Instituto Tecnológico de Costa Rica

Informe final del proyecto:

Estudio e Implementación de Algoritmos de Matemática Simbólica y Gráficos 2D en JAVA para el Desarrollo de Software Educativo

MSc. Alexander Borbón Alpízar
aborbon@itcr.ac.cr

MSc. Walter Mora Flores
wmora2@yahoo.com.mx

MSc. Cristhian Páez Páez
cpaez@itcr.ac.cr

Marzo, 2009

Resumen

Se realizó una investigación donde se estudiaron e implementaron algoritmos de matemática simbólica y gráficos 2D en JAVA; dichos algoritmos fueron utilizados para el desarrollo de un *software* educativo no comercial (de libre acceso a toda persona).

En este informe se muestran los principales logros obtenidos en la investigación y se indican los medios de divulgación de resultados que fueron utilizados durante el desarrollo de la misma.

Abstract

A research was carried where were studied and implemented algorithms of symbolic mathematic, Java 2D graphics, such algorithms were used to develop a non-commercial educational software (freely accessible to anyone).

This report lists the main achievements in the research and the means of dissemination that were used to show the results that were obtained during the development of it.

Palabras clave

Matemática Simbólica, Graficación 2D, Graficación 3D, Software educativo, Algoritmos

Tabla de contenidos

1. Introducción

En los últimos años se ha detectado que la educación matemática costarricense está sufriendo un deterioro alarmante; este se ve reflejado en los resultados de las pruebas nacionales en la educación formal (64% de aprobación en el año 2003, según el INFORME NACIONAL), y en los altos niveles de deserción que muestran los índices publicados por la División de Control de Calidad del Ministerio de Educación Pública. Este problema se agrava por varios factores: el bajo dominio que tienen los educadores de los conceptos básicos (Nación, 30 de setiembre del 2003), los niveles de insatisfacción de los costarricenses con la educación (Ibíd.) y la marcada diferencia de calidad entre la Educación Pública y la Educación Privada a nivel de secundaria.

En este sentido, dentro de los desafíos nacionales planteados en los últimos informes del Estado de la Nación para la Educación, sobresalen “ampliar la cobertura, mejorar la calidad y diversidad de la Educación Secundaria” y “universalizar el acceso efectivo a la Secundaria completa, como mecanismo para dar pasos firmes hacia la reducción de la pobreza y la ampliación de las oportunidades para las futuras generaciones” (Echeverría y otros, 2003).

Los esfuerzos para lograr estos desafíos incluyen el uso de *software* con fines educativos; por ejemplo, *The Geometer’s Sketchpad*, *Mathematica*, *Derive* y otros. Hoy en día, los mayores esfuerzos se centran en la creación de herramientas, de carácter modular, que puedan ajustarse a las necesidades propias de diversos entornos de la enseñanza y el aprendizaje de la matemática.

Para nuestro país, en particular, es necesario el desarrollo de opciones novedosas y oportunas que permitan abordar problemas identificados en los procesos de enseñanza y aprendizaje de las matemáticas, ya que en general, para la mayoría de los profesores de matemática en Latinoamérica, los recursos computacionales disponibles resultan poco prácticos; una posible razón está relacionada con la complejidad y el costo, lo que dificulta el proceso de apropiación de nuevas tecnologías de información y comunicación.

En este sentido, Internet permite la puesta en práctica de dichos entornos, con el aliciente de que permite acceder a la información a cualquier hora y desde “cualquier” lugar, independientemente de la plataforma. El desarrollo de nuevas herramientas para Internet en el área de la matemática, hace posible que el docente pueda auxiliar a los estudiantes, formales o informales, vía Internet o Intranet, en la tarea de participar en la construcción de su propio conocimiento. También, el desarrollo de este tipo de herramientas y entornos virtuales, puede contribuir en el reforzamiento de la formación matemática, tanto de los profesores universitarios como de los profesionales de la enseñanza en primaria, en secundaria y en el nivel técnico.

Estos entornos modulares necesitan herramientas matemáticas que le agreguen interacción. JAVA permite la creación de programas para Internet ejecutables en cualquier plataforma; de esta manera, JAVA se presenta como el programa idóneo para el diseño de dichas herramientas. Sin embargo, para la creación de las herramientas es necesario contar con algoritmos matemáticos eficientes, los cuales utilizan procesos matemáticos complejos.

El fin del proyecto “Centros de Recursos Virtuales en Matemática (CR-USA)” era la creación de un Centro de Recursos Didácticos, apoyados en las tecnologías de información y comunicación, para lo cual se desarrollaron módulos didácticos sobre temas de Educación Media y Diversificada en Matemáticas.

Como ya se ha mencionado, los algoritmos matemáticos que necesitaron las herramientas para estos módulos didácticos, están basados en matemáticas muy avanzadas; probablemente, este sea el principal motivo por el cual no se puedan conseguir, en forma gratuita, dichos algoritmos. Si bien es cierto existe el desarrollo de muchos algoritmos, estos son realizados por empresas con fines comerciales; como por ejemplo, *Mathematica*, *Maple*, *MatLab* y otros, o bien, son accesibles, únicamente, los algoritmos básicos (lectura de expresiones, operaciones elementales y demás).

La investigación buscó generar algoritmos para ayudar a solventar estas dificultades; asimismo, se realiza la difusión gratuita de dichos algoritmos en revistas, Internet, congresos, simposios y artículos, entre otros.

El proyecto tenía como objetivo principal, estudiar e implementar algoritmos matemáticos para temas como: Álgebra, Ecuaciones, Desigualdades, Funciones y Trigonometría. Sus objetivos específicos se enuncian a continuación:

1) Estudiar e implementar algoritmos de matemática simbólica, análisis numérico y gráficos por computadora.

2) Editar y publicar artículos relacionados con los algoritmos implementados, para ser presentados en algunas revistas y congresos.

3) Establecer nexos con docentes de matemática que permitan realimentar el trabajo de la investigación a través del proyecto con CR-USA.

4) Proyectar el trabajo que se viene realizando en la Escuela de Matemática del ITCR en el entorno educativo nacional.

5) Colaborar para solventar las necesidades de herramientas que, eventualmente, se necesiten en los proyectos “Centro de Recursos Virtuales en Matemáticas” y “Recursos Virtuales para el Mejoramiento de la Enseñaza y el Aprendizaje de la Geometría”.

2. Metodología

Para lograr los objetivos propuestos para la investigación se definieron una serie de etapas por seguir:

- I Etapa: Identificación de temas específicos.

  En esta primera etapa se definieron los temas en los que se iba a trabajar, estos temas tenían relación directa con los temas estudiados en secundaria: Álgebra, Ecuaciones, Desigualdades, Funciones y Trigonometría. Para la elección de los temas se tomó en cuenta la importancia de su diseño para el fortalecimiento de la enseñanza y el aprendizaje en secundaria.

  La idea fundamental de esta primera etapa era orientar el estudio y desarrollo de algoritmos y herramientas.

- II Etapa: Estudio de los algoritmos.

  En esta segunda etapa se inició con el estudio propiamente de los algoritmos que se decidieron implementar en la etapa anterior, para agilizar este proceso se repartieron los distintos algoritmos entre los investigadores y se definieron una serie de reuniones periódicas donde cada integrante del proyecto exponía ante sus compañeros el avance de su investigación.

- III Etapa: Recolectar y estudiar la información relacionada con algoritmos de matemática para su implementación en JAVA.

  Una vez que se tienen definidos los algoritmos que se van a implementar se debe realizar una investigación de cómo se puede implementar en un lenguaje de programación como JAVA.

- IV Etapa: Programación de los algoritmos.

  En esta etapa, ahora que se tienen claros los algoritmos a implementar, se llevará a cabo la programación de ellos en JAVA, para esto se crearán las clases y aplicaciones, siempre con base en la información recolectada, haciendo uso de algunos algoritmos matemáticos investigados.

Algunas de estas etapas se entrelazaban y no se iban realizando de forma lineal, es decir, conforme se programaba alguno de los algoritmos se iba estudiando otros para ser implementados después.

3. Revisión de literatura

Como se dijo anteriormente, en la primera etapa de la investigación se definieron los temas específicos que se iban a estudiar.

En álgebra computacional hay dos problemas por resolver; por una parte, está el problema relacionado con algoritmos y teoría subyacente; por otra parte, la programación de estructuras complejas.

En relación con los algoritmos, es importante mencionar que, aunque se requiere una base matemática que se cubre en programas de licenciatura de matemática pura de nuestro país, cada algoritmo como tal requiere de matemáticas muy especializadas; por ejemplo, la factorización de polinomios en un dominio de factorización única $D[x_1,x_2,\dots,x_n]$ que es una de las operaciones más comunes en muchos algoritmos, requiere, además de los temas básicos de inversión de homomorfismos y teorema chino del resto, tópicos más avanzados como iteración lineal p-ádica de Newton y la generalización multivariada del lema de estiramiento (*lifting*) de Hensel que se desarrollan en cuatro o cinco capítulos de algunos libros especializados (por ejemplo, Geddes, K. *et al* “Algorithms for Computer Algebra”).

Además, se utilizaron revistas especializadas (Mathematics of Computation, Theoretical Computer Science) para conocer la versión actualizada de los algoritmos y, de esta manera, evaluar y discriminar cuál variante es más conveniente para nuestros propósitos, y cuáles son las heurísticas que se deben agregar en la implementación; es decir, detrás de cada algoritmo existe teoría matemática muy avanzada.

El segundo problema a resolver es el relacionado con la programación de dichos algoritmos. En este sentido, se necesitan estructuras complejas y un conocimiento avanzado en lenguajes de programación. Este segundo problema, además de complicado, demanda mucho tiempo en términos de programación y pruebas de correctitud.

Se estudiaron algoritmos de álgebra computacional para temas relacionados con polinomios; específicamente, algoritmos para temas como: aritmética en dominios básicos, aproximación p-ádica para ciertos cálculos, máximo común divisor para polinomios, factorización de polinomios, descomposición de polinomios, resolución de sistemas de ecuaciones lineales. El libro *Polynomial Algorithms in Computer Algebra* de la editorial *Springer* fue de gran ayuda para estos temas.

Para la teoría matemática que se requiere en el estudio de los algoritmos, como teoría de grupos, subgrupos, anillos, campos, dominios de factorización única, raíces primitivas y relaciones, entre otros temas, se utilizó el libro *Concrete Abstract Algebra* de la editorial *Cambridge*.

Estos fueron sólo dos de los libros esenciales en el desarrollo del proyecto. La siguiente lista completa el número de obras adquiridas y que permitieron el estudio y la implementación de algoritmos matemáticos.

Computacional Conmutative Algebra 2, editorial Springer.

Some Tapas of Computer Algebra, editorial Springer.

Modern Computer Algebra, editorial Cambridge.

Computer Algebra and Symbolic Computation: Elementary Algorithms, editorial A K Peters.

Computer Algebra and Symbolic Computatios: Mathematical Methods, editorial A K Peters.

Algorithms for Computer Algebra, editorial K A Publishers.

An Introduction to Gröbner Bases, editorial AMS.

4. Resultados obtenidos

Con el desarrollo del proyecto se realizó el diseño y la implementación de un *software* en el que se ejecutaron los módulos implementados (la interfaz); asimismo, se estudió la manera de representar en el computador una expresión matemática. Se trabajó en la adecuación del *software* y del módulo relacionado con la representación en el computador de expresiones matemáticas; este trabajo fue complejo, ya que se necesitaron varias representaciones de las expresiones que se presentan.

El primero de los objetivos específicos se cumplió desde la llegada del material bibliográfico, y es claro que este no tiene una fecha límite para ser alcanzado, ya que el estudio de algoritmos es continuo.

Además, como resultado del primer seminario de estudio sobre teoría de números y álgebra computacional, se publicó un folleto de sus memorias (el cual se puede ver completo en los anexos) y, también, se presentó un taller en el Congreso de Enseñanza de la Matemática Asistida por Computadora. Actualmente, se lleva a cado el segundo seminario de estudio (aun con la finalización del proyecto se sigue con el seminario de estudio).

A lo largo de la ejecución del proyecto, se apoyaron los proyectos “Centro de Recursos Virtuales en Matemáticas” y “Recursos Virtuales para el Mejoramiento de la Enseñaza y el Aprendizaje de la Geometría” con la programación de algoritmos específicos. En general, se colaboró con clases (programas Java) y módulos (programas con interfaz) para dicho proyecto, creando nexos con estos.

El material se ha divulgado en la Revista Virtual Matemática, Educación e Internet, en el I Congreso Internacional de Computación y Matemática, y en el V Congreso Internacional de Enseñanza de la Matemática Asistida por Computadora. Las publicaciones realizadas a partir del proyecto se detallan a continuación y se pueden encontrar en texto completo en los anexos:

Mora, W. (2007). *Cálculo del máximo común divisor de dos polinomios*. Revista Virtual Matemática, Educación e Internet, volumen 8, número 1.

http://www.cidse.itcr.ac.cr/revistamate/HERRAmInternet/MDC_I_Parte/index.html

Borbón, A. (2006) *¿Cómo evaluar expresiones algebraicas en el computador?* Revista Virtual Matemática, Educación e Internet, volumen 7, número 2.

http://www.cidse.itcr.ac.cr/revistamate/ContribucionesV7_n2_2006/Parseador/index.html

Mora, W. (2007). *Probabilidad, Números Primos y Factorización de Enteros. Implementaciones en Java y VBA para Excel.* Revista Virtual Matemática, Educación e Internet, volumen 8, número 2.

http://www.cidse.itcr.ac.cr/revistamate/HERRAmInternet/v8n2-DIC-007/Probabilidad_Primos_Factorizacion.html

Mora, W. (2006). *Criba de Eratóstenes: Cómo colar números primos. Implementación en Java y VBA para Excel.* Revista Virtual Matemática, Educación e Internet, volumen 7, número 2.

http://www.cidse.itcr.ac.cr/revistamate/HERRAmInternet/Criba/Criba.pdf

Mora, W. (2006). *Implementación de un graficador de funciones con interfaz gráfica Swing y Java2D*. Revista Virtual Matemática, Educación e Internet, volumen 7, número 2.

http://www.cidse.itcr.ac.cr/revistamate/HERRAmInternet/Graficador-Swing-java2D/index.html

Borbón, A. (2007). *Programación de Applets en JAVA para Principiantes*. V Congreso Internacional sobre Enseñanza de la Matemática Asistida por Computadora.

Borbón, A. y Páez, C. (2008). *Taller: Evaluación de Software Educativo*. I Congreso Internacional de Computación y Matemática.

A lo largo del proyecto se estudiaron e implementaron, entre otros, los algoritmos siguientes (la descripción de dichos algoritmos se detalla en el primer anexo):

Algoritmo de la división
Máximo común divisor
Algoritmo Extendido de Euclides
Inverso Multiplicativo mod m
Raíz cuadrada de un BigInteger
Criba de Eratóstenes
Colado de primos entre m y n
Factorización por Ensayo y Error
Método rho de Pollard (variante de R. Brent)
Miller-Rabin
Problema Chino del Resto en Z*:* Algoritmo de Garner
Aproximación Racional con Fracciones Continuas
Evaluación de Polinomios. Método de Horner

División de Polinomios en $K[x]; K$ campo

Algoritmo de Euclides

Algoritmo Extendido de Euclides

Algoritmo Primitivo de Euclides

Algoritmo PRS Subresultante

Imagen módulo p de un número, en la representación simétrica de Zp

Representaci´on x-´adica de g

MCDHEU($A;B$)

Factorización Libre de Cuadrados (Algoritmo de Yun)

SquareFreeCF: Factorización Libre de Cuadrados en un Campo Finito

Diferencias Divididas de Newton

Método de Kronecker

Forma de Lagrange del Polinomio Interpolante

Interpolación Cuadrática Inversa

Pesos Baricéntricos

Iteración de Punto fijo

Algoritmo de Bisección

Método de Newton

Método de falsa posición

Método de la secante

Híbrido secante-bisección

Híbrido Newton-bisección

Estiramiento de Hensel univariado

Factorización en un campo finito, Berkelmap

Asimismo, se programaron en JAVA módulos, funciones, rutinas, constantes y otros que se detallan a continuación:

Lectura de expresiones algebraicas

Representación gráfica de expresiones matemáticas

Derivación de expresiones

Graficación de funciones en 2D

Graficación de funciones en 3D

Simplificación básica de expresiones matemáticas

Definición de constantes e y π

Algoritmos numéricos para el cálculo, con número grande de dígitos, de las funciones siguientes:

- Lógicas:
  - Es entero?
  - Es monomio?
  - Es número?
  - Es polinomio?
  - Es primo?
  - Si
- Matrices
  - Transpuesta
  - Reducción de filas de la matriz
  - Determinante
  - Intercambio de filas
  - A la fila i sumarle k veces la fila j
  - Inversa de la matriz
  - A la fila i multiplicarla por k
- Trigonométricas
  - Arcocoseno
  - Arcocotangente
  - Ángulo en posición estándar
  - Arcocosecante
  - Arcoseno
  - Arcotangente
  - Coseno
  - Coseno hiperbólico
  - Cotangente
  - Cotangente hiperbólica
  - Cosecante
  - Cosecante hiperbólica
  - Secante
  - Secante hiperbólica
  - Seno
  - Seno hiperbólico

        - Tangente
        - Tangente hiperbólica
    - Otras
        - Valor absoluto
        - Teorema del resto chino
        - Coeficientes del polinomio
        - Derivada
        - Factorizas un número entero
        - Intersección de conjuntos
        - Logaritmo natural
        - Logaritmo
        - Módulo
        - La parte entera
        - Función pi
        - Primo número *k*
        - Primos de *n* a *m*
        - Generar un número a random
        - Redondear
        - Signo
        - Simplificar
        - Simplificar con números
        - Raíz cuadrada
        - Sustituir
        - Truncar
        - Unión de conjuntos

Los operadores implementados son:

- Y lógico
- Distinto
- División
- Factorial
- Igual
- Mayor
- Menor

Mayor o igual

Menor o igual

Multiplicación

Negación

O lógico

Porcentaje

Potencia

Resta

Suma

Xor

5. Conclusiones y recomendaciones

La realización de este proyecto vino a fortalecer y ayudar otros proyectos como el "Centro de Recursos Virtuales en matemática" y "Recursos Virtuales para el Mejoramiento de la Enseñaza y el Aprendizaje de la Geometría", esto ya que el proyecto brindó algoritmos implementados para ser utilizados por ellos, creemos que esta ayuda entre proyectos debe ser una actividad común en el ambiente académico.

El estudio e implementación de algoritmos de matemática es un estudio contínuo e inacabable, conforme se vayan implementando algunos algoritmos siempre aparecerán otros nuevos más sofisticados, con respecto a esto recomendamos que se pueda continuar con proyectos como este para poder contar con la mayor cantidad de algoritmos ya implementados y listos para ser utilizados.

Se observó que la realización de módulos separados para cada uno de los algoritmos no es viable ya que muchos de los algoritmos avanzados utilizan los algoritmos iniciales haciendo programas muy complejos, con respecto a este proyecto se decidió mejor realizar un sólo programa que reuniera todos los algoritmos juntos.

6. Bibliografía

Ammeraal, L (1998). *Computer Graphics for Java Programmers*. Estados Unidos: John Wiley and Sons.

Arvo, J. (1991). *Graphics Gems II*. Estados Unidos: Morgan Kaufmann.

Brent, R. y **Pollard**, M. (1981). *Factorization of the Eighth Fermat Number*. Mathematics of Computation. Vol. 36 (154), pp. 627-630.

Cohen, J. (2002). *Computer Algebra and Symbolic Computation*. Canadá: A K Peters.

Deitel, H. y **Deitel**, P. (1998). *Cómo Programar en Java*. México: Prentice Hall Hispanoamericana.

Echeverría, C; **Gutiérrez**, M; **Robles**, A; **Román**, I; **Román**, M. y **Vargas**, J. (2003). *Noveno Informe de la Nación*. Capítulo 1. Costa Rica: Estado de la Nación.

Goodman, D. (1998). *JavaScript Bible*. Tercera edición. Estados Unidos: IDG Books.

Glassner, A. (1990). *Graphics Gems*. Estados Unidos: Morgan Kaufmann.

Kiat Shi, T; **Steeb**, W. y **Hardy**, Y. (2000). *SymbolicC++: An Introduction to Computer Algebra*

Using Object-Oriented Programming. Segunda Edición. Gran Bretaña: Athenæum Press.

Lengyel, E. (2002). *Mathematics for 3D Game Programming & Computer Graphics*. Estados Unidos: Charles River Media.

Mignotte, M. (1992). *Mathematics for Computer Algebra*. Estados Unidos: Springer-Verlag.

Montgomery, P. (1987). *Speeding the Pollard and Elliptic Curve Method*. Mathematics of Computation. Vol. 48 (177), pp. 243-264.

Press, W; **Teukolsky**, S; **Vetterling**, W. y **Flannery**, B. (1992). *Numerical Recipes in C*. Segunda Edición. Estados Unidos: Cambridge University Press.

Schildt, H. y **Holmes**, J. (2004). *El Arte de Programar en Java*. México: McGraw-Hill Interamericana.

Schneider, P. y **Eberly**, D. (2003). *Geometric Tools for Computer Graphics*. Estados Unidos: Morgan Kaufmann.

Thau, D. (2000). *The Book of JavaScript*. Estados Unidos: No Starch Press.

von zur Gathen, J. y **Gerhard**, J. (2003). *Modern Computer Algebra*. Segunda Edición. Inglaterra: Cambridge University Press.

von zur Gathen, J. (2003). *Subresultants Revisited*. Theoretical Computer Science. Vol. 297(1-3), pp. 199-239.

Weiss, M. (2000). *Estructuras de Datos en Java*. España: Pearson Educación.

Wu, C. (2001). *An Introduction to Object-Oriented Programming with Java*. Segunda edición. Singapur: McGraw-Hill International.

7. Anexos

En los anexos se encuentra la lista de algoritmos implementados, los artículos publicados a raíz de este proyecto de investigación y las memorias del Seminario de Teoría de Números y Álgebra Computacional.

Algoritmos

Borbón, Alexander; Mora, Walter

Índice general

Lista de Algoritmos

Capítulo 1

Algoritmos

1.1. Simplificación Automática

En el programa realizado se manejan distintos tipos de datos, por ejemplo, están los enteros y los decimales que son números de hasta 16 dígitos, los enteros largos y los decimales largos que son números de más de 16 dígitos, los racionales que se representan como fracciones. Para efecto de los algoritmos que se muestran a continuación se utilizará la siguiente nomenclatura:

- Entero: La expresión puede ser un número entero o entero largo.
- Decimal: La expresión puede ser un número decimal o decimal largo.
- Variable: La expresión representa una variable: x, y, a, etc.
- Constante: La expresión representa una constante: e, pi.
- Número: Se refiere a que la expresión puede ser un número entero, entero largo, decimal, decimal largo, racional o constante.
- Booleano: La expresión representa un valor booleano de falso-verdadero, es decir la expresión contiene true o false.
- Lista: La expresión representa una lista de elementos, donde el orden no importa y no pueden haber elementos repetidos, como un conjunto.
- Fila: La expresión representa una fila de elementos, aquí el orden sí importa, por lo que pueden repetirse elementos en distintas posiciones.
- Matriz: La expresión representa una matriz.

Las expresiones pueden tener funciones ya dadas por el programa y funciones definidas por el usuario (mediante la asignación ":="), en general a todas estas se llamarán de forma genérica como funciones.

La presición en el programa se puede poner en exacta o se puede poner aproximada con una cantidad definida de decimales para los cálculos, en este último caso, todos los números y constantes se manejan como números decimales (o decimales largos según los decimales).

Cuando se indique, por ejemplo, Decimal(u) quiere decir que la expresión u se convirtió a tipo Decimal.

En algunos casos de los algoritmos se indica, por facilidad, el pseudocódigo matemático para lograr el resultado querido, sin embargo, en el programa real se debe manejar de forma distinta. Por ejemplo, en los algoritmos se dirá $u \cdot 10^n$, pero si el número n es un entero largo entonces se maneja como una tira de texto (un String) por lo que en realidad se le debe concatenar n ceros al texto.

El algoritmo 1 muestra el procedimiento principal de la simplificación automática. Básicamente esta función llama a la función de simplificación correspondiente al tipo de la expresión. Para la simplificación automática se supone que cada una de las funciones, operadores y números tiene implementada su propia función de simplificar (más adelante se mostrarán estos algoritmos). Al final se muestra el algoritmo para la simplificación de la suma con números.

Algoritmo 1.1: Simplifica

```
input  : u: una expresión sin simplificar.
output: La expresión simplificada
if Tipo(u) $\in$ {indefinido, variable, booleano} then
    return u;
else if Tipo(u) $\in$ {numero} then
    return SimplificaNumeros(u);
else if Tipo(u) = ":=" then
    return Simplifica(Operando(2));
else
    foreach Operando(u) do
        Simplifica(Operando(u));
        if Tipo(Operando(u))="Indefinido" then
            return "Indefinido";
    if Tipo(u) $\in$ {operador} then
        return SimplificaOperador(u);
    else if Tipo(u) $\in$ {FuncionesUsuario} then
        return SimplificaFuncionUsuario(u);
    else if Tipo(u)="Lista" then
        return OrdenaElementos(u);
    else if Tipo(u) $\in$ {"Matriz", "Fila"} then
        return u;
    else
        return SimplificaFuncion(u);
```

Las siguientes son funciones adicionales que están en el archivo de simplificación automática y que son necesarias a lo largo del programa, muchos de ellos para simplificar otras funciones. El algoritmo 2 muestra el procedimiento para determinar si una expresión representa a un número entero. En este casos si el tipo de la expresión es entera el resultado es directo, la otra opción es que sea de tipo decimal pero no tenga punto (si no tiene decimales entonces es entero).

Algoritmo 1.2: esEntero

```
input  : u: una expresión.
output: Un booleano que indica si la expresión es un entero o no
if Tipo(u) $\in$ {Entero} then
    return true;
else if Tipo(u) $\in$ {Decimal} y el número no tiene punto then
    return true;
return false;
```

El algoritmo 3 muestra el procedimiento para determinar si una expresión representa un número cualquiera. Para este caso si la expresión es de tipo Entero, Decimal, Racional o Constante el resultado es inmediato, pero si es una expresión se debe revisar de manera recursiva si la expresión está formada sólo por números.

Algoritmo 1.3: esNumero

```
input  : u: una expresión.
output: Un booleano que indica si la expresión es un número o no
if Tipo(u) $\in$ {Número} then
    return true;
else if Tipo(u) $\in$ {Variable, Booleano} then
    return false;
else if Tipo(u) $\in$ {Funcion, Operador} then
    foreach Operando(u) do
        if Negacion(esNumero(Operando(u))) then
            return false;
return true;
```

El algoritmo 4 muestra el procedimiento para determinar si una expresión representa un monomio. En este caso primero se revisa si la expresión es un número o una variable en cuyo caso la respuesta es directa, luego verifica si la expresión tiene la forma x^n y, por último, verifica si es un monomio de la forma ax^n o un monomio en varias variables, es decir, con forma $a \cdot {x_0}^{n_0} \cdot {x_1}^{n_1} \cdots {x_k}^{n_k}$.

El algoritmo 5 muestra el procedimiento para determinar si una expresión representa un polinomio. Para esto se debe verificar si la expresión es un monomio

Algoritmo 1.4: esMonomio

```
input  : u: una expresión.
output: Un booleano que indica si la expresión es un monomio o no
if esNumero(u) then
    return true;
else if Tipo(u) = “Variable” then
    return true;
else if Tipo(u) =“Potencia” then
    foreach Operando(u) do
        if Tipo(u.Operando(1))=“Variable” Y esNumero(u.Operando(2))
        Y El número no es negativo then
            return true;
    return false ;
else if Tipo(u) =“Producto” then
    foreach Operando(u) do
        if Negacion(esMonomio(Operando(u))) then
            return false;
    return true ;
return false;
```

o una suma de monomios.

