Cálculo del Máximo Común Divisor de dos Polinomios en $\mathbb{Z}[x],\ \mathbb{Z}[x_1, ..., x_k],\ \mathbb{Z}_p[x]$ y $\mathbb{Q}[x]$.

Teoría, Algoritmos e Implementación en Java.

Primera Parte

Walter Mora Flores

wmora2@yahoo.com.mx

Escuela de Matemática − Centro de Recursos Virtuales (CRV)

Instituto Tecnológico de Costa Rica

1.1. Introducción.

El problema de calcular el máximo común divisor (MCD) de dos polinomios es de importancia fundamental en álgebra computacional. Estos cálculos aparecen como subproblemas en operaciones aritméticas sobre funciones racionales o aparecen como cálculo prominente en factorización de polinomios y en integración simbólica, además de otros cálculos en álgebra.

En general, podemos calcular el MCD de dos polinomios usando una variación del algoritmo de Euclides. El algoritmo de Euclides es conocido desde mucho tiempo atrás, es fácil de entender y de implementar. Sin embargo, desde el punto de vista del álgebra computacional, este algoritmo tiene varios inconvenientes. Desde finales de los sesentas se han desarrollado algoritmos mejorados usando técnicas un poco más sofisticadas.

En esta primera parte vamos a entrar en la teoría básica y en los algoritmos (relativamente) más sencillos, el algoritmo "subresultant PRS" (aquí lo llamaremos PRS subresultante) y el algoritmo heurístico (conocido como "GCDHEU"). Este último algoritmo es muy eficiente en problemas de pocas variables y se usa también como complemento de otros algoritmos. De hecho, se estima que el 90 % de los cálculos de MCD's en MAPLE se hacen con este algoritmo ([**?**]).

No se puede decir con certeza que haya un "mejor" algoritmo para el cálculo del MCD de dos polinomios.

Los algoritmos más usados, para calcular MCD en $\mathbb{Z}[x_1, ..., x_n]$, son "EZ-GCD" (Extended Zassenhaus GCD), GCDHEU y "SPMOD" (Sparse Modular Algorithm) [**?**].

GCDHEU es más veloz que EZGCD y SPMOD en algunos casos, especialmente para polinomios con cuatro o menos variables. En general, SPMOD es más veloz que EZGCD y GCDHEU en problemas donde los polinomios son "ralos", es decir con muchos coeficientes nulos y éstos, en la práctica, son la mayoría.

En la segunda parte, en el próximo número, nos dedicaremos a EZGCD y SPMOD. Estos algoritmos requieren técnicas más sofisticadas basadas en inversión de homomorfismos vía el teorema chino del resto, iteración lineal p-ádica de Newton y construcción de Hensel. Como CGDHEU es un algoritmo modular, aprovechamos para iniciar con parte de la teoría necesaria para los dos primeros algoritmos.

En este trabajo, primero vamos a presentar los preliminares algebraicos, el algoritmo de Euclides, el algoritmo primitivo de Euclides, el algoritmo PRS Subresultante y el algoritmo heurístico, además de el algoritmo extendido de Euclides. Las implementaciones requieren, por simplicidad, construir un par de clases para manejo de polinomios con coeficientes racionales grandes ("BigRational") y para manejo de polinomios con coeficientes enteros grandes ("BigInteger"). Aunque vamos a ver ejemplos de cómo "corren" estos algoritmos en polinomios de varias variables, estas implementaciones no aparecen aquí.

Para mantener el código legible, las implementaciones no aparecen optimizadas, más bien apegadas a la lectura de los algoritmos.

1.2. Preliminares algebraicos.

Un *campo* es un lugar donde usted puede sumar, restar, multiplicar y dividir. Formalmente, es un conjunto F dotado de dos operaciones binarias "+" y "$\cdot$", tales que

1. F es un grupo abeliano respecto a "+", con identidad 0.
2. Los elementos no nulos de F forman un grupo abeliano respecto a "$\cdot$".
3. Se cumple la ley distributiva $a \cdot (b + c) = a \cdot b + a \cdot c$.

El campo F se dice *finito* o *infinito* de acuerdo a si F es finito o infinito. Los ejemplos familiares de campos son $\mathbb{R}, \mathbb{Q}, \mathbb{C}$ y las funciones racionales sobre un campo. En lo que sigue, estaremos en contacto con un campo finito famoso:

$$\mathbb{Z}_p = \{0, 1, ..., p-1\}, \quad \text{dotado de la aritmética mod } p$$

donde p es primo.

- La aritmética módulo p es muy sencilla. Las operaciones con "+" y "$\cdot$" las hacemos en $\mathbb{Z}$ pero el resultado es el *residuo* de la división (en $\mathbb{Z}$) por p.

- Si $a, b \in \mathbb{Z}_p$, la división a/b se entiende como $a \cdot b^{-1}$.

  Ejemplo. En $\mathbb{Z}_5$,

  - $3 + 4 = 7$ corresponde a 2 módulo 5,
  - $4 \cdot 4 = 16$ corresponde a 1 módulo 5, es decir el inverso de 4 es 4 (módulo 5).
  - $3/4 = 3 \cdot 4^{-1} = 3 \cdot 4$ corresponde a 2 módulo 5.

Notación de congruencias.

Sea $p \geq 2$ (el módulo 1 no es de mucho interés). Decimos que $a \equiv b \pmod{p}$ si p divide a $b - a$. Otra forma de verlo es $a \equiv b \pmod{p}$ si $a = pk + b$ con k algún entero.

Ejemplo.

a) $7 \equiv 2 \pmod{5}$
b) $16 \equiv 1 \pmod{5}$
c) $x = 4$ es una solución de la ecuación $4x \equiv 1 \pmod{5}$

El símbolo "$\equiv$" funciona igual que el símbolo "$=$" excepto para la cancelación. En efecto,

1. si $a \equiv b \pmod{p}$ entonces $ka \equiv kb \pmod{p}$, $k \in \mathbb{Z}$;

2. si $a \equiv b \pmod{p}$ y $b \equiv c \pmod{p}$ entonces $a \equiv c \pmod{p}$;
3. si $a \equiv b \pmod{p}$ y $r \equiv s \pmod{p}$ entonces $a + r \equiv c + s \pmod{p}$;
4. si $a \equiv b \pmod{p}$ y $r \equiv s \pmod{p}$ entonces $ar \equiv cs \pmod{p}$;
5. **(cancelación)** si $ca \equiv cb \pmod{p}$ entonces $a \equiv b \pmod{p/\mathrm{mcd}(c,p)}$.

Más adelante vamos a volver a las congruencias.

1.2.1. Dominios de Factorización Única y Dominios Euclidianos.

En el 300 (a.de C.) Euclides dio un algoritmo notablemente simple para calcular el máximo común divisor (MCD) de dos enteros. Las versiones actuales del algoritmo de Euclides cubren no solo el cálculo del MCD para números enteros sino para cualquier par de elementos de un *dominio Euclidiano*. Veamos la definición de dominio Euclidiano.

Un anillo conmutativo $(R, +, \cdot)$ es un conjunto no vacío R cerrado bajo las operaciones binarias "+" y "$\cdot$", tal que $(R, +)$ es un grupo abeliano, "$\cdot$" es asociativa, conmutativa y tiene una identidad y satisface la ley distributiva.

Un *dominio integral* D es un anillo conmutativo con la propiedad adicional (ley de cancelación o equivalentemente, sin divisores de cero).

$$a \cdot b = a \cdot c \quad \text{y} \quad a \neq 0 \implies b = c.$$

Un *dominio Euclidiano* D es un dominio integral con una faceta adicional: una noción de "medida" entre sus elementos. La "medida" de $a \neq 0$ se denota $v(a)$ y corresponde a un entero no negativo tal que

1. $v(a) \leq v(ab)$ si $b \neq 0$;
2. Para todo a, $b \neq 0$ en D, existe $q, r \in D$ (el "cociente" y el "residuo") tal que

$$a = qb + r \quad \text{con} \quad r = 0 \quad \text{o} \quad v(r) < v(b).$$

Ejemplo. Algunos dominios Euclidianos son

a) $\mathbb{Z}$ con $v(n) = |n|$.
b) $F[x] =$ polinomios en la indeterminada x (con coeficientes en F) con $v(p) =$ grado p.
c) Los enteros Gaussianos $\{a + b\sqrt{-1},\ a, b \in \mathbb{Z}\}$, con $v(a + bi) = a^2 + b^2$.

- La propiedad 1 se usa para caracterizar las unidades (elementos invertibles) en D, u es unidad si $v(u) = v(1)$.

Máximo común divisor (MCD).

En un dominio Integral D decimos que a divide a b, simbólicamente $a|b$, si existe $c \in D$ tal que $b = ca$. Si $a|b_i,\ i = 1, 2, ..., n,\ a$ se dice un común divisor de los b_i's. Finalmente, si d es un divisor común de $b_1, b_2, ...b_n$, y si cualquier otro divisor común de los b_i's divide a d, entonces d se dice *un* máximo común divisor de $b_1, b_2, ...b_n$.

Ejemplo.

a) De acuerdo con la definición, 14 y -14 cumplen con la definición de máximo común divisor de $84, -140$ y 210 en $\mathbb{Z}$.

b) Sean $a(x) = 48x^3 - 84x^2 + 42x - 36$ y $b(x) = -4x^3 - 10x^2 + 44x - 30$,
notemos que

- $a(x) = 6(2x - 3)(4x^2 - x + 2)$
- $b(x) = -2(2x - 3)(x - 1)(x + 5)$

Entonces,

- en $\mathbb{Z}[x]$, $\text{MCD}\,(a(x), b(x)) = 4x - 6$.

- en $\mathbb{Q}[x]$, $\text{MCD}\,(a(x), b(x)) = x - 3/2$. ¿Porqué?

  En $\mathbb{Q}[x]$, tanto $g_1(x) = 4x - 6$ como $g_2(x) = x - 3/2$ son divisores comunes de $a(x)$ y $b(x)$ pero $g_1|g_2$ y $g_2|g_1$, es decir son *asociados.* Más adelante veremos que, en el caso de $\mathbb{Q}[x]$, el MCD lo tomamos como un representante de clase.

Unicidad del MCD.

Desde le punto de vista de la matemática, la no-unicidad del máximo común divisor puede ser fastidioso pero de ninguna manera dañino. Desde el punto de vista del software sí necesitamos unicidad, pues necesitamos implementar un *función* `MCD(a,b)` con una única salida.

La unicidad la logramos agregando una propiedad más en la definición. Solo hay que notar que si $a,\ a' \in D$ son MCD de $b_1, b_2, ..., b_n$ entonces a y a' son *asociados,* es decir $a = ub$ y $b = u^{-1}a$ para alguna unidad $u \in D$. En el ejemplo anterior, 14 y -14 son múltiplos uno del otro y en este caso $u = 1$ (los únicas unidades en $\mathbb{Z}$ son ± 1) y en el otro caso $g_1(x) = 4x - 6$ y $g_2(x) = x - 3/2$ son múltiplos uno del otro, las unidades en $Q[x]$ son lo racionales no nulos, en este caso $u = 4$.

La relación "ser asociado de" es una relación de equivalencia en D, por lo que podemos tomar al representante de clase (una vez definido cómo elegirlo) como el único MCD. Formalmente

Definiciones y Teoremas

Sea D un dominio integral.

1. $u \in D$ es una unidad si es invertible, es decir si $u^{-1} \in D$.
2. Dos elementos $a, b \in D$ se dicen *asociados* si $a|b$ y $b|a$.
3. $a, b \in D$ son asociados si y sólo si existe una unidad $u \in D$ tal que $a = ub$ (y entonces $au^{-1} = b$).
4. La relación "es asociado de" es una relación de equivalencia. Decimos que esta relación descompone D en *clases de asociados.*
5. En cada dominio particular D, se define una manera de escoger el representante de cada clase. A cada representante de clase se le llama una *unidad normal.*

Puede haber confusión con los conceptos *unidad* y *unidad normal,* así que se debe tener un poco de cuidado para no confundir las cosas.

- Por ejemplo, en $\mathbb{Z}$, la partición que induce la relación "es asociado de" es $\{0\}, \{1, -1\}, \{2, -2\}, ...$ y si definimos las unidades normales (representantes de clase) como los enteros no negativos, entonces $0, 1, 2, ...$ son unidades normales. Así, 0 no es una unidad, pero es una unidad normal. En los dominios de interés en este trabajo, siempre 0 es una unidad normal y 1 es la unidad normal que representa a la clase de las unidades. También el producto de unidades normales es una unidad normal.
- En $Q[x]$, los asociados de $x - 3/2$ son $\{k(x - 3/2) : k \in \mathbb{Q} - \{0\}, \}$

Definición 1 (Máximo Común Divisor)

En un dominio de integridad D, en el que se ha definido cómo escoger las unidades normales, un elemento $c \in D$ es *el* (único) máximo común divisor de a y b si es un máximo común divisor de a, b y si es una unidad normal.

Ejemplo.

Las unidades normales en $Q[x]$ son polinomios mónicos (es nuestra definición de cómo escoger el representante de clase). Luego, el máximo común divisor de los polinomios $a(x) = 48x^3 - 84x^2 + 42x - 36$ y $b(x) = -4x^3 - 10x^2 + 44x - 30$ es $x - 3/2$.

