Números

Si no tenemos un punto fijo al que queremos llegar, damos vueltas y vueltas.

Luis Sepúlveda

1.1. Introducción

Un *algoritmo* es un método que soluciona un problema: dados datos de entrada (*input*) este arroja datos de salida (*output*) después de una cantidad de pasos.

Podemos clasificar los algoritmos en dos tipos: los *deterministas* y los *probabilístico*. La diferencia radica en el uso de generadores aleatorios para ejecutarlos. Esto es importante pues no disponemos de una fuente real de datos aleatorios en el mundo informático, sólo podemos simularlo. Además los algoritmos probabilístico, difieren con los deterministas en un a cuestión muy importante, existe la posibilidad (de medida 0) de que el algoritmo no termine con los datos que le fueron proporcionados. Sin embargo, la experiencia muestra que los algoritmos probabilísticos son más eficientes que los deterministas en muchos casos y a veces son los únicos disponibles.

Nota: Los algoritmos probabilísticos no deben de confundirse con los métodos que con una alta probabilidad producen un resultado correcto. Si llamamos a algo un algoritmo, quiere decir que este producirá un resultado correcto en un tiempo finito o infinito. Un ejemplo de método no algorítmico es el siguiente: Suponga que N es un número grande y usted sospecha que es un número primo. Usted puede calcular

$$2^{N-1} \pmod N$$

usando un algoritmo que vamos a ver más adelante. Si la congruencia no es cierta, entonces N no es primo por el teorema de Fermat. Sin embargo, si la congruencia es cierta, hay buenas posibilidades de que N sea primo, sin embargo esto no siempre es el caso, por ejemplo $N = 341$ es un contra ejemplo.

Ejercicio 1.1.1. Escriba un algoritmo que calcule el factorial de un número natural n .

Cuando se ejecuta un programa se gastan algunos recursos en el cómputo como: tiempo, espacio (memoria), entre otros. Una función de complejidad es una función $f : \mathbb{N} \rightarrow \mathbb{R}^+$ tal que para todos los casos cuyo *input* sea $\leq s$, podemos garantizar que menos de $f(s)$ recursos serán usados durante el cálculo del algoritmo. El tamaño del input debe de ser definido de acuerdo al problema. Una vez definido nuestro input nos restringiremos a medir el tiempo que se gasta en obtener un output, este tiempo está medido en la cantidad de operaciones de bits, i.e. operaciones lógicas o aritméticas en ceros y unos. El tamaño

de los datos de entrada de un algoritmo serán medidos también en el número de bits que ocupan. Por lo tanto un entero positivo N ocupa $\lfloor \log_2 N \rfloor + 1$ bits.

Definición 1.1.1. Recordemos algunos conceptos de análisis:

- Escribimos $f(s) = O(g(s))$ (igualmente $f \ll g$) si existen constantes s_0 y $C > 0$ que no dependen de s , tales que

$$|f(s)| \leq Cg(s) \quad \text{para toda } s > s_0.$$

- Si T es una lista de parámetros de nuestro problema distintos de s , escribimos $f = O_T(g)$ (igualmente $f \ll_T g$) si las constantes del inciso precedente pueden depender de los valores de los parámetros en T .

Ejemplo 1.1.1. Tenemos que $\ln(x) = O(x)$. Tomemos $x > 0$, es obvio que

$$x < e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!} = 1 + x + \frac{x^2}{2} + \dots$$

Por otra parte el logaritmo es una función no decreciente en $(0, \infty)$ (su derivada es siempre positiva en el dominio de la función) por lo que tomando el logaritmo en los dos lados de la desigualdad tenemos

$$\ln(x) < x \quad \text{para } x > 0,$$

en particular si $x \geq 1$ entonces $\ln(x) \geq 0$ por lo tanto si hacemos $C = 1$ y $s_0 = 1$ obtenemos que $\ln(x) = O(x)$.

Ejemplo 1.1.2. Si tenemos $f(s) = O_\varepsilon(s^{1+\varepsilon})$ para $\varepsilon > 0$, significa que para cada $\varepsilon > 0$ existen constantes $s_0(\varepsilon)$ y $C(\varepsilon)$ que dependen de ε tales que $f(s) \leq C(\varepsilon)s^{1+\varepsilon}$ para todo $s > s_0(\varepsilon)$.

Si s es el tamaño del problema y $f(s)$ un estimado del tiempo de ejecución entonces decimos que

- si $f(s) = O(s)$, el costo es lineal.
- si $f(s) = O(s^d)$ para algún d , que el costo es polinomial.
- si $f(s) = O(\exp(s^d))$, el costo es subexponencial si $d < 1$, sino decimos que es exponencial.

Por supuesto, un algoritmo es bueno, si el tiempo de ejecución aumenta de manera gentil cuando el tamaño del problema aumenta, si un algoritmo es subexponencial no está tan mal pero si es exponencial entonces es malo.

Nota: Tenemos que tener en cuenta que por la naturaleza de los algoritmos probabilísticos, la idea de calcular la complejidad (que en nuestro caso es el tiempo de ejecución) no tiene sentido. Por lo tanto esto es remplazado por una función de complejidad esperada.

En la mayoría de los algoritmos podemos dar una cota superior N del tamaño de los números que se manejarán. El tiempo de ejecución de una operación básica (suma o multiplicación) en $\mathbb{Z}$ tiene un costo de $O(\ln^2 N)$ operaciones en bits.

Ejercicio 1.1.2. Usando el hecho de que un número positivo N puede ser representado con $\lfloor \log_2 N \rfloor + 1$ bits, demuestre que la suma y la multiplicación de dos números entre 1 y N tiene un costo de $O(\ln^2 N)$.

1.2. Algoritmos de potencias

Es importante saber calcular potencias en un grupo de manera eficaz. Si g es un elemento de un grupo $(G, \times)$ y n es un entero, entonces calcular g^n implicaría hacer $|n| - 1$ multiplicaciones ingenuamente. Asimismo, $|n|$ podría ser muy grande, por ejemplo más grande que 10^{100} , por lo que el método ingenuo

sería muy costoso. No obstante, es posible calcular potencias de una manera más eficaz y este es el primer paso para entender la aritmética algorítmica.

La idea para calcular potencias de forma óptima es la siguiente. Escribimos n como

$$\pm 1 \times \sum_i \alpha_i 2^i \quad \text{con } \alpha_i \in \{0, 1\},$$

entonces tenemos

$$g^n = \prod_{i \text{ con } \alpha_i=1} (g^{\pm 1})^{2^i},$$

por lo que si $M = \lfloor \log_2 |n| \rfloor$ entonces necesitamos $M + 1$ multiplicaciones (i.e. el número de dígitos binarios de $|n|$) para calcular los distintos $(g^{\pm 1})^{2^i}$ para $i = 0, \dots, M$ y para obtener el producto, haremos a lo más M multiplicaciones. Para $|n|$ suficientemente grande tenemos $2M + 1 < |n| - 1$. Por lo tanto, usar un algoritmo así es más viable que hacer la multiplicación ingenuamente.

Algoritmo 1: Derecha-Izquierda Binario

Descripción: Dados $g \in G$ y $n \in \mathbb{Z}$, este algoritmo calcula g^n en G . Escribimos 1 para el elemento neutro de G .