El algoritmo 6 muestra el procedimiento para determinar si una expresión tiene prioridad sobre otra para ordenarlas, por ejemplo, la expresión x^2+2x+1 puede ser escrita $2x+1+x^2$, en todo caso el programa lo ordenará de forma que quede $1+2x+x^2$. Cuando se comparan los tipos en la línea 8 lo que se hace es una comparación de letras (no de números) para decidir cuál va primero.

El algoritmo 7 se utiliza para comparar dos números, a las líneas 6 y 13 se llega cuando a o b son números racionales.

El algoritmo 8 se encarga de devolver la base de la expresión que representa una potencia, por ejemplo, en la expresión x^5 devuelve x, si la expresión no es una potencia devuelve la misma expresión (se asume que es una potencia con exponente 1).

El algoritmo 9 es muy similar al anterior, en este caso se devuelve el exponente de la expresión que representa una potencia, por ejemplo, en la expresión x^5 devuelve 5, si la expresión no es una potencia devuelve un 1 (se asume que es una potencia con exponente 1).

El algoritmo 10 es el que simplifica las expresiones con números. La línea 3 se utiliza cuando el número que se da es decimal y la presición que se utiliza es exacta, en este caso se debe convertir el número decimal a racional o entero; si la expresión tiene un punto “.” o una “E” (en notación científica) entonces se pasa a racional, si no lo tienen es un entero. En la línea 22 el k indica el número de decimales con los que está trabajando el programa en forma aproximada. La

Algoritmo 1.5: esPolinomio

input : u: una expresión.
output: Un booleano que indica si la expresión es un polinomio o no

```
if esMonomio(u) then
    return true;
else if Tipo(u) = "Suma" then
    foreach Operando(u) do
        if Negacion(esMonomio(Operando(u))) then
            return false;
    return true ;
return false;
```

Algoritmo 1.6: comparar

input : a: Primera expresión, b: Segunda expresión
output: Un booleano que indica si "a" tiene prioridad sobre "b"

```
if Tipo(a) $\in$ { Entero, Decimal, Racional } then
    if Tipo(b) $\in$ { Entero, Decimal, Racional } then
        return comparaNumeros(a,b);
    else
        return true;
else if Tipo(a)="Constante" then
    if Tipo(b) = "Constante" then
        return Tipo(a) < Tipo(b);
    else if Tipo(b) $\in$ { Entero, Decimal, Racional } then
        return false;
    else
        return true;
else if Tipo(a)="Variable" Y Tipo(b)="Variable" then
    return a<b ;
else if (Tipo(a)="Suma" Y Tipo(b)="Suma") O (Tipo(a)="Producto"
  Y Tipo(b)="Producto") then
    m=NumeroOperandos(a);
    n=NumeroOperandos(b);
    while m>0 Y n>0 do
        if Negacion(esIgual(a.Operando(m), b.Operando(n))) then
            return comparar(a.Operando(m), b.Operando(n)) ;

return false;
```

Algoritmo 1.7: comparaNumeros

```
input  : Dos expresiones numéricas $a$ y $b$ para comparar
output: Un booleano que indica si $a$ es menor que $b$
if Tipo(a)$\in$ {Entero, Decimal} then
    A=a;
else if Tipo(a)="Constante" then
    Poner en modo aproximado;
    A=simplificaNumeros(a);
else
    A=a.Operando(1)/a.Operando(2);
if Tipo(b)$\in$ {Entero, Decimal} then
    B=b;
else if Tipo(b)="Constante" then
    Poner en modo aproximado;
    B=simplificaNumeros(b);
else
    B=b.Operando(1)/b.Operando(2);
return $A < B$;
```

Algoritmo 1.8: base

```
input  : $u$: Una expresión que representa una potencia
output: La base de la potencia $u$
if Tipo(a)$\in$ {Numero} then
    return "Indefinido";
else if Tipo(a)="Potencia" then
    return a.Operando(1);
else
    return a;
```

Algoritmo 1.9: exponente

```
input  : $u$: Una expresión que representa una potencia
output: El exponente de la potencia $u$
if Tipo(a)$\in$ {Numero} then
    return "Indefinido";
else if Tipo(a)="Potencia" then
    return a.Operando(2);
else
    return 1;
```

función simplificarNumeros que se presenta en la línea 29 y posteriores es una función que se debe definir para todos los operadores y funciones del programa, más adelante se presentará el ejemplo de alguna de estas.

1.1.1. Simplificación de la suma

Para simplificar una suma se deben seguir varias reglas básicas de la adición:

- Asociatividad: $a + (b + c) = (a + b) + c = a + b + c$

  Es decir, una suma se puede representar con una sola cabecera con todos sus sumandos a un mismo nivel, tal como se muestra en la figura 1.1 en donde en ambos casos se está representando la misma suma.

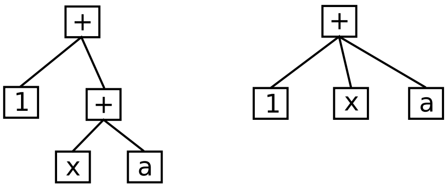


Figura 1.1: Asociatividad de la suma

- Conmutatividad: $a + b = b + a$

  Esta propiedad se utiliza para acomodar los sumandos de acuerdo a la prioridad, tomando el caso de la figura 1.1, el programa acomodaría la suma $1 + x + a$ como $1 + a + x$

- Neutro: $0 + a = a + 0 = a$

  Si uno de los sumandos es cero simplemente se elimina.

- Inverso: $a + (-1 \cdot a) = (-1 \cdot a) + a = 0$

  Si se tiene un término y su inverso aditivo da cero y, por la propiedad anterior, se elimina.

- Distributividad: $a \cdot (b + c) = a \cdot b + a \cdot c$

  Con esta propiedad se pueden sumar términos semejantes, por ejemplo $2x + 3x = (2 + 3)x = 5x$.

Si al eliminar los ceros y los inversos queda sólo un elemento entonces se devuelve ese elemento, es decir, se quita la cabecera de la suma. Si no quedó ningún elemento se devuelve un cero.

El algoritmo 11 es el programa principal para simplificar la suma. En un inicio se verifica si alguno de los sumandos es indefinido o un booleano, en ese caso se devuelve indefinido, el resto llama a la función simplificaSumaRecursiva para hacer la simplificación.

Algoritmo 1.10: simplificaNumeros

```
input  : $u$: Una expresión numérica
output: La expresión simplificada
if Tipo(u)= "Indefinido" then
    return "Indefinido";
else if Presicion=Exacta Y Tipo(u)="Decimal" then
    if "." $\in u$ then
        numerador=$u \cdot 10^n$ con $n$ el número de decimales;
        denominador=$10^n$ con $n$ el número de decimales;
        return SimplificaNumeros(Racional(numerador, denominador));
    else if "E" $\in u$ then
        numerador=$u \cdot 10^n$ con $n$ el exponente;
        denominador=$10^n$ con $n$ el exponente;
        return SimplificaNumeros(Racional(numerador, denominador));
    else
        return Entero(u);
else if Presicion=Aproximado then
    if Tipo(u)="Constante" then
        return Valor($u$);
    else if Tipo(u)="Racional" then
        return $u$.Numerador/$u$.Denominador;
    else if Tipo(u)="Entero" then
        return Decimal($u$);
    else if Tipo(u)="Decimal" then
        return Redondear($u$, k decimales);

if Tipo(u)="Racional" then
    if u.Denominador=0 then
        return "Indefinido";
    else if u.Denominador=1 then
        return Entero($u$.Numerador);
    else if Tipo(u)="Operador" then
        return SimplificarNumeros($u$);
    else if Tipo(u)="Funcion" then
        return SimplificarNumeros($u$);
```

Algoritmo 1.11: simplificarSuma

```
input  : $u$: Una suma
output: La suma simplificada
foreach Sumando(u) do
    if Tipo(u) $\in$ { Indefinido, Booleano } then
        return “Indefinido”
if numeroOperandos(u)=1 then
    return $u$.Operando(1);
else
    simplificado=simplificaSumaRecursiva($u$);
    if Tipo(simplificado)=“Indefinido” then
        return “Indefinido”;
    else if numeroOperandos(simplificado)=0 then
        return 0;
    else if numeroOperandos(simplificado)=1 then
        return simplificado.Operando(1);
    else
        return simplificado;
```

El algoritmo 12 presenta el programa simplificaSumaRecursiva. La idea de esta función es ir trabajando sólo dos sumandos a la vez, por lo que, si hay más, entonces se quita el primero y se llama de manera recursiva a la función (por eso su nombre), esto se hace de la línea 37 en adelante, esto permite ir simplificando y acomodando los sumandos fácilmente. Si alguno de los sumandos es una suma (tal como se mostró en la figura 1.1), entonces se deben unir las sumas, esto lo hace la función UnirSumas que se muestra en el algoritmo 14.

La línea 19 se corre si alguno de los dos son matrices pero no al mismo tiempo por lo que no estaría definido.

En la línea 31 se verifica si el operando2 tiene prioridad sobre el operando1, si es así hay que darles vuelta.

En el caso de la línea 21 es cuando se realiza la suma $ax + bx = (a + b)x$, si la suma da cero se devuelve una suma vacía (la función que la recibe la devuelve como cero), si da uno se devuelve el término y si no se hace la suma, en la línea 28 se supone que el término tiene una multiplicación, por lo que en la siguiente línea sólo se agrega la suma de los números en la primera posición y se devuelve como suma.

El algoritmo 13 es el que realiza la suma de dos matrices.

El algoritmo 14 se utiliza para unir dos sumas en una sola acomodando y simplificando los sumandos. Lo primero es revisar si ambas expresiones que se reciben son sumas, sino se hacen en suma. Luego se van tomando los sumandos en parejas (uno de a y uno de b) y se simplifican, luego se pasan a los siguientes sumandos dependiendo de la simplificación que se haya realizado. Cuando se

Algoritmo 1.12: simplificaSumaRecursiva

```
input  : u: Una suma
output: La suma casi simplificada
if numeroOperandos(u)=2 then
    operando1= u.Operando(1);
    operando2= u.Operando(2);
    if Tipo(operando1)$\neq$"Suma" Y Tipo(operando2)$\neq$"Suma" then
        if Tipo(operando1)$\in$\{ Entero, Decimal, Racional \} Y
        Tipo(operando2)$\in$\{ Entero, Decimal, Racional \} then
            respuesta=simplificarNumeros(u);
            if respuesta=0 then
                return Suma(vacío);
            else
                return Suma(respuesta);
        else if operando1=0 then
            return Suma(operando2);
        else if operando2=0 then
            return Suma(operando1);
        else if Tipo(operando1)="Matriz" Y Tipo(operando2)="Matriz"
        then
            if Tamaño(operando1) $\neq$ Tamaño(operando2) then
                return "Indefinido";
            return sumaMatrices(operando1, operando2);
        else if Tipo(operando1)="Matriz" O Tipo(operando2)="Matriz"
        then
            return "Indefinido";
        else if termino(operando1)=termino(operando2) then
            respuesta=simplificar(Suma(constante(operando1),
            constante(operando2)));
            if respuesta=0 then
                return Suma(vacío);
            else if respuesta=1 then
                return Suma(termino(operando1));
            else
                respuesta2=termino(operando1);
                respuesta2.agrega(respuesta);
                return Suma(respuesta2);
        else if comparar(operando2, operando1) then
            return Suma(operando2, operando1);
        else
            return u;
    else
        return UnirSumas(operando1, operando2);
else
    if Tipo(u)="Indefinido" then
        return "Indefinido";
    v=u;
    v.eliminaOperando(1);
    v=simplificaSumaRecursiva(v);
    return UnirSumas(u.Operando(1), v);
```

Algoritmo 1.13: SumaMatrices

```
input  : u: Dos matrices
output: La suma de las matrices
respuesta=Matriz(vacía);
foreach Fila(operando1) do
    agregaFila(respuesta)  foreach Operando(Fila(operando1)) do
        respuesta.agregar(simplifica(Suma(Operando(Fila(operando1)),
        Operando(Fila(operando2)))));
return sumaMatrices(operando1, operando2);Suma(respuesta);
```

acaban los sumandos de uno de los términos se agregan todos los sumandos de la otra expresión.

El algoritmo 15 se encarga de devolver el término algebraico de una multiplicación, es decir, por ejemplo, para la expresión $3x^2y^5$ devuelve x^2y^5.

El algoritmo 16 es el que se encarga de devolver la constante de la multiplicación, por ejemplo, para la expresión $3x^2y^5$ devuelve 3.

Si la expresión representa una suma de números entonces se debe realizar el procedimiento que se muestra en el algoritmo 17. Cuando alguno de los sumandos es un decimal el resultado se debe revisar si es decimal o entero, esto se debe hacer en la línea 14, en java lo que se hace es tratar de convertir el decimal a entero, si se puede se devuelve como entero, si lanza una excepción entonces se devuelve como decimal. La línea 15 se corre cuando alguno de los números (o ambos) es racional, en este caso se pasan ambos números a racionales y se suma.

1.1.2. Simplificación de la resta

Para simplificar la resta simplemente se expresa como una suma, es decir, $a - b = a + -1 \cdot b$. El programa se muestra en el algoritmo 18.

La simplificación de la resta con números tiene exactamente la misma idea. En el algoritmo 19 se muestra el código.

1.1.3. Simplificación de la multiplicación

Los algoritmos para simplificar un producto son similares a los de la suma, para iniciar se deben seguir varias reglas básicas del producto:

- Asociatividad: $a \cdot (b \cdot c) = (a \cdot b) \cdot c = a \cdot b \cdot c$

  Es decir, un producto se puede representar con una sola cabecera con todos sus sumandos a un mismo nivel, tal como se muestra en la figura 1.2 en donde en ambos casos se representa el mismo producto.

- Conmutatividad: $a \cdot b = b \cdot a$

Algoritmo 1.14: UnirSumas

```
input  : $a$, $b$: Dos sumas
output: La sumas unidas
if Tipo(a)$\neq$ “Suma” then
    $a$=Suma($a$);
else if Tipo(b)$\neq$ “Suma” then
    $b$=Suma($b$);
respuesta=Suma(vacío);
$i$=1;
$j$=1;
while $i$ <=numeroOperandos(a) Y $j$ <=numeroOperandos(a) do
    comodin=Suma($a$.Operando($i$), $b$.Operando($j$));
    comodin=simplificaSumaRecursiva(comodin);
    if numeroOperandos(comodin)=0 then
        $i$++;
        $j$++;
    else if numeroOperandos(comodin)=1 then
        respuesta.agrega(comodin.Operando(1));
        $i$++;
        $j$++;
    else if comodin.Operando1=a.Operando(i) then
        respuesta.agrega(comodin.Operando(1));
        $i$++;
    else if comodin.Operando1=b.Operando(i) then
        respuesta.agrega(comodin.Operando(1));
        $j$++;
if $i$ <=numeroOperando(a) then
    for (; $i$ <=numeroOperando(a); $i$++) do
        respuesta.agrega($a$.Operando($i$));
else if $j$ <=numeroOperando(b) then
    for (; $j$ <=numeroOperando(b); $j$++) do
        respuesta.agrega($b$.Operando($j$));
return respuesta;
```

Algoritmo 1.15: termino

```
input  : u: Una expresión
output: El término algebraico de la multiplicación
if Tipo(u)∈{Número} then
    return "Indefinido";
else if Tipo(u)="Producto" then
    if Tipo(u.Operando(1))∈ { Número } then
        return u.eliminaOperando(1);
    else
        return u;
else
    return Producto(u);
```

Algoritmo 1.16: constantes

```
input  : u: Una multiplicación
output: La constante de la multiplicación
if Tipo(u)∈{Número} then
    return "Indefinido";
else if Tipo(u)="Producto" then
    if Tipo(u.Operando(1))∈ { Número } then
        return u;
return 1;
```

Algoritmo 1.17: simplificarNumerosSuma

```
input  : $u$: Una suma de números
output: La suma simplificada
$a$=u.Operando(1);
$b$=u.Operando(2);
if Tipo(a)∈{“Indefinido”, “Booleano”} O Tipo(b)∈{“Indefinido”, “Booleano”} then
    return “Indefinido;
else if Tipo(a)=Entero Y Tipo(b)=Entero then
    return Entero($a$+$b$);
else if Tipo(a)=Decimal O Tipo(b)=Decimal then
    if Tipo(a)=Racional then
        solucion= $a$.Operando(1)/$a$.Operando(2)+$b$;
    else if Tipo(b)=Racional then
        solucion= $a$+$b$.Operando(1)/$b$.Operando(2);
    else
        solucion= $a$+$b$;
    return EnteroODecimal(solucion);
else
    if Tipo(a)=Entero then
        solucion= Racional($a$)+$b$ ;
    else if Tipo(b)=Entero then
        solucion= $a$+Racional($b$) ;
    else
        solucion= $a$+$b$;
    if Denominador(solucion)=1 then
        return Entero(Numerador(solucion));
    else
        return solucion;
```

Algoritmo 1.18: simplificarResta

```
input  : $a$, $b$: Las expresiones para restar
output: La resta simplificada
if Tipo(a)∈{“Indefinido”, “Booleano”} O Tipo(b)∈{“Indefinido”, “Booleano”} then
    return “Indefinido”;
else
    return simplifica(Suma(a,Producto(-1,b)));
```

Algoritmo 1.19: simplificarNumerosResta

```
input  : $a$, $b$: Las expresiones para restar
output: La resta simplificada
if Tipo(a)∈{"Indefinido", "Booleano"} O Tipo(b)∈{"Indefinido",
"Booleano"} then
    return "Indefinido";
else
    return simplifica(Suma(a,Producto(-1,b)));
```

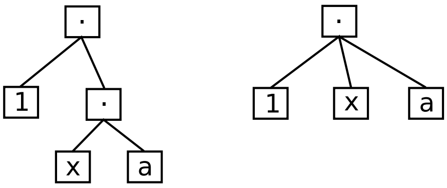


Figura 1.2: Asociatividad de la suma

Esta propiedad se utiliza para acomodar los términos de acuerdo a la prioridad, tomando el caso de la figura 1.2, el programa acomodaría el producto $1 \cdot x \cdot a$ como $1 \cdot a \cdot x$

- Neutro: $1 \cdot a = a \cdot 1 = a$

  Si uno de los términos es uno simplemente se elimina.

- Inverso: $a + \left(a^{-1}\right) = \left(a^{-1}\right) + a = 1$

  Si se tiene un término y su inverso multiplicativo entonces da uno y, por la propiedad anterior, se elimina de un producto.

- Producto por el neutro de la suma: $0 \cdot a = a \cdot 0 = 0$

  Si alguno de los términos de una multiplicación es cero entonces el producto es cero.

Si al eliminar los unos y los inversos queda sólo un elemento entonces se devuelve ese elemento, es decir, se quita la cabecera del producto. Si no quedó ningún elemento se devuelve un uno (el neutro de la suma).

El algoritmo 20 es el programa principal para simplificar el producto. En un inicio se verifica si alguno de los términos es indefinido o un booleano, en ese caso se devuelve indefinido, el resto llama a la función simplificaProductoRecursiva para hacer la simplificación. La variable esCero se utiliza para determinar si alguno de los términos de la multiplicación es un cero, si es así se devuelve un cero.

El algoritmo 21 presenta el programa simplificaMultiplicacionRecursiva. La idea de esta función es ir trabajando sólo dos términos a la vez, por lo que, si hay

Algoritmo 1.20: simplificarProducto

```
input  : u: Un producto
output: El producto simplificado
esCero=falso;
foreach Sumando(u) do
    if Tipo(u) ∈ { Indefinido, Booleano } then
        return "Indefinido"
    else if u=0 then
        esCero=verdadero;
if esCero=verdadero then
    return 0;
else if numeroOperandos(u)=1 then
    return u.Operando(1);
else
    simplificado=simplificaMultiplicacionRecursiva(u);
    if Tipo(simplificado)="Indefinido" then
        return "Indefinido";
    else if numeroOperandos(simplificado)=0 then
        return 1;
    else if numeroOperandos(simplificado)=1 then
        return simplificado.Operando(1);
    else
        return simplificado;
```

más, entonces se quita el primero y se llama de manera recursiva a la función (por eso su nombre), esto se hace de la línea 41 en adelante, esto permite ir simplificando y acomodando los términos fácilmente. Si alguno de los sumandos es un producto (tal como se mostró en la figura 1.2), entonces se deben unir las multiplicaciones, esto lo hace la función UnirMultiplicaciones que se muestra en el algoritmo 25.

Las funciones que multiplican dos matrices, una expresión por una matriz y una multiplicación de dos listas se muestran más adelante en los algoritmos 23, 22 y 24.

En la línea 35 se verifica si el operando2 tiene prioridad sobre el operando1, si es así hay que darles vuelta.

En el caso de la línea 29 es cuando se realiza la multiplicación $a^n \cdot a^m = a^{m+n}$, si la potencia da uno se devuelve un producto vacío (la función que la recibe la devuelve como uno).

El algoritmo 22 es el que realiza la multiplicación de un término por una matriz, en este caso el término se multiplica por cada elemento de la matriz.

El algoritmo 23 es el que realiza la multiplicación de dos matrices.

El algoritmo 24 es el que realiza la multiplicación de dos listas, en este caso el resultado es una lista llena de pares ordenados (en este caso se representan como matrices de dos elementos) representando la combinación de las listas, en donde a cada elemento x de la primera lista se le colocan los elementos de la segunda lista como y.

El algoritmo 25 se utiliza para unir dos multiplicaciones en una sola acomodando y simplificando los términos. Lo primero es revisar si ambas expresiones que se reciben son productos, sino se convierten en productos. Luego se van tomando los términos en parejas (uno de a y uno de b) y se simplifican, luego se pasan a los siguientes térmnos dependiendo de la simplificación que se haya realizado. Cuando se acaban los términos una de las expresiones se agregan todos los términos de la segunda expresión.

Las funciones base y exponente ya se habían definido previamente, sus algoritmos son los número 8 y 9.

Si la expresión representa una multiplicación de números entonces se debe realizar el procedimiento que se muestra en el algoritmo 26. Cuando alguno de los términos es un decimal el resultado se debe revisar si es decimal o entero, esto se debe hacer en la línea 14, en java lo que se hace es tratar de convertir el decimal a entero, si se puede se devuelve como entero, si lanza una excepción entonces se devuelve como decimal. La línea 15 se corre cuando alguno de los números (o ambos) es racional, en este caso se pasan ambos números a racionales y se multiplican.

1.1.4. Simplificación de la división

Para simplificar una división lo que se hace es expresarla como multiplicación, es decir, $\frac{a}{b} = a \cdot b^{-1}$. El programa se muestra en el algoritmo 27.

La simplificación de la división con números tiene exactamente la misma idea. En el algoritmo 28 se muestra el código.

Algoritmo 1.21: simplificaMultiplicacionRecursiva

```
input  : $u$: Un multiplicación
output: La multiplicación casi simplificada
if numeroOperandos(u)=2 then
    operando1= $u$.Operando(1);
    operando2= $u$.Operando(2);
    if Tipo(operando1)$\neq$"Producto" Y Tipo(operando2)$\neq$"Producto" then
        if Tipo(operando1)$\in$\{ Entero, Decimal, Racional \} Y Tipo(operando2)$\in$\{ Entero, Decimal, Racional \} then
            respuesta=simplificarNumeros($u$);
            if respuesta=1 then
                return Producto(vacío);
            else
                return Producto(respuesta);
        else if operando1=1 then
            return Suma(operando2);
        else if operando2=1 then
            return Suma(operando1);
        else if Tipo(operando1)="Matriz" O Tipo(operando2)="Matriz" then
            if Tipo(operando1)$\neq$Tipo(operando2) then
                if Tipo(operando2) $\neq$ "Matriz" then
                    comodin=operando1;
                    operando1=operando2;
                    operando2=comodin;
                return MultiplicaTerminoMatriz(operando1, operando2);
            else if NumeroColumnas(operando1) $\neq$ NumeroFilas(operando2) then
                return "Indefinido";
            return MultiplicaMatrices(operando1, operando2);
        else if Tipo(operando1)="Lista" O Tipo(operando2)="Lista" then
            if Tipo(operando1)$\neq$Tipo(operando2) then
                return "Indefinido";
            return MultiplicaListas(operando1, operando2);
        else if base(operando1)=base(operando2) then
            respuesta=simplificar(Potencia(base(operando1),Suma(Exponente(operando1), Exponente(operando2))));
            if respuesta=1 then
                return Producto(vacío);
            else
                return Producto(respuesta);
        else if comparar(operando2, operando1) then
            return Producto(operando2, operando1);
        else
            return $u$;
    else
        return UnirMultiplicaciones(operando1, operando2);
else
    v=u;
    v.eliminaOperando(1);
    v=simplificaMultiplicacionRecursiva(v);
    return UnirMultiplicaciones(u.Operando(1), v);
```

Algoritmo 1.22: MultiplicaTerminoMatriz

```
input  : $e$: Una expresión, $u$: Una matriz
output: La multiplicación de la expresión $e$ por la matriz $u$

respuesta=Matriz(vacía);
foreach Fila(operando2) do
    agregaFila(respuesta);
    foreach Operando(Fila(operando2)) do
        respuesta.agregar(simplifica(Producto($e$,
        Operando(Fila(operando2)))));
return Producto(respuesta);
```

Algoritmo 1.23: MultiplicaMatrices

```
input  : operando1: La primera matriz, operando2: La segunda matriz
output: La multiplicación de las matrices

respuesta=Matriz(vacía);
foreach Fila(operando1) do
    agregaFila(respuesta);
    foreach Operando(Fila(operando1)) do
        respuesta2=Suma(vacía);
        for $i = 1$; $i <=$NumeroColumnas(operando1); $i++$ do
            respuesta2.agregar(Suma(Operando(Fila(operando1),i),
            Operando(Columna(operando2),i)));
        respuesta.agregar(simplifica(respuesta2));
return respuesta;
```

Algoritmo 1.24: MultiplicaListas

```
input  : operando1: La primera lista, operando2: La segunda lista
output: La multiplicación de las listas

respuesta=Matriz(vacía);
foreach Elemento(operando1) do
    foreach Elemento(operando2) do
        respuesta2=Matriz(vacía);
        agregaFila(respuesta2);
        respuesta2.agregar(Elemento(operando1));
        respuesta2.agregar(Elemento(operando2));
        respuesta.agregar(respuesta2);
return Producto(respuesta);
```

Algoritmo 1.25: UnirMultiplicaciones

```
input  : a, b: Dos multiplicaciones
output: La multiplicaciones unidas
if Tipo(a)≠ "Producto" then
    a=Producto(a);
else if Tipo(b)≠ "Producto" then
    b=Producto(b);
respuesta=Producto(vacío);
i=1;
j=1;
while i <=numeroOperandos(a) Y j <=numeroOperandos(a) do
    comodin=Producto(a.Operando(i), b.Operando(j));
    comodin=simplificaMultiplicacionRecursiva(comodin);
    if numeroOperandos(comodin)=0 then
        i++;
        j++;
    else if numeroOperandos(comodin)=1 then
        respuesta.agrega(comodin.Operando(1));
        i++;
        j++;
    else if comodin.Operando1=a.Operando(i) then
        respuesta.agrega(comodin.Operando(1));
        i++;
    else if comodin.Operando1=b.Operando(i) then
        respuesta.agrega(comodin.Operando(1));
        j++;
if i <=numeroOperando(a) then
    for (; i <=numeroOperando(a); i++) do
        respuesta.agrega(a.Operando(i));
else if j <=numeroOperando(b) then
    for (; j <=numeroOperando(b); j++) do
        respuesta.agrega(b.Operando(j));
return respuesta;
```

Algoritmo 1.26: simplificarNumerosProducto

```
input  : $u$: Una multiplicación de números
output: La multiplicación simplificada
$a$=u.Operando(1);
$b$=u.Operando(2);
if Tipo(a)∈{“Indefinido”, “Booleano”} O Tipo(b)∈{“Indefinido”, “Booleano”} then
    return “Indefinido;
else if Tipo(a)=Entero Y Tipo(b)=Entero then
    return Entero($a \cdot b$);
else if Tipo(a)=Decimal O Tipo(b)=Decimal then
    if Tipo(a)=Racional then
        solucion= $a$.Operando(1)/$a$.Operando(2)$\cdot b$;
    else if Tipo(b)=Racional then
        solucion= $a \cdot b$.Operando(1)/$b$.Operando(2);
    else
        solucion= $a \cdot b$;
    return EnteroODecimal(solucion);
else
    if Tipo(a)=Entero then
        solucion= Racional($a$)$\cdot b$ ;
    else if Tipo(b)=Entero then
        solucion= $a\cdot$Racional($b$) ;
    else
        solucion= $a \cdot b$;
    if Denominador(solucion)=1 then
        return Entero(Numerador(solucion));
    else
        return solucion;
```

Algoritmo 1.27: simplificarDivision

```
input  : $a$, $b$: Las expresiones para dividir
output: La división simplificada
if Tipo(a)∈{“Indefinido”, “Booleano”} O Tipo(b)∈{“Indefinido”, “Booleano”} then
    return “Indefinido”;
else
    return simplifica(Producto(a,Potencia(b,-1)));
```

Algoritmo 1.28: simplificarNumerosDivision

```
input  : $a$, $b$: Las expresiones para dividir
output: La división simplificada
if Tipo(a)∈{ "Indefinido", "Booleano"} O Tipo(b)∈{ "Indefinido",
"Booleano"} then
    return "Indefinido";
else
    return simplifica(Producto(a,Potencia(b,-1)));
```

1.1.5. Simplificación de la potencia

Para simplificar una potencia primero se verifica que ninguno de los elementos sea indefinido o booleano, luego si la base es 1 se retorna un 1 ya que $1^n = 1$, si el exponente es entero se llama a la función simplificaPotenciaEntera cuyo código se muestra más adelante junto con el que simplifica cuando son números; el programa se muestra en el algoritmo 29.