Parte normal, parte unitaria, contenido y parte primitiva.

No hemos hablado tanto sobre unidades normales para tan poquito. En realidad el cálculo eficiente de el MCD de dos polinomios a y b, requiere descomponer ambos polinomios en una *parte unitaria* y una *parte normal.* Seguidamente, la parte normal se separa en una parte puramente numérica (el contenido) y una parte puramente polinomial (la parte primitiva).

Definición 2 (Parte normal y parte unitaria)

1. En un dominio de integridad D, en el que se ha definido cómo elegir las unidades normales, la *la parte normal* de $a \in D$ se denota $n(a)$ y es la unidad normal de la clase que contiene a a.

2. La *parte unitaria* de $a \in D - \{0\}$ se denota $u(a)$ y se define como la única unidad en D tal que $a = u(a)n(a)$.

- $n(0) = 0$ y es conveniente definir $u(0) = 1$.
- En cualquier dominio integral D es conveniente definir $\text{MCD}\,(0,0) = 0$.
- $\text{MCD}\,(a,b) = \text{MCD}\,(b,a)$
- $\text{MCD}\,(a,b) = \text{MCD}\,(n(a), n(b))$
- $\text{MCD}\,(a,0) = n(a)$.

Los siguientes definiciones (y ejemplos) para unidades normales, se deben tomar en cuenta a la hora de las implementaciones.

Definiciones y ejemplos.

a) Las unidades normales en $\mathbb{Z}$ son $0, 1, 2, ...$

b) En $\mathbb{Q}$, como en cualquier campo, las unidades normales son 0 y 1. Esto es así pues todo elemento no nulo es una unidad y pertenece a la clase del 1.

c) Las unidades normales en $D[x]$ son polinomios cuyo coeficiente principal es una unidad normal en D.

- Las unidades normales en $Z[x]$ son polinomios con coeficiente principal en $\{1, 2, ...\}$
- Las unidades normales en $Q[x]$ y $\mathbb{Z}_p[x]$ (p primo) son polinomios mónicos.

d) En $\mathbb{Z}$, $n(a) = |a|$ y $u(a) = sign(a)$

e) Si $a \in \mathbb{Z}[x]$, $u(a(x)) = sign(a_n)$, $n(a(x)) = u(a(x))a(x)$ siendo $a(x) = a_n x^n + ... + a_0$.

- En $\mathbb{Z}[x]$, si $a(x) = -4x^3 - 10x^2 + 44x - 30$ entonces $n(a(x)) = 4x^3 + 10x^2 - 44x + 30$.

f) Si $a \in F[x]$ (con F campo), $n(a(x)) = \dfrac{a(x)}{a_n}$ y $u(a(x)) = a_n$ siendo $a(x) = a_n x^n + ... + a_0$.

- En $\mathbb{Q}[x]$, si $a(x) = -4x^3 - 10x^2 + 44x - 30$ entonces

  $u(a(x)) = -4$ y

  $n(a(x)) = x^3 + 5/2x^2 - 11x + 15/2$ (pues $a = u(a)n(a)$).

Dominios de Factorización Única.

Definición 3

1. Un elemento $p \in D - \{0\}$ se dice *primo* (o irreducible) si p *no* es una unidad y si $p = ab$ entonces o a es una unidad o b es una unidad.
2. Dos elementos $a, b \in D$ se dicen primos relativos si $\text{MCD}\,(a, b) = 1$
3. Un dominio integral se dice dominio de factorización única (DFU) si para todo $a \in D - \{0\}$, o a es una unidad o a se puede expresar como un producto de primos (irreducibles) tal que esta factorización es única excepto por asociados y reordenamiento.
4. En un DFU D, una factorización normal unitaria de $a \in D$ es

$$a = u(a)p_1^{e_1} \cdots p_n^{e_n}$$

donde los primos (o irreducibles) p_i's son unidades normales distintas y $e_i > 0$.

• Si $p \in D$ es primo, también lo es cualquiera de sus asociados.

• Un dominio Euclidiano también permite factorización prima única, por tanto es un DFU.

Ejemplo.

Sean $a(x) = 48x^3 - 84x^2 + 42x - 36$ y $b(x) = -4x^3 - 10x^2 + 44x - 30$.

- $a(x) = (2)(3)(2x - 3)(4x^2 - x + 2)$ en $\mathbb{Z}[x]$
- $b(x) = (-1)(2)(2x - 3)(x - 1)(x + 5)$ en $\mathbb{Z}[x]$

- $\text{MCD}\,(a,b) = x - 3/2$ en $Q[x]$ según nuestra definición de unidad normal en $\mathbb{Q}[x]$.

Existencia del MCD.

Puede ser curioso que haya dominios *no* Euclidianos para los que la existencia de divisores comunes no implica la existencia del MCD y otros donde la existencia del $\text{MCD}\,(a,b)$ no implica que éste se puede expresar como una combinación lineal de a y b.

Es conocido que en el dominio $D = \mathbb{Z}[\sqrt{-5}]$, el conjunto $\{a+b\sqrt{-5},\ a,b \in \mathbb{Z}\}$, los elementos 9 y $6+3\sqrt{-5}$ tienen los divisores comunes 3 y $2+\sqrt{-5}$ pero $\text{MCD}\left(9, 6+3\sqrt{-5}\right)$ no existe. Y también, en el dominio $F[x,y]$ (F campo) $\text{MCD}\,(x,y) = 1$ pero es imposible encontrar $P, Q \in F[x,y]$ tal que $P\,x + Q\,y = 1$.

- En un dominio Euclidiano, el $\text{MCD}\,(a,b)$ existe y además se puede expresar como una combinación lineal de a y b.

- En un DFU podemos garantizar al menos la existencia del MCD (y también calcularlo).

Teorema 1

1. Sea D un dominio Euclidiano. Si $a,b \in D$ no ambos cero, entonces $\text{MCD}\,(a,b)$ existe y es único. Además, existen $t,s \in D$ tal que $\text{MCD}\,(a,b) = sa + tb$ (Teorema de Bezout).

2. Sea D un DFU. Si $a,b \in D$ no ambos cero, entonces $\text{MCD}\,(a,b)$ existe y es único.

- La unicidad se estable como la establecimos con nuestra definición.

- La parte dos del teorema nos dice que el MCD existe no solo en en un dominio Euclidiano sino también en un DFU. Sin embargo, en un DFU no tenemos división Euclidiana, así que el cálculo requiere una *seudo*-división Euclidiana (optimizada). Curiosamente el algoritmo de cálculo en un DFU (por supuesto) se puede usar en un dominio Euclidiano y resulta ser más eficiente.

Los dominios D y $D[x_1, x_2, ...x_n]$.

Mucho de lo que podamos hacer en $D[x]$ (o $D[x_1, x_2, ...x_n]$) depende de D.

Teorema 2

1. Si R es anillo conmutativo también $R[x_1, x_2, ...x_n]$.

2. Si D es dominio integral también $D[x_1, x_2, ...x_n]$. La unidades en $D[x_1, x_2, ...x_n]$ son las unidades de D vistas como polinomios en $D[x_1, x_2, ...x_n]$.

3. Si D es un DFU también $D[x_1, x_2, ...x_n]$.

4. Si D es dominio Euclidiano, $D[x_1, x_2, ...x_n]$ es DFU pero no dominio Euclidiano.

5. Si F es campo, $F[x_1, x_2, ...x_n]$ es DFU pero no dominio Euclidiano excepto que el número de indeterminadas sea uno.

Ejemplo.

a) $\mathbb{Z}$ es un dominio Euclidiano pero $\mathbb{Z}[x]$ es un DFU.

b) $\mathbb{Q}$ y $\mathbb{Z}_p$ (p primo) son campos. $\mathbb{Q}[x]$ y $\mathbb{Z}_p[x]$ son dominios Euclidianos y $\mathbb{Q}[x_1, ..., x_v]$ y $\mathbb{Z}_p[x_1, ..., x_v]$ son DFU.

• Las operaciones de adición y multiplicación en $D[x_1, ..., x_v]$ se definen en términos de las operaciones básicas en $D[x]$. Esto se hace de manera recursiva. Nosotros identificamos $R[x, y]$ con $R[y][x]$, es decir un polinomio en x e y lo identificamos como un polinomio en x con coeficientes en $R[y]$.

Ejemplo.

- $a(x, y) = x^2 + xy + x^2y^2 + xy^3 \in \mathbb{Z}[x, y]$ lo podemos ver como un polinomio en $\mathbb{Z}[y][x]$:

$$g(x, y) = (y^2 + 1)x^2 + (y + y^3)x$$

En general, $D[x_1, ...x_v]$ lo identificamos con $R[x_2, ..., x_v][x_1]$ y entonces

$$D[x_1, ...x_v] = D[x_v][x_{v-1}]...[x_2][x_1]$$

• Asumimos que los términos no nulos de $g(x_1, ..., x_v)$ han sido ordenados según el orden lexicográfico descendente de sus exponentes, entonces el coeficiente principal de a es el coeficiente principal del primer término.

Ejemplo.

- $a(x,y) = x^2 + xy + x^2y^2 + xy^3 \in \mathbb{Z}[x,y]$.
- $a(x,y) = 5x^3y^2 - x^2y^4 - 3x^2y^2 + 7xy^2 + 2xy - 2x + 4y^4 + 5 \in \mathbb{Z}[x,y]$.

• El grado de $a(x_1,...,x_v)$ en la variable i se denota $\text{grado}_i(a(x_1,...,x_v))$

1.3. Algoritmo de Euclides, Algoritmo Primitivo de Euclides y Secuencias de Residuos Polinomiales.

Bien, vamos ahora a dedicarnos a los algoritmos (variaciones del algoritmo de Euclides) para el cálculo del MCD. Aunque iniciamos con el algoritmo de Euclides en un dominio Euclidiano, nos interesa la implementación solo en un DFU, porque esta implementación también se puede usar en un dominio Euclidiano y es más eficiente. La forma extendida del algoritmo de Euclides calcula el MCD y la combinación lineal $sa + tb$ y solo la podemos implementar en un dominio Euclidiano.

Algoritmo de Euclides.

Podemos ver el algoritmo de Euclides para calcular el MCD de dos polinomios $a(x)$ y $b(x)$, con coeficientes en un *campo*, como una construcción de una sucesión de residuos. Si $\text{grado}\, a(x) \geq \text{grado}\, b(x)$, entonces el algoritmo de Euclides construye una sucesión de polinomios $r_0(x), r_1(x), ..., r_k(x)$. La inicialización de esta sucesión es $r_0(x) = a(x)$, $r_1(x) = b(x)$. Luego,

$$\begin{aligned}
r_0(x) &= r_1(x)q_1(x) + r_2(x) \quad \text{con grado } r_2(x) < \text{ grado } r_1(x) \\
r_1(x) &= r_2(x)q_2(x) + r_3(x) \quad \text{con grado } r_3(x) < \text{ grado } r_2(x) \\
\cdots &\quad \cdots \\
r_{k-2}(x) &= r_{k-1}(x)q_{k-1}(x) + r_k(x) \quad \text{con grado } r_k(x) < \text{ grado } r_{k-1}(x) \\
r_{k-1}(x) &= r_k(x)q_k(x) + 0
\end{aligned}$$

Entonces $r_k(x) = \text{MCD}\,(a(x), b(x))$ cuando es normalizado adecuadamente para que se convierta en una unidad normal. Formalmente

Teorema 3

1. Dados $a, b \in D$ $(b \neq 0)$ donde D es un dominio Euclidiano, sean $q, r \in D$ (el cociente y el residuo) que satisfacen

$$a = bq + r \quad \text{con} \quad r = 0 \quad \text{o} \quad v(r) < v(b)$$

Entonces $\text{MCD}\,(a,b) = \text{MCD}\,(b,r)$.

2. Sean $a, b \in D$, D un dominio Euclidiano, con $v(a) \geq v(b) > 0$. Consideremos la sucesión de residuos $r_0(x), r_1(x), r_2(x), ...,$ (con $r_0(x) = a(x)$, $r_1(x) = b(x)$) definida más arriba. Entonces, efectivamente existe un índice $k \geq 1$ tal que $r_{_{k+1}}(x) = 0$ y

$$\text{MCD}\,(a(x), b(x)) = n(r_k(x))$$

Ejemplo.

Sean $a(x) = x^5 - 32$ y $b(x) = x^3 - 8$, polinomios de $\mathbb{Q}[x]$.

El proceso requiere división usual de polinomios.

a) $r_0(x) = x^5 - 32$

b) $r_1(x) = x^3 - 8$

Dividimos $r_0(x)$ por $r_1(x)$,

$x^5 - 32$	$x^3 - 8$
$\cdots$	x^2
residuo: $8x^2 - 32$	

c) $r_2(x) = 8x^2 - 32$.

Ahora, dividimos $r_1(x)$ por $r_2(x)$,

$x^3 - 8$	$8x^2 - 32$
$\cdots$	$x/8$
residuo: $4x - 8$	

d) $r_3(x) = 4x - 8$.

Ahora, dividimos $r_2(x)$ por $r_3(x)$,

$8x^2 - 32$	$4x - 8$
$\cdots$	$2x + 4$
residuo: 0	

e) $r_4(x) = 0$.