- 1 [Iniciar] Hacer $y \leftarrow 1$. Si $n = 0$, regresar y y terminar. Si $n < 0$ sea $N \leftarrow -n$ y $z \leftarrow g^{-1}$. Sino, hacer $N \leftarrow n$ y $z \leftarrow g$.
 - 2 [Multiplicar?] Si N es impar entonces $y \leftarrow z \cdot y$.
 - 3 [Doblegar N] Hacer $N \leftarrow \lfloor N/2 \rfloor$. Si $N = 0$, regresar y como respuesta y terminar el algoritmo. Sino, hacer $z \leftarrow z \cdot z$ e ir al paso 2.
-

Ejercicio 1.2.1. ¿Cuánto tiempo lleva hacer m^n con $m, n \in \mathbb{Z}$?

Podemos hacer varios cambios en este algoritmo. Por ejemplo, podemos hacer la exploración de los dígitos binarios de $|n|$ de izquierda a derecha.

Algoritmo 2: Izquierda-Derecha Binario

Descripción: Dados $g \in G$ y $n \in \mathbb{Z}$, este algoritmo calcula g^n en G . Si $n \neq 0$ suponemos que sabemos cuál es el entero e tal que $2^e \leq |n| < 2^{e+1}$. Escribimos 1 para el elemento neutro de G .

- 1 [Iniciar] Si $n = 0$, regresar 1 y terminar. Si $n < 0$ sea $N \leftarrow -n$ y $z \leftarrow g^{-1}$. Sino, hacer $N \leftarrow n$ y $z \leftarrow g$. Finalmente hacer $y \leftarrow z$, $E \leftarrow 2^e$, $N \leftarrow N - E$.
 - 2 [Terminado?] Si $E = 1$, regresar y y terminar el algoritmo. Sino hacer $E = E/2$.
 - 3 [Multiplicar?] Hacer $y \leftarrow y \cdot y$ y si $N \geq E$, hacer $N \leftarrow N - E$ y $y \leftarrow y \cdot z$. Ir al paso 2.
-

Por ejemplo, si $n = 21$ tenemos $21 = 10101_2$ el algoritmo obedece la factorización siguiente:

$$g^{21} = g((g^4)^4).$$

Note que la parte importante del algoritmo se centra en cuándo un bit es 0 ó 1. Si por ejemplo, hacemos un programa `bit(N, f)` que nos diga el bit que corresponde a la posición f de N , con $f = 0$ siendo el bit correspondiente a la paridad del número N y por supuesto `bit(N, e)=1`; podríamos implementar el siguiente algoritmo:

Algoritmo 3: Izquierda-Derecha Binario usando Bits

Descripción: Dados $g \in G$ y $n \in \mathbb{Z}$, este algoritmo calcula g^n en G . Si $n \neq 0$ suponemos que sabemos cuál es el entero e tal que $2^e \leq |n| < 2^{e+1}$. Escribimos 1 para el elemento neutro de G .

- 1 [Iniciar] Si $n = 0$, regresar 1 y terminar. Si $n < 0$ sea $N \leftarrow -n$ y $z \leftarrow g^{-1}$. Sino, hacer $N \leftarrow n$ y $z \leftarrow g$. Finalmente hacer $y \leftarrow z$, $f \leftarrow e$.
 - 2 [Terminado?] Si $f = 0$, regresar y y terminar el algoritmo. Sino hacer $f = f - 1$.
 - 3 [Multiplicar?] Hacer $y \leftarrow y \cdot y$ y si `bit(N, f)=1`, hacer $y \leftarrow y \cdot z$. Ir al paso 2.
-

Ejercicio 1.2.2. Implemente la función `bit(N, f)` y el algoritmo precedente.

Los últimos dos algoritmos suelen ser muy ventajosos cuando trabajamos en el grupo $G = (\mathbb{Z}/m\mathbb{Z})^*$. Comparándolos con el algoritmo Derecha-Izquierda Binario, son 1.5 veces más rápidos en promedio.

1.3. Los algoritmos de Euclides

Euclides fue sin duda el padre de las matemáticas modernas (Euclides es el profeta matemático por excelencia, nacido antes que Jesús de Nazaret). En su libro Los Elementos, hace el primer tratado riguroso de distintas herramientas matemáticas. Los Elementos consta de 13 tomos, en los primeros se estudia la teoría de números y el álgebra. Entre las pruebas más legendarias está aquella de que los números primos son infinitos. Quizás esta demostración y muchas otras no sean de la autoría de Euclides, pero lo que sí podemos decir es que él se encargó de hacer un compendio que ha trascendido en la historia como uno de los libros fundamentales en matemáticas. El fin de los elementos de Euclides, en sus últimos tomos, es la construcción de los sólidos platónicos.

En esta parte de nuestro curso veremos el algoritmo de Euclides, quizás el primer algoritmo en la historia de las matemáticas. Este algoritmo calcula de una manera eficaz el máximo común divisor de dos enteros a y b . Ingenuamente, uno podría pensar que para obtener el máximo número que simultáneamente a y b uno debería de factorizar estos y luego ver cuáles son los factores comunes y elegir un exponente adecuado que divida a ambos. Sin embargo, el proceso de factorización es difícil y lento, si no fuera así la aritmética algorítmica no existiría.

Para un par de números enteros a y b escribimos (a, b) su máximo común divisor. Recordemos que $\mathbb{Z}$ es un anillo conmutativo, es decir es un grupo abeliano bajo la suma, y tenemos definida una multiplicación que también es abeliana que es compatible con la suma. Recordemos que por el teorema fundamental de la aritmética, todo entero se factoriza de manera única como el producto de una unidad por un producto único de potencias de números primos. El anillo de los números enteros, es entonces un dominio de factorización única.

En general, sabemos que todo dominio de ideales¹ principales DIP² es un dominio de factorización única DFU. En general el inverso no es cierto, pero en el caso de los dominios de Dedekind³ tenemos la siguiente equivalencia:

$$DFU \Leftrightarrow DIP.$$

Por lo tanto, el anillo $\mathbb{Z}$ es un dominio de ideales principales. Si consideramos el ideal generado por a y b , es decir $a\mathbb{Z} + b\mathbb{Z}$, como sabemos que todo ideal de $\mathbb{Z}$ es principal, la pregunta es ¿Qué elemento genera el ideal $a\mathbb{Z} + b\mathbb{Z}$? La respuesta es: el máximo común divisor (a, b) de a y b .

El gran común divisor entonces es el único generador no negativo del ideal de $\mathbb{Z}$ generado por a y b . En particular tenemos las siguientes propiedades: $(a, 0) = (0, a) = |a|$ y $(a, b) = (|a|, |b|)$. Por lo tanto siempre podemos suponer que a y b no son negativos.

Algoritmo 4: Euclides

Descripción: Dados dos enteros no negativos a y b , este algoritmo calcula su máximo común divisor.

- 1 [Trivial?] Si $a = 0$, entonces regresar b .
 - 2 [Pasito Euclidiano] Hacer $r \leftarrow b \bmod a$, $b \leftarrow a$, $a \leftarrow r$ e ir al paso 1.
-

La validez del algoritmo sigue de las siguientes propiedades. Si $a = b \cdot k + r$ entonces $(a, b) = (r, b) = (r, a)$, además $(a, b) = (b, a)$. por lo que el algoritmo cada vez usa números más chicos y eventualmente parará, la pregunta es ¿En cuántos pasos parará?

