# Álgebra Computacional

Álvaro García Tenorio  Miguel Pascual Domínguez

3 de enero de 2019

**Resumen**

En este documento se recopilan unas breves explicaciones sobre el funcionamiento de los algoritmos implementados.

# Índice general

# Capítulo 1

# Algoritmos implementados

## 1.1. Algoritmos auxiliares

### 1.1.1. Algoritmo de exponenciación binaria

Sea $A$ un anillo conmutativo unitario. Consideremos $a \in A$ y $n \in \mathbb{N}$ de manera que $n > 0$. Nuestro objetivo es tener un algoritmo eficiente para calcular $a^n$.

Para ello, tomemos la representación binaria de $n$, es decir, $n = 2^k + \sum_{i=0}^{k-1} n_i 2^i$, así como la sucesión $\{r_i\}_{i=0}^{k}$ definida de la siguiente manera.

$$r_i := \begin{cases} a & \text{si} \quad i = k \\ r_{i+1}^2 a & \text{si} \quad n_i = 1 \\ r_{i+1}^2 & \text{si} \quad n_i = 0 \end{cases}$$

Demostremos que $r_0$ es $a^n$. Para ello, veamos por inducción que, $r_i = a^{\lfloor n/2^i \rfloor}$. La demostración es inmediata, una vez se comprueba que

$$\lfloor n/2^i \rfloor = \sum_{l=0}^{k-i} n_{l+i} 2^l$$

De esta manera, para obtener $a^n$ basta con calcular la sucesión desde el término $k$ hasta el término 0. Como $k = \lfloor \log n \rfloor$, tenemos un algoritmo de exponenciación **lineal respecto del número de cifras del exponente**. Este algoritmo está programado en la clase `Ring<E>`, tal y como se muestra en la figura 1.1.

### 1.1.2. Algoritmo de composición modular rápida

## 1.2. Algoritmo de Euclides simple

Sea $A$ un dominio euclídeo con función de grado $\varphi$. Recordemos, antes de comenzar, el siguiente resultado.

**Lema 1.2.1** (Euclides)**.** *Dado $A$ un dominio, consideremos $a, b, q, r \in A$ que verifican $a = bq + r$. Entonces, si existen, tanto el máximo común divisor de $a$ y $b$, como el de $b$ y $r$, coinciden* ***salvo unidades****.*

Consideramos dos elementos $a, b \in A$, siendo $b \neq 0$. Nuestro objetivo es **calcular el máximo común divisor** de $a$ y $b$.

```java
public E power(E a, BigInteger k) {
    E result = getProductIdentity();
    if (!k.equals(BigInteger.ZERO)) {
        /* Repeated squaring algorithm. */
        String binaryExponent = k.toString(2);
        result = a;
        for (BigInteger i = new BigInteger(Integer.toString(binaryExponent.length() - 2)); i
        .compareTo(BigInteger.ZERO) >= 0; i = i.subtract(BigInteger.ONE)) {
            if (binaryExponent.charAt(binaryExponent.length() - 1 - i.intValue()) == '1') {
                result = multiply(multiply(result, result), a);
            } else {
                result = multiply(result, result);
            }
        }
    }
    return result;
}
```

Figura 1.1: Código Java del algoritmo de exponenciación binaria.

Para ello, dividimos $a$ entre $b$, obteniendo dos elementos $q_1, r_1 \in A$ tales que $a = bq_1 + r_1$, siendo, o bien $r_1 = 0$, o bien $r_1 \neq 0$ y $\varphi(r_1) < \varphi(b)$. Esto es debido a que $A$ es un dominio euclídeo.

Es claro que, si $r_1 = 0$, $b$ es divisor de $a$, y por tanto $b = \text{mcd}(a, b)$, con lo que habríamos terminado. En caso contrario, como por el lema (1.2.1) se cumple que $\text{mcd}(a, b) = \text{mcd}(b, r_1)$, podemos repetir el proceso con $b$ y $r_1$, obteniendo dos nuevos elementos $q_2, r_2 \in A$, cumpliendo hipótesis análogas a las que cumplían $q_1$ y $r_1$.

Nótese que este proceso **solo puede ser repetido un número finito de veces**, ya que, cada vez que repetimos el proceso, la función de grado del resto de la división decrece estrictamente.