Algoritmo 1.29: simplificarPotencia

```
input  : $u$: Una expresión con una potencia $a^b$
output: La potencia simplificada
if Tipo(Base(u))∈{ "Indefinido", "Booleano"} O
Tipo(Exponente(u))∈{ "Indefinido", "Booleano"} then
    return "Indefinido";
else if Base(u)=1 then
    return 1;
else if Tipo(Exponente(u)) ∈ { Entero} then
    return simplificaPotenciaEntera(Base($u$), Exponente($u$));
else if presicion=aproximada Y Base(u)∈{ Número} Y Exponente(u)∈{
Número} then
    return simplificaNumeros($u$);
else
    return $u$;
```

Cuando la potencia es entera se debe utilizar el algoritmo 30, si la base es un número se llama a la función simplificarNumeros cuyo código se muestra después, algunas de las reglas que se utilizan para este algoritmo son:

- Si el exponente es un cero entonces se devuelve un 1 ya que $a^0 = 1$.
- Si el exponente es 1 se devuelve la base ya que $a^1 = a$.
- $(a^b)^c = a^{b \cdot c}$
- $(a \cdot b)^c = a^c \cdot b^c$

Algoritmo 1.30: simplificaPotenciaEntera

```
input  : $a$, $b$: Los elementos de la potencia $a^b$, donde $b$ es entero
output: La potencia simplificada
if Tipo(Base(u))∈{Número} then
    return simplificarNumeros(Potencia($a$,$b$));
else if $b$=0 then
    return 1;
else if $b$=1 then
    return a;
else if Tipo(Base)="Potencia" then
    p=simplifica(Producto($a$.operando(2),$b$));
    if Tipo(p) ∈ {"Entero"} then
        return simplificaPotenciaEntera($a$.operando(1),p);
    return Potencia($a$.operando(1),p);
else if Tipo(a)="Producto" then
    respuesta=Producto(vacío);
    foreach Operando(a) do
        respuesta.agregar(simplificaPotenciaEntera(Operando($a$), $b$));
    return simplifica(respuesta);
else
    return Potencia($a$,$b$);
```

La simplificación de la potencia cuando tanto la base como el exponente son números se muestra en el algoritmo 31. Tal como en la suma, en los casos donde aparece EnteroODecimal es que se debe verificar qué forma tiene el número, se puede ver la explicación el algoritmo 17.

1.1.6. Simplificación de las funciones trigonométricas

En este caso se mostrará como ejemplo la simplificación de la función seno, las demás funciones trigonométricas se simplifican de un modo similar. El programa se muestre en el algoritmo 32.

El algoritmo que simplifica la función seno cuando el argumento es un número se muestra en el algoritmo 33. En un inicio si el ángulo está en grados se pasa a radianes y se lleva el ángulo entre 0 y 2π. La línea 11 calcula seno de forma aproximada, si se está trabajando con números pequeños se utiliza la fución que trae el programa ya definida sino se deben utilizar métodos numéricos para encontrar el seno con una cantidad especificada de decimales. En la línea 64 se regresa el ángulo a grados (de ser necesario).

1.1.7. Simplificación de las funciones de comparación

Como ejemplo, se mostrará los algoritmos para simplificar la función “mayor que”, las demaś funciones de comparación (menor que, mayor o igual que, menor o igual que, igual que) se realizan de una forma similar.

Para simplificar un “mayor que” lo que se verifica es sólo si ambos elementos son números y se llama a la función correspondiente, sino se devuelve el mismo término, el programa se muestra en el algoritmo 34.

Cuando ambos elementos son números lo que se hace es calcularlos de forma aproximada y hacer la comparación, la función se muestra en el algoritmo 35. En la línea 5 se pregunta si no es un tipo de número conocido, esto quiere decir que es una expresión de números tal como $1 + \sqrt{2}$ por lo que hay que aproximar el número.

1.1.8. Simplificación de las funciones Lógicas

En este caso también se mostrará una de las funciones como ejemplo, se mostrará los algoritmos para simplificar la función “Y” lógica, las demaś funciones lógicas (O, XOR, Negación) se realizan de una forma similar.

Para simplificar un “Y” lógico se siguen las reglas del cuadro 1.1, el programa se muestra en el algoritmo 36.

Cuando ambos elementos son números lo que se devuleve es un cero, la función se muestra en el algoritmo 37.

1.2. Derivación

El programa principal permite tres formas de escribir una derivada, Der(u) que calcula la derivada de la expresión u, es este caso lo primero que se hace

Algoritmo 1.31: simplificarNumerosPotencia

```
input  : $u$: Una potencia de números
output: La potencia simplificada
$a$=u.Operando(1);
$b$=u.Operando(2);
if Tipo(b)$\in$ {Entero} then
    if Tipo(a)$\in$ {Entero} then
        if $b > 0$ then
            return $a^b$;
        else if $b = 0$ then
            if $a = 0$ then
                return “Indefinido”;
            else
                return 1;
        else
            if $a = 0$ then
                return “Indefinido”;
            else if $a = 1$ then
                return 1;
            else if $a = -1$ then
                return-1;
            else
                return $1/(a^{(-b)})$;
    else if Tipo(a)=“Decimal” then
        if $b >= 0$ then
            solucion=$a^b$;
        else
            solucion=$a^{-b}$
        return EnteroODecimal(solucion);
    else
        solucion=$Racional(a)^b$;
        if Denominador(solucion)=1 then
            return
        Numerador(solucion);
        else
            return solucion;
else if presicion=aproximado then
    return $Decimal(a)^{Decimal(b)}$;
return “Indefinido”;
```

Algoritmo 1.32: simplificarSeno

```
input  : $u$: Una expresión con un seno.
output: El seno simplificado
if Tipo(u.Operando(1))∈{"Indefinido", "Booleano"} then
    return "Indefinido";
if esNumero(u.Operando(1)) then
    return simplificarNumeros($u$);
else
    return $u$;
```

es determinar la variable de u con la que se va a derivar, si no tiene variables (es un números) entonces se devuelve un cero. La segunda forma es Der(u, v) que indica la variable con respecto a la cual se va a derivar. La tercer forma es Der(u, v, n) que, además de indicar la expresión y la variable también indica el número de derivadas con el entero n, es decir, se calcula la n-ésima derivada.

El algoritmo 38 muestra el programa que simplifica la expresión de la derivada. El If en la línea 5 se utiliza para saber si se indicó la variable con la que se derivará, si el tipo es variable entonces se indicó la variable directamente sino es porque probablemente sea una expresión que no se simplificó todavía por lo que se devuelve la misma expresión. Si no se indicó una variable en el programa entonces se busca la variable en la expresión u con la función buscaVariable cuyo código se muestra más adelantes, si no hay variable es un número y se devuelve 0, sino se agrega la variable en la expresión.

En la línea 15 se pregunta si se está derivando una derivada, en cuyo caso sería una segunda derivada (siempre que tengan la misma variable), por lo que en este caso se suman el número n de ambas derivadas.

La línea 30 se encarga de obtener el número de derivadas (si lo dan), si es entero es directo pero si es decimal se debe revisar si el decimal representa un entero. Por último, se simplifican todas las derivadas que se deben calcular.

El algoritmo 39 muestra la función buscaVariable que se encarga de devolver la primera variable que se encuentre en la expresión.

El algoritmo 40 muestra la función buscaVariable que recibe una expresión y una variable e indica si en la expresión aparece la variable dada.

El algoritmo 41 determina la derivada de una derivada. En la línea 1 se verifica si las derivadas se deben calcular con respecto a la misma variable, si es así y era la primera derivada entonces se agrega un 2 para indicar que ahora se debe calcular la segunda derivada.

La función deriva es la función principal que lleva a cabo la derivación de la expresión llamando a la derivada de cada una de las funciones, operadores y constantes. El código se muestra en el algoritmo 42. Si la expresión es una variable y es con respecto a la que se está derivando entonces se devuelve un 1 sino un cero. Si la expresión es un número entonces se devuelve cero. Para derivar una matriz se deriva cada elemento de la matriz. Una lista se devuelve

Algoritmo 1.33: simplificarNumerosSeno

```
input  : $u$: Una expresión con un seno cuyo argumento es un número.
output: El seno simplificado
if Tipo(u.Operando(1))∈{"Indefinido", "Booleano"} then
    return "Indefinido";
if TipoAngulos=grados then
    angulo=$u$.Operando(1) $\cdot\pi/180$;
comodin2=simplifica(Truncar(angulo $/(2\cdot\pi)$));
if comodin2<0 then
    comodin2=comodin2-1;
comodin=comodin2$\cdot 2\pi$;
angulo=simplificar(angulo-comodin);
if Presicion=Aproximada then
    return sen(angulo);
else
    if angulo=0 O angulo=$\pi$ then
        return 0;
    else if angulo=$\pi/2$ then
        return 1;
    else if angulo=$3\cdot\pi/2$ then
        return 1;
    else if angulo=$\pi/3$ O angulo=$2\cdot\pi/3$ then
        return $1/2\cdot\sqrt{3}$;
    else if angulo=$4\cdot\pi/3$ O angulo=$5\cdot\pi/3$ then
        return $-1/2\cdot\sqrt{3}$;
    else if angulo=$\pi/4$ O angulo=$3\cdot\pi/4$ then
        return $1/2\cdot\sqrt{2}$;
    else if angulo=$5\cdot\pi/4$ O angulo=$7\cdot\pi/4$ then
        return $-1/2\cdot\sqrt{2}$;
    else if angulo=$\pi/5$ O angulo=$4\cdot\pi/5$ then
        return $1/4\cdot\sqrt{10+-2\cdot\sqrt{5}}$;
    else if angulo=$6\cdot\pi/5$ O angulo=$9\cdot\pi/5$ then
        return $-1/4\cdot\sqrt{10+-2\cdot\sqrt{5}}$;
    else if angulo=$2\cdot\pi/5$ O angulo=$3\cdot\pi/5$ then
        return $1/4\cdot\sqrt{10+2\cdot\sqrt{5}}$;
    else if angulo=$7\cdot\pi/5$ O angulo=$8\cdot\pi/5$ then
        return $-1/4\cdot\sqrt{10+2\cdot\sqrt{5}}$;
    else if angulo=$\pi/6$ O angulo=$5\cdot\pi/6$ then
        return $1/2$;
    else if angulo=$7\cdot\pi/6$ O angulo=$11\cdot\pi/6$ then
        return $-1/2$;
    else if angulo=$\pi/8$ O angulo=$7\cdot\pi/8$ then
        return $1/2\cdot\sqrt{2+-1\cdot\sqrt{2}}$;
    else if angulo=$9\cdot\pi/8$ O angulo=$15\cdot\pi/8$ then
        return $-1/2\cdot\sqrt{2+-1\cdot\sqrt{2}}$;
    else if angulo=$3\cdot\pi/8$ O angulo=$5\cdot\pi/8$ then
        return $1/2\cdot\sqrt{2+\sqrt{2}}$;
    else if angulo=$11\cdot\pi/8$ O angulo=$13\cdot\pi/8$ then
        return $-1/2\cdot\sqrt{2+\sqrt{2}}$;
    else if angulo=$\pi/10$ O angulo=$9\cdot\pi/10$ then
        return $1/4\cdot(-1+\sqrt{5})$;
    else if angulo=$11\cdot\pi/10$ O angulo=$19\cdot\pi/10$ then
        return $-1/4\cdot(-1+\sqrt{5})$;
    else if angulo=$3\cdot\pi/10$ O angulo=$7\cdot\pi/10$ then
        return $1/4\cdot(1+\sqrt{5})$;
    else if angulo=$13\cdot\pi/10$ O angulo=$17\cdot\pi/10$ then
```

Algoritmo 1.34: simplificarMayorQue

```
input  : u: Una expresión con un mayor que $a > b$
output: La comparación simplificada
if Tipo(u.Operando(1))∈{“Indefinido”, “Booleano”} O
Tipo(u.Operando(2))∈{“Indefinido”, “Booleano”} then
    return “Indefinido”;
if esNumero(u.Operando(1)) Y esNumero(u.Operando(2)) then
    return simplificarNumeros(u);
else
    return u;
```

Algoritmo 1.35: simplificarNumerosMayorQue

```
input  : u: Una comparación de dos números
output: La comparación simplificada
a=u.Operando(1);
b=u.Operando(2);
if Tipo(a)∈{“Indefinido”, “Booleano”} O Tipo(b)∈{“Indefinido”,
“Booleano”} then
    return “Indefinido”;
if Tipo(a)∉ {“Entero”, “Decimal”, “Racional”} then
    presicion=aproximado;
    a=simplifica(a);
    restablecer(presicion);
else if Tipo(a)={“Racional”} then
    a=Numerador(a)/Denominador(a);
else
    a=Decimal(a);
if Tipo(b)∉ {“Entero”, “Decimal”, “Racional”} then
    presicion=aproximado;
    b=simplifica(b);
    restablecer(presicion);
else if Tipo(b)={“Racional”} then
    b=Numerador(b)/Denominador(b);
else
    b=Decimal(b);
if a¿b then
    return Boolean(verdadero);
else
    return Boolean(falso);
```

Cuadro 1.1: Tabla de verdad para el “Y” lógico

a	**b**	**a Y b**
Verdadero	Verdadero	Verdadero
Verdadero	Falso	Falso
Falso	Verdadero	Falso
Falso	Falso	Falso

Algoritmo 1.36: simplificarY

```
input  : u: Una expresión con un “Y” lógico
output: La expresión simplificada
a=u.Operando(1);
b=u.Operando(2);
if Tipo(a)∈{“Indefinido”} O Tipo(b)∈{“Indefinido”} then
    return “Indefinido”;
if Tipo(a)=“Booleano” then
    if a=Falso then
        return Booleano(Falso);
    else
        if Tipo(b)=“Booleano” then
            if b=Falso then
                return Booleano(Falso);
            else
                return Booleano(Verdadero);
        else
            return b;
else if Tipo(b)=“Booleano” then
    if b=Falso then
        return Booleano(Falso);
    else
        return a;
if esNumero(a) Y esNumero(b) then
    simplificarNumeros(u);
return u;
```

Algoritmo 1.37: simplificarNumerosY

```
input  : $u$: Una comparación de dos números
output: La comparación simplificada
if Tipo(u.Operando(1))∈{ "Indefinido"} O
Tipo(u.Operando(2))∈{ "Indefinido"} then
    return "Indefinido";
return 0;
```

sin derivar. En el caso en que sea un operador, una constante o una función entonces se llama a la función definida en cada una de estas.

1.2.1. Derivación de la suma

La regla de derivación de la suma es simple: $[a+b]' = a' + b'$, es decir, se derivan todos los sumandos, el programa se muestra en el algoritmo 43.

1.2.2. Derivación de la resta

La regla de la resta es idéntica al de la suma $[a-b]' = a' - b'$, el algoritmo también es idéntico, la única diferencia es que en la línea 1 en vez de hacer una suma se hace una resta vacía a la que se le van agregando las derivadas.

1.2.3. Derivación de la multiplicación

La derivada de un multiplicación se calcula con la fórmula: $[a \cdot b]' = a' \cdot b + a \cdot b'$ en el algoritmo 44 se puede ver el programa.

1.2.4. Derivación de la división

Para la derivada de la división se utiliza la fórmula $\left[\frac{a}{b}\right]' = \frac{a' \cdot b - a \cdot b'}{b^2}$, en el algoritmo 45 se muestra el programa.

1.2.5. Derivación de la potencia

La potencia es un poco más delicada ya que si el exponente es un número entonces $x^n = nx^{n-1}$, pero si la base es un número entonces $a^x = a^x \ln a$ y, por último, si hay variables tanto en la base como en el exponente entonces $\left[f(x)^{g(x)}\right]' = f(x)^{g(x)} \cdot [g(x) \cdot \ln(f(x))]'$. El programa se muestra en el algoritmo 46.

1.2.6. Derivación de las funciones trigonométricas

Se pondrá por ejemplo la derivada de la función seno, para las demaś se utiliza una idea similar, el código se muestra en el algoritmo 47

Algoritmo 1.38: simplificarDerivada

```
input  : u: Una derivada
output: La derivada simplificada
foreach Operador(u) do
    if Tipo(Operador(u))∈{“Indefinido”, “Booleano”} then
        return
    “Indefinido”;
if numeroOperandos(u)¿1 then
    if Tipo(u.Operando(2))=“Variable” then
        variable=u.Operando(2);
    else
        return u;
else
    variable=buscaVariable(u);
    if variable=null then
        return 0;
    u.agrega(variable);
if Tipo(u.Operando(1))=“der” then
    if variable=u.Operando(1).Operando(2) then
        if numeroOperandos(u)<=2 then
            n1=1;
        else
            n1=u.Operando(3);
        if numeroOperandos(u.Operando(1))<=2 then
            n1=1;
        else
            n1=u.Operando(1).Operando(3);
        n=n1+n2;
        return der(u.Operando(1).Operando(1), variable, n);
    else
        return u;
num=0;
if numeroOperandos(u)>2 then
    numero=u.Operando(3);
    if Tipo(numero)=“Entero” then
        num=numero;
    else if Tipo(numero)=“Decimal” then
        comparar=redondear(numero);
        if comparar=numero then
            num=numero
    else if Tipo(numero) ∈ {“EnteroLargo”, “DecimalLargo”} then
        Error(“El número es demasiado grande”);
    else
        return u;
u=u.Operando(1);
for i=1; i<=num; i++ do
    expresion=simpifica(deriva(expresion, variable));
return u;
```

Algoritmo 1.39: buscaVariable

```
input  : $u$: Una expresión
output: La primer variable que se encuentre
if Tipo(u)="Variable" then
    return $u$;
else if Tipo(u)∈{ Constante } then
    return null;
else
    encontrado=null;
    for i=1; i<=numeroOperandos(u); i++ do
        encontrado=buscaVariable($u$.Operando(i));
        if encontrado $\neq$ null then
            return encontrado;
return null;
```

Algoritmo 1.40: buscaVariable

```
input  : $u$: Una expresión, $v$: una variable
output: Un booleano que indica si la variable está en la expresión
if Tipo(u)="Variable" then
    if $u = v$ then
        return true;
    else
        return false;
else if Tipo(u)∈{ Constante } then
    return false;
else
    encontrado=null;
    for i=1; i<=numeroOperandos(u); i++ do
        encontrado=buscaVariable($u$.Operando(i), $v$);
        if encontrado = true then
            return true;
return false;
```

Algoritmo 1.41: derivarDerivada

input : u: Una expresión con una derivada, v: La variable con la que se derivará

output: La derivada de la expresión

```
if v=u.Operando(2) then
    if numeroOperandos(u)=2 then
        return u.agregar(2);
    else
        comodin=Suma(u.Operando(3),1);
        return Der(u.Operando(1), u.Operando(2), comodin);
else
    return Der(u,v);
```

Algoritmo 1.42: deriva

input : u: Una expresión para derivar, v: La variable con la que se derivará

output: La derivada de la expresión

```
if Tipo(u)=“Variable” then
    if $u = v$ then
        return 1;
    else
        return 0;
else if Tipo(u ∈{ Número} then
    return 0;
else if Tipou=“Matriz” then
    foreach u.Fila(i) do
        foreach u.Columna(j) do
            u.Cambia(Elemento(i,j), Deriva(Elemento(i,j)));
    return u;
else if Tipo(u)=“Lista” then
    return Der(u, v);
else if Tipo(u)∈ {Operadores} then
    return Operador(Tipo(u)).Derivar(u, v);
else if Tipo(u)∈{ Constantes } then
    return 0;
else if Tipo(u)∈{ Funciones } then
    return Funcion(Tipo(u)).Derivar(u, v);
```

Algoritmo 1.43: derivarSuma

```
input  : $u$: Una suma, $v$: variable
output: La derivada de la suma
derivada=Suma(vacía);
foreach Operando(u) do
    derivada.agregar(Deriva(Operando($u$), $v$));
return derivada;
```

Algoritmo 1.44: derivarMultiplicacion

```
input  : $u$: Una multiplicación, $v$: variable
output: La derivada de la multiplicación
producto1=Producto(vacío);
producto1.agregar(deriva($u$.Operando(1), $v$));
producto1.agregar($u$.eliminarOperando(1));
producto2=Producto(vacío);
producto2.agregar($u$.Operando(1));
producto2.agregar(deriva($u$.eliminarOperando(1), $v$));
derivada=Suma(vacía);
derivada.agregar(producto1);
derivada.agregar(producto2);
return derivada;
```

Algoritmo 1.45: derivarDivisión

```
input  : $u$: Una división, $v$: variable
output: La derivada de la división
producto1=Producto(vacío);
producto1.agregar(deriva($u$.Operando(1), $v$));
producto1.agregar($u$.eliminarOperando(1));
producto2=Producto(vacío);
producto2.agregar($u$.Operando(1));
producto2.agregar(deriva($u$.eliminarOperando(1), $v$));
resta=Resta(producto1, producto2);
derivada=Division(resta, Potencia($u$.eliminarOperando(1), 2));
return derivada;
```

Algoritmo 1.46: derivarPotencia

```
input  : $u$: Una potencia, $v$: variable
output: La derivada de la potencia
if buscaVariable(u.Operando(2), v)=Falso then
    return Producto($u$.Operando(2), Potencia($u$.Operando(1),
    Resta($u$.Operando(2),1)));
else if buscaVariable(u.Operando(1), v)=Falso then
    return Producto($u$, Ln($u$.Operando(1)));
else
    derivada1=Ln($u$.Operando(1));
    derivada2=deriva(Producto($u$.Operando(2), derivada1),$v$);
    return Producto($u$, derivada2);
```

Algoritmo 1.47: derivarSeno

```
input  : $u$: Un seno, $v$: variable
output: La derivada de seno
return Producto(Cos($u$.Operando(1)), Der($u$.Operando(1)));
```

1.3. Matemáticos

Algoritmo 1.48: Algoritmo de la división

```
Data: $a$, $b \in \mathbb{Z}$ no ambos nulos
Result: $q, r$ tal que $a = bq + r,\ \ 0 \leq r < b$
if $b > 0$ then
    return $q = \lfloor\!\lfloor a/b \rfloor\!\rfloor$ , $r = a - qb$
else
    return $q = -\lfloor\!\lfloor a/|b| \rfloor\!\rfloor$ ,;
    ;
    $r = a - qb$
```

Algoritmo 1.49: Máximo común divisor

```
Data: $a$, $b \in \mathbb{Z}$.
Result: $\operatorname{mcd}(a, b)$
if $a = 0$ y $b = 0$ then
    return $\operatorname{mcd}(a, b) = 0$
$c = |a|,\ d = |b|$ ;
while $d \neq 0$ do
    $r = \operatorname{rem}(c, d)$;
    $c = d$;
    $d = r$;
return $\operatorname{mcd}(a, b) = |c|$;
```

Algoritmo 1.50: Algoritmo Extendido de Euclides

```
Data: $a$, $b$ enteros no ambos nulos
Result: mcd $(a,b)$, $t$ y $s$
$c = |a|,\ d = |b|$ ;
$c_1 = 1,\ d_1 = 0$ ;
$c_2 = 0,\ d_2 = 1$ ;
while $d \neq 0$ do
    $q = \mathrm{quo}(c,d)$,      $r = c - qd$,
    $r_1 = c_1 - qd_1$,      $r_2 = c_2 - qd_2$,
    $c = d$,                 $c_1 = d_1$,        $c_2 = d_2$,
    $d = r$,                 $d_1 = r_1$,        $d_2 = r_2$,
return mcd (a,b) $= |c|$,  $s = c_1/\mathrm{sgn}(a) \cdot \mathrm{sgn}(c)$,  $t = c_2/\mathrm{sgn}(b) \cdot \mathrm{sgn}(c)$ ;
```

Algoritmo 1.51: Inverso Multiplicativo mod m.

```
Data: $a \in \mathbb{Z}_m,\ m > 1$
Result: $a^{-1}$ si mcd $(a,m) = 1$.
Calcular $s, t$ tal que $sa + tm =$ mcd $(a,m)$ ;
if $mcd(a,m) > 1$ then
    $a^{-1}$ no existe
else
    return rem $(s,m)$.
```

Algoritmo 1.52: Raíz cuadrada de un BigInteger

```
Data: $N \in \mathbb{N}$
Result: $\left\lfloor \sqrt{N} \right\rfloor \in \mathbb{N}$
if $N = 1$ then
    return 1
if $N > 1$ then
    $x_k = N$ ;
    $x_{k+1} = \mathrm{quo}(N,2)$ ;
    while $x_{k+1} < x_k$ do
        $x_k = x_{k+1}$ ;
        $x_{k+1} = \mathrm{quo}(x_k + \mathrm{quo}(N, x_k), 2)$ ;
    return $x_{k+1}$ ;
```

Algoritmo 1.53: Criba de Eratóstenes

```
Data: $n \in \mathbb{N}$
Result: Primos entre 2 y $n$
máx$= (n-3)/2$ ;
boolean esPrimo[$i$],   $i = 1, 2, ...$,máx;
for $i = 1, 2, ..., máx$ do
    esPrimo[$i$] =True;
$i = 0$ ;
while $(2i+3)(2i+3) \leq n$ do
    $k = i + 1$ ;
    if esPrimo(i) then
        while $(2k+1)(2i+3) \leq n$ do
            esPrimo[$((2k+1)(2i+3)-3)/2$] =False;
            $k = k + 1$;
    $i = i + 1$;
Imprimir;
for $j = 1, 2, ..., máx$ do
    if esPrimo[$j$] =True then
        Imprima $j$
```

Algoritmo 1.54: Colado de primos entre m y n.

```
Data: $n, m \in \mathbb{N}$ con $m < n$.
Result: Primos entre $m$ y $n$
Primo() = una lista de primos $\leq \sqrt{n}$ ;
$min = (m+1-3)/2$; $max = (n-3)/2$ ;
$esPrimo[i]$, $i = min, ..., max$ ;
for $j = min, ..., max$ do
    $esPrimo[j]$ =True;
$np$ = cantidad de primos en la lista Primos;
Suponemos Primo(0) = 2;
for $i = 1, 2, ..., np$ do
    if $m \leq p_i^2$ then
        $k = (p_i - 1)/2$ ;
        while $(2k+1)p_i \leq n$ do
            esPrimo[$((2k+1)p_i - 3)/2$] =False;
            $k = k + 1$;
    if $p_i^2 < m$ then
        $q = (m-1)/p$ ;
        $q_2 = \text{rem}(q, 2)$ ;
        $k = q_2$ ;
        $mp = (2k + 1 - q_2 + q) \cdot p_i$ ;
        while $mp \leq n$ do
            esPrimo[$(mp-3)/2$] =False;
            $k = k + 1$;
            $mp = (2k + 1 - q_2 + q) \cdot p_i$
Imprimir;
for $j = min, ..., max$ do
    if $esPrimo[j] = True$ then
        Imprima $2 * i + 3$
```