¹Un ideal I de un anillo R es un subgrupo abeliano cerrado por la multiplicación de elementos del anillo.

²Un anillo R se un dominio de ideales principales si todo ideal puede ser generado por un elemento de R , es decir si I es un ideal de R , entonces $I = (\alpha)$ para algún $\alpha \in R$.

³Un dominio de Dedekind es un dominio entero, noetheriano, donde todos sus ideales primos son maximales y algebraicamente cerrado en su campo de fracciones.

Teorema 1.3.1. *Supongamos que $a, b \in \{1, \dots, N\}$. Entonces*

1. *El número de pasitos Euclidianos es a lo más igual a*

$$\left\lceil \frac{\ln(\sqrt{5}N)}{\ln((1 + \sqrt{5})/2)} \right\rceil - 2 \approx 2.078 \ln N + 1.672.$$

2. *El número promedio de pasitos Euclidianos es aproximadamente igual*

$$\frac{12 \ln 2}{\pi^2} \ln N + 0.14 \approx 0.843 \ln N + 0.14.$$

Para más información consultar [?].

Nota: En los algoritmos pasados de potencias, asumíamos que trabajábamos con un grupo G y de hecho los algoritmos funcionaban con varios grupos por ejemplo con los polinomios, las matrices, etc. Ahora haga un cálculo con el algoritmo de Euclides precedente, por ejemplo con los polinomios $x^2 - 1$ y $x^2 + 2x + 1$, su máximo común divisor es el polinomio $x + 1$, sin embargo el algoritmo arroja $2x + 2$. ¿Por qué se da este fenómeno?

Existe una variante del Algoritmo de Euclides que es bastante práctica pues no utiliza largas divisiones salvo al inicio, en vez de divisiones utiliza sólo divisiones por 2 que en código binario se traduce en desplazamiento de dígitos. El número de pasos aumenta pero las operaciones son mucho más rápidas y por lo tanto es una ganancia cuando se utilizan enteros muy grandes.

Algoritmo 5: M.C.D. Binario

Descripción: Dados dos enteros no negativos a y b , este algoritmo calcula su máximo común divisor.

- 1 [Reducir tamaño] Si $a < b$, intercambiar a y b . Si $b = 0$ regresar a y terminar el algoritmo. Sino hacer $r \leftarrow a \bmod b$, $a \leftarrow b$ y $b \leftarrow r$.
 - 2 [Calcular potencias de 2] Si $b = 0$ regresar a y terminar el algoritmo. Sino hacer $k \leftarrow 0$, y mientras a y b sean ambos enteros hacer $k \leftarrow k + 1$, $a \leftarrow a/2$ y $b \leftarrow b/2$.
 - 3 [Eliminar las potencias restantes de 2] Si a es par, repetir $a \leftarrow a/2$ hasta que a sea impar. Sino, si b es par, repetir $b \leftarrow b/2$ hasta que b sea impar.
 - 4 [Restar] (Aquí a y b son ambos impares.) Hacer $t \leftarrow (a - b)/2$. Si $t = 0$, regresar $2^k a$ y terminar el algoritmo.
 - 5 [Ciclo] Mientras t sea par hacer $t \leftarrow t/2$. Luego si $t > 0$ hacer $a \leftarrow t$, sino $b \leftarrow -t$ e ir al paso 4.
-

Ejercicio 1.3.1. Verifique la validez del algoritmo.

Nota:

1. El algoritmo precedente es especialmente bueno para calcular el M.C.D. de números muy grandes. Esto es porque no hace divisiones salvo en el primer paso. Por lo tanto se sugiere entonces su uso cuando estemos en la situación ya descrita.
2. Todas las divisiones por 2 hechas por este algoritmo son desplazamiento de dígitos binarios. En particular es deseable que el algoritmo sea programado en el lenguaje más básico posible para poder optimizar la velocidad del algoritmo al máximo posible.
3. Uno podría directamente empezar el algoritmo binario en el paso 2, evitando hacer la primera división. Sin embargo es preferible bajar un poco la magnitud de los números que estamos manejando haciendo más homogéneo el sistema que manejamos, utilizando números menores iguales al más pequeño de a y b .

1.3.1. El algoritmo de Euclides extendido

Sabemos que si $d = (a, b)$ entonces existen enteros u, v tales que $au + bv = d$. Por lo que nos gustaría extender el algoritmo de Euclides y poder calcular un par (u, v) que satisfaga la ecuación precedente. Aunque el par (u, v) no es único, sí lo son $\bmod b/d$ y $\bmod a/d$ respectivamente.

Hay dos maneras de hacer esto. La primera, es la manera usual de ir registrando los coeficientes de las divisiones euclidianas y luego ir hacia atrás. Este es el método más eficiente pero puede requerir mucha memoria. El otro método, requiere menos memoria aunque es un poco más lento. Además requiere usar variables auxiliares para hacer los cálculos conforme se ejecuta el algoritmo.

Algoritmo 6: Euclides Extendido

Descripción: Dados dos enteros no negativos a y b , este algoritmo determina (u, v, d) tal que $au + bv = d$ y $d = (a, b)$. Usamos las variables auxiliares v_1, v_3, t_1, t_3 .

- 1 [Iniciar] Hacer $u \leftarrow 1, d \leftarrow a$. Si $b = 0$, hacer $v \leftarrow 0$ y terminar el algoritmo, sino hacer $v_1 \leftarrow 0$ y $v_3 \leftarrow b$.
 - 2 [Terminado?] Si $v_3 = 0$ hacer $v \leftarrow (d - au)/b$ y terminar el algoritmo.
 - 3 [Pasito Euclidiano] Sea $q \leftarrow \lfloor d/v_3 \rfloor$ y simultáneamente $t_3 \leftarrow d \bmod v_3$. Luego hacer $t_1 \leftarrow u - qv_1, u \leftarrow v_1, d \leftarrow v_3, v_1 \leftarrow t_1, v_3 \leftarrow t_3$ e ir al paso 2.
-

Nota:

- Si introducimos las variables t_2, v_2, v y pedimos que las siguientes relaciones se cumplan cada vez que comenzamos el paso 2:

$$at_1 + bt_2 = t_3, \quad au + bv = d, \quad av_1 + bv_2 = v_3,$$

sin que sea necesario iniciar las variables t_1, t_2, t_3 . Entonces es claro que las sustituciones del paso 3 y las sustituciones $v_2 \leftarrow t_2, t_2 \leftarrow v - qv_2, v \leftarrow v_2$ hacen que las relaciones sigan siendo válidas. Además es claro que al final del algoritmo d contiene el máximo común divisor y que $au + bv = d$.

- Realmente este algoritmo es una extensión del algoritmo de Euclides, sólo que estamos calculando el coeficiente de las relaciones que cumple a en cada paso y actualizándolo. Así al final tenemos u y d , por lo que el paso 2 cuando el algoritmo termina nos da el valor de v .
- Podemos extender de manera similar el algoritmo M.C.D. Binario 5. Para más sobre esto consulte el libro de Cohen [Coh93].