A este método de cálculo del máximo común divisor se le conoce como ***algoritmo de Euclides***, y es precisamente el algoritmo que implementa el método `gcd(E a, E b)` de la clase `EuclideanDomain<E>`.

```java
public E gcd(E a, E b) {
    /* Euclid's algorithm. */
    while (!b.equals(getAddIdentity())) {
        E r = remainder(a, b);
        a = b;
        b = r;
    }
    return a;
}
```

Figura 1.2: Código Java del algoritmo de Euclides.

## 1.3. Algoritmo de Euclides extendido

Basándonos en el algoritmo de Euclides, podemos, además de calcular el máximo común divisor de dos números, extraer una **identidad de Bézout** que los relacione. Partimos de un dominio euclídeo $A$, considerando dos elementos $a, b \in A$, siendo $b$ no nulo.

Procedemos, como en el algoritmo de Euclides, dividiendo $a$ y $b$, obteniendo $q_1, r_1 \in A$ cumpliendo las hipótesis habituales que garantizan que nuestro procedimiento es finito.

Despejando $r_1$ obtenemos que $r_1 = a - q_1 b$. Es decir, tenemos una ***"pseudo–identidad de Bézout"***. Definimos $\alpha_1 := 1$, $\beta_1 := -q_1$ para verlo más claro,

$$r_1 = \alpha_1 a + \beta_1 b.$$

Usando el lema (1.2.1), como en el algoritmo anterior, tenemos que $\mathrm{mcd}(a, b) = \mathrm{mcd}(b, r_1)$, por lo que procedemos a dividir $b$ entre $r_1$, siempre y cuando $r_1 \neq 0$, al final veremos que el procedimiento es válido para todos los casos.

De la división de $b$ y $r_1$ obtenemos la igualdad $r_2 = b - r_1 q_2$, sustituyendo $r_1$ por la pseudo–identidad de Bézout, obtenemos, tras reordenar, una nueva pseudo–identidad de bezout, esta vez para $r_2$, esta es

$$r_2 = \alpha_2 a + \beta_2 b,$$

siendo $\alpha_2 = -\alpha_1 q_2$ y $\beta_2 = 1 - \beta_1 q_2$.

Si repetimos el proceso una vez más, obtendremos una pseudo–identidad para $r_3$, siendo esta

$$r_3 = \alpha_3 a + \beta_3 b,$$

con $\alpha_3 = \alpha_1 - \alpha_2 q_3$ y $\beta_3 = \beta_1 - \beta_2 q_3$.

Por inducción, no es complicado comprobar que la pseudo–identidad para $r_n$ tendrá por coeficientes

$$\alpha_n = \alpha_{n-2} - \alpha_{n-1} q_n$$
$$\beta_n = \beta_{n-2} - \beta_{n-1} q_n$$

Además, si definimos $\alpha_{-1} := 1$, $\alpha_0 = 0$, $\beta_{-1} = 0$ y $\beta_0 = 1$, esta fórmula es válida para todo $n \in \mathbb{N}$.

De esta manera, cuando llegamos a una iteración del procedimiento en la cual $r_l = 0$, es decir, $r_{l-1}$ es el máximo común divisor, para obtener la identidad de Bézout basta recuperar los coeficientes $\alpha_{l-1}$ y $\beta_{l-1}$.

Esto puede ir haciéndose sobre la marcha, con una implementación muy similar a la del cálculo de términos de la sucesión de Fibonacci. A este algoritmo se le conoce como ***algoritmo de Euclides extendido***, y es el algoritmo programado en el método `bezout(E a, E b)` de la clase `EuclideanDomain<E>`.

En el caso particular en el que $A[\mathtt{x}]$ es un anillo de polinomios que es dominio euclídeo, se tiene el siguiente resultado, que usaremos más adelante.

**Corolario 1.3.1** (Grados de los coeficientes)**.** *Dados $f, g \in A[\boldsymbol{x}]$ tales que el algoritmo de Euclides extendido devuelve la identidad de Bézout $uf + vg = h$, entonces se cumple que* $\deg(u) < \deg(g) - \deg(h)$ *y* $\deg(v) < \deg(f) - \deg(h)$.

*Demostración.* Llamemos $\{u_i\}_{i=2}^n$ y $\{v_i\}_{i=2}^n$ a los coeficientes de las pseudo–identidades que va generando el algoritmo en cada iteración.