Finalmente, $\text{MCD}\left(x^5 - 32, x^3 - 8\right) = r_3(x)/4 = x - 2$.

Ejemplo.

Sean $a(x) = x^8 + x^6 - 3x^4 - 3x^3 + 8x^2 + 2x - 5$ y $b(x) = 3x^6 + 5x^4 - 4x^2 - 9x + 21$. Si aplicamos el algoritmo de Euclides para calcular el $\text{MCD}\,(a(x), b(x))$, se obtiene la siguiente sucesión de residuos,

$$r_3(x) = -117/25x^2 - 9x + 441/25,$$

$$r_4(x) = 233150/19773x - 102500/6591,$$

$$r_5(x) = -1288744821/543589225.$$

Por lo tanto, $\text{MCD}\,(a, b) = 1$ (pues el máximo común divisor es un unidad en $\mathbb{Q}[x]$).

• En este ejemplo se puede observar uno de los problemas del algoritmo de Euclides: el crecimiento de los coeficientes.

• Además tenemos otro problema, el algoritmo de Euclides requiere que el dominio de coeficientes sea un campo.

Para calcular el MCD de polinomios en los dominios $\mathbb{Z}[x], \mathbb{Z}[x_1, ..., x_k], \mathbb{Z}_p[x_1, ..., x_k]$ y $\mathbb{Q}[x_1, ..., x_k]$ todos DFU (pero no dominios Euclidianos), no podemos usar directamente el algoritmo de Euclides. En cambio podemos usar una variante: la seudo-división.

Algoritmo Primitivo de Euclides y Sucesiones de Residuos.

Antes de pasar a las definiciones y teoremas, vamos a describir los problemas que tenemos y como queremos resolverlos.

Lo que queremos es, encontrar el MCD en $D[x]$ usando solo aritmética en el dominio $D[x]$ más bien que trabajar en el *campo cociente* de D como nuestro campo de coeficientes. ¿Porqué?, bueno; ya vimos que si queremos encontrar por ejemplo, el MCD de dos polinomios en $\mathbb{Z}[x]$ no podemos recurrir del todo a $\mathbb{Q}[x]$ porque aquí el MCD de los mismos polinomios da otro resultado.

Una manera de usar solo aritmética en D es construir una sucesión de *seudo-residuos* (denotados "prem") usando seudo-división, que sea válida en un DFU.

Si $\text{grado}\,(a(x)) = m$ y $\text{grado}\,(b(x)) = n$ ($m \geq n$), en el algoritmo de Euclides usual hacemos división de polinomios: en cada división, dividimos por el coeficiente principal de $b(x)$, $m - n + 1$ veces antes que el proceso se detenga (los nuevos dividendos son polinomios de grados $m - 1, m - 2, ..., m - n$).

En $D[x]$, la idea es esta: podemos hacer que cada división sea "exacta", multiplicando el coeficiente principal de $a(x)$ por α^{m-n+1} donde α es el coeficiente principal de $b(x)$.

Ejemplo.

Sean $a(x) = x^8 + x^6 - 3x^4 - 3x^3 + 8x^2 + 2x - 5$ y $b(x) = 3x^6 + 5x^4 - 4x^2 - 9x + 21$ en $\mathbb{Z}[x]$.

En este caso $\alpha = 3$ y $m - n + 1 = 3$. Entonces, en vez de dividir $a(x)$ por $b(x)$ (que no se puede en $\mathbb{Z}[x]$), dividimos

$$3^3(x^8 + x^6 - 3x^4 - 3x^3 + 8x^2 + 2x - 5) \quad \text{por} \quad 3x^6 + 5x^4 - 4x^2 - 9x + 21.$$

La división es exacta aunque el dominio de coeficientes sea $\mathbb{Z}$. Obviamente, el problema del crecimiento de los coeficientes en los residuos va a empeorar, de hecho los coeficientes de los residuos crecen exponencialmente. En este caso, los dos últimos *seudo-residuos* (usando *seudo-división*) son

i	$r_i(x)$
4	$125442875143750\,x - 1654608338437500$
5	$125933387955500743100931141992187500$

El resultado al que llegamos es $\text{MCD}\,(a(x), b(x)) = 1$.

• La seudo-división resuelve el problema de aplicar el algoritmo de Euclides en un DFU.

• El problema del creciemiento de los coeficientes lo podemos resolver en una primera instancia y a un costo relativamente alto, dividiendo el seudo-residuo $i+1$ por el máximo común divisor de sus coeficientes (denotado "cont"),

$$r_{i+1} = \frac{\alpha_i^{\delta_i+1} r_i(x) - q_i(x) r_{i-1}(x)}{\beta_i}$$

donde conocemos todos los ingredientes para calcular r_{i+1}, a saber: $\beta_i = \text{cont}(\text{prem}(r_i(x), r_{i-1}(x)))$, es decir el contenido del residuo en la división de $\alpha_i^{\delta_i+1} r_i(x)$ por $r_{i-1}(x)$ ($q_i(x)$ es el cociente), α_i es el coeficiente principal de $r_i(x)$ y $\delta_i = \text{grado}(r_{i-1}(x)) - \text{grado}(r_i(x))$.

Necesitamos algunas cosas antes de establecer el algoritmo.

Definición 4

1. Un polinomio no nulo $a(x) \in D[x]$, con D DFU, se dice *primitivo* si es una unidad normal en $D[x]$ y si sus coeficientes son primos relativos. En particular, si $a(x)$ solo tiene un término no nulo entonces es primitivo si y sólo si es mónico.

2. El *contenido* de un polinomio no nulo $a(x) \in D[x]$, con D DFU, se denota $\text{cont}(a(x))$ y se define como el MCD de los coeficientes de $a(x)$

• Con estas definiciones podemos podemos ver que

$$a(x) = u(a(x))n(a(x)) \; = \; u(a(x))\,\text{cont}(a(x))\,\text{pp}(a(x)$$

donde $\text{pp}(a(x))$ es un polinomio primitivo, llamado la *parte primitiva* de $a(x)$. Es conveniente definir $\text{cont}(0) = 0$ y $\text{pp}(0) = 0$.

Ejemplo.

a) Consideremos $a(x) = 48x^3 - 84x^2 + 42x - 36$ y $b(x) = -4x^3 - 10x^2 + 44x - 30$.

- En $\mathbb{Z}[x]$, $\text{cont}(a) = 6$ y $\text{pp}(a) = 8x^3 - 14x^2 + 7x - 6$
- En $\mathbb{Z}[x]$, $\text{cont}(b) = 2$ y $\text{pp}(b) = 2x^3 + 5x^2 - 22x + 15$.

  (Recordemos que $b(x) = u(b)\text{cont}(b)\text{pp}(b)$ y en $\mathbb{Z}$ $u(b) = -1$)

- En $\mathbb{Q}[x]$, $\text{cont}(a) = 1$ y $\text{pp}(a) = x^3 - 7/4x^2 + 7/8x - 3/4$.
- En $\mathbb{Q}[x]$, $\text{cont}(a) = 1$ y $\text{pp}(a) = x^3 + 5/2x^2 - 11x + 15/2$.

b) En un campo F, $\text{MCD}(a, b) = 1$ (a, b no ambos nulos). En $F[x]$, $\text{cont}(a(x)) = 1$ $(a \neq 0)$ y $\text{pp}(a(x)) = n(a(x))$, es decir $a(x)$ queda mónico.

• En $D[x_1, ..., x_v]$, la parte unitaria y el contenido se definen de igual manera que en D.

Ejemplo.

a) $a(x, y) = (y^2 + 1)x^2 + (y + y^3)x \in \mathbb{Z}[y][x]$.

- $u(a(x, y)) = 1$ pues en $D = \mathbb{Z}[x]$, la parte unitaria es el signo del coeficiente principal.
- $\text{cont}(a(x, y)) = \text{MCD}(y^2 + 1, y + y^3) = y^2 + 1$
- $\text{pp}(a(x, y)) = x^2 + yx$.

- Recordemos que $n(a(x,y)) = \text{cont}(a(x,y))\text{pp}(a(x,y))$.

b) $a(x,y) = yx^2 + (y^2+1)x + y \in \mathbb{Z}[y][x]$.
- $u(a(x,y)) = 1$
- $\text{cont}((a(x,y))) = \text{MCD}(y, y^2+1, y) = 1$
- $\text{pp}((a(x,y))) = yx^2 + (y^2+1)x + y$.

c) $a(x,y) = (-30y)x^3 + (90y^2+15)x^2 - (60y)x + (45y^2) \in \mathbb{Z}[y][x]$.
- $u(a(x,y)) = -1$ pues en $D = \mathbb{Z}[x]$, la parte unitaria es el signo del coeficiente principal.
- Ahora operamos sobre $n(a(x,y)) = (30y)x^3 - (90y^2+15)x^2 + (60y)x - (45y^2)$.

  $\text{cont}(a(x,y)) = \text{MCD}(30y, -90y^2-15, 60y, -45y^2) = 15$

  $\text{pp}(a(x,y)) = (2y)x^3 - (6y^2+1)x^2 + (4y)x - (3y^2)$

Lema 1 (Lema de Gauss)

1. El producto de polinomios primitivos es primitivo

2. $\text{MCD}\,(a(x), b(x)) = \text{MCD}\,(\text{cont}(a(x)), \text{cont}(b(x))) \cdot \text{MCD}\,(\text{pp}(a(x)), \text{pp}(b(x)))$

• El cálculo de $\text{MCD}\,(\text{cont}(a(x)), \text{cont}(b(x)))$ se hace en D, así que nos podemos concentrar en el cálculo del MCD de polinomios primitivos, es decir en el cálculo de $\text{MCD}\,(\text{pp}(a(x)), \text{pp}(b(x)))$

Propiedad de Seudo-División en un DFU

Sea $D[x]$ un dominio de polinomios sobre un DFU D. Para todo $a(x), b(x) \in D[x]$ con $b(x)$ no nulo y $\text{grado}\,(a(x)) \geq \text{grado}\,(b(x))$, existen polinomios únicos $q(x)$, $r(x) \in D[x]$ (llamados seudo-cociente y seudo-residuo) tal que

$$\alpha^{\delta+1} a(x) = b(x)q(x) + r(x), \quad \text{grado}\,(a(x)) \geq \text{grado}\,(b(x))$$

donde α es el coeficiente principal de $b(x)$ y $\delta = m - n$ donde $m = \text{grado}(a(x))$ y $n = \text{grado}\,(b(x))$.

• Para efectos de implementación, usamos la notación "pquo $(a(x), b(x))$" para el seudo-cociente y "prem $(a(x), b(x))$" para el seudo-residuo.

• Es conveniente extender la definición de "pquo" y "prem" para el caso grado $(a(x)) <$ grado $(b(x))$, haciendo pquo $(a(x), b(x)) = 0$ y prem $(a(x), b(x)) = a(x)$.

• "pquo" y "prem" se obtienen haciendo la división de polinomios usual (entre $\alpha^{\delta+1}a(x)$ y $b(x)$), solo que ahora la división es exacta en el dominio de coeficientes D.

Cálculo del MCD en $D[x]$.

La propiedad de seudo-división nos da, de manera directa, un algoritmo para calcular el MCD en $D[x]$ con D DFU. Como habíamos notado antes, basta con restringir nuestra atención a la parte primitiva de los polinomios, es decir nos restringimos al cálculo del MCD para polinomios primitivos.

Teorema 4

Sea $D[x]$ un dominio de polinomios sobre un DFU. Dados polinomios *primitivos* $a(x), b(x) \in D[x]$ con $b(x)$ no nulo y grado $(a(x)) \geq$ grado $(b(x))$, sean $q(x)$ y $r(x)$ el seudo-cociente y el seudo-residuo, entonces

$$\text{MCD}\,(a(x), b(x)) = \text{MCD}\,(b(x), \text{pp}(r(x))) \tag{1.3.1}$$

Prueba. Usamos la propiedad de seudo-división. Si $a(x)$ y $b(x)$ tienen grado m y n, respectivamente, y si δ es el coeficiente principal de $b(x)$, entonces

$$\delta^{m-n+1}a(x) = b(x)q(x) + r(x)$$

Luego, aplicando las propiedades de MCD y usando el hecho de que $a(x)$, $b(x)$ son primitivos, tenemos

$$\begin{aligned} \text{MCD}\left(\delta^{m-n+1}a(x), b(x)\right) &= \text{MCD}\,(b(x), r(x)) \\ &= \text{MCD}\left(\delta^{m-n+1}, 1\right) \cdot \text{MCD}\,(a(x), b(x)) \\ &= \text{MCD}\,(a(x), b(x)) \end{aligned}$$

De manera similar,

$$\begin{aligned} \text{MCD}\,(b(x), r(x)) &= \text{MCD}\,(1, \text{cont}(r(x))) \cdot \text{MCD}\,(b(x), \text{pp}(r(x))) \\ &= \text{MCD}\,(b(x), \text{pp}(r(x)).) \end{aligned}$$

• La ecuación 1.3.1 define un método de iteración para calcular el MCD de dos polinomios primitivos en $D[x]$ y esta iteración es finita pues grado $(r(x))$ $<$ grado $(b(x))$ en cada paso.