Algoritmo 1.55: Factorización por Ensayo y Error.

```
Data: $N \in \mathbb{N},\ G \leq \sqrt{N}$
Result: Un factor $p \leq G$ de $N$ si hubiera.
$p = 5$;
if $N$ es divisible por $2$ o $3$ then
    Imprimir factor;
else
    while $p \leq G$ do
        if $N$ es divisible por $p$ o $p+2$ then
            Imprimir factor;
            break;
        end
        $p = p + 6$
    end
end
```

Algoritmo 1.56: Método rho de Pollard (variante de R. Brent)

```
Data: $N \in \mathbb{N},\ f,\ x_0$
Result: Un factor $p$ de $N$ o mensaje de falla.
salir=false;
$k = 0,\ x_0 =$ Random$(2, N-1)$;
$x_i = x_0$;
while salir=False do
    $i = 2^k - 1$;
    for $j = i+1, i+2, ..., 2i+1$ do
        $x_j = f(x_0) \pmod{N}$;
        if $x_i = x_j$ then
            salir=True;
            Imprimir "El método falló. Reintentar cambiando $f$ o $x_0$";
            Exit For;
        $g = \text{mcd}\,(x_i - x_j, N)$;
        if $1 < g < N$ then
            salir=True;
            Imprimir $N = N/g \cdot g$;
            Exit For;
        $x_0 = x_j$;
    $x_i = x_j$;
    $k{+}{+}$;
```

Algoritmo 1.57: Miller-Rabin

```
Data: $n \geq 3$ y un parámetro de seguridad $t \geq 1$.
Result: "$n$ es primo" o "$n$ es compuesto".
Calcule $r$ y $s$ tal que $n-1 = 2^s r$, $r$ impar;
for $i = 1, 2, ..., t$ do
    $a =$ Random$(2, n-2)$;
    $y = a^r \pmod{n}$;
    if $y \neq 1$ y $y \neq n-1$ then
        $j = 1$;
        while $j \leq s-1$ y $y \neq n-1$ do
            $y = y^2 \pmod{n}$;
            if $y = 1$ then
                return "Compuesto";
            $j = j + 1$;
        if $y \neq n-1$ then
            return "Compuesto";
return "Primo";
```

Algoritmo 1.58: Problema Chino del Resto en $\mathbb{Z}$. Algoritmo de Garner

```
Data: $(u_0, u_1, ...u_n)$, $(m_0, m_1, ..., m_n)$ con $m_i \in \mathbb{Z}$ positivos y primos
        relativos dos a dos y $u_i \in \mathbb{Z}_{m_i}$ .
Result: $u \in \mathbb{Z}_m$ con $m = \prod_{i=0}^{n} m_i$ tal que
        $u \equiv u_i \pmod{m_i}$, $i = 0, 1, ..., n$.
Cálculo de inversos;
for $k = 1$ to $n$ do
    producto $= \varphi_{m_k}(m_0)$ ;
    for $i = 1$ to $k - 1$ do
        producto $= \varphi_{m_k}(\text{producto} \cdot m_i)$ ;
    $\gamma_k = (\text{producto})^{-1}\,(\text{mód}\, m_k)$ ;
Cálculo de los $v_k$ ;
$v_0 = u_0$ ;
$j = 0$ ;
for $k = 1$ to $n$ do
    temp $= v_{k-1}$ ;
    $j = k - 2$ ;
    while $j \geq 0$ do
        temp $= \varphi_{m_k}(\text{temp} \cdot m_j + v_j)$ ;
        $j = j - 1$ ;
    $v_k = \varphi_{m_k}((u_k - \text{temp})\gamma_k)$ ;
Pasar $u$ a base $10$ ;
$u = v_n$ ;
$j = n - 1$ ;
while $j \geq 0$ do
    $u = u \cdot m_j + v_j$ ;
    $j = j - 1$ ;
return $u$ ;
```

Algoritmo 1.59: Aproximación Racional con Fracciones Continuas.

```
Data: $x \in \mathbb{R} - \mathbb{Z},\ x \neq 0$.
Result: $\frac{A_n}{B_n}$ tal que $\left(x - \frac{A_n}{B_n}\right) < 10^{-15}$.
$X = x$ ;
$A_1 = 1$ ;
$B_1 = 0$ ;
$A_2 = 0$ ;
$B_2 = 1$ ;
$A_n = 1$ ;
$B_n = 1$ ;
if $x \neq 0$ then
    while $\left(x - \frac{A_n}{B_n}\right) > 10^{-15}$ do
        $b = \lfloor X \rfloor$ ;
        $A_n = bA_1 + A_2$ ;
        $B_n = bB_1 + B_2$ ;
        $A_2 = A_1$ ;
        $B_2 = B_1$ ;
        $A_1 = A_n$ ;
        $B_1 = B_n$ ;
        $X = X - b$ ;
        $X = 1/X$ ;
return $A_n/B_n$
```

Algoritmo 1.60: Evaluación de Polinomios. Método de Horner.

```
Data: Un polinomio $A(x) = a_0 + ... + a_n x^n$ y $x_0 \in \mathbb{R}$
Result: $A(x_0)$
valor= $a_n$;
$j = n - 1$;
while $j \geq 0$ do
    valor= $a_j + x_0 \cdot$valor;
    $j = j - 1$;
return valor
```

Algoritmo 1.61: División de Polinomios en $K[x]$, K campo.

```
Data: Polinomios $A$, $B$ en $K[x]$
Result: Polinomios $Q$, $R$ tal que $A = Q \cdot B + R$ con grado(R)¡grado(B)
$R = A$;
$Q = 0$;
while $grado(R) \geq grado(B)$ do
    $\delta = \text{grado}(R) - \text{grado}(B)$;
    $M = \frac{cp(R)}{cp(B)}\, x^{\delta}$;
    $Q = Q + M$;
    $R = R - B \cdot M$;
return $Q$, $R$.
```

Algoritmo 1.62: Algoritmo de Euclides

```
Data: Polinomios $A$, $B$ en $F[x]$, $F$ campo.
Result: mcd AB normalizado
$C = n(A)$;
$D = n(B)$;
while $D \neq 0$ do
    $R = rem(C, D)$;
    $C = D$;
    $D = R$;
return $n(C)$
```

Algoritmo 1.63: Algoritmo Extendido de Euclides

```
Data: Polinomios $A$, $B$ en $F[x]$, $F$ campo.
Result: mcd $(A, B)$ normalizado, $T$, $S \in F[x]$ tal que
            mcd $(A, B) = SA + TB$
$C = n(A),\ D = n(B)$;
$C_1 = 1,\ D_1 = 0$;
$C_2 = 0,\ D_2 = 1$;
while $D \neq 0$ do
    $Q = \text{quo}(C, D)$,      $R = C - QD$,
    $R_1 = C_1 - QD_1$,    $R_2 = C_2 - QD_2$,
    $C = D$,               $C_1 = D_1$,        $C_2 = D_2$,
    $D = R$,               $D_1 = R_1$,        $D_2 = R_2$,
$G = n(C)$;
return $\left\{ \text{mcd} = G,\ S = \frac{C_1}{u(A) \cdot u(C),}\ \ T = \frac{C_2}{u(B) \cdot u(C)} \right\}$;
```

Algoritmo 1.64: Algoritmo Primitivo de Euclides.

```
Data: Polinomios $A(x), B(x) \in D[x]$, $D$ DFU.
Result: $G(x) = \text{mcd}\,(A(x), B(x))$
$C(x) = \text{pp}(A(x))$ ;
$D(x) = \text{pp}(B(x))$ ;
while $D(x) \neq 0$ do
    $R(x) = \text{prem}(C(x), D(x))$ ;
    $C(x) = D(x)$ ;
    $D(x) = \text{pp}(R(x))$ ;
$\lambda = \text{mcd}\,(\text{cont}(A(x)), \text{cont}(B(x)))$ ;
$G(x) = \lambda\, C(x)$ ;
return $G(x)$ ;
```

Algoritmo 1.65: Algoritmo PRS Subresultante.

```
Data: Polinomios $A(x), B(x) \in D[x]$, $\text{grado}\, A(x) \geq \text{grado}\, B(x)$, $D$
      DFU.
Result: $\text{MCD}\,(A(x), B(x))$
$r_0 = A(x),\ r_1 = B(x)$ ;
$deg_0 = \text{grado}(r_0),\ deg_1 = \text{grado}(r_1),\ cp_0 =$ 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
    $c = \alpha_1\, r_0,\ q = \text{quo}(c, r_1),\ r_0 = r_1 w\ r_1 = \text{quo}(c - q \cdot r_1, \beta_1)$ ;
    $deg_0 = \text{grado}(r_0),\ deg_1 = \text{grado}(r_1)$ ;
    $cp_0 =$ 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
        $\psi_1 = \text{quo}(-cp_0^{\delta_0}, \psi_0^{\delta_0-1})$ ;
    else
        $\psi_1 = -cp_0^{\delta_0} \cdot \psi_0$ ,;
    $\beta_1 = -cp_0 \cdot \psi_1^{\delta_1}$ ;
return $\text{mcd}\,(\text{cont}(A(x)),\ \text{cont}(B(x))) \cdot \text{pp}(r_0)$;
```

Algoritmo 1.66: Imagen módulo p de un número, en la representación simétrica de $\mathbb{Z}_p$.

```
Data: $m, p \in \mathbb{Z}$
Result: $m \pmod{p}$ en la representación simétrica de $\mathbb{Z}_p$.
$u = \mathrm{rem}(m, p)$ ;
if $u > p/2$ then
    $u = u - p$
return $u$ ;
```

Algoritmo 1.67: Representación ξ-ádica de γ.

```
Data: $\gamma,\ \xi \in \mathbb{Z}$
Result: $u_0, u_1, ..., u_d$ tal que $\gamma = u_0 + u_1\xi + ... + u_d\xi^d$ con $\xi^{d+1} > 2|\gamma|$
        y $-\xi/2 < u_i \leq \xi/2$ .
$e = \gamma$ ;
$i = 0$ ;
while $e \neq 0$ do
    $u_i = \phi_\xi(e)$ ;
    $e = (e - u_i)/\xi$ ;
    $i = i + 1$ ;
return $u_0, u_1, ..., u_d$ ;
```

Algoritmo 1.68: MCDHEU(A, B).

```
Data: $A, B \in \mathbb{Z}[x]$ ambos polinomios primitivos.
Result: $\operatorname{mcd}(A, B)$ si el resultado de la búsqueda heurística da
        resultado, sino retorna $-1$
if $grado\ A = grado\ B = 0$ then
    return $\gamma = \operatorname{mcd}(A, B) \in \mathbb{Z}$
$\xi = 2 \cdot \text{Mín}\{||A||_\infty, ||B||_\infty\} + 2$ ;
$i = 0$ ;
while $i < 7$ do
    if $length(\xi) \cdot máx\{grado\ A, grado\ B,\} > 5000$ then
        return $-1$ //$mcd \geq 0,\ -1$ se usa como indicador de fallo
    $\gamma = \text{MCDHEU}(\phi_{(x-\xi)}(A), \phi_{(x-\xi)}(B))$ //llamada recursiva;
    if $\gamma \neq -1$ then
        Generar $G$ a partir de la expansión $\xi-$ádica de $\gamma$
    //División en $\mathbb{Q}[x]$;
    if $G|A\ \ y\ \ G|B$ then
        return $\text{pp}(G)$
    Crear un nuevo punto de evaluación;
    $\xi = \text{quo}(\xi \times 73794,\ 27011)$
return $-1$ ;
```

Algoritmo 1.69: Factorización Libre de Cuadrados (Algoritmo de Yun)

```
Data: $A(x) \in D[x]$, $A(x)$ primitivo, $D$ DFU de característica cero.
Result: Factorización Libre de Cuadrados de $A(x)$.
$i = 1$; $salida = 1$;
$B(x) = A'(x)$, $C(x) = \text{mcd}\,(A(x),\ B(x))$;
if $C(x) = 1$ then
    $W(x) = A(x)$
else
    $W(x) = A(x)/C(x)$;
    $Y(x) = B(x)/C(x)$;
    $Z(x) = Y(x) - W'(x)$;
    while $Z(x) \neq 0$ do
        $G(x) = \text{mcd}\,(W(x),\ Z(x))$;
        $salida = salida \cdot G(x)^i$;
        $i = i + 1$;
        $W(x) = W(x)/G(x)$;
        $Y(x) = Z(x)/G(x)$;
        $Z(x) = Y(x) - W'(x)$;
$salida = salida \cdot W(x)^i$;
return $salida$
```

Algoritmo 1.70: SquareFreeCF: Factorización Libre de Cuadrados en un Campo Finito.

```
Data: Polinomio mónico $A(x) \in GF(p^r)$, $p$ primo.
Result: Factorización Libre de Cuadrados de $A(x)$
$i = 1$, $salida = 1$, $B(x) = A'(x)$;
if $B(x) \neq 0$ then
    $C(x) = \operatorname{mcd}(A(x), B(x))$;
    $W(x) = A(x)/C(x)$;
    while $W(x) \neq 1$ do
        $Y(x) = \operatorname{mcd}(W(x),\ C(x))$;
        $Z(x) = W(x)/Y(x)$;
        $salida = salida \cdot Z(x)^i$;
        $i = i + 1$;
        $W(x) = Y(x)$;
        $C(x) = C(x)/Y(x)$;
        if $C(x) \neq 1$ then
            $C(x) = C(x)^{1/p}$;
            $salida = salida \cdot (\text{SquareFreeCF}(C(x)))^p$
else
    $A(x) = A(x)^{1/p}$;
    $salida = (\text{SquareFreeCF}(A(x)))^p$
return $salida$
```

Algoritmo 1.71: SquareFreeCF: Factorización Libre de Cuadrados en un Campo Finito.

```
Data: Polinomio mónico $A(x) \in GF(p^r)$, $p$ primo.
Result: Factorización Libre de Cuadrados de $A(x)$
$i = 1$, $salida = 1$, $B(x) = A'(x)$;
if $B(x) \neq 0$ then
    $C(x) = \text{mcd}\,(A(x),\ B(x))$;
    $W(x) = A(x)/C(x)$;
    while $W(x) \neq 1$ do
        $Y(x) = \text{mcd}\,(W(x),\ C(x))$;
        $Z(x) = W(x)/Y(x)$;
        $salida = salida \cdot Z(x)^i$;
        $i = i + 1$;
        $W(x) = Y(x)$;
        $C(x) = C(x)/Y(x)$;
        if $C(x) \neq 1$ then
            $C(x) = C(x)^{1/p}$;
            $salida = salida \cdot (\text{SquareFreeCF}(C(x)))^p$
else
    $A(x) = A(x)^{1/p}$;
    $salida = (\text{SquareFreeCF}(A(x)))^p$
return $salida$
```

Algoritmo 1.74: Diferencias Divididas de Newton

```
Data: $\{(x_i, y_i)\}_{i=0,1,\ldots,n}$ con los $x_i$'s distintos.
Result: Coeficientes del polinomio interpolante: $F_{0,0}$, $F_{1,1}, \ldots, F_{n,n}$
for $i = 1$ to $n$ do
    $F_{i,0} = y_i$
;
for $i = 1$ to $n$ do
    for $j = 1$ to $i$ do
        $F_{i,j} = \dfrac{F_{i,j-1} - F_{i-1,j-1}}{x_i - x_{i-j}}$
return $F_{0,0}$, $F_{1,1}, \ldots, F_{n,n}$
```

Algoritmo 1.72: Método de Kronecker.

```
Data: $P(x) \in \mathbb{Z}[x]$ y $x_0 \in \mathbb{Z}$.
Result: Factor no trivial $A(x)$ de $P$ de grado $\leq [\![\text{grado}P/2]\!]$ (si existe).
$m = [\![\text{grado}P/2]\!]$ ;
for $j = 0$ to $m$ do
    Calcule $P(x_0 + j)$ ;
    if $P(x_0 + j) = 0$ then
        $A(x) = x - x_0 - i$ ;
        $FactorEncontrado = true$ ;
        break;
    else
        Factorice $P(x_0 + j)$ y construya $D_j$ ;
if FactorEncontrado=false then
    for $s = 1$ to $m$ do
        foreach $(d_0, ..., d_s) \in D_0 \times ... \times D_s$ con $(d_0, ..., d_s) \neq (1, ..., 1)$ do
            Calcule el polinomio interpolante $\overline{A}(x)$ ;
            if $\overline{A}(x) \mid P(x)$ then
                $A(x) = \overline{A}(x)$ ;
                $FactorEncontrado = true$ ;
                break;
        if $FactorEncontrado = true$ then
            break;
return $A(x)$ ;
```

Algoritmo 1.73: Forma de Lagrange del Polinomio Interpolante.

```
Data: $(x_0, y_0), ..., (x_m, y_m)$
Result: Coeficientes $a_0, a_1, ..., a_m$ del polinomio interpolante.
$a_0 = y_0$;
$s = a_j - a_0$;
$f = x_j - x_0$;
for $j = 1$ to $m$ do
    $s = y_j - a_0$;  $f = x_j - x_0$;
    for $k = 1$ to $j - 1$ do
        $s = s - a_k \cdot f$;
        $f = (x_j - x_k) \cdot f$;
    return $a_j = s/f$ ;
```

Algoritmo 1.75: Interpolación Cuadrática Inversa.

```
**Data**: una función $f$ y las aproximaciones $x_0, x_1, x_2$
**Result**: una aproximación $p_3$ a un cero de $f$.
Parámetros;
Máximo número de iteraciones;
$\delta$ = Tolerancia.;
Defina $k = 2$;
$p_0 = x_0,\, p_1 = x_1,\, p_2 = x_2.$ ;
$p_3 = p_2,\, p_2 = p_1,\, p_1 = p_0.$ ;
while $|p_3 - p_2| < \delta \quad o \quad k < N$ do
    $p_0 = p_1,\, p_1 = p_2,\, p_2 = p_3.$ ;
    $f_0 = f(p_0),\, f_1 = f(p_1),\, f_2 = p(p_2)$ ;
    $R = \frac{f_1}{f_2},\; S = \frac{f_1}{f_0},\; T = \frac{f_0}{f_2},$ ;
    $P = S(T(R-T)(p_2 - p_1) - (1-R)(p_1 - p_0))$ ;
    $Q = (T-1)(R-1)(S-1)$ ;
    $p_3 = p_1 + \frac{P}{Q}$ ;
    $k = k + 1$ ;
end
return $p_3$ ;
return Estimación del error $|p_3 - p_2|.$ ;
```

Algoritmo 1.76: Pesos Baricéntricos

```
Data: $n+1$ nodos distintos $\{x_i\}_{i=0,1,...,n}$ .
Result: Pesos baricéntricos $\omega_{k}^{(n)},\ k=0,1,...,n$
if $\{x_i\}_{i=0,...,n}$ son nodos de TChebyshev then
    $w_{k}^{(n)} = (-1)^k \sin\dfrac{(2k+1)\pi}{2n+2},\ k=0,...,n$
else
    $w_{0}^{(0)} = 1$ ;
    for $j=1$ to $n$ do
        for $k=0$ to $j-1$ do
            $w_{k}^{(j)} = (x_k - x_j) w_{k}^{(j-1)}$
        $w_{j}^{(j)} = \prod_{k=0}^{j-1}(x_j - x_k)$ ;
    for $k=0$ to $n$ do
        $w_{k}^{(n)} = 1/w_{k}^{(n)}$
return $w_{0}^{(n)}, w_{1}^{(n)}, ..., w_{n}^{(n)}$
```

esp

Algoritmo 1.77: Iteración de Punto fijo.

```
Data: Una función continua $g$, $x_0$, $\delta$, maxItr.
Result: Si hay convergencia, una aproximación $x_1$ de un punto fijo.
$k = 0$;
repeat
    $x_1 = g(x_0)$;
    $dx = |x_1 - x_0|$;
    $x_0 = x_1$;
    $k = k+1$;
until $dx \leq \delta(|x_0|+1)$ o $k >$ maxItr ;
return $x_1$
```

Algoritmo 1.78: Algoritmo de Bisección.

```
Data: $a$, $b$, $\delta$ y $f$ continua en [a,b] con $f(a)f(b) < 0$.
Result: Una aproximación $m$ de un cero $x^*$ de $f$ en $]a, b[$.
$k = 0$;
repeat
    $m = a + 0{,}5(b - a)$;
    $dx = (b - a)/2$;
    if $Sgn(f(a)) <> Sgn(f(m))$ then
        $b = m$;
    else
        $a = m$
    $k = k + 1$
until $dx \leq \delta$ or $f(m) = 0$ ;
return $m$
```

Algoritmo 1.79: Método de Newton

```
Data: $f \in C[a, b]$, $x_0$, $\delta$, maxItr.
Result: Si la iteración converge, una aproximación $x_n$ de un cero de $f$
        en $[a, b]$ y una estimación del error.
$k = 0$ ;
$x_k = x_0$;
repeat
    $dx = \dfrac{f(x_k)}{f'(x_k)}$;
    $x_{k+1} = x_k - dx$;
    $x_k = x_{k+1}$;
    $k = k + 1$;
until $dx \leq \delta\ (|x_{k+1}| + 1)$ or $k \leq maxItr$ ;
return $x_k$ y $dx$
```

Algoritmo 1.80: Método de falsa posición

```
Data: $f \in C[a,b]$ con $f(a)f(b) < 0$ , $\delta$ , maxItr
Result: Una aproximación $x_n$ de un cero de $f$.
k = 0 ;
$a_n = a,\ b_n = b.$ ;
repeat
    $x_n = \dfrac{a_n - b_n}{f(a_n) - f(b_n)} f(a_n)$ ;
    if $f(x_n)f(a_n) > 0$ then
        $a_{n+1} = x_n$ ;
        $b_{n+1} = b_n$
    else
        $a_{n+1} = a_n$ ;
        $b_{n+1} = x_n$
    $k = k + 1$ ;
until ( $x_n - a_n \leq \delta \ \vee\ b_n - x_n \leq \delta \ \vee\ k \geq maxItr$ ) ;
return $x_k$
```

Algoritmo 1.81: Método de la secante.

```
Data: Una función continua $f$, las aproximaciones iniciales $x_0$ y $x_1$ , $\delta$
      y maxItr
Result: Si la iteración converge a un cero, una aproximación $x_k$ del cero.
$j = 0$ ;
$x_{k-1} = x_0$ ;
$x_k = x_1$ ;
while $|x_k - x_{k-1}| > \delta\ (|x_k| + 1)$ y $j < Nmax$ do
    $x_{k+1} = x_k - f(x_k) \cdot \dfrac{(x_k - x_{k-1})}{f(x_k) - f(x_{k-1})}$ ;
    $x_{k-1} = x_k$ ;
    $x_k = x_{k+1}$ ;
    $j = j + 1$ ;
return $x_{k+1}$ ;
```

Algoritmo 1.82: Híbrido secante-bisección.

```
Data: una función continua $f$ y $a$, $b$ tal que $f(a)f(b) < 0$ y $\delta$
Result: una aproximación del cero.
$c = a$;
$n = 0$;
repeat
    if $f(a)f(b) < 0$ then
        Aplicar $SecanteItr(a, b)$
    else
        if TestSecante =true then
            Aplicar $SecanteItr(a, b)$
        else
            Aplicar bisección $b = b - 0{,}5(c - b)$;
    Intervalo para bisección;
    if $f(a)f(b) < 0$ then
        $c = a$
    $n = n + 1$
until ( $|c - b| < \delta(|b| + 1)$ ) Or ( $n >$ maxItr Or $f(b) < \delta$ ) ;
return $b$;
```

Algoritmo 1.83: Híbrido Newton-bisección.

```
Data: Una función continua $f$, las aproximaciones iniciales $a$ y $b$ y $\delta$
Result: Una aproximación $x_k$ del cero.
$k = 0, x_0 = a$ ;
repeat
    test1 =
    $(f'(x_0) > 0 \wedge (a - x_0) f'(x_0) < -1 \cdot f(x_0) \wedge (b - x_0) \cdot f'(x_0) > -1 \cdot f(x_0))$
    test2 =
    $(f'(x_0) < 0 \wedge (a - x_0) f'(x_0) > -1 \cdot f(x_0) \wedge (b - x_0) \cdot f'(x_0) < -1 \cdot f(x_0))$
    if test1 Or test2 Or $f(x_0) = 0$ then
        $x_1 = x_0 - f(x_0)/f'(x_0)$;
        $dx = |x_1 - x_0|$;
        $x_0 = x_1$;
        if $Sgn(f(a)) <> Sgn(f(x_1))$ then
            $b = x_1$
        else
            $a = x_1$
    else
        $x1 = a + 0{,}5 * (b - a)$;
        $dx = (b - a)/2$;
        $x0 = x1$;
        if $Sgn(f(a)) <> Sgn(f(x_1))$ then
            $b = x_1$
        else
            $a = x_1$
    $k = k + 1$;
until $dx < delta$ Or $k > maxItr$ ;
return $x_1$
```

Algoritmo 1.84: Estiramiento de Hensel univariado

```
Data: $a(x)$ primitivo, un primo $p$ que no divida $a_n$, dos polinomios
      primos relativos $u(x)$, $w(x)$ en $\mathbb{Z}_p[x]$ tales que
      $a(x) = u(x)w(x)\ (mod\, p)$, una cota $B$ de los coeficientes de $a(x)$ y
      cualquiera de sus posibles factores con grados que no excedan
      $\max\{deg(u(x)), deg(w(x))\}$. Opcionalmente un entero $\gamma \in \mathbb{Z}$
      múltiplo de lcoef($uf(x)$) donde $uf(x)$ es uno de los factores de
      $a(x)$ en $\mathbb{Z}[x]$ que se debe calcular
Result: Si existen polinomios $uf(x)$ y $wf(x)$ tales que
        $a(x) = uf(x)wf(x) \in \mathbb{Z}[x]$ y $n(uf(x)) \equiv n(u(x))(mod\, p))$,
        $n(wf(x)) \equiv n(w(x))(mod\, p))$, entonces $uf$ y $wf$ serán
        calculado. Sino, "No factoriza"
Definir un polinomio y sus factores módulo $p$;
$a = lcoeff(a(x))$;
if $\gamma$ indefinido then
    $\gamma = \alpha$
;
$a(x) = \gamma a(x)$;
$u(x) = \phi_p(\gamma n(u(x))$;
$w(x) = \phi_p(\alpha\, n(w(x)))$;
Aplicamos Algoritmo Extendido de Euclides a $u(x)$, $w(x) \in Z_p[x]$;
```

Algoritmo 1.85: Factorización en un campo finito, Berkelmap

```
Data: $a(x)$ primitivo, un primo $p$ que no divida $a_n$
Result: Si existen polinomios $uf(x)$ y $wf(x)$ tales que
        $a(x) = uf(x)wf(x) \in \mathbb{Z}[x]$ y $n(uf(x)) \equiv n(u(x))(mod\, p))$,
        $n(wf(x)) \equiv n(w(x))(mod\, p))$, entonces $uf$ y $wf$ serán
        calculado. Sino, "No factoriza"
```

¿Cómo evaluar expresiones matemáticas en el computador?

Alexander Borbón A.
Escuela de Matemática
Instituto Tecnológico de Costa Rica

Fecha de recibido: Octubre, 2005. Fecha de aceptación: Abril, 2006.