Por inducción se demuestra muy fácilmente que $\deg(v_i) = \deg(f) - \deg(r_{i-1})$ y que $\deg(u_i) = \deg(g) - \deg(r_{i-1})$.

Finalmente, como $\deg(r_n) < \deg(r_{n-1})$, se tienen ambas desigualdades. ■

```java
public Pair<E> bezout(E a, E b) {
    /* Extended Euclid's algorithm. */
    E alphaMinus1 = getAddIdentity();
    E alphaMinus2 = getProductIdentity();
    E betaMinus1 = getProductIdentity();
    E betaMinus2 = getAddIdentity();
    while (!b.equals(getAddIdentity())) {
        /* Computes division of a and b */
        E q = quotient(a, b);
        E r = remainder(a, b);

        /*
        * Just follow the formula for the new pseudo-bezout identity coefficients
        */
        E alphaAux = add(alphaMinus2, getAddInverse(multiply(q, alphaMinus1)));
        E betaAux = add(betaMinus2, getAddInverse(multiply(q, betaMinus1)));

        /* Maintaining alphaMinusX and betaMinusX coherent */
        alphaMinus2 = alphaMinus1;
        betaMinus2 = betaMinus1;

        alphaMinus1 = alphaAux;
        betaMinus1 = betaAux;

        /* By euclid's lemma gcd(a,b) = gcd(b,r) */
        a = b;
        b = r;
    }
    return new Pair<E>(alphaMinus2, betaMinus2);
}
```

Figura 1.3: Código Java del algoritmo de Euclides extendido.

## 1.4. Algoritmo de Euclides primitivo

Nuestro objetivo es desarrollar un algoritmo que nos permita calcular el máximo común divisor de dos polinomios $f, g \in A[\mathtt{x}]$, siendo $A[\mathtt{x}]$ un dominio de factorización única. Antes de comenzar, recordemos brevemente un par de resultados.

**Definición 1.4.1** (Contenido de un polinomio)**.** Dado un polinomio $f \in A[\mathtt{x}]$, siendo $A$ un dominio de factorización única, llamamos ***contenido*** de $f$ al máximo común divisor de sus coeficientes. Lo denotaremos por $\mathrm{c}(f)$. Si $\mathrm{c}(f) = 1$, diremos que $f$ es ***primitivo***.

**Definición 1.4.2** (Parte primitiva)**.** Dado un polinomio $f \in A[\mathtt{x}]$, siendo $A$ un DFU, es claro que $\mathrm{c}(f)$ divide a $f$, luego $\mathrm{c}(f)\alpha = f$. A dicho polinomio $\alpha$ lo llamamos ***parte primitiva*** de $f$, y lo denotamos por $\mathrm{pp}(f)$.

**Lema 1.4.1** (Buen comportamiento del contenido)**.** *Dados dos polinomios $f, g \in A[\boldsymbol{x}]$, siendo $A$ un DFU, se tiene que $\mathrm{c}(fg) = \mathrm{c}(f)\mathrm{c}(g)$.*

**Lema 1.4.2** (Lema de Gauss)**.** *$A[\boldsymbol{x}]$ es un DFU si y solo si $A$ es un DFU.*

**Lema 1.4.3** (Pseudodivisión de polinomios)**.** *Sea $A$ un anillo conmutativo unitario, y $f, g \in A[\boldsymbol{x}]$, entonces, existen $q, r \in A[\boldsymbol{x}]$, con $\deg(r) < \deg(g)$ tales que $\mathrm{l}(g)^k f = gq + r$, siendo $k \geq 0$.*

Con estos resultados en la mano, podemos adaptar el algoritmo de Euclides para calcular el máximo común divisor de dos polinomios primitivos $f, g \in A[\mathtt{x}]$, siendo $A[\mathtt{x}]$ un $DFU$. Nuestra adaptación se basará en el siguiente resultado.

**Proposición 1.4.4** (Lema de Euclides primitivo)**.** *Dados $f, g \in A[\boldsymbol{x}]$ primitivos, entonces* $\mathrm{mcd}(f, g) = \mathrm{mcd}(g, \mathrm{pp}(r))$*, siendo $r$ el resto de la pseudodivisión de $f$ y $g$.*

*Demostración.* Es sencillo demostrar que el máximo común divisor de dos polinomios primitivos es primitivo, a partir de ahí, consideremos $d_1 = \mathrm{mcd}(f, g)$ y $d_2 = \mathrm{mcd}(g, \mathrm{pp}(r))$.