• En el algoritmo se calcula la sucesión de residuos $\mathrm{pp}(r(x))$, por esto, a este algoritmo se le llama el algoritmo primitivo de Euclides.

Algoritmo 1.3.1: Algoritmo Primitivo de Euclides.

Entrada: Polinomios $a(x), b(x) \in D[x]$, D DFU.
Salida: $c(x) = \mathrm{MCD}\,(a(x), b(x))$

```
c(x) = pp(a(x));
d(x) = pp(b(x));
while d(x) ≠ 0 do
    r(x) = prem(c(x), d(x));
    c(x) = d(x);
    d(x) = pp(r(x));
λ = MCD(cont(a(x)), cont(b(x)));
g(x) = λ c(x);
return g(x);
```

Ejemplo.

Sean $a(x), b(x) \in \mathbb{Z}[x]$. $a(x) = x^8 + x^6 - 3x^4 - 3x^3 + 8x^2 + 2x - 5$ y $b(x) = 3x^6 + 5x^4 - 4x^2 - 9x + 21$.

La sucesión de valores calculada por el algoritmo para $r(x)$, $c(x)$ y $d(x)$ es

n	$r(x)$	$c(x)$	$d(x)$
0	$-$	$x^8 + x^6 - 3x^4 - 3x^3 + 8x^2 + 2x - 5$	$3x^6 + 5x^4 - 4x^2 - 9x + 21$
1	$-$	$27x^8 + 27x^6 - 81x^4 - 81x^3 + 216x^2 + 54x - 135$	$x^6 + 5x^4 - 4x^2 - 9x + 21$
2	$5x^4 - x^2 + 3$	$375x^6 + 625x^4 - 500x^2 - 1125x + 2625$	$5x^4 - x^2 + 3$
3	$13x^2 + 25x - 49$	$10985x^4 - 2197x^2 + 6591$	$13x^2 + 25x - 49$
4	$4663x - 6150$	$282666397x^2 + 543589225x - 1065434881$	$4663x - 6150$
5	0	$4663x - 6150$	1

Cuadro 1.1: Algoritmo Primitivo de Euclides aplicado a $a(x)$ y $b(x)$

Lo que retorna el algoritmo es 1. Observe que $a(x)$ y $b(x)$ son primitivos, así que no hay cambio en la iteración $n = 0$.

Ejemplo en $\mathbb{Z}[x, y]$.

Sean $a(x,y) = (y^2+1)x^2+(y+y^3)\,x$ y $b(x,y) = yx^2+(y^2+1)x+y$. Para calcular $\text{MCD}\,(a(x,y), b(x,y))$, vemos a estos polinomios como elementos de $\mathbb{Z}[y][x]$.

Paso 1. $c(x) = \text{pp}(a(x,y)) = x^2+yx$, pues $u(a(x,y)) = 1$ y $\text{cont}(a(x,y)) = \text{MCD}(y^2+1, y+y^3) = y^2+1$.

Paso 2. $d(x) = \text{pp}(b(x,y)) = yx^2 + (y^2+1)x + y$.

Paso 3. **While** $d(x) \neq 0$ **do**

Paso 3.1 $r(x) = \text{prem}(c(x), d(x)) = -x - y$, pues

$yx^2 + y^2x$	$yx^2 + (y^2+1)x + y$
$-yx^2 - (y^2+1)x - y$	1
residuo: $-x - y$	

Paso 3.2 $c(x) = d(x)$

Paso 3.3 $d(x) = \text{pp}(r(x)) = x + y$ pues $r(x) = u(r(x))\text{cont}(r(x))\text{pp}(x) = (-1)\cdot 1 \cdot (x+y)$

Paso 3.4 $r(x) = \text{prem}(c(x), d(x)) = 0$, pues

$yx^2 + (y^2+1)x + y$	$x + y$
$-yx^2 - y^2x$	$yx + 1$
$x + y$	
$-x - y$	
residuo: 0	

Paso 3.5 $c(x) = d(x) = x + y$

Paso 3.6 $d(x) = \text{pp}(r(x)) = 0$.

Fin del **While.**

Paso 4. $\lambda = \text{MCD}(\text{cont}(a(x)), \text{cont}(b(x))) = \text{MCD}\left(y^2+1, 1\right) = 1$

Paso 5. $g(x) = \lambda\, c(x) = x + y$.

Retorna: $\text{MCD}\,(a(x,y), b(x,y)) = x + y$.

• Este algoritmo, por supuesto, también se puede usar en el dominio Euclidiano $F[x]$ (F campo).

• En $\mathbb{Z}[x]$, el tiempo estimado de ejecución de este algoritmo (en términos del número de operaciones con palabras de máquina) es

$$\boldsymbol{O}\left(n^3\, m\, \varrho\, \log^2(nA)\right)$$

donde m, n son los grados de los polinomios, la constante A acota superiormente a ambos $||a(x)||_\infty$ y a $||b(x)||_\infty$ y $\varrho = \text{Máx}_{1\leq i\leq k}\left\{\text{grado}\, r_{i-1}(x) - \text{grado}\, r_i(x)\right\}$ con k el subíndice del último residuo.

1.4. Algoritmos PRS y el Algoritmo PRS Subresultante

En el algoritmo primitivo de Euclides, en cada iteración calculamos

$$r_i(x) = \text{prem}(c(x), d(x))$$

es decir, en cada iteración se hace seudo-división de $\text{pp}(r_{i-1}(x))$ por $\text{pp}(r_i(x))$.

Así, el algoritmo construye una sucesión de seudo-residuos $r_0(x), r_1(x), ..., r_k(x)$ con inicialización $r_0(x) = \text{pp}(a(x))$, $r_1(x) = \text{pp}(b(x))$ y

$$\begin{aligned}
\alpha_1 r_0(x) &= r_1(x)q_1(x) + r_2(x) \\
\alpha_2\, r_1(x) &= r_2(x)q_2(x) + r_3(x) \\
\cdots &\quad \cdots \\
\alpha_{k-1}\, r_{k-2}(x) &= r_{k-1}(x)q_{k-1}(x) + r_k(x) \\
\alpha_k\, r_{k-1}(x) &= r_k(x)q_k(x).
\end{aligned}$$

con $\alpha_i = r_i^{\delta_i+1}$ donde $r_i = \text{cp}(r_i(x))$ (cp=coeficiente principal) y $\delta_i = \text{grado}\, r_{i-1}(x) - \text{grado}\, r_i(x)$.

Ejemplo. En el cuadro 1.1 se puede observar la relación entre los coeficientes principales de los residuos $r_i(x)$ y r_{i-1}

i	$r_i(x)$	$\alpha_i\, r_{i-1}$		
0	$-$			
1	$-$	$3^3\, r_0(x)$	$=$	$27x^8 + 27x^6 - 81x^4 - 81x^3 + 216x^2 + 54x - 135$
2	$5x^4 - x^2 + 3$	$5^3 \cdot 3\, r_1(x)$	$=$	$375x^6 + 625x^4 - 500x^2 - 1125x + 2625$
3	$13x^2 + 25x - 49$	$13^3 \cdot 5\, r_2(x)$	$=$	$10985x^4 - 2197x^2 + 6591$
4	$4663x - 6150$	$4663^3 \cdot 13\, r_3(x)$	$=$	$282666397x^2 + 543589225x - 1065434881$

Cuadro 1.2: $r_i(x) = \text{prem}(c(x), d(x))$

Esta sucesión de residuos es un caso particular de un caso más general, las llamadas *Sucesiones de Residuos Polinomiales* (PRS).

Definición 5 (Sucesión de Residuos Polinomiales)

Sean $a(x), a(x) \in R[x]$ con $\text{grado}\, a(x) \geq \text{grado}\, b(x)$. Una sucesión de residuos polinomiales (PRS, por sus siglas en inglés) para $a(x)$ y $b(x)$ es una sucesión de polinomios $r_0(x), r_1(x), ..., r_k(x) \in R[x]$ que satisfacen

1. $r_0(x) = a(x)$, $r_1(x) = a(x)$ (inicialización).

2. $\alpha_i\, r_{i-1}(x) = q_i(x) r_i + \beta_i\, r_{i+1}(x)$

3. $\text{prem}(\, r_{k-1}(x),\, r_k(x)) = 0\,.$

El principal objetivo en la construcción de PRS para dos polinomios dados es, además de mantener todas las operaciones en el dominio R, escoger β_i de tal manera que los coeficientes de los residuos se mantengan tan pequeños como sea posible y que este proceso sea lo más "barato" (menor costo) posible. Inicialmente la teoría fue desarrollada por Sylvester y Trudi en el siglo diecinueve (mientras desarrollaban la teoría de ecuaciones) y el algoritmo PRS Subresultante es una variación perfeccionada desarrollada por Collins y Brown a finales de los sesentas (ver [**?**]).

- Si $\alpha_i = cp_i^{\delta_i+1}$ donde $cp_i =$ coeficiente principal de $r_i(x)$ y $\beta_i = 1$, obtenemos el llamado PRS Euclidiano. El resultado es un crecimiento exponencial (en el número de bits) de los coeficientes.

- En el otro extremo,

$$\alpha_i = cp_i^{\delta_i+1} \quad \text{y} \quad \beta_i = \text{cont}(\text{prem}(r_{k-1}(x),\, r_k(x)))$$

  es decir, dividimos los seudo-residuos por el máximo común divisor de sus coeficientes. Esta escogencia tiene éxito en mantener los coeficientes los más pequeños posibles pero el costo de calcular los MCD's generalmente no es bajo. Esta variación es llamada PRS primitiva.

- El siguiente paso fue tratar de hallar divisores del contenido sin calcular MCD's. Cerca de 1970 Collins, Brown y Traub, reinventaron la teoría de polinomios subresultantes como variantes de las matrices de Sylvester ([**?**]) y hallaron que coincidían con los residuos en el algoritmo de Euclides, excepto por un factor. Ellos dieron fórmulas para calcular este factor e introdujeron el concepto de PRS. El resultado final es el Algoritmo "PRS Subresultante" que permite un crecimiento lineal de los coeficientes y mantiene el cálculo en el dominio D.

  En el Algoritmo "PRS Subresultante",

$$\alpha_i = \text{cp}_i^{\delta_i+1}, \quad \beta_1 = (-1)^{\delta_1+1}, \quad \beta_i = -\text{cp}_{i-1}\, \psi_i^{\delta_i}, \;\; 2 \le i \le k$$

  donde cp_i es el coeficiente principal de $r_i(x)$, $\delta_i = \text{grado}\, r_{i-1}(x) - \text{grador}_i(x)$ y ψ_i se define de manera recursiva: $\psi_1 = -1$, $\psi_i = (-\text{cp}_{i-1})^{\delta_{i-1}} \psi_{i-1}^{1-\delta_{i-1}}$; $2 \le i \le k$

  Si ponemos $q_i(x) = \text{pquo}(r_{i-1}(x), r_i(x))$, entonces el siguiente residuo se calcula como

$$r_{i+1} = \frac{\alpha_i\, r_{i-1}(x) \;-\; q_i(x) r_i}{\beta_i} \qquad \text{(división "exacta")}$$

Hay que notar que en $\psi_i = (-\mathrm{cp}r_{i-1})^{\delta_{i-1}}\psi_{i-1}^{1-\delta_{i-1}}$ se tiene $1-\delta_{i-1} \leq 0$, pero se trata de una "división exacta" si $1-\delta_{i-1} < 0$ (esto fue un problema abierto hasta el 2003, [**?**]).

- El tiempo estimado de ejecución de este algoritmo, en algunos casos, es similar al del algoritmo primitivo de Euclides.

  En estimaciones ([**?**]) en términos de número de operaciones de bits sobre polinomios de grado a lo sumo n y con coeficientes de longitud a lo sumo n (en bits, menor o igual a 2^n), se obtuvo, tiempos de orden $\boldsymbol{O}\left(n^6\right)$, ignorando factores logarítmicos, para el algoritmo primitivo de Euclides y también para el algoritmo "PRS Subresultante" (ver [**?**]). En estos mismos experimentos se obtuvo tiempos de $\boldsymbol{O}\left(n^4\right)$ para el algoritmo heurístico y tiempos de $\boldsymbol{O}\left(n^4\right)$ y $\boldsymbol{O}\left(n^3\right)$ para otros dos algoritmos modulares.

- En las notas de implementación de *Mathematica* se indica que se implementa SPMOD y, en el improbable caso de que este algoritmo falle, *Mathematica* salta al algoritmo PRS Subresultante.

En el algoritmo que sigue, se pone $\mathbf{quo}(a,b)$ para indicar el cociente de la división usual. Hay que notar que $\mathbf{pquo}(r_{i-1}(x), r_i(x)) = \mathbf{quo}(\alpha_i\, r_{i-1}(x), r_i(x))$

Algoritmo 1.4.1: Algoritmo PRS Subresultante.

