

Álgebra Computacional

Álvaro García Tenorio Miguel Pascual Domínguez

25 de diciembre de 2018

Resumen

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

Índice general

1. Algoritmos implementados	3
1.1. Algoritmos auxiliares	3
1.1.1. Algoritmo de exponenciación binaria	3
1.1.2. Algoritmo de composición modular rápida	3
1.2. Algoritmo de Euclides simple	3
1.3. Algoritmo de Euclides extendido	4
1.4. Algoritmo de Euclides primitivo	5
1.5. Algoritmo del teorema chino de los restos	7
1.6. Algoritmo de inversión de elementos en \mathbb{F}_q	8
1.7. Criterio de irreducibilidad en $\mathbb{F}_q[x]$	9
1.8. Algoritmo de Cantor y Zassenhaus en $\mathbb{F}_q[x]$	9
1.8.1. Fase I	9
1.8.2. Fase II	9
1.8.3. Fase III	9
1.9. Algoritmo de Berlekamp en $\mathbb{F}_q[x]$	9
1.10. Algoritmo de Kronecker en $\mathbb{Z}[x]$	9
1.11. Algoritmo de factorización modular en $\mathbb{Z}[x]$	9
1.12. Criterio de primalidad AKS en \mathbb{Z}	9
1.13. Logaritmo discreto	9

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.

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

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>`.

1.1.2. Algoritmo de composición modular rápida

1.2. Algoritmo de Euclides simple

Sea A un dominio euclídeo con función de grado φ . 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 .

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.

```

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.

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>`.

```

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 $\text{mcd}(a, b) = \text{mcd}(b, r_1)$, 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

$$\begin{aligned}\alpha_n &= \alpha_{n-2} - \alpha_{n-1} q_n \\ \beta_n &= \beta_{n-2} - \beta_{n-1} q_n\end{aligned}$$

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 α_{l-1} y β_{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 $A[x]$ 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 $f, g \in A[x]$ 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. ■

1.4. Algoritmo de Euclides primitivo

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

```

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.

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

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

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

Lema 1.4.2 (Lema de Gauss). $A[x]$ 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 $f, g \in A[x]$, entonces, existen $q, r \in A[x]$, con $\deg(r) < \deg(g)$ tales que $l(g)^k f = gq + r$, 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 $f, g \in A[x]$, siendo $A[x]$ un DFU. Nuestra adaptación se basará en el siguiente resultado.

Proposición 1.4.4 (Lema de Euclides primitivo). *Dados $f, g \in A[x]$ primitivos, entonces $\text{mcd}(f, g) = \text{mcd}(g, pp(r))$, 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 $d_1 = \text{mcd}(f, g)$ y $d_2 = \text{mcd}(g, pp(r))$.

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

Recíprocamente, d_2 divide a g y a $pp(r)$, luego d_2 divide a $l(g)^k f$, es decir, $l(g)^k f = \beta d_2$. Veamos que $l(g)^k$ divide a β . Esto ocurre si y solo si $l(g)^k$ divide a $c(\beta)$, pero, $c(\beta) = c(\beta d_2) = c(l(g)^k f) = l(g)^k$. 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.

```

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 $I_1, \dots, I_r \subset A$ comaximales dos a dos, se cumple que la aplicación*

$$\begin{aligned} \phi: A &\rightarrow A/I_1 \times \dots \times A/I_r \\ a &\mapsto (a + I_1, \dots, a + I_r) \end{aligned}$$

es sobreyectiva.

Demostración. En primer lugar, es sencillo demostrar por inducción que, para todo $i \in \{1, \dots, r\}$ 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 $(a_1 + I_1, \dots, a_r + I_r)$, es sencillo demostrar que $\phi(\sum_{i=1}^r a_i v_i) = (a_1 + I_1, \dots, a_r + I_r)$, quedando el teorema demostrado. ■

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

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 $i \in \{1, \dots, r\}$. 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 α y β 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

El algoritmo de inversión se basa en el siguiente resultado.

Proposición 1.6.1. *Dado A un dominio euclídeo con unidad 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 (1.6.1) 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.

Se puede consultar el código de este algoritmo en los métodos `getProductInverse(T a)` de la clases `PrimeModuleIntegers` (que representan a los cuerpos \mathbb{Z}_p) y `PrimeQuotients`, que representan a los cuerpos \mathbb{F}_q .

```

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 $\mathbb{F}_q[x]$

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

1.8.1. Fase I

1.8.2. Fase II

1.8.3. Fase III

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

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

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

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

1.13. Logaritmo discreto