Es claro que $d_1$ divide a $f$ y a $g$. Como, haciendo la pseudodivisión, $\mathrm{l}(g)^k f = gq + r$, tenemos que $d_1$ divide a $r$, luego $d_1\alpha = \mathrm{c}(r)\mathrm{pp}(r)$. Veamos que $\mathrm{c}(r)$ divide a $\alpha$, en efecto, esto sucede si y solo si $\mathrm{c}(r)$ divide a $\mathrm{c}(\alpha)$. Pero tenemos que $\mathrm{c}(\alpha) = \mathrm{c}(d_1\alpha) = \mathrm{c}(r)\mathrm{pp}(r) = \mathrm{c}(r)$. Luego, por la propiedad cancelativa, $d_1$ divide a $\mathrm{pp}(r)$, y, por ser $d_2 = \mathrm{mcd}(g, \mathrm{pp}(r))$, se cumple que $d_1$ divide a $d_2$.

Recíprocramente, $d_2$ divide a $g$ y a $\mathrm{pp}(r)$, luego $d_2$ divide a $\mathrm{l}(g)^k f$, es decir, $\mathrm{l}(g)^k f = \beta d_2$. Veamos que $\mathrm{l}(g)^k$ divide a $\beta$. Esto ocurre si y solo si $\mathrm{l}(g)^k$ divide a $\mathrm{c}(\beta)$, pero, $\mathrm{c}(\beta) = \mathrm{c}(\beta d_2) = \mathrm{c}(\mathrm{l}(g)^k f) = \mathrm{l}(g)^k$. Luego, de nuevo, por la propiedad cancelativa, $d_2$ divide a $f$, y, por ser $d_1$ es máximo común divisor de $f$ y $g$, $d_2$ divide a $d_1$.

En definitiva, $d_1$ y $d_2$ son asociados, como queríamos demostrar. ■

Esto nos permite adaptar de manera directa el algoritmo de Euclides visto anteriormente.

```java
public Polynomial<E> gcd(Polynomial<E> a, Polynomial<E> b) {
    /* Primitive euclid's algorithm */
    while (!b.equals(getAddIdentity())) {
        Polynomial<E> r = polyRing.pseudoDivision(a, b).getThird();
        E c = content(a);
        r = primitivePart(r, c);
        a = b;
        b = r;
    }
    return a;
}
```

Figura 1.4: Código Java del algoritmo de Euclides primitivo.

## 1.5. Algoritmo del teorema chino de los restos

Recordamos brevemente el teorema chino de los restos.

**Teorema 1.5.1** (Teorema chino de los restos)**.** *Dado un anillo conmutativo y con unidad $A$, y $r$ ideales $I_1, \ldots, I_r \subset A$ comaximales dos a dos, se cumple que la aplicación*

$$\begin{aligned} \phi: \quad & A \to A/I_1 \times \cdots \times A/I_r \\ & a \mapsto (a + I_1, \ldots, a + I_r) \end{aligned}$$

*es sobreyectiva.*

*Demostración.* En primer lugar, es sencillo demostrar por inducción que, para todo $i \in \{1, \ldots, r\}$ $I_i$ e $\prod_{j \neq i} I_j$ son comaximales, o sea que, para ciertos $u_i \in I_i$ y $v_i \in \prod_{j \neq i} I_j$, se cumple que $u_i + v_i = 1$.

Dicho esto, si consideramos $(a_1 + I_1, \ldots, a_r + I_r)$, es sencillo demostrar que $\phi(\sum_{i=1}^r a_i v_i) = (a_1 + I_1, \ldots, a_r + I_r)$, quedando el teorema demostrado. ■

Lo que tratamos de hacer nosotros, por medio de un algoritmo, es hallar una imagen inversa $\phi^{-1}$ de un elemento $(a_1 + I_1, \ldots, a_r + I_r) \in A/I_1 \times \cdots \times A/I_r$.

Por la demostración del teorema, tenemos que dicha imagen inversa es $\sum_{i=1}^{r} a_i v_i$, luego solo tenemos que encontrar los coeficientes $v_i$. Esto es una tarea imposible en la mayoría de circunstancias, pero no si $A$ es un DE.