Entrada: Polinomios $a(x), b(x) \in D[x]$, $\operatorname{grado} a(x) \geq \operatorname{grado} b(x)$, D DFU.
Salida: $c(x) = \mathrm{MCD}\,(a(x), b(x))$

$r_0 = a(x)$;
$r_1 = b(x)$;
$deg_0 = \mathrm{grado}(r_0)$, $deg_1 = \mathrm{grado}(r_1)$, $cp_0 =$ coeficiente principal de r_0;
$cp_1 =$ coeficiente principal de r_1, $\delta_1 = deg_0 - deg_1$, $\delta_0 = \delta_1$;
$\alpha_1 = cp_1^{\delta_1+1}$, $\beta_1 = (-1)^{\delta_1+1}$, $\psi_1 = -1$, $\psi_0 = -1$;
while $r_1 \neq 0$ **do**
    $c = \alpha_1\, r_0$, $q = \mathrm{quo}(c, r_1)$, $r_0 = r_1 w$ $r_1 = \mathrm{quo}(c - q \cdot r_1, \beta_1)$;
    $deg_0 = \mathrm{grado}(r_0)$, $deg_1 = \mathrm{grado}(r_1)$;
    $cp_0 =$ coeficiente principal de r_0;
    $cp_1 =$ coeficiente principal de r_1;
    $\delta_0 = \delta_1$, $\delta_1 = deg_0 - deg_1$;
    $\alpha_1 = cp_1^{\delta_1+1}$, $\psi_0 = \psi_1$;
    if $\delta_0 > 0$ **then**
        $\psi_1 = \mathrm{quo}(-cp_0^{\delta_0}, \psi_0^{\delta_0-1})$;
    else
        $\psi_1 = -cp_0^{\delta_0} \cdot \psi_0$
    $\beta_1 = -cp_0 \cdot \psi_1^{\delta_1}$;
Normalizar salida;
$r_0 =$ MCD $(\mathrm{cont}(a(x)), \mathrm{cont}(b(x))) \cdot \mathrm{pp}(r_0)$;
return r_0;

• Hay que agregar la línea 32 pues antes de esta línea, $r_0(x)$ es solo un asociado del $\text{MCD}\,(a(x), b(x))$.

Ejemplo.

Sean $a(x) = x^8 + x^6 - 3x^4 - 3x^3 + 8x^2 + 2x - 5$ y $b(x) = 3x^6 + 5x^4 - 4x^2 - 9x + 21$.

Al aplicar el algoritmo PRS Subresultante, tal y como está descrito más arriba obtenemos los restos

i	$r_i(x)$
2	$15x^4 - 3x^2 + 9$
3	$65x^2 + 125x - 245$
4	$9326x - 12300$
5	260708

Ejemplo.

Sean $a(x) = 48x^3 - 84x^2 + 42x - 36$ y $b(x) = -4x^3 - 10x^2 + 44x - 30$.

Al aplicar el algoritmo PRS Subresultante, tal y como está descrito más arriba, antes de la línea 32, tenemos $r_0 = 1232640x - 1848960$. Con la normalización queda $r_0 = \text{MCD}\,(6, 2) \cdot (2x - 3) == 4x - 6$.

1.5. Algoritmo Heurístico.

Primero vamos a ver una idea muy general.

Consideremos dos polinomios $a(x)$, $b(x) \in \mathbb{Z}[x]$. Sea $\xi \in \mathbb{Z}$ y calculemos $\text{MCD}\,(a(\xi), b(\xi)) = \gamma$. Veamos que podría pasar

Sean $a(x) = 4x^3 - 16x^2 + 4x + 24$ y $b(x) = 8x^3 - 152x + 240$.

- $a(x) = 4(x - 3)(x - 2)(x + 1)$,
- $b(x) = 8(x - 3)(x - 2)(x + 5)$,
- $\text{MCD}\,(a(x), b(x)) = 4(x - 3)(x - 2) = 4x^2 - 20x + 24$ en $\mathbb{Z}[x]$.

Sea $g(x) = \text{MCD}\,(a(x), b(x))$.

- si $\xi = 5$ entonces $g(5) = 48 \neq 24 = \text{MCD}\,(a(5), b(5)) = \gamma$,
- si $\xi = 10$ entonces $g(10) = 224 = \text{MCD}\,(a(10), b(10)) = \gamma$,

Si $\gamma = u_0 + u_1\xi + ... + u_n\xi^n$, $u_i \in \mathbb{Z}_\xi$, para un ξ suficientemente grande, podría pasar que el polinomio $\overline{g}(x) = u_0 + u_1x + ... + u_nx^n$ sea exactamente $g(x)$.

En nuestro ejemplo,

- si $\xi = 10$, $\gamma = 224$.
- $224 = 4 + 2 \times 10 + 2 \times 10^2$
- $\overline{g}(x) = 4 + 2x + 2x^2$ pero no divide a $a(x)$ ni a $b(x)$.
- si $\xi = 50$, $\gamma = 9024$.
- $9024 = 24 - 20 \times 50 + 4 \times 50^2$
- $\overline{g}(x) = 24 - 20x + 4x^2$ que si divide $a(x)$ y a $b(x)$ y de hecho es el MCD.

La idea es que si $\gamma = u_0 + u_1\xi + ... + u_n\xi^n$, $u_i \in \mathbb{Z}_\xi$, para un ξ suficientemente grande, podría pasar que el polinomio $\overline{G}(x) = u_0 + u_1x + ... + u_nx^n$ sea exactamente $G(x)$. Las condiciones de prueba se establecen más adelante.

1.5.1. Representación ξ-ádica de un número y de un polinomio.

La representación ξ-ádica de $\gamma \in \mathbb{Z}$ es

$$\gamma = u_0 + u_1\xi + ... + u_d\xi^d \tag{1.5.2}$$

con $u_i \in \mathbb{Z}_\xi$. Aquí d es el más pequeño entero tal que $\xi^{d+1} > 2|\gamma|$.

Para el cálculo de los u_i's es más conveniente la representación simétrica de $\mathbb{Z}_\xi$, a saber

$$\mathbb{Z}_\xi = \{i \in \mathbb{Z} \ : \ \xi/2 < i \leq \xi/2\}$$

Por ejemplo, $\mathbb{Z}_5 = \{-2, -1, 0, 1, 2\}$ y $\mathbb{Z}_6 = \{-2, -1, 0, 1, 2, 3\}$.

Esta representación permite valores de γ negativos.

En el algoritmo que sigue, rem (a, b) denota el residuo de la división de a por b.

Algoritmo 1.5.1: Imagen módulo p de un número en la representación simétrica de $\mathbb{Z}_p$.

Entrada: $m, p \in \mathbb{Z}$

Salida: $u = m \,(\text{mód}\, p)$ en la representación simétrica de $\mathbb{Z}_p$, es decir $-p/2 < u \leq p/2$

```
1 u = rem(m, p);
2 if u > p/2 then
3 |  u = u − p
4 return u;
```

Para ir introduciendo notación que usaremos en el futuro, sea $\phi_p : \mathbb{Z} \longrightarrow \mathbb{Z}_p$ el homomorfismo definido por $\phi_p(a) = a \,(\text{mód}\, p)$, es decir el residuo de la división por p pero en la representación simétrica.

- De la ecuación 1.5.2, $\gamma \equiv u_0 \,(\text{mod}\, \xi)$ y entonces,

$$u_0 = \phi_\xi(\gamma) \tag{1.5.3}$$

- $\gamma - u_0$ divide ξ por lo que, de acuerdo al item anterior,

$$\frac{\gamma - u_0}{\xi} = u_1 + u_2\xi + ... + u_n\xi^{d-1}$$

de donde $u_1 = \phi_\xi\left(\dfrac{\gamma - u_0}{\xi}\right)$

- Continuando de esta manera

$$u_i = \phi_\xi\left(\frac{\gamma \ - \ (u_0 + u_1\xi + ... + u_{i-1}\xi^{i-1})}{\xi^i}\right), \ i = 1, ..., d \tag{1.5.4}$$

Ejemplo. $\xi = 50$, y $\gamma = 9024$ entonces $9024 = 24 - 20 \times 50 + 4 \times 50^2$. Es decir $u_0 = 24$, $u_1 = -20$ y $u_2 = 4$.

- La representación $\xi-$ádica de un polinomio $a(x) \in \mathbb{Z}[x]$ es

$$a(x) = \sum_e u_e x^e \quad \text{con} \quad u_e = \sum_{i=0}^{\boldsymbol{n}} u_{e,i}\xi^i, \ u_{e,i} \in \mathbb{Z}_\xi.$$

con n el entero más pequeño tal que $\xi^{n+1} > 2|u_{\max}|$, donde $u_{\max} = \max_e |u_e|$.

- $a(x) = \sum_e u_e x^e = \sum_e \left(\sum_{i=0}^{n} u_{e,i} \xi^i \right) x^e = u_0(x) + u_1(x)\xi + ... + u_n(x)\xi^n.$

- Las fórmulas para el caso entero permanecen válidas. Si $\phi_\xi : \mathbb{Z}[x] \longrightarrow \mathbb{Z}_\xi[x]$ es el homomorfismo que aplicado sobre $a(x) = \sum a_i x^i$ devuelve $\phi_\xi(a(x)) = \sum a_i (\text{mód}\, \xi)\, x^i,$ entonces

$$u_0(x) = \phi_\xi(u(x))$$

$$u_i(x) = \phi_\xi \left(\frac{u(x) \ - \ (u_0(x) + u_1(x)\xi + ... + u_{i-1}(x)\xi^{i-1})}{\xi^i} \right), \ i = 1, ..., n$$

Ejemplo.

1. Sea $a(x) = 4 + 7x - 9x^3$ y $\xi = 6.$
   - $a(x) = 3x^3 + x - 2 \ + \ (-2x^3 + x + 1) \cdot 6^1$
   - $u_0(x) = 3x^3 + x - 2$ y $u_1(x) = (-2x^3 + x + 1).$
2. Sea $a(x) = 4 + x$ y $\xi = 4.$
   - $a(x) = x \ + \ 1 \cdot 4^1$
   - $u_0(x) = x$ y $u_1(x) = 1.$

El algoritmo para calcular la representación $\xi-$ádica de $\gamma \in \mathbb{Z},$ sería (recuerde la definición de ϕ_ξ),

Algoritmo 1.5.2: Representación ξ-ádica de γ .

Entrada: $\gamma, \xi \in \mathbb{Z}$

Salida: $u_0, u_1, ..., u_d$ tal que $\gamma = u_0 + u_1\xi + ... + u_d\xi^d$ con $\xi^{d+1} > 2|\gamma|$ y $-\xi/2 < u_i \leq \xi/2$.

```
1 e = γ ;
2 i = 0 ;
3 while e ≠ 0 do
4 |   u_i = φ_ξ(e) ;
5 |   e = (e − u_i)/ξ ;
6 |   i = i + 1 ;
7 return u_0, u_1, ..., u_d ;
```

• Cuando necesitemos la reconstrucción de $\overline{g}(x),$ hacemos una pequeña modificación, agregamos $\overline{g} = 0$ y en el "while" actualizamos $\overline{g} = \overline{g} + u_i \cdot x^i$

• Es necesario implementar la versión polinomial también. En este caso lo que se reconstruye es un polinomio, pero en otra variable. Más adelante veremos esto.

1.5.2. Reconstrucción del Máximo Común Divisor

En general, el procedimiento es como sigue: El teorema principal establece una cota inferior para ξ. Tomamos un valor de ξ superior a esta cota y calculamos el polinomio G usando una representación ξ-ádica de $\gamma = \text{MCD}\,(A(\xi), B(\xi))$. Este polinomio es el $\text{MCD}\,(A, B)$ si pasa la prueba de divisibilidad. En otro caso, volvemos a tomar un nuevo valor de ξ, y así sucesivamente.

En el teorema se toma la parte primitiva de la reconstrucción G pues hay polinomios que al evaluarlos, no importa si se evalúan en números grandes, siempre tienen un factor común que es ajeno a la factorización y por tanto, sin remover el contenido, el criterio de divisibilidad por A y B fallaría siempre. Pero hay que garantizar que al remover el contenido de G, no estamos también quitando factores del verdadero $\text{MCD}\,(A, B)$. La escogencia de ξ en el teorema, garantiza esto último.

▷ Ejemplo 1 *Los polinomios* $A(x) = x^3 - 3x^2 + 2x$ *y* $B(x) = x^3 + 6x^2 + 11x + 6$ *son primos relativos, es decir* $MCD\,(A, B) = 1$. *Al evaluar* A *y* B, *siempre hay un factor común, múltiplo de* 6 *y este factor aparece en la reconstrucción* $\xi-$*ádica* G, *por eso hay que remover su contenido.*

El algoritmo se basa en el teorema que sigue,

Teorema 5 *Sean* $A(x), B(x) \in \mathbb{Z}[x]$ *ambos polinomios primitivos. Sea* $\xi \in \mathbb{Z}$ *tal que* $\xi \geq 2 \cdot Mín\{\|A||_\infty, ||B||_\infty\} + 2$ *donde* $||A||_\infty$ *denota el coeficiente numérico más grande de* A *(en valor absoluto). Si* $\overline{G}(x)$ *es el polinomio que se obtiene a partir de la representación* $\xi-$*ádica de* $\gamma = MCD\,(A(\xi), B(\xi))$ *y si* $pp(\overline{G}) \mid A$ *y* $pp(\overline{G}) \mid B$ *entonces* $pp(\overline{G}) = MCD\,(A, B)$.

Prueba. Ver [**?**].