Resumen

En este artículo se muestra una forma de programar un evaluador de expresiones matemáticas en JAVA. El programa se construye paso a paso y se explican detalladamente las partes más importantes del mismo. El evaluador consta de dos partes o módulos, el primero se encarga de convertir la expresión digitada a notación postfija que es más sencilla para el computador; el segundo es el que evalúa la expresión que se obtuvo en un valor específico. Para poder comprender y reescribir este programa se necesita tener conocimientos básicos en la programación en JAVA, sin embargo, se explicará el uso de varias primitivas utilizadas y de algunos conceptos básicos de programación.

Palabras Clave: Programación, JAVA, funciones, expresiones matemáticas, software matemático.

Notación Postfija

El primer módulo del programa se encargará de traducir la expresión que digita el usuario (en notación infija) en una expresión más simple, esta segunda expresión se dice que está en notación postfija; esta forma de escribir una expresión matemática tiene la ventaja que se evalúa de forma lineal por lo que es más sencillo para una computadora "entenderlo".

Al inicio talvez se ve un poco complejo, pero no es así; lo que hace el lenguaje es colocar primero los números con los que va a operar y luego escribe la operación, por ejemplo $a+b$ se escribiría "a b +";

$(5-8)*4$ se escribiría "5 8 - 4 *", otros ejemplos son:

Notación infija	Notación postfija
$\text{sen}(x)$	x sen
$1+3*4$	1 3 4 * +
$\text{sen}(9+x)$	9 x + sen

Notación infija	Notación postfija
$3*6$	3 6 *
$4-\text{sen}(x)$	4 x sen -
$\text{sen}(x^2)+4*x$	x 2 ^ sen 4 x * +

Para evaluar una expresión en notación infija en un valor específico para x se deben seguir las reglas matemáticas para la prioridad de las operaciones:

1. Las potencias tienen prioridad sobre cualquier operación

2. La multiplicación y la división tienen prioridad sobre la suma y la resta.
3. Si se presenta un paréntesis, se deben realizar primero las operaciones dentro de éste. Si hay un paréntesis dentro de otro tiene prioridad el paréntesis interno.

Por el contrario, en la notación postfija siempre se trabaja de izquierda a derecha; note además que la notación postfija no tiene paréntesis por la forma lineal en que se lee. Comparemos un ejemplo en donde se evalúa un valor en una expresión utilizando estas dos notaciones.

Evaluación en notación infija y postfija

Tomemos un ejemplo sencillo como $4+x^3$ cuando $x=2$, esta expresión se escribe en la notaciones anteriores de la siguiente manera:

Notación infija: 4 + x ^ 3

Notación postfija: 4 x 3 ^ +

Para evaluar $4+x^3$ en notación infija primero se debe tomar el 2 (en vez de la x), elevarlo al cubo y luego sumarle 4; se nota que esto no está en forma lineal, es decir, se debe ir primero a la segunda parte de la expresión, evaluarla y luego sumarle al resultado el cuatro.

Es muy complejo hacer que un programa evalue de esta forma. Si la expresión "$4+x^3$" se traduce a la forma "4 x 3 ^ +", entonces se convierte en una expresión más sencilla; para evaluar esta expresión en $x=2$, el algoritmo es leer el texto de izquierda a derecha, si se encuentra un número lo apila[4] y las operaciones se irán realizando conforme aparezcan.

Así, en el ejemplo "4 x 3 ^ +" se toma el 4 y se mete en una pila de números, el segundo valor que se toma es la x, en vez de ésta introducimos el 2 (que es el valor que estamos evaluando), luego se toma el 3 y se mete nuevamente en la pila de números, es decir, tenemos una pila de números como sigue

3
2
4

Ahora sigue una potencia, por lo que se toman los dos últimos números de la pila (recuerde que tomamos los dos de arriba: 2 y 3) y se realiza la potencia $2^3=8$, así, el 2 y el 3 se sustituyen en la pila por 8, se obtiene la pila

8
4

Por último, sigue un signo de suma, éste también se realiza con dos números, por lo que se toman el 4 y el 8, el resultado es $4+8=12$ y en la pila queda un 12; como ya se acabó la expresión entonces el resultado es 12.

Aunque a primera vista esta forma de evaluar parece más compleja, para un programa no lo es, ya que se siguen los pasos en forma lineal (la expresión se lee de izquierda a derecha), la algoritmo estaría compuesto por tres reglas:

1. Si lo que sigue en la expresión es un número, se agrega a la pila de números.
2. Si sigue una operación que ocupa dos números (como la suma y la resta) se sacan los dos últimos números de la pila, se realiza la operación y se introduce el resultado en la pila.

3. Si es una función que ocupa un solo número (como seno o coseno) entonces se saca un número de la pila, se evalúa y se guarda el resultado.

Con este ejemplo se observa que evaluar una expresión en notación postfija es más sencillo que hacerlo en notación infija, por lo que lo primero que se hará el programa es "traducir" la expresión que digite el usuario a dicha notación. Veamos primero la teoría sobre cómo se hará esto.

Traducción a notación postfija

Para la traducción a notación postfija se utilizarán dos pilas, una en donde se guardarán los números y otra para los operadores y los paréntesis[5]. Aunque se dice que una de las pilas maneja números, esto no es muy cierto, en realidad las dos pilas se trabajarán con texto (*String*), esto permite concatenar varias tiras fácilmente y se pueden manejar expresiones que no son números como si lo fueran.

Para la traducción del lenguaje natural a notación postfija es fundamental manejar la prioridad de las operaciones y el uso de paréntesis.

Como se dijo al inicio del artículo, para evaluar una expresión matemática se deben seguir las siguientes reglas:

1. Primero se evalúan las potencias.
2. A continuación siguen las multiplicaciones, las divisiones y el resto de la división entera (%).
3. Por último se realizan las sumas y las restas.
4. Si hay paréntesis, se hace primero la expresión que está dentro del paréntesis interno.

Por esto, para la prioridad se les dará a las operaciones los siguientes valores:

Operador	Prioridad
+, -	0
*, /, %	1
^	2

Por lo que en nuestro programa ocupamos una función como la que sigue:

```
private int prioridad(char s) {
   if (s=='+' || s=='-')
      return 0;
   else if (s=='*' || s=='/' || s=='%')
      return 1;
   else if (s=='^')
      return 2;

   return -1;
} //Fin de la funcion prioridad
```

Esta función recibe un caracter *s*, dependiendo de la operación que represente este caracter se devuelve el valor de la prioridad correspondiente, si recibe un operador que no corresponde con las operaciones se devuelve -1.

Otro punto importante es que todas las funciones que reciben un parámetro (seno, coseno, tangente,...) se manejan como un paréntesis que abre "(", el usuario debe digitarlas con ese paréntesis, es decir: "sen(", "cos(", "tan(", ... Se manejan así porque cuando se digita el paréntesis que cierra ")", este paréntesis saca todo lo que hay en la pila hasta que encuentra el de apertura o una función (lo que quiere decir que en términos prácticos funcionan igual con una ligera diferencia que luego se verá).

A continuación se muestran varios ejemplos para observar el algoritmo que se debe seguir para traducir una expresión de notación infija a postfija.

Una expresión simple

Como primer ejemplo se va a traducir una expresión simple como 3*x+4, la prioridad indica que primero se tiene que hacer la multiplicación y luego la suma; para esto, recuerde que la función le asigna prioridad 0 a la suma y 1 a la multiplicación y se dará la regla que una prioridad inferior saca de la pila cualquier operación con prioridad igual o superior a ella.

Dado esto, se pueden notar que:

1. Los operadores "+" y "-" son los que tienen menor prioridad (0), por lo que siempre sacarán todos los operadores precedentes.
2. Los operadores "*", "/" y "%" sacan a la potencia "^" y a ellos mismos.
3. El operador "^" es el que tiene mayor prioridad, no saca a nadie, ni siquiera a sí mismo ya que la prioridad para realizar las potencias es de derecha a izquierda (contrario a todas las demás operaciones matemáticas), es decir: $2^{3^4} = 2^{(3^4)}$ o de manera escrita 2^3^4=2^(3^4)

Por lo tanto, se van a sacar los elementos de la expresión 3*x+4; primero se toma el 3 y se introduce en la pila de números, luego se toma el * y se mete en la pila de operadores, se obtiene

Números
3

Operadores
*

Luego se saca la x, ésta se debe manejar como un número pues cuando se evalúe así será, esta se mete en la pila de números

Números
x
3

Operadores
*

Ahora se debe sacar un + que se tendría que introducir en la pila de operadores, pero tiene prioridad cero que es menor que la prioridad de la multiplicación (que es 1), por lo que la multiplicación se debe hacer antes que la suma (la multiplicación tienen prioridad sobre la suma). Para este caso se sacan dos números de la pila y los "multiplicamos" obteniendo "3 x *" (recuerde que en realidad manejamos texto, cuando se dice que lo multiplicamos se quiere decir que se escribe la multiplicación en notación postfija). El resultado se introduce en la pila como un número (ahora se maneja como si toda esta expresión fuera un número porque es como si ya se hubiera evaluado) y la suma se mete en la pila de operadores, se obtienen las pilas

Números
3 x *

Operadores
+

Ahora se saca el número 4 y se mete en la pila de números

Números
4
3 x *

Operadores
+

Aquí ya se acaba la expresión por lo que se saca todo lo que queda en la pila. Para cada operador se deben tomar los valores necesarios de la pila de números; en este caso sólo hay una suma y dos números, al realizar la operación se sacan los dos números y se escribe en postfijo "3 x * 4 +" pues se colocan los dos números primero y luego la operación.

Una expresión con funciones

Se tomará como segundo ejemplo una expresión con más elementos, por ejemplo:

$$1+3*\tan(2*(1+x)-1)$$

En este caso se toma el 1, el + y el 3

Números
3
1

Operadores
+

Sigue la multiplicación, pero esta sólo saca la potencia y las de su mismo nivel por lo que no saca la suma, simplemente se agrega a los operadores

Números
3
1

Operadores
*
+

Sigue la función tangente que se maneja dentro de los operadores como un paréntesis que abre (en la pila le quitamos el paréntesis que en postfijo no se necesita)

Números
3
1

Operadores
tan
*
+

Ahora se deben incluir el 2 y el *. A la hora de incluir *, tangente no funciona como un operador sino como el inicio de otra expresión por lo que * no lo puede sacar; aquí no hay problema con la función prioridad porque a tan le asignaría un -1 que no lo saca nadie. También se agrega el paréntesis de apertura

Números
2
3
1

Operadores
(
*
tan
*
+

Ahora incluimos el 1, el + y la x

Números
x
1
2
3
1

Operadores
+
(
*
tan
*
+

Viene el cierre de paréntesis, este sacaría todos los elementos hasta que haya una apertura (un paréntesis o una función). En este caso solo debe sacar el + y se quita el paréntesis de la pila.

Números
1 x +
2
3
1

Operadores
*
tan
*
+

Sigue un - que tiene menor prioridad que el *, por lo que tenemos que sacar la multiplicación antes de meter el menos.

Números
2 1 x + *
3
1

Operadores
-
tan
*
+

Sigue un uno y luego un cierre de paréntesis que sacaría hasta tangente. En este caso, se debe sacar el menos que queda; la tangente, contrario al paréntesis de apertura, se debe agregar al final del texto para que se evalúe en la expresión.

Números
2 1 x + * 1 - tan
3
1

Operadores
*
+

Aquí se acaba la expresión por lo que se debe sacar todo lo que queda, obteniendo como resultado "1 3 2 1 x + * 1 - tan * +"

Se tomará como segundo ejemplo una expresión con más elementos, por ejemplo:

$$1+3*\tan(2*(1+x)-1)$$

En este caso se toma el 1, el + y el 3

Números
3
1

Operadores
+

Sigue la multiplicación, pero esta sólo saca la potencia y las de su mismo nivel por lo que no saca la suma, simplemente se agrega a los operadores

Números
3
1

Operadores
*
+

Sigue la función tangente que se maneja dentro de los operadores como un paréntesis que abre (en la pila le quitamos el paréntesis que en postfijo no se necesita)

Números
3
1

Operadores
tan
*
+

Ahora se deben incluir el 2 y el *. A la hora de incluir *, tangente no funciona como un operador sino como el inicio de otra expresión por lo que * no lo puede sacar; aquí no hay problema con la función prioridad porque a tan le asignaría un -1 que no lo saca nadie. También se agrega el paréntesis de apertura

Números
2
3
1

Operadores
(
*
tan
*
+

Ahora incluimos el 1, el + y la x

Números
x
1
2
3
1

Operadores
+
(
*
tan
*
+

Viene el cierre de paréntesis, este sacaría todos los elementos hasta que haya una apertura (un paréntesis o una función). En este caso solo debe sacar el + y se quita el paréntesis de la pila.

Números
1 x +
2
3
1

Operadores
*
tan
*
+

Sigue un - que tiene menor prioridad que el *, por lo que tenemos que sacar la multiplicación antes de meter el menos.

Números
2 1 x + *
3
1

Operadores
-
tan
*
+

Sigue un uno y luego un cierre de paréntesis que sacaría hasta tangente. En este caso, se debe sacar el menos que queda; la tangente, contrario al paréntesis de apertura, se debe agregar al final del texto para que se evalúe en la expresión.

Números
2 1 x + * 1 - tan
3
1

Operadores
*
+

Aquí se acaba la expresión por lo que se debe sacar todo lo que queda, obteniendo como resultado "1 3 2 1 x + * 1 - tan * +"

Programa que traduce de notacion infija a postfija

Recuerde que para esta parte del programa se va a suponer que la expresión que digita el usuario no tiene errores, luego se darán consejos para detectarlos.

En los ejemplos de la sección anterior se observó que para realizar la traducción de la expresión se necesitan dos pilas de *Strings* (texto), una para los números y otra para los operadores. JAVA ya posee una clase que maneja pilas, esta clase se llama *Stack* que se encuentra dentro del paquete *java.util*, lo primero que tenemos que hacer es llamar a esta librería y hacer nuestra clase *Parseador*[6]; el inicio de nuestro programa se debe ver como:

```
import java.util.*;

public class Parseador{
...
}//fin de Parseador
```

Para crear una nueva pila se debe definir como un nuevo objeto:

```
Stack nuevaPila = new Stack();
```

La clase *Stack* se maneja con objetos (introduce objetos y saca objetos); para introducir un nuevo objeto dentro de la pila se utiliza la instrucción

```
nuevaPila.push(Objeto);
```

Para sacar un objeto de la pila (recuerde que una pila saca el último objeto que se introdujo) utilizamos

```
nuevaPila.pop();
```

Para "mirar" un objeto de la pila sin sacarlo se usa

```
nuevaPila.peek()
```

Además se puede preguntar si la pila está vacía con la instrucción

```
nuevaPila.empty()
```

que devuelve *True* si está vacía o *False* si no lo está.

Para empezar con la clase *Parseador*, definimos la variable global *ultimaParseada* como sigue:

```
private String ultimaParseada;
```

Esta guarda un registro de la última expresión parseada (o traducida) en notación postfija, la expresión se guarda por si alguien quiere evaluar sin tener que dar la expresión en donde se evalúa.

El constructor de nuestra clase lo único que hace es poner *ultimaParseada* en 0.

```
public Parseador(){
   ultimaParseada="0";
}
```

La función parsear se define de tal forma que recibe un texto con la expresión en notación infija y devuelve otro texto con la expresión en notación postfija. La función lanza una excepción (*SintaxException*) si encuentra que la expresión está mal digitada.

```
public String parsear(String expresion) throws SintaxException{
   Stack PilaNumeros=new Stack(); //Pila de números
   Stack PilaOperadores= new Stack(); //Pila de operadores
   String fragmento;
   int pos=0, tamano=0;
   byte cont=1;
   final String funciones[]={"1 2 3 4 5 6 7 8 9 0 ( ) x e + - * / ^ %",
         "pi",
         "ln(",
         "log( abs( sen( sin( cos( tan( sec( csc( cot( sgn(",
         "rnd() asen( asin( acos( atan( asec( acsc( acot( senh( sinh( cosh( tanh(
              sech( csch( coth( sqrt(",
         "round( asenh( acosh( atanh( asech( acsch( acoth("};
   final private String parentesis="( ln log abs sen sin cos tan sec csc cot asen
asin
              acos atan asec acsc acot senh sinh cosh tanh sech csch coth sqrt
round";
   final private String operadoresBinarios="+ - * / ^ %";

   //La expresión sin espacios ni mayúsculas.
   String expr=quitaEspacios(expresion.toLowerCase());
```

La función necesita dos pilas: *PilaNumeros* y *PilaOperadores*.

La variable *fragmento* se encargará de guardar el fragmento de texto (*String*) que se esté utilizando en el momento (ya sea un número, un operador, una función, etc.). La variable *pos* va a marcar la posición del caracter que se está procesando actualmente en el *String*.

La variable *funciones* contiene un arreglo de textos (*String*), en la primera posición tiene todas las expresiones de un caracter que se aceptarán, en la segunda posición están las expresiones de dos caracteres y así hasta llegar a la posición seis.

La variable *parentesis* contiene a todas las expresiones que funcionarán como paréntesis de apertura.

En *operadoresBinarios* se guardan todos los operadores que reciben dos parámetros.

Todas estas definiciones se hacen como texto (*String*) para después poder comparar si la expresión que el usuario digitó concuerda con alguno de ellos.

Se define también el *String* en donde se guarda la expresión sin espacios ni mayúsculas (*expr*); la función *toLowerCase()* ya está implementada en JAVA en la clase *String* mientras que *quitaEspacios* es una función que se define de la siguiente manera

```
private String quitaEspacios(String expresion){
   String unspacedString = "";  //Variable donde guarda la función

   //Le quita los espacios a la expresión que leyó
   for(int i = 0; i < expresion.length(); i++){
      if(expresion.charAt(i) != ' ')
         unspacedString += expresion.charAt(i);
   }//for

   return unspacedString;
}//quitaEspacios
```

Esta función va tomando cada uno de los caracteres de la expresión, si el caracter no es un espacio lo agrega al texto sin espacios *unspacedString*.

La excepción que lanza el programa también se define como una clase privada

```
private class SintaxException extends ArithmeticException{
   public SintaxException(){
      super("Error de sintaxis en el polinomio");
   }

   public SintaxException(String e){
      super(e);
   }
}
```

Volviendo a la función parsear, lo que seguiría es realizar un ciclo mientras no se acabe la expresión, es decir

```
   try{
      while(pos<expr.length()){
```

Lo que se haría dentro del *while* es ver si lo que sigue en la expresión es algo válido y tomar decisiones dependiendo si es un número, un operador, un paréntesis o una función.

El código:

```
   tamano=0;
   cont=1;
   while (tamano==0 && cont<=6){
      if(pos+cont<=expr.length() &&
           funciones[cont-1].indexOf(expr.substring(pos, pos+cont))!=-1){
         tamano=cont;
      }
      cont++;
   }
```

Hace que el contador vaya de 1 a 6 que es la máxima cantidad de caracteres que tiene una función y se inicializa *tamano* en cero. Luego se pregunta si la posición actual (*pos*) más el contador (*cont*) es menor de la longitud del texto y si el fragmento de texto que sigue está en alguna de las funciones, si esto pasa el siguiente texto que se tiene que procesar es de tamaño *cont*.

Ahora se van tomando algunos casos con respecto al tamaño encontrado.

```
if (tamano==0){
   ultimaParseada="0";
   throw new SintaxException("Error en la expresión");
```

Si *tamano* continúa siendo cero quiere decir que el fragmento de texto que sigue no coincidió con ninguna función ni con algo válido por lo que se lanza una excepción y se pone la última expresión parseada en 0.

```
}else if(tamano==1){
```

Pero si el tamaño es uno tenemos varias opciones, la primera es que sea un número

```
if(isNum(expr.substring(pos,pos+tamano))){
   fragmento="";
do{
   fragmento=fragmento+expr.charAt(pos);
   pos++;
}while(pos<expr.length() && (isNum(expr.substring(pos,pos+tamano)) ||
      expr.charAt(pos) == '.' || expr.charAt(pos) == ','));
   try{
      Double.parseDouble(fragmento);
   }catch(NumberFormatException e){
      ultimaParseada="0";
      throw new SintaxException("Número mal digitado");
   }
   PilaNumeros.push(new String(fragmento));
   pos--;
```

En la primera línea, para preguntar si el caracter es un número se utiliza la función *isNum* definida por

```
private boolean isNum(char s) {
   if (s >= '0' && (s <= '9'))
      return true;
   else
      return false;
} //Fin de la funcion isNum
```

Que devuelve *True* si el caracter que recibe es un número (está entre 0 y 9) o *False* si no lo es.

Si el caracter actual es un número existe un problema, no se sabe cuántos dígitos tenga este número, por lo que se sacan todos los caracteres que sigan que sean números, puntos o comas.

En la variable *fragmento* se va a guardar el número, por lo que al inicio se debe vaciar (*fragmento=" "*). Luego se hace un ciclo mientras no se acabe la expresión y el caracter que sigue sea un número, un punto o una coma: *while(pos < expr.length() && (isNum(expr.substring(pos,pos+tamano))* ||

expr.charAt(pos) == ‘.' || *expr.charAt(pos) == ‘,')){* .

Luego, la variable *fragmento* se trata de convertir a *double* y se mete en la pila de números; si no la puede convertir el programa lanza una excepción en el bloque *try-catch*.

De esta manera se maneja un número cualquiera. Ahora la segunda posibilidad con tamaño uno es que el caracter sea x o e. Estos dos casos se manejan como un número que se mete en la pila.

```
   else if (expr.charAt(pos)=='x' || expr.charAt(pos)=='e'){
      PilaNumeros.push(expr.substring(pos,pos+tamano));
```

Si el caracter que sigue es uno de los operadores +, *, / y % (el menos ‘-' se maneja igual, pero se recomienda ponerlo en un caso aparte para manejar posteriormente los menos unarios) se deben sacar todos los operadores con prioridad mayor o igual a ellos y meter el operador en la pila

```
   }else if (expr.charAt(pos)=='+' || expr.charAt(pos)=='*' ||
         expr.charAt(pos)=='/' || expr.charAt(pos)=='%'){
      sacaOperadores(PilaNumeros, PilaOperadores, expr.substring(pos,pos+tamano));
```

En este caso se declara una función (ya que se utilizará mucho al manejar detalles posteriores) que realice el trabajo de sacar operadores de la pila, esta función se define de la siguiente manera

```
private void sacaOperadores(Stack PilaNumeros, Stack PilaOperadores, String
operador){
   final String parentesis="( ln log abs sen sin cos tan sec csc cot sgn asen asin
         acos atan asec acsc acot senh sinh cosh tanh sech csch coth sqrt round
asenh
         asinh acosh atanh asech acsch acoth";
   while(!PilaOperadores.empty() &&
         parentesis.indexOf((String)PilaOperadores.peek( ))==-1 &&
         ((String)PilaOperadores.peek()).length()==1 &&
         prioridad(((String)PilaOperadores.peek()).charAt(0))>=
         prioridad(operador.charAt(0))){
      sacaOperador(PilaNumeros, PilaOperadores);
   }
   PilaOperadores.push(operador);
}
```

Aquí se vuelve a declarar la variable *parentesis* para el *while*. En este *while* se indica que se deben sacar operadores mientras haya algo en la pila, lo que siga en la pila no sea un paréntesis, sea de un solo caracter y cuya prioridad sea mayor o igual a la prioridad del operador que se está procesando en ese momento; luego se guarda el operador en la pila correspondiente.

Observe que el *while* llama a *sacaOperador* que es una función definida posteriormente cuyo código es

```
private void sacaOperador(Stack Numeros, Stack operadores) throws
EmptyStackException{
   String operador, a, b;
   final String operadoresBinarios="+ - * / ^ %";
   try{
      operador=(String)operadores.pop();

      if(operadoresBinarios.indexOf(operador)!=-1){
         b=(String)Numeros.pop();
         a=(String)Numeros.pop();
         Numeros.push(new String(a+" "+b+" "+operador));
      }else{
         a=(String)Numeros.pop();
         Numeros.push(new String(a+" "+operador));
      }
   }catch(EmptyStackException e){
      throw e;
   }
}//sacaOperador
```

Esta función se encarga de sacar dos números de la pila correspondiente, esto si es un operador binario o un número si es unario; luego introduce el nuevo número en notación postfija en la pila. Esta función recibe las dos pilas y no devuelve nada, si se le acaban los elementos a alguna de las pilas se lanza una excepción. Esta es la última función externa que se va a necesitar para traducir la expresión.

Con la potencia (^) es más sencillo ya que este operador no saca a nadie, el código quedaría de la siguiente manera.

```
   }else if (expr.charAt(pos)=='^'){
      PilaOperadores.push(new String("^"));
```

En este caso, simplemente se mete el operador a su pila correspondiente.

Si el caracter fuera un paréntesis de apertura simplemente se mete en la pila de operadores.

```
   }else if (expr.charAt(pos)=='('){
      PilaOperadores.push(new String("("));
```

Y si el caracter es un paréntesis de cierre se deben sacar operadores hasta encontrar una apertura de paréntesis en la pila.

```
   }else if (expr.charAt(pos)==')'){
      while(!PilaOperadores.empty() &&
            parentesis.indexOf(((String) PilaOperadores.peek()))==-1){
         sacaOperador(PilaNumeros, PilaOperadores);
      }
      if(!((String)PilaOperadores.peek()).equals("(")){
         PilaNumeros.push(new String(((String)PilaNumeros.pop()) + " " +
               PilaOperadores.pop()));
      }else{
         PilaOperadores.pop();
      }
   }
```

Si el paréntesis de apertura no era el caracter "(" (es decir, era una función) entonces la concatena al final del texto en la notación postfija, si era un paréntesis simplemente lo desecha (recuerde que en notación postfija no hay paréntesis).

Aquí se terminan de procesar todos los elementos posibles de un solo caracter, ahora se pasará al caso de dos o más caracteres

```
   }else if(tamano>=2){
      fragmento=expr.substring(pos,pos+tamano);
      if(fragmento.equals("pi")){
         PilaNumeros.push(fragmento);
      }else if(fragmento.equals("rnd()")){
         PilaNumeros.push("rnd");
      }else{
         PilaOperadores.push(fragmento.substring(0,fragmento.length()-1));
      }
   }
   pos+=tamano;
}//Fin del while
```

En este caso se toma la expresión en *fragmento*, si *fragmento* es igual a "*pi*", lo metemos en la pila de números; lo mismo hacemos si es *rnd()*.

Cualquier otra posibilidad de tamaño mayor que dos es que sea una función por lo que se mete en la pila de operadores quitándole el paréntesis. Al final se aumenta la posición de acuerdo al tamaño del texto procesado para pasar al siguiente caracter en la expresión.

Ya cuando se acabó la expresión se debe procesar todo lo que quedó en las pilas en el proceso final, esto se hace con un while que saque todos los operadores que quedaron

```
      //Procesa al final
      while(!PilaOperadores.empty()){
         sacaOperador(PilaNumeros, PilaOperadores);
      }

   }catch(EmptyStackException e){
      ultimaParseada="0";
      throw new SintaxException("Expresión mal digitada");
   }

   ultimaParseada=((String)PilaNumeros.pop());

   if(!PilaNumeros.empty()){
      ultimaParseada="0";
      throw new SintaxException("Error en la expresión");
   }

   return ultimaParseada;
}//Parsear
```

Si hubo algún error con las pilas en el proceso se lanza una excepción y se pone a *ultimaParseada* en cero. Al final se devuelve *ultimaParseada*.

Con esto ya acabamos el programa que convierte una expresión matemática del lenguaje natural a la notación postfija.

Evaluación de expresiones postfijas

Ahora que se tiene el traductor de expresiones en notación postfija, lo que hace falta es el segundo módulo, es decir, el que evalúa una expresión en notación postfija en un número dado.

Al igual que para el módulo anterior, se va a iniciar mostrando varios ejemplos que realizan esta evaluación para observar cuál es el procedimiento que se debe seguir.

Evaluación en notación postfija: primer ejemplo

En este caso vamos a iniciar evaluando en la expresión "x 2 ^ sen 4 x * +" cuando $x = 3$. Recuerde que para evaluar en postfijo se leía la expresión de izquierda a derecha y se seguían tres reglas a saber:

1. Si lo que sigue en la expresión es un número, se agrega a la pila de números.
2. Si sigue una operación que ocupa dos números (como la suma y la resta) se sacan los dos últimos números de la pila, se realiza la operación y se introduce en la pila el resultado.
3. Si es una función que ocupa un solo número (como seno o coseno) entonces se saca un número de la pila, se evalúa y se guarda el resultado.