El algoritmo extendido de Euclides, es muy útil en muchos contextos. Por ejemplo, imaginemos que queremos saber si un número b es invertible módulo m y en caso de que lo sea encontrar su inverso. En este caso, aplicando el algoritmo podemos encontrar (u, v, d) tales que

$$ub + vm = d,$$

si $d > 0$ entonces b no es invertible. Si por el contrario, $d = 1$ tenemos que el inverso de b es u . Hay otras maneras de calcular $b^{-1} \bmod m$ cuando la factorización de m es conocida. Por ejemplo, si m es primo, por el teorema de Fermat tenemos que

$$b^{m-1} \equiv 1 \pmod{m},$$

por lo tanto $b^{-1} = b^{m-2} \bmod m$ y por los algoritmos de potencias que vimos este número es fácil de calcular. En general si $(b, m) = 1$ tenemos que

$$b^{\varphi(m)} \equiv 1 \pmod{m},$$

donde $\varphi(m)$ es la función φ de Euler. Entonces, el inverso de b módulo m puede ser obtenido calculando

$$b^{-1} = b^{\varphi(m)-1} \pmod{m}.$$

1.3.2. El Teorema Chino del Residuo

El siguiente teorema es un clásico en la teoría de números:

Teorema 1.3.2 (Teorema Chino del Residuo). Sean $m_1, \dots, m_k$ y $x_1, \dots, x_k$ enteros. Supongamos que para cada par (i, j) tenemos

$$x_i \equiv x_j \pmod{\text{mcd}(m_i, m_j)}.$$

Entonces existe un entero x tal que $x \equiv x_i \pmod{m_i}$ para $1 \leq i \leq k$. Además, x es único módulo el mínimo común múltiplo de $m_1, \dots, m_k$.

Demostración: El **caso** $k = 1$ es obvio.

Veamos el **caso** $k = 2$. Sean m_1, m_2, x_1, x_2 como en la hipótesis, entonces existen u, v tales que

$$um_1 + vm_2 = d = \text{mcd}(m_1, m_2)$$

en particular

$$1 = u \frac{m_1}{d} + v \frac{m_2}{d}.$$

Sea $x = x_2 u \frac{m_1}{d} + x_1 v \frac{m_2}{d}$, entonces

$$\begin{aligned} x = x_2 u \frac{m_1}{d} + x_1 v \frac{m_2}{d} &= \underbrace{(x_1 + kd)}_{x_2 \equiv x_1 \pmod{d}} u \frac{m_1}{d} + x_1 v \frac{m_2}{d} \\ &= (x_1 + kd) u \frac{m_1}{d} + x_1 v \frac{m_2}{d} \\ &= x_1 \left(u \frac{m_1}{d} + v \frac{m_2}{d} \right) + 2kd \\ &= x_1 + kd; \end{aligned}$$

de donde concluimos que $x \equiv x_1 \pmod{d}$. Un cálculo similar nos da que $x \equiv x_2 \pmod{d}$.

Supongamos que es cierto para el **caso** $k = n$.

Veamos el **caso** $k = n + 1$. Sean $m_1, m_2, \dots, m_n, m_{n+1}, x_1, x_2, \dots, x_n, x_{n+1}$ como en la hipótesis. Sea X una solución para el problema $k = n$, es decir

$$X \equiv x_i \pmod{m_i} \quad \text{para } i = 1, \dots, n.$$

Supongamos que Y es una solución para el problema $k = n + 1$, entonces

$$Y \equiv x_i \pmod{m_i} \quad \text{para } i = 1, \dots, n, n + 1,$$

en particular como Y también es solución para el problema con $k = n$, por la unicidad (probada más adelante), tenemos que

$$Y \equiv X \pmod{\text{mcm}(m_1, \dots, m_n)}.$$

Además Y tendría que satisfacer:

$$Y \equiv x_{n+1} \pmod{m_{n+1}}.$$

Sea $M = \text{mcm}(m_1, \dots, m_n)$ y $D = \text{mcd}(M, m_{n+1})$, entonces existen enteros u y v tales que

$$uM + vm_{n+1} = D,$$

entonces

$$u \frac{M}{D} + v \frac{m_{n+1}}{D} = 1.$$

Sea $Y = ux_{n+1} \frac{M}{D} + vX \frac{m_{n+1}}{D}$. Para $i = 1, \dots, n$ sea $d_i = \text{mcd}(m_{n+1}, m_i)$, entonces tenemos

$$\begin{aligned} Y = ux_{n+1} \frac{M}{D} + vX \frac{m_{n+1}}{D} &= u \underbrace{(x_i + k_i d_i)}_{x_{n+1} \equiv x_i \pmod{d_i}} \frac{M}{D} + vX \frac{m_{n+1}}{D} \\ &= ux_i \frac{M}{D} + vX \frac{m_{n+1}}{D} + k_i d_i \frac{M}{D} \\ &\equiv ux_i \frac{M}{D} + vx_i \frac{m_{n+1}}{D} + k_i \underbrace{d_i \frac{M}{D}}_{\text{divisible por } m_i} \pmod{m_i} \\ &\equiv x_i \underbrace{\left(u \frac{M}{D} + v \frac{m_{n+1}}{D} \right)}_{=1} \pmod{m_i} \\ &\equiv x_i \pmod{m_i}. \end{aligned}$$

Proposición 1.3.1. $m_i \mid d_i \frac{M}{D}$.

Demostración de la proposición: Sea $\widehat{M}$ el entero tal que $M = \text{mcm}(m_i, \widehat{M})$, i.e.

$$\widehat{M} = \text{mcm}(m_1, \dots, m_{i-1}, \widehat{m}_i, m_{i+1}, \dots, m_n).$$

Entonces

$$\begin{aligned} d_i \frac{M}{D} &= d_i \frac{m_i \widehat{M}}{(m_i, \widehat{M}) \left(\frac{m_i \widehat{M}}{(m_i, \widehat{M})}, m_{n+1} \right)} \\ &= d_i \frac{m_i \widehat{M}}{(m_i, \widehat{M}) (m_i, m_{n+1}) \left(\frac{\widehat{M}}{(m_i, \widehat{M})}, m_{n+1} \right)} \\ &= \frac{m_i \widehat{M}}{(m_i, \widehat{M}) \left(\frac{\widehat{M}}{(m_i, \widehat{M})}, m_{n+1} \right)} \\ &= \frac{m_i \widehat{M}}{\left(\widehat{M}, (m_i, \widehat{M}) m_{n+1} \right)}, \end{aligned}$$

y el resultado sigue fácilmente del hecho que $\frac{\widehat{M}}{\left(\widehat{M}, (m_i, \widehat{M}) m_{n+1} \right)}$ es un entero. □

Finalmente, también tenemos que

$$Y = ux_{n+1} \frac{M}{D} + vX \frac{m_{n+1}}{D} \equiv x_{n+1} \pmod{m_{n+1}}.$$

Unicidad: Supongamos que y es tal que $y \equiv x_i \pmod{m_i}$ para todo $i = 1, \dots, m_k$. Entonces $x - y \equiv 0 \pmod{m_i}$ para todo $i = 1, \dots, m_k$, esto implica que $x - y \equiv 0 \pmod{\text{mcm}(m_1, \dots, m_k)}$. ■

Corolario 1.3.1. Sean $m_1, \dots, m_k$ enteros coprimos a pares, es decir tales que $\text{mcd}(m_i, m_j) = 1$ para $i \neq j$. Entonces, para cualesquiera enteros x_i , existe un entero x , único módulo $\prod m_i$, tal que

$$x \equiv x_i \pmod{m_i} \quad \text{para } 1 \leq i \leq k.$$

Queremos un algoritmo para calcular x . Consideraremos sólo el caso cuando los m_i son coprimos a pares. Sea $M = \prod_{1 \leq i \leq k} m_i$ y $M_i = M/m_i$. Como los m_i son coprimos a pares, $(M_i, m_i) = 1$ entonces por el algoritmo extendido de Euclides podemos encontrar a_i tal que $a_i M_i \equiv 1 \pmod{m_i}$. Si hacemos

$$x = \sum_{1 \leq i \leq k} a_i M_i x_i,$$

es claro que x satisface las condiciones requeridas. Entonces, podemos regresar $x \pmod{M}$ como el resultado.