En tal caso, $A$ también es un dominio de ideales principales, luego podemos suponer que el ideal $I_i$ está generado por el elemento $b_i$ para todo $i \in \{1, \ldots, r\}$. Asimismo, es sencillo comprobar que el ideal $\prod_{j \neq i} I_j$ está generado por $q := \prod_{j \neq i} b_j$.

Por la comaximalidad de $I_i$ y $\prod_{j \neq i} I_j$, habrá un elemento de $I_i$, es decir $b_i\alpha$ para cierto $\alpha \in A$ y un elemento de $\prod_{j \neq i} I_j$, o sea, $q\beta$ para cierto $\beta \in A$ de manera que $b_i\alpha + q\beta = 1$. Estos $\alpha$ y $\beta$ los podemos hallar mediante una identidad de Bézout. De esta manera, basta definir $v_i := q\beta$, con lo que ya tendríamos el algoritmo. Este algoritmo está implementado

```
    public E chineseRemainderInverse(List<E> ideals, List<E> remainders) {

    /* m is a generator of the product of ideals. */
    E m = getProductIdentity();
    for (int i = 0; i < ideals.size(); i++) {
        m = multiply(m, ideals.get(i));
    }

    E inverse = getAddIdentity();
    for (int i = 0; i < remainders.size(); i++) {
        /* q is the generator of the product of ideals except the i-th one. */
        E q = quotient(m, ideals.get(i));
        E coefficient = bezout(q, ideals.get(i)).getFirst();
        inverse = add(inverse, multiply(multiply(remainders.get(i), coefficient), q));
    }
    return inverse;
}
```

Figura 1.5: Código Java del algoritmo del Teorema Chino de los Restos.

en el método `chineseRemainderInverse` de la clase `EuclideanDomain`.

## 1.6. Algoritmo de inversión de elementos en $\mathbb{F}_q$

Antes de comenzar, veamos un pequeño lema auxiliar.

**Lema 1.6.1** (Identidad de Bézout y coprimalidad)**.** *Dados $a, b \in A$, siendo $A$ un DE, $a$ y $b$ son coprimos si y solo si existen $\alpha, \beta \in A$ tales que $\alpha a + \beta b = 1$.*

*Demostración.* La implicación a la derecha es evidente. Recíprocamente, si existen $\alpha, \beta \in A$ tales que $\alpha a + \beta b = 1$, entonces, si $d := \text{mcd}(a, b)$, es claro que $d$ divide a $\alpha a + \beta b = 1$, luego $d$ es una unidad. En otras palabras, $a$ y $b$ son coprimos. ■

El algoritmo de inversión se basa en el siguiente resultado.

**Proposición 1.6.2.** *Dado $A$ un dominio euclídeo y $a, m \in A$. $A$ es invertible en $A/mA$ si y solo si* $\text{mcd}(a, m) = 1$.

*Demostración.* En efecto, $a$ es una unidad si y solo si existe un $c \in A$ tal que $ac = 1 + \lambda m$. Despejando la igualdad obtenemos que $ac - \lambda m = 1$, que es una identidad de Bézout con el elemento neutro del producto. ■

Basta por tanto con darse cuenta de que el inverso de $a$ en la proposición (1.6.2) es el coeficiente de la identidad de Bézout que acompaña a $a$. Como podemos calcular los coeficientes de dicha identidad mediante el algoritmo de Euclides extendido, el algoritmo para obtener el inverso es inmediato.

Únicamente hay que ser cuidadoso, sobre todo en $\mathbb{F}_q$, pues la identidad de Bézout obtenida puede estar expresada, no respecto del elemento neutro del producto, sino respecto de cualquier otra unidad.

```
public FiniteFieldElement getProductInverse(FiniteFieldElement a) {
    /*
    * The greater common divisor will always be a unit, we just normalize it to be
    * the product identity, so the product inverse matches with the first
    * coefficient of the Bezout's identity.
    */
    BigInteger factor = baseField.getProductInverse
    (polyRing.gcd(a.getPolynomial(), irrPolMod).leading());
    return new FiniteFieldElement(polyRing.remainder(
    polyRing.intMultiply((polyRing.bezout(a.getPolynomial(), irrPolMod).getFirst()),
    factor), irrPolMod), irrPolMod);
}
```

Figura 1.6: Código Java del algoritmo de inversión de elementos en cuerpos finitos de orden potencia de primo.