Algoritmo

Hay que tener algunos cuidados en el algoritmo que vamos a presentar: Se supone que las pruebas de divisibilidad se hacen en $\mathbb{Q}[x]$, esto es equivalente a remover el contenido y entonces hacer división sobre los enteros. Pero entonces, debemos *remover el contenido* solamente del polinomio retornado.

En el algoritmo que sigue, usamos el homomorfismo de evaluación $\phi_{(x_1-\xi)} : \mathbb{Z}[x] \longrightarrow \mathbb{Z}$ definido por $\phi_{(x-\xi)}(A(x)) = A(\xi)$

Algoritmo 1.5.3: MCDHEU(A, B).

Entrada: $A, B \in \mathbb{Z}[x]$ ambos polinomios primitivos.
Salida: $\text{MCD}\,(A, B)$ si el resultado de la búsqueda heurística da resultado, sino retorna -1

```
1  if grado A = grado B = 0 then
2      return γ = MCD(A, B) ∈ ℤ
3  ξ = 2 · Mín{||A||∞, ||B||∞} + 2;
4  i = 0;
5  while i < 7 do
6      if length(ξ) · máx{grado A, grado B, } > 5000 then
7          return −1 //MCD ≥ 0, −1 se usa como indicador de fallo
8      γ = MCDHEU(φ(x−ξ)(A), φ(x−ξ)(B)) //llamada recursiva;
9      if γ ≠ −1 then
10         Generar G a partir de la expansión ξ−ádica de γ
11     //División en ℚ[x];
12     if G|A y G|B then
13         return pp(G)
14     Crear un nuevo punto de evaluación;
15     ξ = quo(ξ × 73794, 27011)
16 return −1;
```

- En al línea 6 se impone una restricción sobre la longitud de ξ (longitud en número de bits). Después de todo, el algoritmo es heurístico, así que se trata de asegurar que el cálculo no sea demasiado costoso.
- En la línea 11, la división se hace en $\mathbb{Q}$, es decir usando el método de división para polinomios con coeficientes en $\mathbb{Q}$. Esto tiene como efecto remover el contenido del divisor, resultando en un test de divisibilidad sobre los enteros. Solo hay que tener el cuidado de dividir por el contenido de G a la hora de retornar G (en caso de éxito).
- La línea 14 lo que procura es tener algún grado de "aleatoriedad" en la escogencia del siguiente punto de evaluación de tal manera que si hay un fallo en la primera escogencia, no haya una tendencia a que esto se repita ([**?**]).
- El algoritmo que se presenta aquí es la versión *optimizada* que aparece en [**?**]. Aunque manejamos una versión en $\mathbb{Z}[x_1, ..., x_k]$, en al implementación solo consideramos, en esta primera parte, el caso de polinomios en una indeterminada.

Ejemplo en $\mathbb{Z}[x, y]$. Sean $a(x, y) = (y^2 + 1)x^2 + (y + y^3)\,x$ y $b(x, y) = yx^2 + (y^2 + 1)x + y$. El algoritmo heurístico es recursivo. En este ejemplo pasaría lo siguiente:

Llamada 1. Entran $A(x,y) = (y^2+1)x^2 + (y+y^3)\,x$ y $B(x,y) = yx^2 + (y^2+1)x + y$.
- $vars = \{x, y\}$
- $\xi_1 = 2 \cdot 1 + 2 = 4$, pues $||A||_\infty = 1$ y $||B||_\infty = 1$.
- $var = x$
- $\cdots$

Llamada 2. $\gamma_2 = \text{MCDHEU}(A_1 = 16(y^2+1) + 4(y+y^3),\ B_1 = 16y + 4(y^2+1) + y)$
- $vars = \{y\}$
- $\xi_2 = 2 \cdot 17 + 2 = 36$, pues $||A_1||_\infty = 16$ y $||B_1||_\infty = 17$.
- $var = y$
- $\cdots$

Llamada 3.. $\gamma_3 = \text{MCDHEU}(A_1(36) = 207520, B_1(36) = 5800)$
- vars=$\{\}$
- Retorna $\gamma_3 = 40$.
- Nos devolvemos hacia **Llamada 2.**

Llamada 2. Entra a reconstrucción de G_2 con $\gamma_3 = 40$ y $\xi_2 = 36$
- $var = y$
- Representación ξ_2-ádica

$40 = 4 + 1 \cdot 36^1$

$G_2 = 4 + y$
- Test: $(\, G_2 \,|\, A_1(y)$ y $G_2 \,|\, B_1(y)\,) \longrightarrow$ **true**
- Retorna $G_2 = 4 + y$
- Nos devolvemos hacia **Llamada 1.**

Llamada 1. Entra a reconstrucción de G_1 con $\gamma_2 = 4 + y$ y $\xi_1 = 4$
- $var = x$
- Representación ξ_1-ádica

$4 + y = y \cdot 4^0 + 1 \cdot 4^1$

$G_1 = y + 1 \cdot x$
- Test: $(\, G_1 \,|\, A(x,y)$ y $G_1 \,|\, B(x,y)\,) \longrightarrow$ **true**
- Retorna $G = x + y$

$\therefore$ $\text{MCD}\,(a(x,y), b(x,y)) = x + y$

1.6. Algoritmo Extendido de Euclides.

El teorema de Bezout nos dice que si a y b son dos elementos (no ambos nulos), en un dominio Euclidiano D, existen $s, t \in D$ tal que $\text{MCD}\,(a,b) = sa + tb$.

En varios algoritmos que vamos a ver vamos a ver más adelante, vamos a usar extensamente este resultado.

Por ahora necesitamos concentrarnos en el cálculo de s y t. Esto se puede lograr directamente de la aplicación del algoritmo de Euclides.

Ejemplo.

- $\mathrm{mcd}(78, 32) = 2$. En efecto;

$$\begin{aligned} 78 &= 32 \cdot 2 + 14 \\ 32 &= 14 \cdot 2 + 4 \\ 14 &= 4 \cdot 3 + 2 \\ 4 &= 2 \cdot 2 + 0 \end{aligned}$$

- De acuerdo a la identidad de Bézout, existen $s, t \in \mathbb{Z}$ tal que $s \cdot 78 + t \cdot 32 = 2$. En este caso, una posibilidad es $7 \cdot 78 - 17 \cdot 32 = 2$, es decir $s = 7$ y $t = -17$.

  s y t se obtuvieron así: primero despejamos los residuos en el algoritmo de Euclides de abajo hacia arriba, iniciando con el máximo común divisor,

$$\begin{aligned} 78 &= 32 \cdot 2 + 14 \quad \longrightarrow \quad 14 = 78 - 32 \cdot 2 \\ 32 &= 14 \cdot 2 + 4 \quad \longrightarrow \quad 4 = 32 - 14 \cdot 2 \quad \uparrow \\ 14 &= 4 \cdot 3 + 2 \quad \longrightarrow \quad 2 = 14 - 4 \cdot 3 \quad \uparrow \\ 4 &= 2 \cdot 2 + 0 \end{aligned}$$

  Ahora hacemos sustitución hacia atrás, sustituyendo las expresiones de los residuos. En cada paso se ha subraya el residuo que se sustituye

$$\begin{aligned} 2 &= 14 - \underline{4} \cdot 3 \\ &= 14 - (32 - 14 \cdot 2)3 \\ &= \underline{14} \cdot 7 - 32 \cdot 3 \\ &= (78 - 32 \cdot 2)7 - 32 \cdot 3 \\ &= 7 \cdot 78 - 17 \cdot 32 \end{aligned}$$

El algoritmo extendido de Euclides es lo mismo que el algoritmo de Euclides, excepto que calcula una sucesión de residuos $r_i(x)$ junto con dos sucesiones $s_i(x)$ y $t_i(x)$ tales que

$$r_i(x) = a(x)s_i(x) + b(x)t_i(x).$$

Aquí

$$\begin{aligned} s_{i+1}(x) &= s_{i-1}(x) - s_i(x)q_i(x) \\ t_{i+1}(x) &= t_{i-1}(x) - t_i(x)q_i(x) \end{aligned}$$

El cociente $q_i(x)$ esta definido por la división $r_{i-1}(x) = r_i(x)q_i(x) + r_{i+1}(x)$.

Las condiciones iniciales para estas sucesiones son $s_0(x) = t_1(x) = 1$ y $s_1(x) = t_0(x) = 0$.

El algoritmo es el siguiente

Algoritmo 1.6.1: Algoritmo Extendido de Euclides.

Entrada: $a, b \in D$ dominio Euclidiano
Salida: MCD $(a, b) = g$ y $s, t \in D$ tal que $g = sa + tb$.

```
1  c = n(a), d = n(b);
2  c_1 = 1, d_1 = 0;
3  c_2 = 0, d_2 = 1;
4  while d! = 0 do
5  |   q = quo(c, d), r = c − qd;
6  |   r_1 = c_1 − qd_1, r_2 = c_2 − qd_2;
7  |   c = d, c_1 = d_1, c_2 = d_2;
8  |   d = r, d_1 = r_1, d_2 = r_2;
9  MCD = n(c);
10 s = c_1/(u(a) ∗ u(c)), t = c_2/(u(b) ∗ u(c));
11 return g, s, t;
```

- Recordemos que, por convenio, $u(0) = 1$.

- La correctitud del algoritmo se prueba en [**?**].

  - En el algoritmo anterior,

    - en el dominio Euclidiano $D = \mathbb{Z}$ entonces $n(a) = |a|$ y $u(a) = \text{sgn}(a)$ con $u(0) = 1$.

    - en el dominio Euclidiano $D = F[x]$ con F campo, entonces $n(a(x)) = a(x)/a_n$ y $u(a) = a_n$ donde a_n es el coeficiente principal de $a(x)$.

Ejemplo.

Sean $a(x) = 48x^3 - 84x^2 + 42x - 36$ y $b(x) = -4x^3 - 10x^2 + 44x - 30$. El algoritmo extendido de Euclides no se puede aplicar en $\mathbb{Z}[x]$ porque este dominio no es Euclidiano, pero si lo podemos aplicar en $\mathbb{Q}[x]$. En este caso

a) $g(x) = x - 3/2$

b) $s = \dfrac{17x}{6420} + \dfrac{3}{215}$

$c)\ \ t = \dfrac{17x}{535} + \dfrac{71}{2140}$

1.7. Implementaciones en Java

En esta sección vamos a ver las implementaciones de los algoritmos en Java. Recordemos que las clases y los métodos de esta sección no están optimizados, más bien la implementación sigue la lectura de los algoritmos, lo cual no significa que sean ineficientes.
Para mantener la claridad y la simplicidad, implementamos una clase para polinomios con coeficientes enteros y otra para polinomios con coeficientes racionales. Los algoritmos que se implementan como métodos de estas clases.

En lo que sigue, conviene tener a mano la API de Java, en particular conviene tener visibles los métodos de la clase BigInteger.

1.7.1. Una clase para polinomios

Lo primero que necesitamos es una clase `Bpolinomios` para polinomios con coeficientes enteros con los métodos necesarios para implementar los algoritmos. Usamos la clase BigInteger de Java.

La manera obvia de representar un polinomio $a(x) = \sum_{i=0}^{n} a_i x^i$ es con una arreglo de coeficientes $a = (a_0\ a_1\ ...\ a_n)$. A esta representación se le llama *representación densa.*

En muchas aplicaciones, los polinomios son en su mayoría, *ralos,* es decir con muchos coeficientes nulos: por ejemplo $a(x) = x^{1000} - 1$. Esto no parece bueno para la representación densa. Hay varias maneras de representar polinomios. Una manera requiere *listas.* Por ejemplo, el polinomio $a(x) = x^{1000} - 1$ se representa con la lista $(1\ 1000\ -1\ 0)$ y $a(x, y) = (2x^3 + 1)y^7 + (4x^5 - 5x^2 + 9)y^4 + 1$ se representa con la lista $((2\ 3\ 1\ 0)\ 7\ (4\ 5\ -5\ 2\ 9\ 0)\ 4\ (1\ 0)\ 0)$.

En esta primera parte, usamos la representación densa porque las operaciones con polinomios son fáciles de implementar y esto nos permite ver mejor los algoritmos. Más adelante tendremos que recurrir a alguna representación rala.