Podemos implementar este método explícitamente, sin embargo queremos mejorarlo un poco antes de hacerlo. Observe que las constantes necesarias a_i son pequeñas, de hecho menores que m_i , pero los M_i o los productos $a_i M_i$ que son necesarios pueden ser muy grandes. Hay un modo muy ingenioso para

evitar tener números así de grandes, y este es la base del siguiente algoritmo.

Algoritmo 7: Algoritmo Chino

Descripción: Dados enteros m_i con $1 \leq i \leq k$, coprimos a pares y enteros x_i , este algoritmo encuentra un entero x tal que $x \equiv x_i \pmod{m_i}$ para todo i .

- 1 [Iniciar] Sea $j \leftarrow 2$, $C_1 \leftarrow 1$. Además, si no es muy costoso reordenar los m_i (y por lo tanto los x_i) de modo que estén en orden creciente, i.e. $m_i < m_{i+1}$.
- 2 [Pre-calcular] Hacer $p \leftarrow m_1 m_2 \cdots m_{j-1} \pmod{m_j}$. Calcular (u, v, d) tales que $up + vm_j = d = \text{mcd}(p, m_j)$, i.e. aplique el algoritmo extendido de Euclides. Si $d > 1$ regresar un mensaje de error (los m_i no son coprimos a pares). Sino, hacer $C_j \leftarrow u$, $j \leftarrow j + 1$, e ir al paso 2 si $j \leq k$.
- 3 [Calcular constantes auxiliares] Hacer $y_1 \leftarrow x_1 \pmod{m_1}$, y para $j = 2, \dots, k$ calcular

$$y_j \leftarrow (x_j - (y_1 + m_1(y_2 + m_2(y_3 + \cdots + m_{j-2}y_{j-1}) \cdots)))C_j \pmod{m_j}.$$

- 4 [Terminar] Regresar $x \leftarrow y_1 + m_1(y_2 + m_2(y_3 + \cdots + m_{k-1}y_k) \cdots)$.
-

Nota:

- Los pasos 1 y 2 son un pre-cálculo que se necesita hacer sólo una vez cuando los m_i estén fijos y los x_i varíen.
- Al final del algoritmo tendremos $0 \leq x < M = \prod m_i$.

Un algoritmo de este tipo para calcular x con los m_i no necesariamente coprimos a pares, sería un poco complejo de realizar, en este caso es mejor usar la fórmula de inducción que utilizamos en la prueba del Teorema 1.3.2. El algoritmo siguiente es un modelo de cómo hacer dicho algoritmo. En este caso se supone que los m_i con coprimos nuevamente, pero una adaptación sencilla nos da el algoritmo general.

Algoritmo 8: Algoritmo Chino Inductivo

Descripción: Dados enteros m_i con $1 \leq i \leq k$, coprimos a pares y enteros x_i , este algoritmo encuentra un entero x tal que $x \equiv x_i \pmod{m_i}$ para todo i .

- 1 [Iniciar] Hacer $i \leftarrow 1$, $m \leftarrow m_1$, $x \leftarrow x_1$.
 - 2 [Terminado?] Si $i = k$ regresar x y terminar el algoritmo. Sino, hacer $i = i + 1$, y con el Algoritmo de Euclides extendido calcular u y v tales que $um + vm_i = 1$.
 - 3 [Calcular el siguiente x] Hacer $x \leftarrow umx_i + vm_i x$, $m \leftarrow mm_i$, $x \leftarrow x \pmod{m}$ e ir al paso 2.
-

Ejercicio 1.3.2. Implemente el algoritmo Chino Inductivo en el caso en que los m_i no sean coprimos a pares.

1.3.3. El símbolo de Legendre

Los grupos $(\mathbb{Z}/n\mathbb{Z})^*$

Si A es un anillo conmutativo con unidad, el conjunto de los elementos invertibles de A bajo la multiplicación forman un grupo denotado A^* y llamado las unidades de A . En el caso en que A es un campo entonces $A^* = A - \{0\}$ y viceversa, si podemos *dividir* por todos los elementos no nulos, entonces A es un campo. El siguiente teorema nos da la estructura de $(\mathbb{Z}/n\mathbb{Z})^*$.

Teorema 1.3.3. *Tenemos*

$$|(\mathbb{Z}/n\mathbb{Z})^*| = \varphi(n) = n \prod_{p|n} \left(1 - \frac{1}{p}\right),$$

y más precisamente

$$(\mathbb{Z}/n\mathbb{Z})^* \simeq \prod_{p^\alpha || n} (\mathbb{Z}/p^\alpha \mathbb{Z})^*,$$

donde

$$(\mathbb{Z}/p^\alpha\mathbb{Z})^* = \begin{cases} \mathbb{Z}/(p-1)p^{\alpha-1}\mathbb{Z} & \text{si } p \geq 3 \text{ o } p = 2 \text{ y } \alpha \leq 2, \\ \mathbb{Z}/2\mathbb{Z} \times \mathbb{Z}/2^{\alpha-2}\mathbb{Z} & \text{si } p = 2 \text{ y } \alpha \geq 3. \end{cases}$$

Observemos que la última línea del teorema nos dice que $(\mathbb{Z}/n\mathbb{Z})^*$ es el producto de grupos cíclicos. En el caso de que $(\mathbb{Z}/n\mathbb{Z})^*$ sea cíclico, un elemento de g que lo genere $(\mathbb{Z}/n\mathbb{Z})^*$ se llamará una raíz primitiva módulo n . Recordemos que el orden de un elemento g de un grupo es el menor entero positivo n tal que g^n es igual a la identidad. Cuando un grupo es finito, el orden de cualquier elemento divide el orden del grupo (Teorema de Lagrange). Por lo tanto, g es una raíz primitiva de $(\mathbb{Z}/n\mathbb{Z})^*$ si y solamente si su orden es igual a $\varphi(n)$. Los siguientes resultados son corolarios de las discusiones precedentes.

Proposición 1.3.2. ■ (Fermat) Si p es primo y a no es divisible por p , entonces tenemos que

$$a^{p-1} \equiv 1 \pmod{p}.$$

■ (Euler) Más generalmente, si n es un entero positivo, entonces para cualquier entero a coprimo a n tenemos

$$a^{\varphi(n)} \equiv 1 \pmod{n},$$

e incluso

$$a^{\varphi(n)/2} \equiv 1 \pmod{n}$$

si n es diferente de $2, 4, p^\alpha$ o $2p^\alpha$ con p un primo impar.

El siguiente algoritmo calcula el orden de un elemento g de un grupo finito G .

Algoritmo 9: Orden de un elemento

Descripción: Dado un grupo finito G de cardinalidad $h = |G|$, y un elemento $g \in G$, este algoritmo calcula el orden de $g \in G$. Denotamos por 1 el elemento identidad de G .