## 1.7. Criterio de irreducibilidad en $\mathbb{F}_q[\mathtt{x}]$

Antes de comenzar, conviene repasar algunos hechos acerca de los cuerpos finitos y los polinomios sobre los mismos.

**Teorema 1.7.1** (Generalización del pequeño teorema de Fermat)**.** *Para cada $d \geq 1$, consideremos $h := \boldsymbol{x}^{q^d} - \boldsymbol{x} \in \mathbb{F}_q[\boldsymbol{x}]$. Se cumple que $h$ es el producto de todos los polinomios irreducibles mónicos de $\mathbb{F}_q[\boldsymbol{x}]$ cuyo grado divide a $d$.*

*Demostración.* Veamos en primer lugar que $h$ **es libre de cuadrados.**

Aplicando el pequeño teorema de Fermat sobre $\mathbb{F}_{q^d}[\mathtt{x}]$ tenemos que

$$h = \prod_{a \in \mathbb{F}_{q^d}} (\mathtt{x} - a)$$

Por tanto, si hubiera un $g \in \mathbb{F}_q[\mathtt{x}]$ con $\deg(g) \geq 1$ de manera que $g^2$ divida a $h$, entonces tendríamos que $g^2\alpha = \prod_{a \in \mathbb{F}_{q^d}} (\mathtt{x} - a)$, y, como $\mathbb{F}_{q^d}$ es un DFU, cada factor de $g^2$ se corresponde con uno de los factores del productorio.

O sea, que para cierto $b \in \mathbb{F}_{q^d}$, $\mathtt{x} - b$ divide a $g$, luego, $(\mathtt{x} - b)\beta = g$, o, lo que es lo mismo, $(\mathtt{x} - b)^2\beta^2 = g^2$, y, como habíamos supuesto que $g^2\alpha = h$, tendríamos que $h = (\mathtt{x} - b)^2\beta^2\alpha$, lo cual es absurdo, pues entonces el productorio contendría un factor cuadrático, contradiciendo el pequeño teorema de Fermat.

Probemos ahora que para todo $f$ mónico e irreducible de grado $n$, $f$ **divide a** $h$ **si y solo si** $n$ **divide a** $d$**.**

Si $f$ divide a $h$, por el pequeño teorema de Fermat aplicado a $\mathbb{F}_{q^d}$ tendríamos que $f$ es el producto de un conjunto selecto $A$ de polinomios de la forma $\mathtt{x} - a$ con $a \in A \subset \mathbb{F}_{q^d}$.

Tomando alguno de estos $a \in A$, podemos considerar el anillo de $q^n$ elementos

$$\frac{\mathbb{F}_q[\mathtt{x}]}{f} \cong \mathbb{F}_q[a] \subset \mathbb{F}_{q^d}$$

Por ende, $q^d$ debe ser potencia de $q^n$, y esto implica que $n$ divide a $d$.

Recíprocamente, si $n$ divide a $d$, se cumple que $q^d - 1$ divide a $q^n - 1$ (el factor es $\gamma := q^{d-n} + q^{d-2n} + \cdots + 1$), asimismo, también se cumple que $\mathtt{x}^{q^n-1} - 1$ divide a $\mathtt{x}^{q^d-1} - 1$, en este caso, el factor es $\xi := \mathtt{x}^{(q^n-1)(\gamma-1)} + \mathtt{x}^{(q^n-1)(\gamma-2)} + \cdots + 1$. Con lo que finalmente concluímos que $\mathtt{x}^{q^n} - \mathtt{x}$ divide a $h$.

Si de nuevo consideramos el anillo $\frac{\mathbb{F}_q[\mathtt{x}]}{f} \cong \mathbb{F}_{q^n}$ junto con el elemento $a := \mathtt{x} \mod f$, que es una raíz de $f$, tenemos que, por el pequeño teorema de Fermat aplicado a $\mathbb{F}_{q^n}$, $\mathtt{x} - a$ divide a $\mathtt{x}^{q^n} - \mathtt{x}$, que a su vez divide a $h$.