Nota: no todos los métodos están implementados, así que se requiere completar la clase. Los métodos que faltan no presentan ningún problema e implementarlos de seguro que ayudará a agregar diversión.

```
public class Bpolinomio
{
    public final static Bpolinomio ZERO = new Bpolinomio(BigInteger.ZERO, 0);
    public final static Bpolinomio ONE  = new Bpolinomio(BigInteger.ONE, 0);


    BigInteger[] coef;   // coeficientes
```

```
int deg;             // grado

// a * x^b
public Bpolinomio(BigInteger a, int b)
{
    coef = new BigInteger[b+1];// 0+a_1x+a_2x^2+...+a_nx^n
    for (int i = 0; i < b; i++)
    {
        coef[i] = BigInteger.ZERO;
    }
    coef[b] = a;
    deg = degree();
}//

public Bpolinomio(){//definir sin argumentos...}

// -this
public  Bpolinomio negate()
{
        Bpolinomio a = this;
        return  a.times(new Bpolinomio(new BigInteger(-1+""), 0));
}//

//Evaluar
public BigInteger evaluate(BigInteger xs)
{
    BigInteger brp = BigInteger.ZERO;
    for (int i = deg; i >= 0; i--)
            brp = coef[i].add(xs.multiply(brp));
    return brp;
}//

//comparación de polinomios
public int  compareTo(Bpolinomio b)
{
    int si =0;    //0 para "true"
    Bpolinomio    a = this;

      if(a.deg != b.deg)
      {
       si= 1;
    }else{
          for (int i = 0; i <= a.deg; i++)
          {
              if(a.coef[i].compareTo(b.coef[i])!=0 )
              { si = 1;
                break;
              }
```

```
            }
          }
       return si;
 }//

 // return c = a + b en Z.
public Bpolinomio plus(Bpolinomio b)
{
    Bpolinomio a = this;
                       //0 +...+ 0*x^max(a.deg.b.deg)
    Bpolinomio c = new Bpolinomio(BigInteger.ZERO, Math.max(a.deg, b.deg));
    for (int i = 0; i <= a.deg; i++) c.coef[i] = c.coef[i].add(a.coef[i]);
    for (int i = 0; i <= b.deg; i++) c.coef[i] = c.coef[i].add(b.coef[i]);
    c.deg = c.degree();
    return c;
 }//

 // return c = a - b
 public Bpolinomio minus(Bpolinomio b)
 {
         Bpolinomio a = this;
         Bpolinomio c = new Bpolinomio(BigInteger.ZERO, Math.max(a.deg, b.deg));
         for (int i = 0; i <= a.deg; i++) c.coef[i] = c.coef[i].add(a.coef[i]);
         for (int i = 0; i <= b.deg; i++) c.coef[i] = c.coef[i].add(b.coef[i].negate());
         c.deg = c.degree();
         return c;
 }//

 // return (a * b)
 public Bpolinomio times(Bpolinomio b)
 {
         Bpolinomio a = this;
         Bpolinomio c = new Bpolinomio(BigInteger.ZERO, a.deg + b.deg);
         for (int i = 0; i <= a.deg; i++)
         {
             for (int j = 0; j <= b.deg; j++)
                 c.coef[i+j] = c.coef[i+j].add(a.coef[i].multiply(b.coef[j]));
         }
         c.deg = c.degree();
         return c;
 }//
 //return k*a (k constante) en Z.
 public Bpolinomio times(BigInteger k)
 {
     Bpolinomio a = this;
     Bpolinomio c = new Bpolinomio(BigInteger.ZERO, a.deg);
     for (int i = 0; i <= a.deg; i++)
     {
```

```
                c.coef[i] = c.coef[i].add(a.coef[i].multiply(k));
        }
        c.deg = c.degree();
        return c;
    }//
    public Bpolinomio pow(int k){//...}

    //return quo(this,b).
    public Bpolinomio divides(Bpolinomio b){//...}

    //return quo(this,bi)
    public Bpolinomio divides(BigInteger bi){//...}


   // this to (mod p)
   public Bpolinomio toMod(BigInteger  p)
   {
      Bpolinomio a = this;
      Bpolinomio c = new Bpolinomio(BigInteger.ZERO, a.deg);
      for (int i = 0; i <= a.deg; i++) c.coef[i] = a.coef[i].mod(p);
      c.deg = c.degree(); //corriente
      return c;
  }

    // return quo(a,b) mod p primo
    public Bpolinomio divides(Bpolinomio pb, BigInteger p){//...}

    public String toString(){ //imprimir el polinomio...}

    public static Bpolinomio leer(String txt){ //leer el polinomio...}

    //Pruebas
   public static void main(String[] args)
   {
    Bpolinomio p = new Bpolinomio(new BigInteger("2"),2);//2x^2
    System.out.print(""+p);
   }
}//
```

• `toString` toma un polinomio $(a_0\ a_1\ ...\ a_n)$ y lo imprime como `a_0+a_1 x+...+a_nx^n`. No presenta dificultad. Una vez implementado, la instrucción `System.out.print(""+a)` imprime el polinomio `a`.

• Una manera sencilla para implementar `leer()` (en esta primera parte), se logra usando la clase `StringTokenizer`. En el apéndice aparece el código para este método.

1.7.2. Clase BigRational

Para el manejo de racionales grandes se crea una clase `BigRational`.

```
import java.math.BigInteger;
import java.util.Vector;

public class BigRational
{
    public final static BigRational ZERO  = new BigRational(0);
    public final static BigRational ONE   = new BigRational(1);

    private BigInteger num;
    private BigInteger den;

    public BigRational()
    {
     BigRational r = new BigRational(IZERO, IONE );
     num = r.num;
     den = r.den;
     return;
    }

    public BigRational(BigInteger numerador, BigInteger denominador)
    {
        if (denominador.equals(BigInteger.ZERO))
            throw new RuntimeException("Denominador es cero");

         //simplificar fracción. GCD está implementado en BigInteger
        BigInteger g = numerador.gcd(denominador);
        num = numerador.divide(g);
        den = denominador.divide(g);

        // Asegura invariante den >0
        if (den.compareTo(BigInteger.ZERO) < 0)
        {
            den = den.negate();
            num = num.negate();
        }
    }//
    public BigRational(BigInteger numerador){//...}

    public BigRational(int numerador, int denominador){//...}

    public BigRational(int numerador){//...}

    public BigRational(String s) throws NumberFormatException
```

```
    { //...}

    //return string
    public String toString()
    {
        if (den.equals(BigInteger.ONE)) return num + "";
        else                            return num + "/" + den;
    }

    // return { -1, 0, + 1 }
    public int compareTo(BigRational b)
    {
        BigRational a = this;
        return a.num.multiply(b.den).compareTo(a.den.multiply(b.num));
    }

    // return a * b
    public BigRational times(BigRational b)
    {
        BigRational a = this;
        return new BigRational(a.num.multiply(b.num), a.den.multiply(b.den));
    }

    // return a + b
    public BigRational plus(BigRational b){//...}

    // return -a
    public BigRational negate(){//...}

    // return a - b
    public BigRational minus(BigRational b){//...}

    // return 1 / a
    public BigRational reciprocal(){//...}

    // return a / b
    public BigRational divide(BigRational b){//...}

    //Pruebas
    public static void main(String[] args)
    {
       BigRational r = new BigRational(9,4);
       System.out.println(""+r);
    }
}
```

1.7.3. Clase Qpolinomio

Una vez implementada una clase `BigRational`, podemos implementar una clase más general, para polinomios con coeficientes racionales siguiendo el código de la clase `Bpolinomio`

```
import java.util.*;
import java.math.BigInteger;
import java.util.Vector;

public class Qpolinomio
{
    public final static Qpolinomio ZERO = new Qpolinomio(BigRational.ZERO, 0);
    public final static Qpolinomio ONE = new Qpolinomio(BigRational.ONE, 0);

    BigRational[] coef;   // coeficientes
    int deg;              // grado del polinomio (0 for the zero polynomial)

    // a * x^b
    public Qpolinomio(BigRational a, int b)
    {
        coef = new BigRational[b+1];// 0+a_1x+a_2x^2+...+a_nx^n
        for (int i = 0; i < b; i++)
        {
            coef[i] = BigRational.ZERO;
        }
        coef[b] = a;
        deg = degree();
    }

    //etc, etc, etc....

}//
```

1.7.4. Algoritmos

Contenido y parte primitiva

```
  //contenido en Z[x]
  public  BigInteger  cont()
  {
    Bpolinomio a   = this;
    BigInteger mcd = BigInteger.ZERO;
    int dega       = a.deg;
```

```
    if(dega==0)
    {mcd= a.coef[dega]; //0 si 0, k si kx^0.
     }else{
          mcd = a.coef[0].gcd(a.coef[1]);
          if(dega >1)
            for (int i = 2; i <= dega; i++)
                 mcd = mcd.gcd(a.coef[i]);
         }
    return mcd;
  }

   //parte primitiva en Z_p[x]
  public  Bpolinomio  toPP(BigInteger p)
  {
    Bpolinomio a   = this;
    if(a.compareTo(Bpolinomio.ZERO)==0) return Bpolinomio.ZERO;

     return a.divides(new Bpolinomio(a.coef[a.deg],0), p); //div mod p.
   }


  // parte primitiva en  Z[x]
  public  Bpolinomio  toPP()
  {
    Bpolinomio a   = this;
    int dega       = a.deg;
    Bpolinomio c   = new Bpolinomio(a.coef[dega], dega);
    BigInteger mcd = a.cont();
    BigInteger sgn = new BigInteger(""+(a.coef[dega]).signum()); //u(a(a))=a_m, con signo

    if(dega==0)
    {  if(a.coef[dega].compareTo(BigInteger.ZERO)==0) return Bpolinomio.ZERO;
       if(a.coef[dega].compareTo(BigInteger.ZERO)!=0) return Bpolinomio.ONE;
    }else{
           for (int i = 0; i <= dega; i++)
           {     c.coef[i] = a.coef[i].divide(mcd);
                 c.deg = c.degree();
           }
         }
    return c.times(sgn);
  }
```

MCD Primitivo

```
// a, b en Z[x] y devuelve MCD(a,b)
```

```
public  Bpolinomio  MCDprimitivo(Bpolinomio b)
{
        Bpolinomio a     = this;
        BigInteger lda   = (a.cont()).gcd(b.cont());//BigInteger
        Bpolinomio c     = a.toPP();
        Bpolinomio d     = b.toPP();
        Bpolinomio r,q;
        BigInteger cpd;
        int degc,degd;

     while(d.compareTo(Bpolinomio.ZERO)!=0)
     {   degc = c.deg;
         degd = d.deg;
         cpd  = new BigInteger(""+d.coef[degd]);
         c = c.times(cpd.pow(Math.abs(degc-degd)+1));
         q = c.divides(d); // no importa el de mayor grado!
         r = c.minus(q.times(d)); //c=dq+r -> r=c-dq
         c = d;
         d = r.toPP();
     }
    return c.times(lda);
}//
```

Ejercicio. Implementar MCD Primitivo para $\mathbb{Z}_p[x]$.

PRS Subresultante.

```
 // a, b en Z[x] y devuelve MCD(a,b)
public  Bpolinomio  PRS_SR(Bpolinomio b)
{
      //Agregar caso especial deg b > deg a.
       Bpolinomio a        = this;
       Bpolinomio ri       = b;
       Bpolinomio rim1     = a; //ri menos 1

       Bpolinomio c,q;
       BigInteger xii,xiim1;
       BigInteger bei,cri,crim1;
       BigInteger alfi;
       int di,dim1,d3,degr0, degri,degrim1;

       degrim1 = rim1.deg;
       degri   = ri.deg;
```

```
       //casos especiales aquí...

       cri     = new BigInteger(""+ri.coef[degri]);
       crim1   = new BigInteger(""+rim1.coef[degrim1]);
       di      = degrim1-degri;
       dim1    = di;
       alfi    = cri.pow(di+1); //alfa^{d2+1}
       bei     = ((BigInteger.ONE).negate()).pow(di+1); //bei=(-1)^{di+1}
       xii     = (BigInteger.ONE).negate();
       xiim1   = (BigInteger.ONE).negate();

       while(ri.compareTo(Bpolinomio.ZERO)!=0)
       { c    = rim1.times(alfi);
         q    = c.divides(ri);
         rim1 = ri;
         ri   = (c.minus(q.times(ri))).divides(bei); //r_i+1
         degrim1 = rim1.deg;
         degri   = ri.deg;
         cri     = new BigInteger(""+ri.coef[degri]);
         crim1   = new BigInteger(""+rim1.coef[degrim1]);
         dim1    = di;
         di      = degrim1-degri;
         alfi    = cri.pow( di+1); //alfa^{d2+1}
         xiim1   = xii;
         if(dim1 > 0)
         {     xii     = ((crim1.negate()).pow(dim1)).divide(xiim1.pow(dim1-1));
         }else xii     = ((crim1.negate()).pow(dim1)).multiply(xiim1);

         bei     = (crim1.negate()).multiply(xii.pow(di)); //bei=(-1)^{di+1}
       }
     //normalizar
     rim1=rim1.toPP();
     rim1=rim1.times((a.cont()).gcd( b.cont()));
     return rim1;
}//
```