- 1 [Iniciar] Calcular la factorización en primos de h , es decir $h = p_1^{v_1} p_2^{v_2} \cdots p_k^{v_k}$, y hacer $e \leftarrow h$, $i \leftarrow 0$.
 - 2 [Siguiente p_i] Hacer $i \leftarrow i + 1$. Si $i > k$, regresar e y terminar. Sino, hacer $e \leftarrow e/p_i^{v_i}$, $g_1 \leftarrow g^e$.
 - 3 [Calcular el orden local] Mientras $g_1 \neq 1$, hacer $g_1 \leftarrow g_1^{p_i}$ y $e \leftarrow e \cdot p_i$. Ir al paso 2.
-

Ya que este algoritmo usa la factorización de h , esto puede acarrear dificultades cuando el orden del grupo es muy largo.

Sea p un primo. Para encontrar una raíz primitiva módulo p parece que no hay mejor manera para hacerlo que la siguiente: Probar $g = 2$, $g = 3$, etc. hasta que g sea una raíz primitiva. Uno debe de evitar hacer cálculos innecesarios, por ejemplo si $g = g_0^k$, entonces si g es una raíz primitiva, es lo mismo para g_0 .

Para saber cuándo g es una raíz primitiva, podemos calcular el orden de g usando el algoritmo pasado. Sin embargo es mejor proceder con el siguiente algoritmo.

Algoritmo 10: Raíz primitiva

Descripción: Dado un primo impar p , el algoritmo encuentra una raíz primitiva módulo p .

- 1 [Iniciar a] Hacer $a \leftarrow 1$, y factorizar $p - 1 = p_1^{v_1} p_2^{v_2} \cdots p_k^{v_k}$.
 - 2 [Control inicial] Hacer $a \leftarrow a + 1$ e $i \leftarrow 1$.
 - 3 [Verificar p_i] Calcular $e \leftarrow a^{p-1/p_i}$. Si $e = 1$ ir al paso 2. Sino hacer $i \leftarrow i + 1$.
 - 4 [Terminado?] Si $i > k$ regresar a y terminar el algoritmo, Sino, ir al paso 3.
-

Si n no es primo, pero es tal que existe una raíz primitiva módulo n (es decir el grupo multiplicativo $(\mathbb{Z}/n\mathbb{Z})^*$ es cíclico), podríamos modificar los algoritmos anteriores y usarlos para encontrar la raíz primitiva. Sin embargo es más factible proceder en el siguiente modo.

Primero, si $n = 2$ o $n = 4$, $g = n - 1$ es una raíz primitiva. Cuando $n = 2^a$ es una potencia de 2 con $a \geq 3$, $(\mathbb{Z}/n\mathbb{Z})^*$ ya no es cíclico (ver 1.3.3), pero es isomorfo a un producto de $\mathbb{Z}/2\mathbb{Z}$ con un grupo cíclico de orden 2^{a-2} . Entonces $g = 5$ es siempre un generador de este grupo cíclico.

Cuando $n = p^a$ es una potencia de un primo impar, con $a \geq 2$, entonces usamos el lema siguiente.

Lema 1.3.1. *Sea p un número primo impar, y sea g una raíz primitiva módulo p . Entonces g o $g + p$ es una raíz primitiva módulo toda potencia de p .*

La demostración de este lema se puede consultar en [Coh93, Lema 1.4.5].

Por lo tanto, para encontrar una raíz primitiva módulo p^a para p primo y $a \geq 2$, podemos hacer el siguiente procedimiento: calcular una raíz primitiva g módulo p usando el algoritmo 11, después calcular $g_1 = g^{p-1} \pmod{p^2}$. Si $g_1 \neq 1$, entonces g es una raíz primitiva módulo p^a para todo a , sino, $g + p$ lo es.

Finalmente, notemos que si p es un primo impar, entonces g es una raíz primitiva módulo p^a . Entonces g o $g + p^a$ es una raíz primitiva módulo $2p^a$.

El símbolo de Legendre-Jacobi-Kronecker

Sea p un primo impar. Entonces es fácil de ver que dado un entero a , la congruencia

$$x^2 \equiv a \pmod{p}$$

puede no tener solución, puede tener una solución si $a \equiv 0 \pmod{p}$, o puede tener dos soluciones, en tal caso decimos que a es un residuo cuadrático módulo p .

Definimos el símbolo de Legendre $\left(\frac{a}{p}\right)$ como -1 si la congruencia anterior no tiene solución, 0 si $a \equiv 0 \pmod{p}$ y 1 si a es un residuo cuadrático. El número de las soluciones módulo p para la congruencia anterior es $1 + \left(\frac{a}{p}\right)$. Además, uno puede fácilmente ver que este símbolo tiene las siguientes propiedades.

Proposición 1.3.3. *1. El símbolo de Legendre es multiplicativo, i.e.*

$$\left(\frac{a}{p}\right) \left(\frac{b}{p}\right) = \left(\frac{ab}{p}\right).$$

2. Tenemos la congruencia

$$a^{(p-1)/2} \equiv \left(\frac{a}{p}\right) \pmod{p}.$$

3. Existen tantos residuos cuadráticos como no residuos cuadráticos módulo p , y cada uno de estos son $(p-1)/2$.

El símbolo de Legendre es fundamental en muchos problemas. Así que tenemos que estudiar los métodos para calcularlo. Una idea es usar la congruencia $a^{(p-1)/2} \equiv \left(\frac{a}{p}\right) \pmod{p}$. Usando un buen algoritmo para calcular potencias, podemos hacer este cálculo en tiempo $O(\ln^3 p)$. Esto se puede mejorar usando la reciprocidad cuadrática, que es en sí un resultado fundamental en teoría de números.

Teorema 1.3.4. *Sea p un primo impar, entonces*

1.

$$\left(\frac{-1}{p}\right) \equiv (-1)^{(p-1)/2}, \quad \left(\frac{2}{p}\right) = (-1)^{(p^2-1)/8}.$$

2. Si q es un primo impar diferente de p , entonces tenemos la siguiente ley de reciprocidad:

$$\left(\frac{p}{q}\right) \left(\frac{q}{p}\right) = (-1)^{(p-1)(q-1)/4}.$$

La prueba puede ser consultada en cualquier buen libro introductorio de teoría de números.

Este teorema nos ayuda a calcular el símbolo de Legendre, ya que $\left(\frac{a}{p}\right)$ es multiplicativo en a y depende solamente de $a \pmod{p}$. Usar directamente el teorema de reciprocidad 1.3.4 implicaría tener que factorizar un entero, lo cual es una tarea en general difícil, afortunadamente con el siguiente teorema, que es una extensión del precedente, podemos evitar este teorema.