En definitiva, $\mathtt{x} - a$ divide a $f$, y $\mathtt{x} - a$ divide a $h$, luego divide al máximo común divisor de ambos dos, que es un elemento de $\mathbb{F}_q[\mathtt{x}]$, y, como $f$ es irreducible, dicho máximo común divisor será el propio $f$, concluyéndose finalmente que $f$ divide a $h$. ■

## 1.8. Algoritmo de Cantor y Zassenhaus en $\mathbb{F}_q[\mathtt{x}]$

### 1.8.1. Fase I

### 1.8.2. Fase II

### 1.8.3. Fase III

## 1.9. Algoritmo de Berlekamp en $\mathbb{F}_q[\mathtt{x}]$

## 1.10. Algoritmo de Kronecker en $\mathbb{Z}[\mathtt{x}]$

## 1.11. Algoritmo de factorización modular en $\mathbb{Z}[\mathtt{x}]$

## 1.12. Criterio de primalidad AKS en $\mathbb{Z}$

## 1.13. Logaritmo discreto

Dado un grupo multiplicativo cíclico $G$, sabemos que este tiene al menos un generador $g \in G$. En estas condiciones, el siguiente concepto está bien definido.

**Definición 1.13.1** (Logaritmo discreto)**.** Se define el ***logaritmo discreto*** en base $g$ de un elemento $v \in G$, como el entero $x$ que verifica que $v = g^x$. Sabemos que este siempre existe, por ser $g$ un generador del grupo cíclico.

Hay un algoritmo evidente de fuerza bruta que calcula el logaritmo discreto de cualquier elemento. Este símplemente iría probando con cada una de las potencias hasta que salga el elemento deseado. La complejidad de este algoritmo es $\mathcal{O}(n)$, siendo $n$ el orden de $G$.

Una posible mejora consistiría en considerar $m := \lceil \sqrt{n} \rceil$. Es claro que $x \in \{0, \ldots, n-1\}$, entonces, dividiendo el número $x$ buscado entre $m$, obtenemos la igualdad $x = a + bm$ (siendo $a, b \in \{0, \ldots, m-1\}$), y, por tanto $v = g^{a+bm}$, o, lo que es lo mismo, $vg^{-bm} = g^a$.

Dicho esto, calcularemos las potencias sucesivas de $g$ (junto con su exponente asociado), las cuales serán almacenadas en una tabla que nos permita realizar búsquedas en tiempo

constante (en nuestro caso concreto haremos uso de los `HashMap` que proporciona la librería estándar de Java).

Tras esto, calcularemos $g^{-m}$ mediante el algoritmo de exponenciación rápida. Un vez calculado, iremos generando la sucesión $\{vg^{-im}\}_{i=0}^{m-1}$ (lo cual podemos hacer partiendo de $v$ y multiplicando por el ya calculado $g^{-m}$ en cada paso). Cada vez que generemos un nuevo término, miraremos en la tabla si el término generado se corresponde con alguno de los allí presentes.

En caso afirmativo, hemos encontrado los entero $a$ y $b$ buscados, en caso negativo, debemos seguir generando términos. Este algoritmo tiene una complejidad del orden de $\mathcal{O}(\sqrt{n})$ en tiempo y espacio, lo cual mejora sustancialmente el algoritmo de fuerza bruta. No obstante, nótese que sigue siendo un algoritmo exponencial respecto del número de cifras del orden del grupo.

```
public BigInteger discreteLogarithm(E generator, E a) {
    Integers Z = new Integers();

    BigInteger m = Z.add(Z.sqrtFloor(Z.add(getOrder(), Z.getAddInverse(
    Z.getProductIdentity()))), Z.getProductIdentity());

    HashMap<E, BigInteger> L = new HashMap<E, BigInteger>();
    E aux = getProductIdentity();
    L.put(aux, BigInteger.ZERO);
    for (BigInteger j = BigInteger.ONE; j.compareTo(m) < 0;
        j = j.add(BigInteger.ONE)) {
        aux = multiply(aux, generator);
        L.put(aux, j);
    }

    E gm = power(getProductInverse(generator), m);
    E w = a;
    for (BigInteger i = BigInteger.ZERO; i.compareTo(m) < 0;
        i = i.add(BigInteger.ONE)) {
        BigInteger match = L.get(w);
        if (match == null) {
            w = multiply(w, gm);
        } else {
            return Z.add(Z.multiply(i, m), match);
        }
    }

    /* This return is never reached. */
    return null;
}
```

Figura 1.7: Código Java del algoritmo de cómputo del logaritmo discreto para el grupo multiplicativo de cuerpos finitos.