Por lo tanto en este caso se inicia tomando la x y se introduce un 3 en la pila (es el valor que se evalúa en vez de la x), luego se toma el 2 y también se introduce en la pila obteniendo

2
3

Luego sigue un ^ por lo que se toman los dos valores y se obtiene $3^2 = 9$, nos queda una pila de un elemento

9

Ahora sigue sen, esta función se realiza sobre un solo número (el único que hay en la pila), se obtiene $\text{sen}(9) = 0{,}412118485241757$, este es el nuevo valor en la pila

0.412118485241757

Sigue agregar a la pila un 4 y un 3 (en vez de la x)

3
4
0.412118485241757

Sigue una multiplicación, por lo que tomamos el 4 y el 3 de la pila y le metemos $4 \cdot 3 = 12$ a la pila, nos queda

12
0.412118485241757

Por último, queda una suma, es decir $0{,}412118485241757 + 12 = 12{,}412118485241757$ que es el resultado final.

Evaluación en notación postfija: segundo ejemplo

Un segundo ejemplo antes de programar el módulo será evaluar la expresión

$$\operatorname{asen}\left(\tan\left(\ln\left(x+\pi\right)\right)\right) + x^{x^2+3\cdot x+5}$$

que en notación postfija se escribe "x pi + ln tan asen x x 2 ^ 3 x * + 5 + ^ +" cuando $x = -1$.

Se toma la x, es decir, se introduce el -1 en la pila, luego pi

3.14159265358979
-1

Ahora sigue la suma de estos términos

2.14159265358979

Ahora se le saca logaritmo natural a este valor

0.761549782880893

Sigue el cálculo de la tangente

0.953405606022648

Ahora se calcula el arcoseno

1.26433002209559

Sigue introducir dos veces -1 (en vez de x) en la pila y un dos

2
-1
-1
1.26433002209559

Ahora sigue una exponente, $(-1)^2$ =1

1
-1
1.26433002209559

Se agrega un 3 y un -1

-1
3
1
-1
1.26433002209559

Se multiplica $3 \cdot -1 = -3$

-3
1
-1
1.26433002209559

Se suma $1 + -3 = -2$

-2
-1
1.26433002209559

Se agrega un 5

5
-2
-1
1.26433002209559

Se suma, $-2 + 5 = 3$

3
-1
1.26433002209559

Sigue un exponente $(-1)^3 = -1$

-1
1.26433002209559

Por último, se suman $1{,}26433002209559 + -1 = 0{,}26433002209559$, que es el resultado final.

Función que evalúa expresiones

Para hacer este módulo se inicia declarando la función para evaluar, la cual recibe la expresión en notación postfija y el número que se va a evaluar (que es un *double*); la función devuelve el resultado de la evaluación (que también es un *double*). En este caso la función se llamará *f* para simular que se evalúa en una función *f(x)*. Esta función lanzará una excepción (*ArithmeticException*) si encuentra un error en la expresión.

```
public double f(String expresionParseada, double x) throws ArithmeticException{
   Stack pilaEvaluar = new Stack(); //Pila de double's para evaluar
   double a, b;
   StringTokenizer tokens=new StringTokenizer(expresionParseada);
   String tokenActual;
   ...
```

En esta función se declaran algunas variables. En un principio se declara la pila que guardará los números (*pilaEvaluar*), luego se declaran dos números `a` y `b` en donde se guardarán los elementos que se sacan de la pila para ser operados.

El *StringTokenizer* es otra clase que se define en la librería *java.util*, ésta toma un texto y lo separa en unidades lexicográficas (llamadas lexemas o "tokens" en inglés), cada palabra separada por un espacio es un lexema; por ejemplo, el texto "2 3 + sen" tiene cuatro lexemas: "2", "3", "+" y "sen". El *tokenActual* guarda la unidad lexicográfica que se procesa en un momento dado.

Para saber si hay más lexemas en una expresión se utiliza *tokens.hasMoreTokens()* que devuelve *True* o *False* y para sacar el siguiente lexema se usa *tokens.nextToken()*. Así, lo que falta en nuestra función es ir pasando por los lexemas e ir haciendo cálculos o guardando números según corresponda, el código sería el siguiente

```
try{
   while(tokens.hasMoreTokens()){
      tokenActual=tokens.nextToken();
      ...
   }//while
}catch(EmptyStackException e){
   throw new ArithmeticException("Expresión mal parseada");
}catch(NumberFormatException e){
   throw new ArithmeticException("Expresión mal digitada");
}catch(ArithmeticException e){
   throw new ArithmeticException("Valor no real en la expresión");
}

a=((Double)pilaEvaluar.pop()).doubleValue();

if(!pilaEvaluar.empty())
   throw new ArithmeticException("Expresión mal digitada");

return a;

}//funcion f
```

Los bloques *try-catch-throw* son los que lanzan las excepciones si en algún momento se acabaron los elementos en la pila (*EmptyStackException*), si hubo algún número que no pudo entender (*NumberFormatException*) o si hubo algún cálculo que no corresponde a un número real (*ArithmeticException*).

El proceso de analizar lexemas se repetirá mientras hallan más lexemas *while(tokens.hasMoreTokens()){* y se saca el siguiente lexema con la instrucción *tokenActual=tokens.nextToken();*.

Al final del proceso, sacamos el último valor de la pila con la instrucción *a=((Double)pilaEvaluar.pop()). doubleValue();* lo que hace es sacar el elemento con *pilaEvaluar.pop()*, luego lo convierte a un objeto *Double* (recuerde que la pila trabaja con objetos) y, por último, calcula su valor double primitivo (con *doubleValue()*).

Si este no era el último valor en la pila, quiere decir que hubo un error al evaluar (faltaron operadores) y se lanza una excepción con el bloque

```
if(!pilaEvaluar.empty())
   throw new ArithmeticException("Expresión mal digitada");
```

Si no hubo ningún error entonces devuelve el valor encontrado.

Ahora veamos el bloque de código que hace falta dentro del *while*.

Si el lexema actual es alguno de los números "e", "pi" o "x", simplemente lo metemos en la pila, tomemos por ejemplo el número "e":

```
   if(tokenActual.equals("e")){
      pilaEvaluar.push(new Double(Math.E));
```

Al meter el número *Math.E* en la pila (este número es un *double*) se debe introducir como un objeto *Double*, por eso se utiliza el código *new Double*.

Cuando se tiene que meter "*x*" lo que se hace es introducir en la pila el valor que recibe la función. En todos los siguientes casos de este *if* utilizamos *elseif*; en este caso se utilizó *if* por ser el primero.

Por otro lado, si el lexema siguiente es un operador que recibe dos números, entonces primero se saca los dos números de la pila, y se guarda el resultado de la operación en la pila, tomando como ejemplo la suma.

```
   }else if(tokenActual.equals("+")){
      b=((Double)pilaEvaluar.pop()).doubleValue();
      a=((Double)pilaEvaluar.pop()).doubleValue();
      pilaEvaluar.push(new Double(a+b));
```

De igual manera se hace para funciones con un solo número, como logaritmo.

```
   }else if(tokenActual.equals("ln")){
      a=((Double)pilaEvaluar.pop()).doubleValue();
      pilaEvaluar.push(new Double(Math.log(a)));
```

Por último, si no fue nada de lo anterior, la única posibilidad que queda es que sea un número, este caso se toma el lexema y se trata de convertir en un *double* con *Double.parseDouble(tokenActual),* si no puede automáticamente se lanza la excepción; el número se convierte en un nuevo objeto *Double* y se mete en la pila. El código quedaría

```
   }else{
      pilaEvaluar.push(new Double(Double.parseDouble( tokenActual)));
   }
```

Agregando los casos correspondientes para cada una de las funciones que se admiten, ya se tiene un evaluador de expresiones en notación postfija funcional.

En el programa de ejemplo que se muestra al final del artículo, se admite lo siguiente:

- Números "especiales": e, pi, x.
- Operadores: +, -, *, /, %, ^.
- Funciones: ln, log, abs, sen, sin, cos, tan, sec, csc, cot, sgn, asen, asin, acos, atan, asec, acsc, acot, senh, sinh, cosh, tanh, sech, sech, coth, sqrt, asenh, asinh, acosh, atanh, asech, acsch, acoth, round.
- Números reales, por ejemplo 2,15, -20,5, etc.

El evaluador se puede modificar para poder admitir la variable y o la z por si queremos evaluar

funciones de varias variables, además se pueden agregar otras funciones con modificaciones mínimas (otra variable se manejaría igual que la x, para otra función habría que manejarla como las demás funciones con su cálculo correspondiente).

Este evaluador también lo sobrecargamos para que pueda admitir la forma *f(x)* que evaluaría la función en la última expresión parseada (recuerde que la última expresión parseada se guardaba en el módulo anterior en una variable global llamada *ultimaParseada*).

```
public double f(double x) throws ArithmeticException{
   try{
      return f(ultimaParseada,x);
   }catch(ArithmeticException e){
      throw e;
   }
}//Fin de la funcion f
```

Esta función se puede probar introduciendo cualquier expresión en notación postfija y un número para evaluar.

Uniendo estos dos programas, se tiene una poderosa herramienta para evaluar expresiones y funciones matemáticas.

Para poder usar esta clase dentro de un programa se debe crear un objeto (*Obj*) de tipo *Parseador*.

```
Parseador miparser = new Parseador();
```

Para parsear una expresión *expr* se escribe *miparser.parsear(expr)*, la función devuelve un *String* con *expr* en notación postfija, además el programa también guarda de manera automática la última expresión parseada. Para evaluar el número x en la expresión se utilizar *miparser.f(x)* para evaluar en la última expresión o se puede pasar una expresión en notación postfija escribiendo *miparser.f(exprEnPostfija, x)*.

Así, por ejemplo, el siguiente es el código de un applet[7] en donde se utiliza la clase *Parseador*

```
import java.applet.*;
import java.awt.*;

public class PruebaParseador extends java.applet.Applet {
  //Constructor del parseador
  Parseador miparser=new Parseador();
  //Expresión a parsear
  String expresion=new String();
  //Valor en el que se va a evaluar
  double valor=0;
  //Textfield donde se digita la expresión a parsear
  TextField inputexpresion = new TextField("x + 5");
  //Textfield donde se digita el valor a evaluar en la expresión
  TextField inputvalor = new TextField("0",5);
  //Botón para evaluar
  Button boton= new Button("Evaluar la expresión");
  //Resultado de parsear la expresión
  TextField outputparseo = new TextField("          ");
  //Resultado de la evaluación en la expresión
  TextField outputevaluar = new TextField("         ");
  //Label donde se dan los errores
  Label info = new Label("Información en extremo importante           ",
Label.CENTER);

  public void init(){ //Todo se pone en el applet
    add(inputexpresion);
    add(inputvalor);
    add(boton);
    add(outputparseo);
    add(outputevaluar);
    add(info);
  }//init

  public boolean action(Event evt, Object arg){
    if (evt.target instanceof Button){ //Si se apretó el botón
      try{
        info.setText(""); //Se pone el Label de los errores vacío
        expresion=inputexpresion.getText(); //Se lee la expresión
        //Se lee el valor a evaluar
        valor=Double.valueOf(inputvalor.getText()).doubleValue();
        //Se parsea la expresión
        outputparseo.setText(miparser.parsear(expresion));
        //Se evalúa el valor y se redondea
        outputevaluar.setText(""+redondeo(miparser.f(valor),5));
      }catch(Exception e){ //Si hubo error lo pone en el Label correspondiente
        info.setText(e.toString());
      }
    }//if del botón
    return true;
  }//action

  /*
  *Se redondea un número con los decimales dados
  */
  private double redondeo(double numero, int decimales){
    return
((double)Math.round(numero*Math.pow(10,decimales)))/Math.pow(10,decimales);
  }

}//PolCero
```

El applet se puede ver bajar y ver funcionando en la sección correspondiente a los programas, si se copia el texto se debe pegar en un archivo que se llame PruebaParseador.java y se debe tener una página HTML de prueba para verlo, la más sencilla es una página que tenga por código:

```
<HTML>
<HEAD>
</HEAD>
<BODY>
<CENTER>
<APPLET
  code  = "PruebaParseador.class"
  width = "500"
  height    = "300"
  >
</APPLET>
</CENTER>
</BODY>
</HTML>
```

Este applet tiene dos cajas de texto (*TextField*), una es donde se digita la expresión a evaluar (*inputexpresion*) y la otra es donde se digita el valor a evaluar(*inputvalor*), por defecto estas cajas iniciar con ´´x+5" y "0" respectivamente. El applet tiene un botón (*boton*), al ser presionado, se lee lo que el usuario escribió en las cajas de texto y se escribe el resultado de parsear la expresión con el texto *outputparseo.setText(miparser.parsear(expresion));*, observe que el código que hace la traducción es *miparser.parsear(expresion)* y el resultado se escribe en la etiqueta *outputparseo*; algo similar se hace con el resultado de la evaluación, sólo que antes se redondea el resultado con el código *redondeo(miparser.f(valor),5)*

Comentarios finales

Para finalizar, se comentará un poco las ideas para verificar si una expresión que digitó el usuario está bien escrita, lo cual no verificamos en el artículo pero sí está en el programa final. Para esto, lo que se hizo fue declarar una variable global *anterior* que se encarga de guardar un número dependiendo si lo último que tradujo en una expresión fue un número, un operador, etc. de la siguiente manera:

Valor	Última subexpresión traducida	Significa
0	nada	nada
1	Número, pi, e, x	Números
2	+, -, *, /, ^, %	Operadores
3	(, sen(, cos(, etc.	Paréntesis de apertura
4	)	Paréntesis de cierre

De esta forma se deben cumplir las siguientes reglas:

- Si no se había traducido nada (*anterior=0*) entonces se puede admitir cualquier cosa menos (+ * / ^ %).
- Si lo anterior fue un número puede seguir cualquier cosa.
- Si lo anterior fue un operador puede seguir cualquier cosa menos otro operador (con excepción de -).
- Si lo anterior fue un paréntesis de apertura puede seguir cualquier cosa menos (+ * / ^ %)
- Si lo anterior fue un cierre de paréntesis debe seguir un operador, un número (en cuyo caso hay una multiplicación oculta), un paréntesis de apertura (también hay una multiplicación oculta) u

otro paréntesis de cierre, estas multiplicaciones ocultas deben ser agregadas antes de poner el número o el paréntesis.

- Para un menos unario se debe agregar un -1 en la pila de números y un "por" en la de operadores (sacando los correspondientes operadores con mayor prioridad); el menos unario se da si no había nada anterior al menos o si era otro operador o un paréntesis de apertura.

El código completo con su correspondiente archivo, el archivo de prueba y la página HTML se encuentran al final de este artículo junto con el código similar en un módulo de Visual Basic que puede ser usado en cualquier programa de este lenguaje.

Otro programa recomendado que es similar al que se desarrolló en el artículo y que puede ser estudiado, bajado con su código completo y utilizado en otros programas libremente es: JEP[5]

Programas

> Parseador_Java.Zip

> Parseador_VB.Zip

Bibliografía

1
Aho, Alfred; Hopcroft, John; Ullman, Jeffrey.(1983) Data structures and algorithms. Massachusetts : Addison-Wesley.

2
Deitel H. y Deitel P. (1998) *Cómo programar en Java*. [Traducción del libro *Java how to program*] Primera edición. Prentice-Hall, México.

3
Murillo, M.; Soto, A. y Alfredo, J. (2000) *Matemática básica con aplicaciones*. EUNED. San José, C.R.

4
Sitio Web de JAVA: www.java.sun.com

5
Sitio Web de JEP - Java Expresion Parser: http://www.singularsys.com/jep

Implementación de un graficador de funciones con interfaz gráfica Swing y Java2D

Walter Mora F
Escuela de Matemática
Instituto Tecnológico de Costa Rica

Resumen:

Se presenta una introducción a Java Swing y Java2D, por medio de la implementación de un graficador de funciones de una variable real. Para leer la función se usa JEP (Java Expression Parser).

Palabras claves: Java, Swing, Java2D, JEP, funciones, gráfico de una función.

Introducción

Para la implementación de un graficador de funciones de una variable real primero necesitamos definir una interfaz gráfica, es decir, los componentes necesarios y cómo acomodarlos. Después de crear un método de graficación, necesitamos definir el manejo de eventos de botón, de campo de texto y de ratón para interactuar de manera adecuada con el usuario.

Esta presentación se inicia describiendo la manera en se pueden crear y acomodar los componentes del graficador usando la interfaz gráfica de usuario (GUI) Swing. Luego se implementa un método que lee una función introducida por el usuario y construye su gráfico en el dominio de pantalla usando Java2D. Por último se maneja los eventos de mouse para interactuar con el gráfico.

Observe que, para visualizar los applets, necesitará la versión 1.5 (o mayor) de la máquina virtual de Java.

Un 'parser' para leer la función

Para graficar una función necesitamos

1. Leer la función
2. Calcular los pares ordenados y unirlos con un segmento de recta

Para leer la función debemos usar un 'parser', es decir una clase que nos permita leer y evaluar la fórmula que define a la función. Aquí, en vez de implementar un 'parser' (que sería bastante laborioso), vamos a usar uno que está disponible de manera gratuita en internet: JEP (Java Expression Parser,http://sourceforge.net/projects/jep/). Al momento de esta publicación, la versión actual es la 2.3.1

Primero vamos a implementar un pequeño applet para ver como usar el evaluador

Object 1

La manera fácil para poder usar JEP con el IDE JCreator (http://www.jcreator.com/) es descargar JEP y poner la carpeta 'org' (que viene en la subcarpeta 'build') en la subcarpeta 'classes' del proyecto.

Si usa otro IDE, deberá hacer los cambios adecuados para que Java pueda encontrar el archivo jep231.jar.

En nuestro programa, se deben importar dos bibliotecas

```
import org.nfunk.jep.*;
import org.nfunk.jep.type.*;
```

La segunda biblioteca nos permite un buen manejo de los números complejos, para trabajar con funciones como $y=\ln(x)$ y $y=x^{1/3}$

Para leer y evaluar funciones se debe crear un objeto JEP con algunas características

```
private JEP miEvaluador = new JEP();
miEvaluador.addStandardFunctions();
miEvaluador.addStandardConstants();
miEvaluador.addComplex();
//habilitar 'sen'
miEvaluador.addFunction("sen", new org.nfunk.jep.function.Sine();
miEvaluador.addVariable("x", 0);
miEvaluador.setImplicitMul(true); //permite 2x en vez de 2*x
```

Para evaluar, se asigna una función (al leer una función, ésta se convierte en la función actual), se declara la variable y se evalúa en algún valor. Todo esto lo hacemos con un método.

```
double evaluar(JEP Miparser,String fun, double valorx)
{
   Miparser.parseExpression(fun);
   Miparser.addVariable("x", valorx);
   errorEnExpresion = Miparser.hasError(); //hay error?
  return Miparser.getValue();
}
```

el código completo es

Evaluador.java

```
import javax.swing.*;
import javax.swing.event.*;
//////////////////////
import java.awt.*;
import java.awt.event.*;
import org.nfunk.jep.*;
import org.nfunk.jep.type.*;

public class Evaluador extends JApplet
{
  private JEP miEvaluador;
  double vx;
  boolean errorEnExpresion; //si hay error de sintaxis en la función
  boolean errorEnNumero ;   //si hay error de sintaxis en el valor x
  JTextField Tfun;                //leer función
  JTextField Tvalorx;           //leer valor x para evaluar
  JButton BtnEvaluar;
  JLabel Mensaje;

  public void init()
  {
   Container content = getContentPane();

   Tfun = new JTextField("2x^2-3x+sen(x)",10);
   Tvalorx = new JTextField("4",3);
   BtnEvaluar = new JButton("Evaluar");
   Mensaje = new JLabel(""); //para imprimir resultado de evaluación

   content.setLayout(new FlowLayout());

   content.add(Tfun);
   content.add(Tvalorx);
   content.add(BtnEvaluar);
   content.add(Mensaje);
   //Manejo de eventos
   ManejadorDeEvento ManejadorDevt = new ManejadorDeEvento();
   Tfun.addActionListener(ManejadorDevt);
   Tvalorx.addActionListener(ManejadorDevt);
   BtnEvaluar.addActionListener(ManejadorDevt);

   //parser JEP
   miEvaluador = new JEP();
   miEvaluador.addStandardFunctions(); //agrega las funciones comunes
   miEvaluador.addStandardConstants();
   miEvaluador.addComplex();
   miEvaluador.addFunction("sen", new org.nfunk.jep.function.Sine());
   miEvaluador.addVariable("x", 0);
   miEvaluador.setImplicitMul(true); //permite 2x en vez de 2*x
  }

  private class ManejadorDeEvento implements ActionListener
  {
     public void actionPerformed (ActionEvent evt)
     {
       Object source = evt.getSource ();
       // si se presiona el botón o se da 'enter' en algún campo de texto
       if( source == BtnEvaluar || source == Tfun || source ==
Tvalorx)
          LeerImprimir();
     }
  }//


 void LeerImprimir()
  {
     try
     {
      Double dx = new Double( Tvalorx.getText() ); //leer el número
      vx = dx.doubleValue();
      errorEnNumero=false;
      Tvalorx.setForeground(Color.black);

     }catch(NumberFormatException e){ errorEnNumero=true; //número
incorrecto
                                       Tvalorx.setForeground(Color.red);
                                     }
     vx=evaluar(miEvaluador,Tfun.getText(),vx);
```

```
          if(!errorEnExpresion && !errorEnNumero)
          {
            Mensaje.setText(""+vx);
           }else Mensaje.setText("Hay un error.");

        }//

       double evaluar(JEP Miparser,String fun, double valorx)
       {
          Miparser.parseExpression(fun);
          Miparser.addVariable("x", valorx);
          errorEnExpresion = Miparser.hasError(); //hay error?
          return Miparser.getValue();
       }//
     }
```

Método de graficación.

Ahora que podemos leer funciones y evaluarlas, ya podemos implementar el método que construye el gráfico de la función.

> Coordenadas de pantalla y coordenadas reales

Primero debemos establecer un factor de escala para el eje X y el eje Y. Digamos 'escalaX = 30 pixeles' y 'escalaY=30 pixeles'.

Ahora debemos manejar la conversión entre coordenadas en números reales y coordenadas de pantalla. Supongamos que la variable 'aReal' corresponde al valor como número real de la coordenada de pantalla 'a'

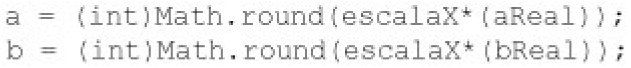
- Conversión: números reales a coordenadas de pantalla

```
a = (int)Math.round(escalaX * (aReal ) );
```

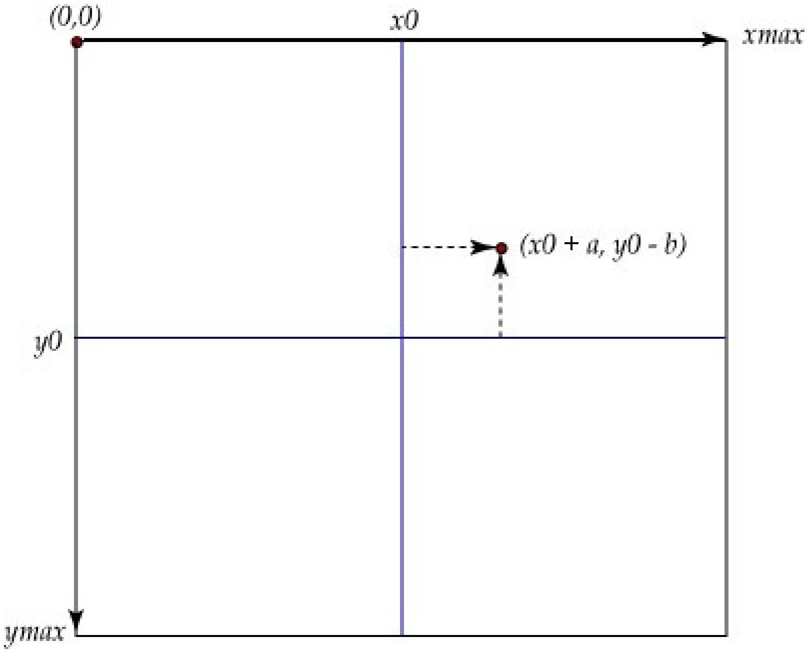
- Conversión: coordenadas de pantalla a números reales

```
aReal = 1.0*a/escalaX;
```

Vamos a suponer que el origen del sistema, en coordenadas de pantalla, es (x0,y0). En la siguiente figura se observa la representación del par ordenado (aReal, bReal) en coordenadas de pantalla (a,b) respecto al origen (x0,y0).



```
a = (int)Math.round(escalaX*(aReal));
b = (int)Math.round(escalaX*(bReal));
```



> Métodos de Java2D

Para dibujar el gráfico de la función necesitamos un dominio [xmin, xmax] y un conjunto de pares ordenados $\{(x,\ f(x)\ ):\ x\ \in\ [xmin, xmax]\}$ en coordenadas de pantalla.

Estos pares ordenados los unimos luego con segmentos de recta. Para crear segmentos y puntos (discos) vamos a usar los métodos gráficos de Java2D. Para esto necesitamos la biblioteca `java.awt.geom.*`.

- Definimos un par de constantes para controlar el grosor (stroke) y las líneas punteadas (dashed)

```
final static BasicStroke grosor1 = new BasicStroke(1.5f);
final static float dash1[] = {5.0f};
final static BasicStroke dashed = new BasicStroke(1.0f,

BasicStroke.CAP_BUTT,

BasicStroke.JOIN_MITER,
                                                  5.0f, dash1,
0.0f);
```

- Para habilitar anti-aliasing se usa el método

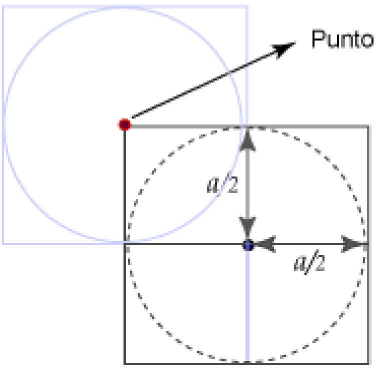
```
         setRenderingHint(RenderingHints.KEY_ANTIALIASING,
                RenderingHints.VALUE_ANTIALIAS_ON);
```

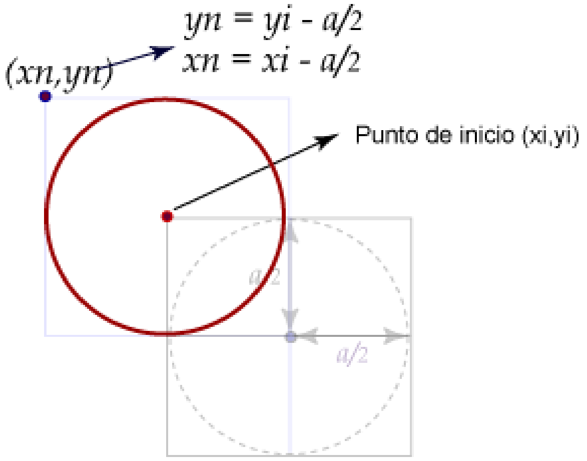
- Segmento de (x1, y1) a (x2, y2)

```
draw(new Line2D.Double(x1,y1,x2,y2));
```

- Círculo con centro en (xi, yi) y radio 'a'

```
draw(new Ellipse2D.Double(xi-a/2,yi-a/2,a,a));
```



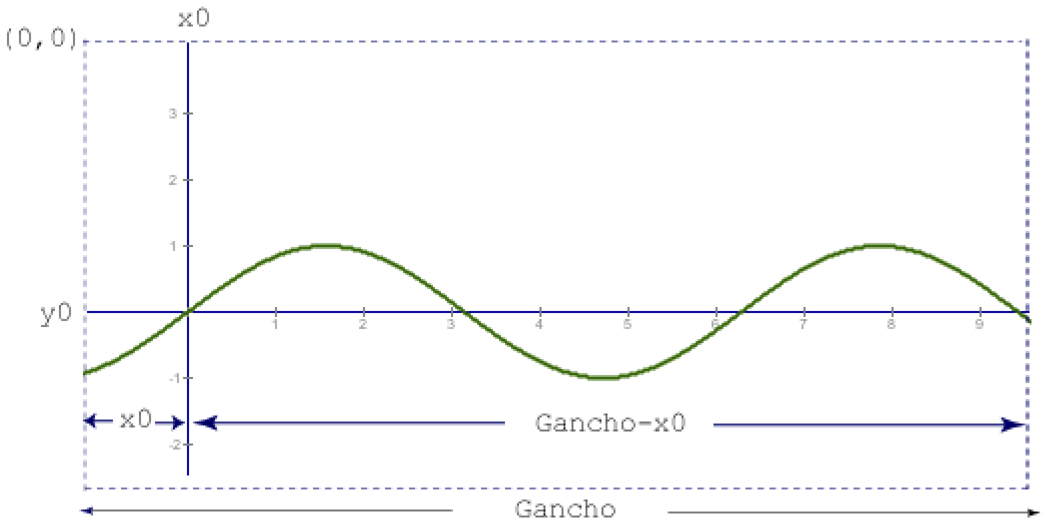


`g.draw(new Ellipse2D.Double(xi - a/2, yi-a/2, a, a ));`

(Java2D)

- Dominio de graficación.