Definición 1.3.1. Definimos el símbolo de Kronecker $\left(\frac{a}{b}\right)$ para todo a, b en $\mathbb{Z}$ de la siguiente manera:

1. Si $b = 0$, entonces $\left(\frac{a}{0}\right) = 1$ si $a = \pm 1$, en otro caso $\left(\frac{a}{0}\right)$ es igual a 0.
2. Para $b \neq 0$, escribimos $b = \prod p$, donde los p son primos no necesariamente distintos, incluyendo 2, o $p = -1$ si b es negativo. Entonces definimos

$$\left(\frac{a}{b}\right) = \prod \left(\frac{a}{p}\right),$$

donde $\left(\frac{a}{p}\right)$ es el símbolo de Legendre definido antes para $p > 2$, y donde definimos

$$\left(\frac{a}{2}\right) = \begin{cases} 0, & \text{si } a \text{ es par} \\ (-1)^{(a^2-1)/8}, & \text{si } a \text{ es impar.}; \end{cases}$$

y también

$$\left(\frac{a}{-1}\right) = \begin{cases} 1, & \text{si } a \geq 0 \\ -1, & \text{si } a < 0. \end{cases}$$

Definición 1.3.2. Una función f es periódica de período b , si para todo x

$$f(x+b) = f(x).$$

Ejemplo 1.3.1. Las funciones trigonométricas seno y coseno, son periódicas de período 2π .

De las propiedades del símbolo de Legendre, y en particular de la reciprocidad cuadrática, podemos demostrar las siguientes propiedades del símbolo de Kronecker.

Teorema 1.3.5. 1. $\left(\frac{a}{b}\right) = 0$ si y sólo si $(a, b) \neq 1$.

2. Para todo a, b y c tenemos

$$\left(\frac{ab}{c}\right) = \left(\frac{a}{b}\right) \left(\frac{b}{c}\right), \quad \left(\frac{a}{bc}\right) = \left(\frac{a}{b}\right) \left(\frac{a}{c}\right) \quad \text{si } bc \neq 0.$$

3. Si fijamos $b > 0$, el símbolo $\left(\frac{a}{b}\right)$ es periódico en a de período b si $b \not\equiv 0 \pmod{4}$, sino es periódico de período $4b$.
4. Si fijamos $a \neq 0$, el símbolo $\left(\frac{a}{b}\right)$ es periódico en b de período $|a|$ si $a \equiv 0$ o 1 módulo 4 , de otro modo es periódico de período $4|a|$.
5. Las fórmulas del teorema de reciprocidad 1.3.4 si p y q son enteros positivos impares, no necesariamente primos.

Una consecuencia del teorema precedente es que con él podemos diseñar un algoritmo para calcular los símbolos de Legendre, y más generalmente los de Kronecker.

Algoritmo 11: Kronecker

Descripción: Dados $a, b \in \mathbb{Z}$, este algoritmo calcula el símbolo de Kronecker $\left(\frac{a}{b}\right)$ (por lo tanto el símbolo de Legendre cuando b es un primo impar).

- 1 [Verificar si $b = 0$] Si $b = 0$ entonces regresar 0 si $|a| \neq 1$, 1 si $|a| = 1$ y terminar el algoritmo.
- 2 [Remover los 2's de b] Si a y b son ambos pares, regresar 0 y terminar el algoritmo. Sino, hacer $v \leftarrow 0$ y mientras b sea par hacer $v = v + 1$ y $b \leftarrow b/2$. Luego si v es par hacer $k \leftarrow 1$, sino hacer $k \leftarrow (-1)^{(a^2-1)/8}$ (por inspección, no calculando directamente $(a^2 - 1)/8$). Finalmente si $b < 0$ haga $b \leftarrow -b$, y si además $a < 0$, hacer $k \leftarrow -k$.
- 3 [Terminado?] (Aquí b es impar y $b > 0$.) Si $a = 0$ entonces regresar 0 si $b > 1$, y k si $b = 1$, y terminar el algoritmo. Sino, hacer $v \leftarrow 0$ y mientras a sea par hacer $v \leftarrow v + 1$ y $a \leftarrow a/2$. Si v es impar hacer $k \leftarrow (-1)^{(b^2-1)/8}k$.
- 4 [Aplicar reciprocidad] Hacer

$$k \leftarrow (-1)^{(a-1)(b-1)/4}k,$$

(usando múltiples `if` y no multiplicando), y luego hacer $r \leftarrow |a|$, $a \leftarrow b \pmod{r}$, $b \leftarrow r$ y vuelva al paso 3.

Apéndice A

PARI/GP

A.1. ¿Qué es PARI/GP?

PARI/GP es un sistema de álgebra computacional muy utilizado, diseñado para cálculos rápidos en Teoría de Números (factorización, Teoría Algebraica de Números, curvas elípticas,...) pero que también incluye un gran número de otras funciones útiles para operar con objetos matemáticos como matrices, polinomios, series de potencias, números algebraicos, etc., además de muchas funciones trascendentes. PARI también se puede obtener como una colección de rutinas en C para mayor rapidez de cálculo.

A.2. Operaciones básicas

Para abrir PARI/GP (en linux) basta abrir una terminal y escribir `gp` en la consola. Un texto como el siguiente aparecerá:

```
GP/PARI CALCULATOR Version 2.10.0 (development 19770-eda143c)
  amd64 running linux (x86-64/GMP-5.1.3 kernel) 64-bit version
compiled: Nov 10 2016, gcc version 4.8.4 (Ubuntu 4.8.4-2ubuntu1~14.04)
      threading engine: single
      (readline v6.3 enabled, extended help enabled)
```

Copyright (C) 2000-2016 The PARI Group

PARI/GP is free software, covered by the GNU General Public License, and comes WITHOUT ANY WARRANTY WHATSOEVER.

Type `? for help`, `\q to quit`.

Type `?15 for how to get moral (and possibly technical) support`.

```
parisizemax = 2000003072, primelimit = 500000
?
```

Lo primero que pueden observar es la versión que están utilizando de PARI/GP, si su versión es muy antigua recuerden que pueden descargar la más reciente en <https://pari.math.u-bordeaux.fr/>.

En segundo lugar vienen las especificaciones de la máquina que se está utilizando.

Luego viene la última fecha de compilación y algunas otras características asociadas con el software de la máquina en que se está ejecutando.

Además vienen algunos anuncios sobre PARI/GP: es un programa gratuito de uso libre y que por supuesto no viene con garantía. Este anuncio está seguido por algunos comandos para pedir ayuda de

los cuales hablaremos más tarde, además del comando para cerrar el programa: \q.

Luego vienen dos parámetros:

- `parisizemax` es el máximo lo cual el stack (pila) de memoria que puede usar GP o cualquier programa usando la librería PARI. El tamaño está en bytes y se recomienda poner el valor de acuerdo a las capacidades de su máquina. PARI tratará de hacer operaciones en dentro de un límite, pero si es necesario incrementará el stack hasta el valor máximo. Un valor de `parisizemax` de entre un cuarto y la mitad de la memoria ram total de la computadora es recomendado.
- `primelimit`, la calculadora GP pre-calcula una lista de todos los primos menores que `primelimit` cuando iniciamos GP. Estos son usados por muchas funciones y evita hacer cálculos innecesarios.

Finalmente hay un signo ? es ahí donde escribimos nuestras instrucciones que queremos ejecutar.

Por ejemplo, sumar dos enteros

```
? 2+2
%1 = 4
```

y multiplicar el último resultado por un entero

```
? 2*%
%2 = 8
```

Por lo tanto % simboliza el último resultado, y si queremos un resultado en específico basta poner su número por ejemplo

```
? %1*%2
%3 = 32
```

PARI siempre va a privilegiar el uso de fracciones, sin embargo cuando queramos saber la expansión decimal de una división basta con poner un punto después de algunos de los números en el cálculo, por ejemplo:

```
? 1/2+1/3+1/5+1/7
%7 = 247/210
? 1.+1/2+1/3+1/5+1/7
%8 = 2.1761904761904761904761904761905
```

Las operaciones basicas son llevadas entonces de manera normal, por ejemplo como es de esperarse PARI no divide por 0

```
? 1/0
*** at top-level: 1/0
***      ^__
*** _/_: impossible inverse in gdiv: 0.
*** Break loop: type 'break' to go back to GP prompt
break>
```

Para salir del circuito de error basta con escribir `break`.