Algoritmo Heurístico.

```
public  BigInteger NormaInfinito()
{
   Bpolinomio u = this;
   BigInteger maxabs=u.coef[0].abs();
   if(u.deg >0)
     for(int i=1; i<= u.deg; i++)
         maxabs=maxabs.max(u.coef[i].abs());
   return maxabs;
```

```
}//

//homorfismo psi(xi,u)= u mod(xi), en representación simétrica
public static  BigInteger psi(BigInteger xi, BigInteger gamma)
{
      BigInteger salida;
      BigInteger DOS  = new BigInteger("2");

      salida = gamma.mod(xi);
      //representación simétrica de Z_p =]-p/2,...-1,0,1,...,p/2], excluye -p/2
      if(salida.compareTo(xi.divide(DOS))==1)
          salida = salida.add(xi.negate());
      return salida;
}

//Para gamma = u0+...+ukxi^k, devuelve u0+u1x+...+ukx^k
public Bpolinomio Reconstruccion_xi_adica(BigInteger gamma, BigInteger xi)
{
      Bpolinomio g   = Bpolinomio.ZERO;
      BigInteger sumui, ui;

      ui    = g.psi(xi, gamma);
      g     = g.plus(new Bpolinomio(ui,0));
      sumui = ui;
      int i=1;
      while(gamma.add((sumui).negate()).compareTo(BigInteger.ZERO)!=0)
      {
          ui= g.psi(xi, (gamma.add((sumui).negate()).divide(xi.pow(i))));
          g = g.plus(new Bpolinomio(ui,i));
          sumui=sumui.add(ui.multiply(xi.pow(i))); //pow(i) pues es para paso i+1.
          i++;
      }
      return g;
}//

Como necesitamos dividir en \mt{\Q,} la clase {\tt QPolinomio}
debe de estar presente.\\

\begin{verbatim}
//homomorfismo psi(xi,u)= u mod(xi), en representaci\'on sim\'etrica
 public static  BigInteger psi(BigInteger xi, BigInteger gamma)
 {
      BigInteger salida;
      BigInteger DOS  = new BigInteger("2");

      salida = gamma.mod(xi);
      //representación simétrica de Z_p =]-p/2,...-1,0,1,...,p/2], excluye -p/2
      if(salida.compareTo(xi.divide(DOS))==1)
```

```
          salida = salida.add(xi.negate());
      return salida;
}

 public BPolinomio toMod_rs(BigInteger  p)
 {
 BPolinomio a = this;
 BPolinomio c = new BPolinomio(BigInteger.ZERO, a.deg);
 for (int i = 0; i <= a.deg; i++) c.coef[i] = psi(a.coef[i],p);
      c.deg = c.degree(); //corriente
 return c;
  }
 //devuelve u0,u1,...,un tal que u=u0+u1p+u2p^2+...+unp^n
  // u.p_adica(u,p)
  public  Vector p_adica(BigInteger u, BigInteger p)
  {
  Vector salida   = new Vector(1);
  BigInteger dosu = (u.multiply(new BigInteger("2"))).abs();
  BigInteger sumui, ui;
  int i =1;
  ui    = psi(u,p);
  salida.addElement(ui);
  sumui = ui;

  while(u.add((sumui).negate()).compareTo(BigInteger.ZERO)!=0)
  {
    ui= psi((u.add((sumui).negate()).divide(p.pow(i))),p);
    salida.addElement(ui);
    sumui=sumui.add(ui.multiply(p.pow(i))); //pow(i) pues es para paso i+1.
    i++;
   }
  return salida;
  }

 public  BigInteger NormaInfinito()
 {
   BPolinomio u = this;
   BigInteger maxabs=u.coef[0].abs();
   if(u.deg >0)
     for(int i=1; i<= u.deg; i++)
         maxabs=maxabs.max(u.coef[i].abs());
   return maxabs;
 }//


 // Como gamma = u0+...+ukxi^k, devuelve u0+u1x+...+ukx^k
 public BPolinomio Reconstruccion_xi_adica(BigInteger gamma, BigInteger xi)
 {
```

```
      BPolinomio g   = BPolinomio.ZERO;
      BigInteger sumui, ui;

      ui    = g.psi(xi, gamma);
      g     = g.plus(new BPolinomio(ui,0));
      sumui = ui;
       int i=1;
      while(gamma.add((sumui).negate()).compareTo(BigInteger.ZERO)!=0)
      {
          ui= g.psi(xi, (gamma.add((sumui).negate()).divide(xi.pow(i))));
          g = g.plus(new BPolinomio(ui,i));
          sumui=sumui.add(ui.multiply(xi.pow(i))); //pow(i) pues es para paso i+1.
          i++;
      }
      return g;
}//

//MCD Eur\'istico
public BPolinomio MCDHeuBPolinomio(BPolinomio B) {
 //GRADO
    BPPolinomio A  = this;
    BigInteger lda = (A.cont()).gcd(B.cont());//BigInteger
    BPPolinomio c  = A.toPP();
    BPPolinomio d  = B.toPP();
    if(d.deg > c.deg ){c=d; d = A.toPP();}

 //vars={} pues aplicamos sobre A,B primitivos en Z[x],
 //llamamos MCDHeu(phi_{x-xhi}(A),phi_{x-xhi}(B))
 BPolinomio G       = BPolinomio.ZERO;
 BPolinomio gamma   = new BPolinomio();
 BigInteger BI2     = new BigInteger("2");
 QPolinomio QG,QA,QB,r1,r2;



 QA = QpolObj.leer(A.toString()); //lo lee como QPolinomio
 QB = QpolObj.leer(B.toString());

 if(A.deg==0 && B.deg ==0)
   return new BPolinomio(A.coef[0].gcd(B.coef[0]),0); //MCD(A,B) en Z[x]

 BigInteger xi        = (BI2.multiply(A.NormaInfinito().min(B.NormaInfinito()))).add(BI2);
 BPolinomio failflag  = (BPolinomio.ONE).negate(); //-1,  MCD debe ser normal
                        //number of bits in the ordinary binary representation si A>0
 int lengthxi         = xi.bitLength();

 for(int i= 1; i<=6; i++)
 {
```

```
  if(lengthxi*Math.max(A.deg, B.deg)>5000)
    return failflag; //sale

  gamma=(new BPolinomio(A.evaluate(xi),0)).MCDHeuBPolinomio(new BPolinomio(B.evaluate(xi),0));

  if(gamma.compareTo(failflag)!=0)
  { //si gamma es una constante
   if(gamma.deg==0)
     G = G.Reconstruccion_xi_adica(gamma.coef[0], xi);
  }

   //Viene divisi\'on en Q[x]
    QG = QpolObj.leer(G.toString()); //lo lee como QPolinomio
    //Test de divisibilidad
    r1 = QA.minus((QA.QuoQPolinomio(QG)).times(QG));
    r2 = QB.minus((QB.QuoQPolinomio(QG)).times(QG));

    if(r1.compareTo(QPolinomio.ZERO)==0 && r2.compareTo(QPolinomio.ZERO)==0)
    { if(G.deg==0)
      {
            return new BPolinomio(lda,0);
      }else return  (G.toPP()).times(lda);
    }
    //sino sali\'o, crear un  nuevo punto de evaluaci\'on
    xi = (xi.multiply(new BigInteger("73794"))).divide(new BigInteger("27011"));
 }
 return failflag; //-1 si no encuentra algo.
}
```

Esta implementación contempla el caso en el que A y B no son primitivos. Para hacer una corrida de prueba, en el método `main` de la clase `BPolinomio` escribimos

```
    //No primitivos
    A = B.leer("48x^3 - 84x^2 + 42x - 36");
    B = B.leer("-4 x^3 - 10x^2 + 44x - 30");
    System.out.print(" 1.)  "+A.MCDHeuBPPolinomio(B)+"\n\n");

    //Primitivos
    A = B.leer("  8x^3 - 14x^2 + 7x - 6");
    B = B.leer(" -2x^3 - 5x^2 + 22x - 15");
    System.out.print(" 2.)  "+A.MCDHeuBPPolinomio(B)+"\n\n");
```

La salida es

```
1.)  4x^1-6

2.)  2x^1-3
```

Algoritmo Extendido de Euclides (método en `Qpolinomio`).

```
//retorna g = MCD(a(x),b(x)) y t(x) , s(x)
public static Qpolinomio[]  MCD_ext(Qpolinomio a, Qpolinomio b)
{
      Qpolinomio[] salida1 = new Qpolinomio[3];
      Qpolinomio   an      = new Qpolinomio(a.coef[a.deg],0);
      Qpolinomio   bn      = new Qpolinomio(b.coef[b.deg],0);
      Qpolinomio   cn;
      Qpolinomio   c,d,c1,d1,c2,d2,q,r,r1,r2,s,t;

         c  = a.divides(an);
         d  = b.divides(bn);
         c1 = Qpolinomio.ONE;
         d1 = Qpolinomio.ZERO;
         c2 = Qpolinomio.ZERO;
         d2 = Qpolinomio.ONE;

         int j=1;
         while(d.compareTo(Qpolinomio.ZERO)!=0)
         {
           q = c.divides(d);
           r = c.plus(q.times(d).negate());
           r1= c1.plus(q.times(d1).negate());
           r2= c2.plus(q.times(d2).negate());
           c = d;
           c1= d1;
           c2= d2;
           d = r;
           d1= r1;
           d2= r2;
          j++;
         }
         cn = new Qpolinomio(c.coef[c.deg],0);
          c = c.divides(cn);
         salida1[0]=c;
         salida1[1]=c1.divides(an.times(cn));
         salida1[2]=c2.divides(bn.times(cn));
         return salida1;
}//
```

Ejercicio. Implemente al algoritmo extendido de Euclides en $\mathbb{Z}_p[x]$ (en la clase `Bpolinomio`)

Apéndice A

Métodos Adicionales

Método para leer polinomios en la clase `Bpolinomio`.

```
public static Bpolinomio leer(String txt)
{
Bpolinomio salida = new Bpolinomio();
String  polyStr;

polyStr= normalizar(txt);

StringTokenizer termStrings = new StringTokenizer(polyStr, "+-", true);
boolean nextTermIsNegative = false;

while (termStrings.hasMoreTokens())
{
   String termToken = termStrings.nextToken();

   if (termToken.equals("-"))
   {
    nextTermIsNegative = true;
   }else if (termToken.equals("+"))
   {
    nextTermIsNegative = false;
   }else{
       StringTokenizer numberStrings = new StringTokenizer(termToken, "*^", false);
       BigInteger  coeff = new BigInteger(""+1);
       int expt;
       String c1 = numberStrings.nextToken();
       if (c1.equals("x"))
       {
        // "x" or "x^n"
        if (numberStrings.hasMoreTokens())
       {
        //  "x^n"
```

```
        String n = numberStrings.nextToken();
        expt = Integer.parseInt(n);
      } else {
              // "x"
              expt = 1;
              }
              } else {
                        // "a_i" or "a_i*x" or "a_i*x^n"
                        String ai=c1;
                        coeff = new BigInteger(ai);

                        if (numberStrings.hasMoreTokens())
                        {
                        //  "a_i*x" or "a_i*x^n"
                        String x = numberStrings.nextToken();

                        if (numberStrings.hasMoreTokens())
                        {
                        //  "a_i*x^n"
                        String n = numberStrings.nextToken();
                        expt = Integer.parseInt(n);
                        } else {
                                // "a_1*x"
                                 expt = 1;
                                 }
                                 } else {
                                            // "a_0"
                                            expt = 0;
                                           }
                                  }
                       // Si coeff es precedido por '-'
                       if (nextTermIsNegative)
                       {
                           coeff = coeff.negate();
                       }
                       //Acumular términos en "salida"
                       int cmp = coeff.compareTo(BigInteger.ZERO);
                       if( cmp !=0)
                       {
                         salida = salida.plus(new Bpolinomio(coeff,expt));
                       }
                  }
              }
              return salida;
}//
```

Bibliografía

[1] Lindsay N. Childs. *A Concrete Introduction to Higher Algebra.* Springer-Verlag New York, 1995.

[2] K.O. Geddes, S.R. Czapor y G. Labahn. *Algorithms for Computer Algebra.* Kluwer Academic Publishers. 1992.

[3] Raymond Séroul, *Programming for Mathematicians.* Springer, 2000.

[4] Joachim von zur Gathen, Jürgen Gerhard. "*Modern Computer Algebra*". Cambridge University Press, 2003.

[5] Maurice Mignotte. "*Mathematics for Computer Algebra*". Springer,1992.

[6] Niels Lauritzen. "*Concrete Abstract Algebra*". Cambridge University Press, 2005.

[7] John B. Fraleigh. "*A First Course in Abstract Algebra*". Addison Wesley; 2nd edition, 1968.

[8] Gautschi, W. *Numerical Analysis. An Introduction.* Birkhäuser, 1997.

[9] Lipson, J. *Elements of Algebra and Algebraic Computing.* Adison Wesley Co., 1981.

[10] F. Winkler *Polynomial Algoritms for Computer Algebra.* Springer Verlag/Wien., 1996.

[11] R. McEliece *Finite Fields for Computer Scientists and Engineers.* Kluwer Academic Publishers, 1987.

[12] D. Knuth *The Art of Computer Programming. Semi-numerical Algoritms.* Addison-Wesley, 1998.

[13] H.-C. P. Liao y R. J. Fateman. "Evaluation of the heuristic polynomial GCD". ISSAC, pp 240-247, 1995.

[14] J. von zur Gathen y T. Lücking. "Subresultants Revisited." Theoretical Computer Science, 297(1-3):199-239,2003.

[15] P. Bernard "A correct proof of the heuristic GCD algorithm."
En `http://www-fourier.ujf-grenoble.fr/~parisse/publi/gcdheu.pdf` (consultada Julio, 2007).

[16] E. Kaltofen, M. Monagan y A. Wittkopf. "On the Modular Polynomial GCD Algorithm over Integer, Finite Fields an Number Field".
En `http://www.cecm.sfu.ca/CAG/products2001.shtml` (consultada Julio, 2007).

[17] R. Sedgewick, K. Wayne. *Introduction to Programming in Java. An Interdisciplinary Approach.* Addison-Wiley. 2008.