En este graficador vamos a graficar en todo el dominio de pantalla, pensando en que más adelante vamos a arrastrar el origen de coordenadas con el mouse. Una vez que hemos elegido el origen de coordenadas (x0,y0) -en coordenadas de pantalla- podemos establecer `xmin` y `xmax` como los extremos de la región de graficación y luego pasar estas coordenadas de pantalla a números reales para poder calcular los pares ordenados



Así, haciendo la conversión a números reales

```
xmin  = -1.0*x0/escalaX;
xmax = (1.0*(Gancho-x0)/escalaX);
```

Ahora ya podemos crear el método gráfico.

Método 'Graficar'

```
void Graficar(Graphics ap, int xg, int yg)
{
  int xi=0,yi=0,xi1=0,yi1=0,numPuntos=1;
  int cxmin,cxmax,cymin,cymax;
  double valxi=0.0, valxi1=0.0, valyi=0.0,valyi1=0.0;
  Complex valC; //manejo de complejos en JEP
  double imgx;

  //convertimos el objeto ap en un objeto Graphics2D para usar los métodos Java2D
  Graphics2D g = (Graphics2D) ap;
  g.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
RenderingHints.VALUE_ANTIALIAS_ON);

  g.setFont(ft10);
  g.setPaint(new Color(0,0,150));

  //eje Y
  g.draw(new Line2D.Double(xg, 10, xg, Galto-10));
  //eje X
  g.draw(new Line2D.Double(10, yg, Gancho-10, yg));

  xmin=-1.0*xg/escalaX;
  xmax=(1.0*(Gancho-xg)/escalaX);
  cxmin=(int)Math.round(xmin); //pantalla
  cxmax=(int)Math.round(xmax);
  cymin=(int)Math.round(1.0*(yg-Galto)/escalaY);
  cymax=(int)Math.round(1.0*yg/escalaY);

  numPuntos=Gancho; //num pixels

  g.setPaint(Color.gray);
  g.setFont(ft7);


  //marcas en los ejes (ticks)
  if(escalaX>5)
  {
    for(int i=cxmin+1;i<cxmax;i++)
    {  g.draw(new Line2D.Double(xg+i*escalaX, yg-2, xg+i*escalaX , yg+2));
       if(i>0)
          g.drawString(""+i, xg+i*escalaX-2, yg+12);
       if(i<0)
         g.drawString(""+i, xg+i*escalaX-6, yg+12);
    }
  }

  if(escalaY>5)
  {
    for(int i=cymin+1;i<cymax;i++)
    {  g.draw(new Line2D.Double(xg-2, yg-i*escalaY, xg+2 , yg-i*escalaY));
       if(i>0)
          g.drawString(""+i, xg-12,yg-i*escalaY+3 );
       if(i<0)
```

```
          g.drawString(""+i, xg-14,yg-i*escalaY+3 );
    }
   }
   g.setPaint(new Color(50,100,0));

   g.setStroke(grosor1);
   miEvaluador.parseExpression(Tffun.getText());
   errorEnExpresion = miEvaluador.hasError(); //hay error?

   if(!errorEnExpresion)
   {
     Tffun.setForeground(Color.black);

     for(int i=0;i<numPuntos-1;i++)
     {
       valxi   =xmin +i*1.0/escalaX;
       valxi1  =xmin+(i+1)*1.0/escalaX;
       miEvaluador.addVariable("x", valxi);
       valyi   = miEvaluador.getValue();
       miEvaluador.addVariable("x", valxi1);
       valyi1  = miEvaluador.getValue();
       xi      =(int)Math.round(escalaX*(valxi));
       yi      =(int)Math.round(escalaY*valyi);
       xi1     =(int)Math.round(escalaX*(valxi1));
       yi1     =(int)Math.round(escalaY*valyi1);

      //control de discontinuidades groseras y complejos
       valC = miEvaluador.getComplexValue();
       imgx = (double)Math.abs(valC.im());
      if(dist(valxi,valyi,valxi1,valyi1)< 1000 && imgx==0)
      {
        g.draw(new Line2D.Double(xg+xi,yg-yi,xg+xi1,yg-yi1));
      }
     }//fin del for
   }else{mensaje.setText(":. Hay un error.");
         Tffun.setForeground(Color.red);
  }
}//
```

Ahora agregamos este método a al clase interna `ZonaGrafica` y lo llamamos en `paintComponet()`

```
class ZonaGrafica extends JPanel
{
   public void paintComponent(Graphics g)
   {
     super.paintComponent(g);
     Graficar(g, x0, y0);     //x0,y0 se inicalizan en init
     showStatus("http://www.cidse.itcr.ac.cr/");
   }

   void Graficar(Graphics ap, int xg, int yg)
  {
   .....
  }
}
```

Claro, todavía falta agregar en el constructor de ZonaGrafica, los eventos de arrastre del mouse

Manejo de eventos

Bien, ya tenemos casi todos los ingredientes. Sólo falta agregar el manejo de eventos, es decir,

I. si el usuario presiona el botón graficar o da enter en el campo de texto de la función, se debe dibujar el gráfico

II. si el usuario arrastra el mouse sobre la zona gráfica, se debe arrastrar el gráfico

III. si el usuario usa los deslizadores (sliders) se ejecuta el cambio de escala en el eje respectivo

IV. si el usuario entra al panel del logo (el mouse entra al panel) entonces el curso cambia a una "manita" indicando que, dando un clic o doble clic, irá al sitio web del CRV.

V. un frame de ayuda (acerca de la sintaxis de las fucniones)

Para hacer estas tareas debemos implementar los manejadores de eventos respectivos.

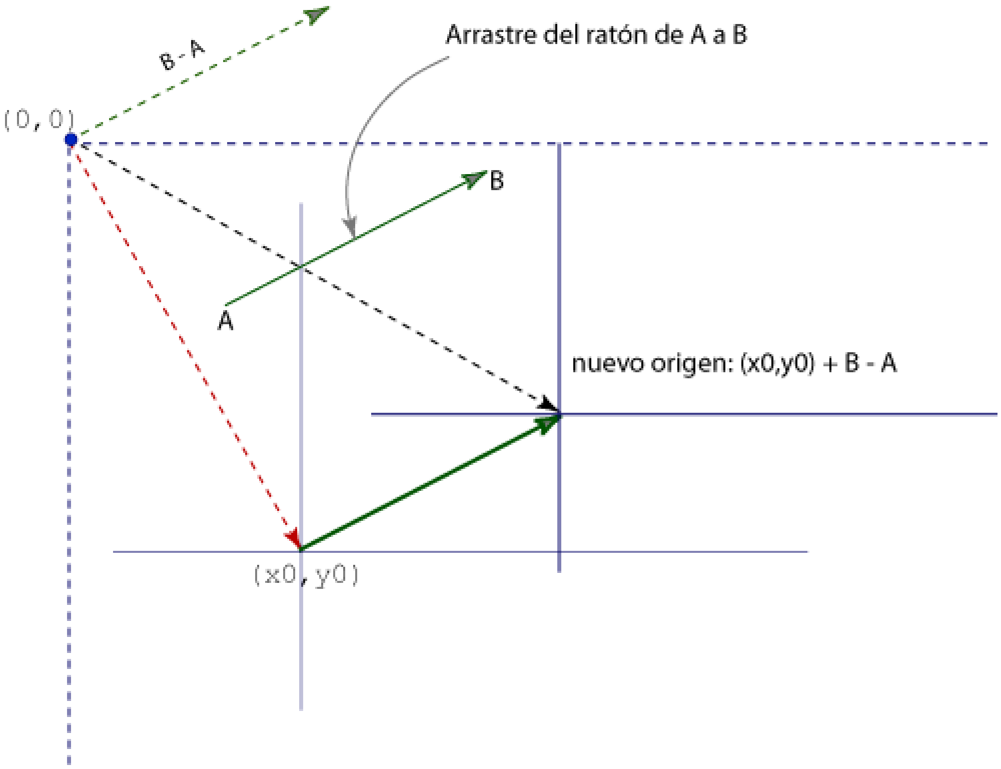
I. Manejador de eventos para campo de texto y para botón

Agregamos un auditor (listener) al campo de texto y al botón que "escucha" los eventos de 'dar enter' en el campo de texto o presionar el botón. Cuando esto sucede, solamente ejecutamos `ZG.repaint();` con lo que se ejecuta el método gráfico en el panel ZG

```
...

 public void init()
  { ...

  ManejadorDeEvento ManejadorDevt = new ManejadorDeEvento();
    Tffun.addActionListener(ManejadorDevt);
    BtnGraficar.addActionListener(ManejadorDevt);
  }//

private class ManejadorDeEvento implements ActionListener
{
  public void actionPerformed (ActionEvent evt)
  {
    Object source = evt.getSource ();
    // si se presiona el botón o se da 'enter' en algún campo de texto
    if ( source == BtnGraficar || source == Tffun)
    {
      ZG.repaint();
    }
  }
}
```

II. Arrastre del mouse

Para implementar la respuesta al arrastre del mouse, debemos localizar el punto inicial de arrastre A y el punto final B y redefinir el origen (x0,y0) de acuerdo al vector B-A. Esto lo hacemos en la clase interna 'ZonaGrafica'



```
class ZonaGrafica extends JPanel  implements MouseListener, MouseMotionListener
{
  int offsetX, offsetY;
  boolean dragging;

  ZonaGrafica()
  {
   offsetX=x0; offsetY=y0;
   addMouseListener(this);        //auditores para eventos de mouse
   addMouseMotionListener(this);
  }
  //manejo de eventos de mouse
  public void mousePressed(MouseEvent evt)
  {
   if (dragging)
   return;
   int x = evt.getX(); // clic inicial
   int y = evt.getY();
   offsetX = x - x0;
   offsetY = y - y0;
   dragging = true;
  }

  public void mouseReleased(MouseEvent evt)
  {
   dragging = false;
   repaint();
  }

  public void mouseDragged(MouseEvent evt)
```

```
{
  if (dragging == false)
  return;
  int x = evt.getX(); // posición del mouse
  int y = evt.getY();
  x0 = x - offsetX;  // mover origen usando suma vectorial
  y0 = y - offsetY;
  repaint();
}

 //el resto hace nada
public void mouseMoved(MouseEvent evt) {}
public void mouseClicked(MouseEvent evt) { }
public void mouseEntered(MouseEvent evt) { }
public void mouseExited(MouseEvent evt) { }

public void paintComponent(Graphics g)
{ ...
```

III. Deslizadores (sliders)

Agregar un par de deslizadores en el panel SP (instancia de SliderPanel) y un manejador de eventos que escuchan si hay algún cambio en cada deslizador. Si se escucha algún cambio, se redefine la variable escalaX o la variable escalaY, según corresponda y se ejecuta `ZG.repaint()` para actualizar en 'tiempo real'.

```
import javax.swing.event.*;

...

class SliderPanel extends JPanel
{
  JSlider xSlider,ySlider; // Manejo de escala

  SliderPanel()
  {
   setLayout(new GridLayout(1,2));
   SliderListener auditor = new SliderListener();
   //escala X
   xSlider = new JSlider(JSlider.VERTICAL, 1, 200, 20);
   xSlider.addChangeListener(auditor);
   add(xSlider);
   //escalaY
   ySlider = new JSlider(JSlider.VERTICAL, 1, 200, 20);
   ySlider.addChangeListener(auditor);
   add(ySlider);

   //xSlider.setLabelTable(xSlider.createStandardLabels(20));
   //xSlider.setMajorTickSpacing(200);
   xSlider.setMinorTickSpacing(20);
   xSlider.setPaintTicks(true);
   xSlider.setPaintLabels(true);
```

```
           //ySlider.setMajorTickSpacing(200);
           ySlider.setMinorTickSpacing(20);
           ySlider.setPaintTicks(true);
           ySlider.setPaintLabels(true);
          }//

          class SliderListener implements ChangeListener
          {
            public void stateChanged(ChangeEvent e)
            {
              JSlider source = (JSlider)e.getSource();
              ajusteEscala();

            }
          }

         public void ajusteEscala()
          { // se ejecuta si se 'oyó' algún cambio en algún Slider
           escalaX =(int) xSlider.getValue();
           escalaY =(int) ySlider.getValue();
           ZG.repaint();
          }//
        } //fin class
```

IV. Implementar LogoPanel

El manejo propuesto en LogoPanel no requiere mucho detalle, solo agregar una propiedad que defina comportamiento requerido. Esta propiedad se la podemos agregar en init().

```
...
public void init()
{ ...

 //Logopanel
  LogoPanel = new JPanel();
  LogoPanel.add(new JLabel(logocrv));
  LogoPanel.addMouseListener (new MouseAdapter ()
  {
   public void mouseClicked (MouseEvent e)
   {
     if(e.getClickCount() >0)
     {
       try{URL elURL= new URL("http://www.cidse.itcr.ac.cr/crv/index.html");
           getAppletContext().showDocument(elURL);
          }catch(MalformedURLException ae){}
     }
   }
    });

   LogoPanel.setCursor(new Cursor(Cursor.HAND_CURSOR));
  //fin logoPanel
```

```
...
```

V. Ventana (JFrame) de ayuda

Para la ventana de ayuda creamos un JFrame con un JTextArea con algunas indicaciones acerca d ela sintaxis de las funciones en JEP. Debemos agregar un auditor al botón 'Ayuda' para que levante la ventana cuando es presionado.

```
...

public void init()
{ ...

 BtnAyuda.addActionListener(ManejadorDevt);
}

...

private class ManejadorDeEvento implements ActionListener
{
  public void actionPerformed (ActionEvent evt)
  {
    Object source = evt.getSource ();
    ...
    if(source == BtnAyuda)
    {
      fFrame.setVisible (true);
    }//
  }
}//

class AyudaJFrame extends JFrame
{

   JTextArea p;

   GraficadorClasico fApplet;

   AyudaJFrame(GraficadorClasico applet)
   {
    super ("Ayuda");
    fApplet=applet;
    Container content_pane = getContentPane ();

    p = new JTextArea(30,40);
    p.setText(information());
    p.setEditable(false);

    JScrollPane sp = new JScrollPane(p);
    content_pane.add(sp,BorderLayout.CENTER);

    pack ();
    setDefaultCloseOperation (JFrame.DISPOSE_ON_CLOSE);
   }
   public void actionPerformed (ActionEvent e)
   {
     //nada por hoy
   }


  String information(){
    String message =
    " :.\n"
+ " Mover ejes : arrastre el mouse\n\n"
+" ------ ------ EJEMPLO\n"
+ " + suma x+2\n"
+ " - resta x-5\n"
+ " * multiplicación 3*x\n"
+ " / división -1/x\n"
+ " () agrupación (x+2)/(3*x)\n"
+ " ^ potenciación (-3*x)^2\n"
+ " % resto de la división x%5\n"
+ " RAIZ(x) raíz cuadrada RAIZ(x)\n"
+ " sqrt() raíz cuadrada sqrt(x)\n"
+ " mod() resto de la división mod(x,5)\n"
+ " sen() seno 4*sen(x^2)\n"
+ " cos() coseno 6*cos(-3*x)\n"
+ " tan() tangente 3*tan(x)\n"
+ " atan() arcotangente atan(x-3)\n"
+ " asin() arcoseno asen((x+5)/(3^x))\n"
+ " acos() arcocoseno 2-acos(-x+3)\n"
+ " sinh() seno hiperbólico sinh(x)\n"
+ " cosh() coseno hiperbólico -3*cosh(1/x)\n"
+ " tanh() tangente hiperbólica tanh(x)/2\n"
+ " asinh() arcoseno hiperbólico 2*asinh(x)/3\n"
+ " acosh() arcocoseno hiperbólico (2+acosh(x))/(1-x)\n"
+ " atanh() arcotangente hiperbólica atanh(x)*(3-x^(1/x))\n"
+ " ln() logaritmo natural ln(x)+1\n"
+ " log() logaritmo decimal -2*log(x)-1\n"
+ " abs() valor absoluto abs(x-2)\n"
+ " rand() valor aleatorio rand()\n"
+ " re() parte real de un Complejo re(2+9*i)\n"
+ " im() parte imaginaria im(-8+7*i)\n"
+ " angle() ángulo en pos. estándar angle(x,2)\n\n"
+ " pi 3,141592653589793 pi+cos(x)\n"
+ " e 2,718281828459045 e+1\n"
+ " Usa JEP,(Nathan Funk http://sourceforge.net/projects/jep/\n\n";

return message;
}//información

} // class AyudaFrame
```

El código completo junto con las imágenes y JEP, están en la carpeta de proyecto GraficadorClásico, en la subcarpeta "src". Este proyecto fue desarrollado con JCreator 3.5.

El applet se ve así

Bibliografía

1. Java Tutorial (http://java.sun.com/docs/books/tutorial/). Consultado en Abril, 2006.

2. Deitel, H. Deitel, P. "Cómo programar en Java". Pearson Educación, México, 2004.

3. JEP: Java Expression Parser. http://sourceforge.net/projects/jep/. Consultado en Abril, 2006

Sección Tecnologías de Internet
Revista digital Matemática, Educación e Internet (www.cidse.itcr.ac.cr/revistamate/). Vol. 9, N^o 1. 2008

Probabilidad, Números Primos y Factorización de Enteros. Implementaciones en Java y VBA para Excel.

Walter Mora F.
wmora2@yahoo.com.mx
Escuela de Matemática
Instituto Tecnológico de Costa Rica

Palabras claves: Teoría de números, probabilidad, densidad, primos, factorización, algoritmos.

1.1 Introducción

> *"God may not play dice with the universe, but something strange is going on with the prime numbers." Paul Erdös, 1913-1996.*

> *"Interestingly, the error* $O(\sqrt{n \ln^3 n})$ *predicted by the Riemann hypothesis is essentially the same type of error one would have expected if the primes were distributed randomly. (The law of large numbers.) Thus the Riemann hypothesis asserts (in some sense) that the primes are pseudorandom - they behave randomly, even though they are actually deterministic. But there could be some sort of "conspiracy" between members of the sequence to secretly behave in a highly "biase" or "non-random" manner. How does one disprove a conspiracy?." Terence Tao, Field Medal 2006.*

Este trabajo muestra cómo algunos métodos estadísticos y probabilísticos son usados en teoría de números. Aunque en este contexto no se puede definir una medida de probabilidad en el sentido del modelo axiomático, los métodos probabilísticos se han usado para orientar investigaciones acerca del comportamiento en promedio, de los números primos. Algunos de estos resultados se usan para hacer estimaciones acerca de la eficiencia en promedio de algunos algoritmos para factorizar enteros. Consideramos dos métodos de factorización, “ensayo y error” y el método “rho” de Pollard. Los enteros grandes tienen generalmente factores primos pequeños. Es normal tratar de detectar factores menores que 10^8 con el método de ensayo y error y factores de hasta doce o trece dígitos con alguna variación eficiente del método rho de Pollard. Los números “duros” de factorizar requieren algoritmos más sofisticados (un número duro podría ser, a la fecha, un entero de unos trescientos dígitos con solo dos factores primos muy grandes). Aún así, estos algoritmos (a la fecha) han tenido algún éxito solo con números (duros) de hasta unas doscientos dígitos, tras un largo esfuerzo computacional.

Además de discutir algunos algoritmos, se presenta la implementación en Java. En el apéndice se presentan algunas implementaciones en VBA Excel.

1.2 A los números primos les gustan los juegos de azar.

1.2.1 ¿La probabilidad de que un número natural, tomado al azar, sea divisible por p es $1/p$?

¿Qué significa “tomar un número natural al azar”?. Los naturales son un conjunto infinito, así que no tiene sentido decir que vamos a tomar un número al azar. Lo que si podemos es tomar un número de manera aleatoria en un conjunto finito $\{1,2,...,n\}$ y luego (atendiendo a la noción frecuencista de probabilidad) ver que pasa si n se hace grande (i.e. $n \longrightarrow \infty$).

Hagamos un pequeño experimento: Fijamos un número p y seleccionamos de manera aleatoria un número en el conjunto $\{1,2,...,n\}$ y verificamos si es divisible por p. El experimento lo repetimos m veces y calculamos la frecuencia relativa.

En la tabla que sigue, hacemos este experimento varias veces para n,m y p.

n	m	p	Frecuencia relativa
100000	10000	5	0.1944
			0.2083
			0.2053
			0.1993
10000000	100000	5	0.20093
			0.19946
			0.1997
			0.20089
100000000	1000000	5	0.199574
			0.199647

Tabla 1.1

Y efectivamente, parece que "la probabilidad" de que un número tomado al azar en el conjunto $\{1,2,...,n\}$ sea divisible por p es $1/p$

De una manera sintética: Sea $E_p(n) =$ los múltiplos de p en el conjunto $\{1,2,...,n\}$. Podemos calcular la la proporción de estos múltiplos en este conjunto, es decir, podemos calcular $\frac{E_p(n)}{n}$ para varios valores de n

n	Múltiplos de $p=5$	Proporción
100	20	0.2
10230	2046	0.2
100009	20001	0.199992
1000000	199999	0.199999

Tabla 1.2

Parece que en el conjunto $\{1,2,...,n\}$, la proporción de los múltiplos de $p=5$ se aproxima a $1/5$, conforme n se hace grande. ¿Significa esto que la probabilidad de que un número natural, tomado al azar, sea divisible por 5 es $1/5$?. Por ahora, lo único que podemos decir es que este experimento sugiere que la densidad (o la proporción) de los múltiplos de 5 en $\{1,2,\ldots,n\}$ parece ser $1/5$ conforme n se hace grande. Generalizando,

Definición 1.1 Sea E un conjunto de enteros positivos con alguna propiedad especial y sea $E(N)=E\bigcap\{1,2,\ldots,N\}.$ La densidad (o medida relativa) de E se define como

$$D[E]=\lim_{n\to\infty}\frac{E(n)}{n}$$

siempre y cuando este límite exista.

¿Es esta densidad una medida de probabilidad?. Para establecer una medida de probabilidad P, en el modelo axiomático, necesitamos un conjunto Ω (“espacio muestral”). En Ω hay una colección E de subconjuntos E_i (una $\sigma-$álgebra sobre Ω), llamados “eventos”, con medida de probabilidad $P(E_i)$ conocida. La idea es extender estas medidas a una colección más amplia de subconjuntos de Ω. P es una medida de probabilidad si cumple los axiomas

1. $P(E_i)\geq 0$ para todo $E_i\in\Omega$,
2. Si $\{E_j\}$ es una colección de conjuntos disjuntos dos a dos en F, entonces

$$P\left(\bigcup_j E_j\right)=\sum_j P(E_j),$$

3. $P(\Omega)=1$

Cuando el espacio muestral Ω es finito y los posibles resultados tienen igual probabilidad entonces $P(E)=\dfrac{|E|}{|\Omega|}$ define una medida de probabilidad.

La densidad D no es una medida de probabilidad porque no cumple el axioma 2.

> La idea de la demostración [21] usa el Teorema de Mertens (ver más adelante). Si denotamos con E_p los múltiplos positivos de p y si suponemos que hay una medida de probabilidad P en $\mathbb{Z}^+$ tal que $P(E_p)=1/p$, entonces $P\left(E_p\cap E_q\right)=(1-1/p)(1-1/q)$ para p,q primos distintos. De manera

inductiva y utilizando el teorema de Mertens se llega a que $P(\{m\}) = 0$ para cada entero positivo m. Luego,

$$P\left(\bigcup_{m\in\mathbb{Z}^+} \{m\}\right) = 1 \neq \sum_m P(\{m\}) = 0.$$

Aunque en el esquema frecuencista se puede ver la densidad como la "probabilidad" de que un entero positivo, escogido aleatoriamente, pertenezca a E, aquí identificamos este término con *densidad o proporción*. Tenemos,

Teorema 1.1 *La densidad de los naturales divisibles por* p *es* $\frac{1}{p}$, *es decir, si* E_p *es el conjunto de enteros positivos divisibles por* p, *entonces*

$$D[E_p] = \lim_{n\to\infty} \frac{E_p(n)}{n} = \frac{1}{p}$$

Prueba: Para calcular el límite necesitamos una expresión analítica para $E_p(n)$. Como existen p, r tales que $n = pk + r$ con $0 \leq r < p$, entonces $kp \leq n < (k+1)\,p$, es decir, hay exactamente k múltiplos positivos de p que son menores o iguales a n. Luego $E_p(n) = k = \frac{n-r}{p}$.

Por lo tanto,

$$\begin{aligned} D[E_p] &= \lim_{n\to\infty} \frac{E_p(n)}{n} = \lim_{n\to\infty} \frac{\frac{n-r}{p}}{n} \\ &= \lim_{n\to\infty} \frac{n-r}{pn} = \lim_{n\to\infty} \frac{1}{p} - \frac{r}{pn} = \frac{1}{p} \end{aligned}$$

1.2.2 Teorema de los Números Primos

$\pi(x)$ denota la cantidad de primos que no exceden x. Por ejemplo, $\pi(2) = 1$, $\pi(10) = 4$ y $\pi(\sqrt{1000}) = 11$.

Para la función $\pi(x)$ no hay una fórmula sencilla. Algunas fórmulas actuales son variaciones un poco más eficientes que la fórmula recursiva de Legendre (1808).

Fórmula de Legendre.

Esta fórmula esta basada en el principio de inclusión-exclusión. Básicamente dice que el conjunto $\{1,2,...,[\![x]\!]\}$ es la unión del entero $1,$ los primos $\leq x$ y los enteros compuestos $\leq x,$

$$[\![x]\!] = 1+\pi(x)+\#\{\text{ enteros compuestos} \leq x\}$$

Un entero compuesto en $A=\{1,2,...,[\![x]\!]\}$ tiene al menos un divisor primo menor o igual a $\sqrt{x}$[1]. Esto nos ayuda a detectar los números compuestos en $A:$ solo tenemos que contar los elementos de A con un divisor primo $\leq \sqrt{x}.$

$\lfloor\!\lfloor x\rfloor\!\rfloor = n$ si $n \leq x < n+1.$ Entonces si $\lfloor\!\lfloor x/p\rfloor\!\rfloor = k$ significa que $kp \leq x,$ i.e. $p < 2p < ... < k\cdot p \leq x.$ Luego, la cantidad de enteros $\leq x$ divisibles por p es $\lfloor\!\lfloor x/p\rfloor\!\rfloor.$

Ahora, ¿#{ enteros compuestos $\leq x\}$ es igual a al conteo total de los múltiplos de cada primo $p_i \leq \sqrt{x}$? No, pues este conteo incluye a los propios primos $p_i,$ así que hay que reponer con $\pi(\sqrt{x})$ para hacer una corrección. Pero también habría que restar los compuestos que son divisibles por p_i y p_j pues fueron contados dos veces, pero esto haría que los números divisibles por $p_i,\, p_j,\, p_k$ fueran descontados una vez más de lo necesario así que hay que agregar una corrección para estos números, y así sucesivamente.