A continuación se muestra un diccionario de algunas funciones que son bastante útiles, para más funciones se puede acceder a la documentación de PARI como mostraremos más adelante:

Matemáticas	PARI	Descripción
	random(N)	Arroja un entero aleatorio entre [0,N)
	randomprime(N)	Arroja un primo aleatorio entre [0,N)
$\sum_{n=a}^b f(n)$	sum(n=a,b,f(n))	Sumatoria de $n = a$ a b de la expresión $f(n)$
$\prod_{n=a}^b f(n)$	prod(n=a,b,f(n))	Producto de $n = a$ a b de la expresión $f(n)$
A^T	mattranspose(A)	Transpuesta de la matriz A
trace, det(A)	trace(A), matdet(A)	Traza y determinante de la matriz A
deg(f)	poldegree(f)	Grado del polinomio f
$\Delta(f)$	poldisc(f)	Discriminante del polinomio f
$\sqrt{x}, \sqrt[n]{x}$	sqrt(x), sqrtn(x,n)	Raíz cuadrada y raíz n -ésima de x
$\log(x), e^x$	log(x), exp(x)	Logaritmo y exponencial de x
$\sin(x), \cos(x), \tan(x)$	sin(x), cos(x), tan(x)	Funciones trigonométricas de x
$\lfloor x \rfloor, \lceil x \rceil, \{x\}$	floor(x), ceil(x), frac(x)	Funciones techo, piso y parte fraccional
	prime(n)	n -ésimo primo
	primes(n)	Vector de los primeros n primos
	factor(n)	Factorización de n
$x!$	x! o factorial(x)	n factorial
$\binom{x}{y}$	binomial(x,y)	Coficiente binomial
$\varphi(n)$	eulerphi(x)	Función φ de Euler

A.3. Ayuda

El pequeño diccionario precedente, es sólo un ejemplo de todas las funciones disponibles en PARI. En general podemos acceder a la tarjeta de referencia, donde nos muestran más funciones:

? ??refcard

En general cuando necesitamos ayuda en PARI debemos de recurrir al signo ?. Si escribimos este signo, nos muestra el siguiente diálogo:

? ?

```

Help topics: for a list of relevant subtopics, type ?n for n in
0: user-defined functions (aliases, installed and user functions)
1: Standard monadic or dyadic OPERATORS
2: CONVERSIONS and similar elementary functions
3: TRANSCENDENTAL functions
4: NUMBER THEORETICAL functions
5: ELLIPTIC CURVES
6: L-FUNCTIONS
7: MODULAR SYMBOLS
8: General NUMBER FIELDS
9: Associative and central simple ALGEBRAS
10: POLYNOMIALS and power series
11: Vectors, matrices, LINEAR ALGEBRA and sets
12: SUMS, products, integrals and similar functions
13: GRAPHIC functions
14: PROGRAMMING under GP
15: The PARI community

```

Also:

```

? functionname (short on-line help)
?\             (keyboard shortcuts)
?.            (member functions)

```

Extended help (if available):

```

??          (opens the full user's manual in a dvi previewer)
?? tutorial / refcard / libpari (tutorial/reference card/libpari manual)
?? keyword  (long help text about "keyword" from the user's manual)
??? keyword (a propos: list of related functions).

```

Es decir, si escribimos por ejemplo `?5` obtenemos una lista de todas las funciones relacionadas con curvas elípticas. O si ponemos `??tutorial` podemos acceder a un tutorial general de PARI. Además si tenemos dudas sobre los parámetros en una función podemos escribir `?nombredefuncion` y podremos tener una breve reseña sobre la función y sus parámetros, por ejemplo:

```

? ?for
for(X=a,b,seq): the sequence is evaluated, X going from a up to b. If b is set
to +oo, the loop will not stop.

```

Si queremos una descripción más detallada basta con escribir `??nombredefuncion`. Y si queremos una lista de las funciones relacionadas con la función escribimos `???nombredefuncion`.

Es importante que usted se familiarice con el uso de la ayuda en PARI, ya que esto evitará perder la cabeza y la moral cuando nuestro código presente errores.

A.4. Estructuras y Variables

PARI no acepta llamar objetos de algunas formas, por ejemplo: `Pi`, `Euler`, `Catalan` están reservados para las constantes de mismo nombre. La variable `I` es $i = \sqrt{-1}$ y `O` está reservada para referirse a la O de Landau (big-Oh notation).

Estructuras matemáticas:

- Podemos definir números enteros, reales, racionales, complejos y p-ádicos:

```
? N=2017; x=2.11113; r=4/3; z=x+I*r; a=r+O(7^10);
```

- Vectores renglón $(1 \ 0 \ 1)$ y columna $\begin{pmatrix} 2 \\ 1 \\ 0 \end{pmatrix}$:

```
? [1,0,1]; [2,1,0]~;
```

- Un vector con los enteros de 1 a 10:

```
? [1..10]
%17 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

- La matriz

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

la escribimos

```
? [a,b;c,d]
```

A.5. Declarar funciones

El objetivo de usar PARI no es hacer cálculos individuales sino hacer funciones que sirvan en general y poder recurrir a ellas. Así pues es importante saber definir y hacer funciones que calculen lo que queremos.

Por ejemplo si queremos programar la función factorial podemos hacer el siguiente código:

```

Factorial(n)=
{
if(n!=0,return(factorial(n-1)*n),1);
}

```

En general una función se declara de la manera siguiente:

```

NombreFuncion(Variable1,...,VariableN)=
{
my(V1,V2,...,VM); \\variables locales

METODO;

return(X);
}

```

Podemos escribir las funciones en un bloc de notas, y guardar el archivo con la extensión *.gp, el archivo puede contener una o varias funciones. Desde PARI podemos ejecutar el archivo de la manera siguiente

```
? \r /home/usuario/Factorial.gp
```

y podemos llamar todas las funciones definidas en el archivo.

Nota:

- Algunas veces es necesario que una función regrese más de un dato, entonces la manera de regresar más de un dato es por medio de un arreglo. Por ejemplo, un arreglo en PARI de cuatro variables se define como `[x1,x2,x3,x4]`, si queremos regresar este arreglo la instrucción sería: `return([x1,x2,x3,x4]);`.
- En las funciones condicionales, por ejemplo `if`, `for` y `while`, tenemos que verificar una propiedad. Muchas veces esta propiedad es la igualdad entre dos objetos A y B . Para verificar esto, el código sería

```
A==B;
```

resultando 1 si A es igual a B y 0 sino. Para asignar el valor B a la variable A la instrucción es

```
A=B;
```

por lo tanto no debemos confundir el uso de doble `==` para verificar igualdad con el uso de un simple `=` para asignar un valor.

Bibliografía

- [Coh93] Henri Cohen. *A course in computational algebraic number theory*, volume 138 of *Graduate Texts in Mathematics*. Springer-Verlag, Berlin, 1993.