EJEMPLO 1.1 Si $x=30,$ los primos menores que $\lfloor\!\lfloor\sqrt{30}\rfloor\!\rfloor = 5$ son $2,3$ y $5.$

$\lfloor\!\lfloor 30/2\rfloor\!\rfloor = 15$ cuenta $\{2,4,6,8,10,12,14,16,18,20,22,24,26,28,30\}$
$\lfloor\!\lfloor 30/3\rfloor\!\rfloor = 10$ cuenta $\{3,6,9,12,15,18,21,24,27,30\}$
$\lfloor\!\lfloor 30/5\rfloor\!\rfloor = 6$ cuenta $\{5,10,15,20,25,30\}$

En el conteo $\lfloor\!\lfloor 30/2\rfloor\!\rfloor + \lfloor\!\lfloor 30/3\rfloor\!\rfloor + \lfloor\!\lfloor 30/5\rfloor\!\rfloor:$

- se contaron los primos $2,3$ y $5.$

---

[1] Esto es así pues si $n \in A$ y si $n=ab,$ no podría pasar que a y b sean ambos $\geq \sqrt{x}$ pues sería una contradicción pues $n \leq x.$

- $6, 12, 18, 24, 30$ fueron contados dos veces como múltiplos de $2,\ 3$
- $10, 20, 30$ fueron contados dos veces como múltiplos de $2,\ 5$
- $15, 30$ fueron contados dos veces como múltiplos de $3,\ 5$
- 30 fue contado tres veces como múltiplo de $2, 3$ y 5.

Finalmente,

$$\begin{aligned}\#\{\text{ enteros compuestos} \leq 30\} &= [\![30/2]\!] + [\![30/3]\!] + [\![30/5]\!] \\ &- [\![30/(2\cdot 3)]\!] - [\![30/(2\cdot 5)]\!] - [\![30/(3\cdot 5)]\!] \\ &+ [\![30/(2\cdot 3 + \cdot 5)]\!] \\ &= 31 - 3 - 5 - 3 - 2 + 1 = 19\end{aligned}$$

El último sumando se agrega pues el 30 fue contado tres veces pero también se resto tres veces.

Observe ahora que en $\{1, 2, ..., 30\}$ hay 19 compuestos y el 1, así que quedan 10 primos.

Sea p_i el $i-$ésimo primo. La fórmula de Legendre es,

$$1 + \pi(x) = \pi\left(\sqrt{x}\right) + [\![x]\!] - \sum_{p_i \leq \sqrt{x}} \left[\frac{x}{p_i}\right] + \sum_{p_i < p_j \leq \sqrt{x}} \left[\frac{x}{p_i p_j}\right] - \sum_{p_i < p_j < p_k \leq \sqrt{x}} \left[\frac{x}{p_i p_j p_k}\right] + \cdots$$

Para efectos de implementación es mejor poner $\alpha = \pi(\sqrt{x})$ y entonces la fórmula queda

$$1 + \pi(x) = \pi\left(\sqrt{x}\right) + [\![x]\!] - \sum_{i \leq \alpha} \left[\frac{x}{p_i}\right] + \sum_{i < j \leq \alpha} \left[\frac{x}{p_i p_j}\right] - \sum_{i < j < k \leq \alpha} \left[\frac{x}{p_i p_j p_k}\right] + \cdots$$

EJEMPLO 1.2 Calcular $\pi(100)$
Solución: Como $\sqrt{100} = 10$, solo usamos los primos $\{2, 3, 5, 7\}$.

$$\begin{aligned}
1+\pi(100) &= \pi(10)+\lfloor\lfloor 100\rfloor\rfloor \\
&\quad -(\lfloor\lfloor 100/2\rfloor\rfloor+\lfloor\lfloor 100/3\rfloor\rfloor+\lfloor\lfloor 100/5\rfloor\rfloor+\lfloor\lfloor 100/7\rfloor\rfloor) \\
&\quad +\lfloor\lfloor 100/2\cdot 3\rfloor\rfloor+\lfloor\lfloor 100/2\cdot 5\rfloor\rfloor+\lfloor\lfloor 100/2\cdot 7\rfloor\rfloor+\lfloor\lfloor 100/3\cdot 5\rfloor\rfloor+\lfloor\lfloor 100/3\cdot 7\rfloor\rfloor+\lfloor\lfloor 100/5\cdot 7\rfloor\rfloor \\
&\quad -(\lfloor\lfloor 100/2\cdot 3\cdot 5\rfloor\rfloor+\lfloor\lfloor 100/2\cdot 3\cdot 7\rfloor\rfloor+\lfloor\lfloor 100/2\cdot 3\cdot 7\rfloor\rfloor+\lfloor\lfloor 100/3\cdot 5\cdot 7\rfloor\rfloor) \\
&\quad +\lfloor\lfloor 100/3\cdot 3\cdot 5\cdot 7\rfloor\rfloor \\
&= 4+100-(50+33+20+14)+(16+10+7+6+4+2)-(3+2+0+1)+0 \;=\; 26
\end{aligned}$$

El problema con esta fórmula es la cantidad de cálculos que se necesita para calcular las correcciones.

Las cantidad de partes enteras $\lfloor\lfloor x/(p_{i_1}p_{i_2}\cdots p_{i_k})\rfloor\rfloor$ corresponde a la cantidad de subconjuntos no vacíos $\{i_1,i_2,\cdots,i_k\}$ de $\{1,2,...,\alpha\}$, es decir, hay que calcular $2^{\alpha}-1$ partes enteras.

Si quisieramos calcular $\pi(10^{33})$, entonces, puesto que $\sqrt{10^{33}}=10^{18}$, tendríamos que tener los primos $\leq 10^{18}$ y calcular las partes enteras $\left[x/(p_{k_1}p_{k_2}...p_{k_j})\right]$ que corresponden al cálculo de todos los subconjuntos de $\{1,2,...,\pi(10^{18})\}$. Como $\pi(10^{18})=24739954287740860$, tendríamos que calcular

$$2^{24739954287740860}-1 \quad \text{partes enteras.}$$

que constituye un número nada razonable de cálculos.

Fórmula de Meisel.

La fórmula de Meisel es un re-arreglo de la fórmula de Legendre. Pongamos

$$\text{Legendre}(x,\alpha)=\sum_{i\leq\alpha}\left[\frac{x}{p_i}\right]-\sum_{i<j\leq\alpha}\left[\frac{x}{p_ip_j}\right]+\sum_{i<j<k\leq\alpha}\left[\frac{x}{p_ip_jp_k}\right]+\cdots$$

Así $\pi(x)=\llbracket x\rrbracket-1+\alpha-\text{Legendre}(x,\alpha)$ donde $\alpha=\pi(\sqrt{x})$, es decir, $\text{Legendre}(x,\alpha)-\alpha$ cuenta la cantidad de números compuestos $\leq x$ o, en otras palabras, los números $\leq x$ con al menos un divisor primo inferior a $\alpha=\sqrt{x}$.

Ahora $\text{Legendre}(x,\alpha)$ va a tener un significado más amplio: Si $\alpha\in\mathbb{N}$,

$$\text{Legendre}(x,\alpha)=\sum_{i\leq\alpha}\left[\frac{x}{p_i}\right]-\sum_{i<j\leq\alpha}\left[\frac{x}{p_ip_j}\right]+\sum_{i<j<k\leq\alpha}\left[\frac{x}{p_ip_jp_k}\right]+\cdots$$

es decir, $\text{Legendre}(x,\alpha)-\alpha$ cuenta los compuestos $\leq x$ que son divisibles por primos $\leq p_\alpha$. La resta es necesaria pues la manera de contar cuenta también los primos $p_1,\ p_2,\ ...,p_\alpha$

Ahora, dividamos los enteros en cuatro grupos: $\{1\}$, $\{\text{primos }\leq x\}$, $C_3\cup C_4=$ los compuestos $\leq x$.

$[\![x]\!]=1+\pi(x)+\#C_3+\#C_4$

$\#C_3$: Es la cantidad de números compuestos $\leq x$ con al menos un divisor primo $\leq p_\alpha$, es decir $\text{Legendre}(x,\alpha)-\alpha$.

$\#C_4$ son los compuestos $\leq x$ cuyos divisores primos son $>p_\alpha$: Aquí es donde entra en juego la escogencia de α para determinar la cantidad de factores primos de estos números.

Sea p_i el $i-$ésimo primo. Sean p_α y p_β tal que $p_\alpha^3\leq x<p_{\alpha+1}^3$ y $p_\beta^2\leq x<p_{\beta+1}^2$. En otras palabras: $\alpha=\pi(\sqrt[3]{x})$ y $\beta=\pi(\sqrt{x})$.

Consideremos la descomposición prima de $n\in C_4$, $n=p_{i_1}\cdot p_{i_2}\cdots p_{i_k}$ con $\alpha<p_{i_1}<p_{i_2}<\ldots<p_{i_k}$ y $k\geq 2$. Como $p_{\alpha+1}^k\leq p_{i_1}\cdot p_{i_2}\cdots p_{i_k}\leq x<p_{\alpha+1}^3\implies k=2$.

Así que estos números en C_4 son de la forma $p_{\alpha+k}p_j\leq x,\ a+k\leq j,\ k=1,2,...$
Pero la cantidad de números $p_{\alpha+k}p_j$ es igual a la cantidad de p_j's tal que $p_j\leq x/p_{\alpha+k}$: $\pi(x/p_{\alpha+k})-(\alpha+k)$.

Además $\alpha<\alpha+k\leq\beta$ pues si $\alpha+k=\beta,\ p_\beta\cdot p_\beta=p_\beta^2\leq x$ pero $p_{\beta+1}p_j\geq p_{\beta+1}^2>x$.

Así, usando la fórmula $\sum_{i=1}^{n-1}i=n(n-1)/2$,

$$\#C_4=\sum_{\alpha<i\leq\beta}\{\pi(x/p_i)-(i-1)\}=\frac{1}{2}\beta(\beta-1)-\frac{1}{2}\alpha(\alpha-1)+\sum_{\alpha<i\leq\beta}\pi(x/p_i)$$

¿Cuál es la ganancia?

Mientras que con la fórmula de Legendre necesitamos conocer $\pi(\sqrt{x})$ y calcular con primos $\leq \sqrt{x}$, con la fórmula de Meisel solo necesitamos conocer hasta $\pi(\sqrt[3]{x})$ y calcular con primos $\leq \sqrt[3]{x} < \sqrt{x}$.

EJEMPLO 1.3 **1.1** Calcule $\pi(100)$ usando la fórmula de Meisel.

Solución: **Meisel:** Como $\alpha = \pi(\sqrt[3]{100}) = 2$ y $\beta = \pi(\sqrt{100}) = 4$, solo vamos a usar los primos $p_1 = 2,\ p_2 = 3,\ p_3 = 5,\ p_4 = 7$.

$$\begin{aligned} \text{Legendre}(100,2) &= [\![100/2]\!] + [\![100/3]\!] + [\![100/2\cdot 3]\!] \\ &= 50 + 33 - 16 = 67 \end{aligned}$$

$$\begin{aligned} \text{Meisel}(100,2,4) &= \pi(100/5) + \pi(100/7) \\ &= \pi(20) + \pi(4) = 8 + 6 = 14 \end{aligned}$$

Así, $\pi(100) = 100 + 6 - 0 - 67 - 14 = 25$

A la fecha (2007) se conoce $\pi(x)$ hasta $x = 10^{22}$. *Mathematica* (Wolfram Research Inc.) implementa $\pi(x)$ con el comando `PrimePi`$[x]$ hasta $x \approx 8 \times 10^{13}$. En esta implementación, si x es pequeño, se calcula $\pi(x)$ usando colado y si x es grande se usa el algoritmo Lagarias-Miller-Odlyzko.

Estimación asintótica de $\pi(x)$. Teorema de los números primos.

La frecuencia relativa $\pi(n)/n$ calcula la proporción de primos en el conjunto $A = \{1, 2, ..., n\}$. Aunque la distribución de los primos entre los enteros es muy irregular, el comportamiento promedio si parece ser agradable. Basado en un estudio empírico de tablas de números primos, Legendre y Gauss (en 1792, a la edad de 15 años) conjeturan que la ley que gobierna el cociente $\pi(n)/n$ es aproximadamente igual a $\dfrac{1}{\ln(n)}$.

En [9] se indica que Gauss y Legendre llegaron a este resultado, de manera independiente, estudiando la densidad de primos en intervalos que difieren en potencias de diez: notaron que la proporción de primos en intervalos centrados en $x = 10^n$ decrece lentamente y disminuye aproximadamente a la mitad cada vez que pasamos de x a x^2. Este fenómeno es muy bien modelado por $1/\ln(x)$ pues $1/\ln(x^2) = 0.5/\ln(x)$.

EJEMPLO 1.4 Frecuencia relativa y estimación.

n	$\pi(n)$	$\pi(n)/n$	$1/\ln(n)$
10^7	664579	0.0664579	0.0620420
10^8	5761455	0.0576145	0.0542868
10^9	50847534	0.0508475	0.0482549
10^{10}	455052511	0.0455052	0.0434294
10^{11}	4118054813	0.0411805	0.0394813
10^{11}	37607912018	0.0376079	0.0361912

Tabla 1.3

Acerca de este descubrimiento, Gauss escribió a uno de sus ex-alumnos, Johann Franz Encke, en 1849

> *"Cuando era un muchacho considere el problema de cuántos primos había hasta un punto dado. Lo que encontré fue que la densidad de primos alrededor de x es aproximadamente $1/\ln(x)$."*

La manera de interpretar esto es que si n es un número "cercano" a x, entonces es primo con "probabilidad" $1/\ln(x)$. Claro, un número dado es o no es primo, pero esta manera de ver las cosas ayuda a entender de manera muy intuitiva muchas cosas acerca de los primos.

Lo que afirma Gauss es lo siguiente: Si Δx es "pequeño" comparado con x (en el mundillo asintótico esto quiere decir que $\Delta x/x \to 0$ conforme $x \to \infty$) entonces

$$\frac{\pi(x+\Delta x)-\pi(x)}{\Delta x} \approx \frac{1}{\ln(x)}$$

$\dfrac{\pi(x+\Delta x)-\pi(x)}{\Delta x}$ es la densidad de primos en le intervalo $[x,\, x+\Delta x]$ y $\dfrac{1}{\ln(x)}$ es el promedio estimado en este intervalo.

Por esto decimos: $1/\ln(x)$ es la “probabilidad” de que un número $n,$ en las cercanías de $x,$ sea primo.

Para hacer un experimento, podemos tomar $\Delta x = \sqrt{x}$ (que claramente es dominada por x),

x	$\pi(x+\Delta x)-\pi(x)$	$\dfrac{\pi(x+\Delta x)-\pi(x)}{\Delta x}$	$\dfrac{1}{\ln(x)}$
10	2	0.632	0.434
100	4	0.4	0.217
1000	5	0.158	0.144
10000	11	0.11	0.108
100000	24	0.075	0.086
1000000	75	0.075	0.072
10000000	197	0.0622	0.062
100000000	551	0.0551	0.054
1000000000	1510	0.0477	0.048
10000000000	4306	0.0430	0.043
100000000000	12491	0.0395	0.039
1000000000000	36249	0.0362	0.036

Hadamard y de la Vallée Poussin probaron en 1896, usando métodos basados en análisis complejo, el

Teorema 1.2 (Teorema de los Números Primos) *Sea* $\mathrm{li}(x)=\displaystyle\int_2^x \frac{dt}{\ln(t)}.\,\pi(x) \sim \mathrm{li}(x),$ *es decir,* $\lim_{x\to\infty}\dfrac{\pi(x)}{\mathrm{li}(x)}=1$

La conjetura de Legendre era $\pi(x) \sim x/\ln(x)$. Esta expresión se usa mucho cuando se hacen estimaciones "gruesas":

Teorema 1.3 $\text{li}(x) \sim x/\ln(x)$, *es decir* $\lim_{x\to\infty} \dfrac{\pi(x)}{x/\ln(x)} = 1$

Prueba: Para relacionar $\text{li}(x)$ con $x/\ln(x)$, integramos por partes,

$$\begin{aligned}
\text{li}(x) &= \int_2^x \frac{dt}{\ln(t)}, \quad \text{tomamos} \quad u = 1/\ln(t) \quad \text{y} \quad dv = dt,\\
&= \left.\frac{t}{\ln(t)}\right|_2^x - \int_2^x t\cdot\frac{-1/t\,dt}{\ln^2(t)}\\
&= \frac{x}{\ln(x)} + \int_2^x \frac{dt}{\ln^2(t)} + K_1, \quad \text{tomamos} \quad u = 1/\ln^2(t) \quad \text{y} \quad dv = dt,\\
&= \frac{x}{\ln(x)} + \frac{x}{\ln^2(x)} + 2\int_2^x \frac{dt}{\ln^3(t)} + K_2
\end{aligned}$$

Ahora vamos a mostrar que $2\displaystyle\int_2^x \frac{dt}{\ln^3(t)} + K_2 = O\left(\frac{x}{\ln^2 x}\right)$. Para esto, vamos a usar el hecho de que $\sqrt{x} = O\left(x/\ln^2 x\right)$.

Primero que todo observemos que solo necesitamos mostrar que $\displaystyle\int_2^x \frac{dt}{\ln^3(t)} = O\left(\frac{x}{\ln^2(x)}\right)$ pues como $\dfrac{x}{\ln^2(x)}$ tiende a infinito, podemos despreciar K_2. Además podemos ajustar la constante involucrada en la definición de la $O-$grande para "absorver" el coeficiente 2.

Como $\displaystyle\int_2^x = \int_2^e + \int_e^{\sqrt{x}} + \int_{\sqrt{x}}^x = K + \int_e^{\sqrt{x}} + \int_{\sqrt{x}}^x$, nos vamos a concentrar en estas dos últimas integrales.

Puesto que $e \leq t \implies \dfrac{1}{\ln^3 t} < 1$. Luego,

$$\int_e^{\sqrt{x}} \frac{dt}{\ln^3(t)} < \int_e^{\sqrt{x}} 1\,dt = \sqrt{x} - e < \sqrt{x}, \quad \text{es decir,} \quad \int_e^{\sqrt{x}} \frac{dt}{\ln^3(t)} = O\left(x/\ln^2 x\right).$$

Ahora, la otra integral. Puesto que $t < x$ entonces $\frac{x}{t} > 1.$ Multiplicando la segunda integral por $\frac{x}{t}$ obtenemos,

$\int_{\sqrt{x}}^{x} \frac{dt}{\ln^3(t)} < x \int_{\sqrt{x}}^{x} \frac{dt}{t\,\ln^3(t)}.$

Usando la sustitución $u = \ln t,$

$$\begin{aligned} x\int_{\sqrt{x}}^{x} \frac{dt}{t\,\ln^3(t)} &= x\left(\frac{\ln^{-2} t}{-2}\right)\Bigg|_{\sqrt{x}}^{x} \\ &= x\left(\frac{1}{2\ln^2\sqrt{x}} - \frac{1}{2\ln^2 x}\right) \\ &< x\,\frac{1}{2\ln^2\sqrt{x}} = \frac{x}{\ln^2 x} = O\left(x/\ln^2 x\right) \end{aligned}$$

Finalmente, $\mathrm{li}(x) = \frac{x}{\ln x} + O\left(x/\ln^2 x\right).$

La definición de $O-$grande nos permite dividir a ambos lados por $x/\ln x,$ entonces

$$\frac{\mathrm{li}(x)}{x/\ln x} = 1 + O(1/\ln x)$$

y como $1/\ln x \longrightarrow 0$ conforme $x \longrightarrow \infty,$

$$\mathrm{li}(x) \sim x/\ln(x)$$

como queríamos.

$\lim_{x\to\infty} \frac{\pi(x)}{x/\ln(x)} = 1$ debe entenderse en el sentido de que $x/\ln(x)$ aproxima $\pi(x)$ con un *error relativo* que se aproxima a cero conforme $x \to \infty$, aunque el error absoluto nos puede parecer muy grande.

Por ejemplo, si $n = 10^{13}$ (un número pequeño, de unos 13 dígitos solamente) entonces, una estimación de $\pi(10^{13}) = 346065536839$ sería $10^{13}/\ln(10^{13}) \approx 334072678387.$ El error relativo es $(\pi(n) - n/\ln(n))/\pi(n) = 0.034,$ es decir un 3.4%.

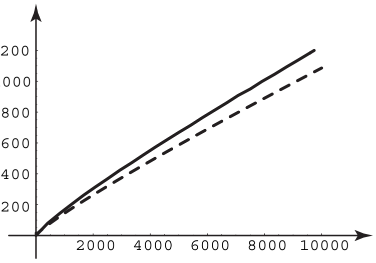


Figura 1.1 Comparando $x/\ln(x)$ con $\pi(x)$.

1.2.3 Teorema de Mertens.

¿Qué mide $\left(1-\frac{1}{2}\right)\left(1-\frac{1}{3}\right)\left(1-\frac{1}{5}\right)\cdots = \prod_{\substack{2\leq p\leq G,\\ p \text{ primo}}}\left(1-\frac{1}{p}\right)$?

$1/p$ es, a secas, la proporción de números en el conjunto $\{1,2,...,n\}$ que son divisibles por p. Luego $1-1/p$ sería la proporción de números en este conjunto que no son divisibles por p.

Aquí estamos asumiendo demasiado porque esta proporción no es exactamente $1/p$. Este número solo es una aproximación.

Si “ser divisible por p” es un evento independiente de “ser divisible por q”, $\left(1-\frac{1}{p}\right)\left(1-\frac{1}{q}\right)$ sería la proporción de números en el conjunto $\{1,2,...,n\}$, que *no* son divisibles por p ni por q.

En general, $\prod_{\substack{2\leq p\leq G,\\ p \text{ primo},}}\left(1-\frac{1}{p}\right)$ sería una estimación de la proporción de números en el conjunto $\{1,2,...,n\}$, que son divisibles por ninguno de los primos menores o iguales a G: Esto si tiene utilidad práctica, como veremos más adelante.

Hay que tener algunos cuidados con esta fórmula. Si la "probabilidad" de que un que un número n sea primo es la probabilidad de que no sea divisible por un primo $p \leq \sqrt{x}$, entonces podríamos concluir erróneamente que $\Pr[X es primo] = \prod_{\substack{2\leq p\leq\sqrt{x},\\ p\text{ primo}}} \left(1-\frac{1}{p}\right)$

Esta conclusión no es correcta pues $\prod_{\substack{2\leq p\leq\sqrt{x},\\ p\text{ primo}}} \left(1-\frac{1}{p}\right) \not\sim 1/\ln(x)$ como estable el Teorema de Mertens (Teorema 1.4).

EJEMPLO 1.5 Hagamos un experimento. Sea $d_n = \#\{m \leq n : m \text{ es divisible por } 2, 3, 5, \text{o } 7\}$.

n	d_n	d_n/n
103790	80066	0.7714230658059543
949971	732835	0.7714288120374201
400044	308605	0.7714276429592745
117131	90359	0.7714354013881893
124679	96181	0.7714290297483939

Tabla 1.4

La proporción de números naturales $\leq n$ divisibles por $2,3,5$ es ≈ 0.7714. Así, $1-0.7714 = 0.2286$ es la proporción de números en $\{1,2,...,n\}$ que *no* son divisibles por los primos $2,3,5$ y 7.

Y efectivamente, $\left(1-\frac{1}{2}\right)\left(1-\frac{1}{3}\right)\left(1-\frac{1}{5}\right)\left(1-\frac{1}{7}\right) = 0.228571.$

Si intentamos calcular el producto para cantidades cada vez grandes de primos, rápidamente empezaremos a tener problemas con el computador. En vez de esto, podemos usar el

Teorema 1.4 (Fórmula de Mertens)

$$\prod_{\substack{2\leq p\leq x,\\ p \text{ primo}}} \left(1-\frac{1}{p}\right) = \frac{e^{-\gamma}}{\ln(x)} + O(1/\ln(x)^2), \quad \gamma \text{ es la constante de Euler}$$

Para efectos prácticos consideramos la expresión

$$\prod_{\substack{2\leq p\leq x,\\ p \text{ primo}}} \left(1-\frac{1}{p}\right) \sim \frac{e^{-\gamma}}{\ln(x)} \approx \frac{0.5615}{\ln(x)} \quad \text{si } x \longrightarrow \infty \tag{1.1}$$

Sustituyendo en (1.1), x por $x^{0.5}$ encontramos que

$$\prod_{\substack{2\leq p\leq \sqrt{x},\\ p \text{ primo}}} \left(1-\frac{1}{p}\right) \sim \frac{2e^{-\gamma}}{\ln(x)} \approx \frac{1.12292}{\ln(x)}, \quad \text{si } x \longrightarrow \infty$$

EJEMPLO 1.6 Usando la fórmula de Mertens.

x	$\prod_{\text{primos } p\leq\sqrt{x}} (1-1/p)$	$\frac{2e^{-\gamma}}{\ln(x)}$
100000	0.0965	0.0975
100000000000000	0.034833774529614024	0.03483410793219253

Tabla 1.5

También, multiplicando (1.1) por 2, la fórmula

$$\prod_{\substack{3\leq p,\\ p \text{ primo}}}^{G} \left(1-\frac{1}{p}\right) \sim \frac{2e^{-\gamma}}{\ln(G)} \approx \frac{1.12292}{\ln(G)}$$

nos daría la proporción aproximada de números impares que no tienen un factor primo $\leq G$.

EJEMPLO 1.7 Calculando la proporción aproximada de impares sin factores primos $\leq G$.

G	Proporción approx de impares sin factores primos $\leq G$.
100	0.243839
1000	0.162559
10000	0.121919
100000	0.0975355
1000000	0.0812796
10000000	0.0696682
100000000	0.0609597
1000000000	0.0541864
10000000000	0.0487678

Tabla 1.6

Esta tabla nos informa que "típicamente", los números grandes tienen factores primos pequeños.

En resumen: El teorema de los números primos establece que $\pi(x)$ es aproximadamente igual a $x/\ln x$ en el sentido de que $\pi(x)/(x/\ln x)$ converge a 1 conforme $x \longrightarrow \infty$. Se cree que la densidad de primos en las cercanías de x es aproximadamente $1/\ln x$, es decir, un entero tomado aleatoriamente en las cercanías de x tiene una probabilidad $1/\ln x$ de ser primo.

El producto $\prod_{\substack{3 \leq p,\\ p \text{ primo}}}^{G} \left(1 - \frac{1}{p}\right)$ se puede usar para obtener la proporción aproximada de números impares que no tienen un factor primo $\leq G$.

También podemos estimar otras cosas. Para contar la cantidad de primos que hay entre $\sqrt{x}$ y x: Escribimos todos los números desde 2 hasta x, luego quitamos el 2 y todos sus múltiplos, luego quitamos el 3 y todos sus múltiplos, luego quitamos el 5 y todos sus múltiplos, seguimos así hasta llegar al último primo $p_k \leq \sqrt{x}$ el cual quitamos al igual que sus múltiplos. Como cualquier entero compuesto n, entre $\sqrt{x}$ y x, tiene que tener un factor primo $\leq \sqrt{n}$ entonces este entero fue quitado a esta altura del proceso. Lo que nos queda son solo los primos entre $\sqrt{x}$ y x. Este proceso de colado es una variación de la Criba ("colador") de Eratóstenes.

Por ejemplo, podemos colar $\{2,...12\}$ para dejar solo los primos entre $[\sqrt{12}\,] = 3$ y 12 :

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

así que solo quedan $7, 11$.

Para hacer un "colado aleatorio" para estimar la cantidad de primos entre $\sqrt{x}$ y x, podemos ver $\displaystyle\prod_{p\leq\sqrt{x}}\left(1-\frac{1}{p}\right)$ como una estimación de la proporción de números sin factores primos inferiores a $\sqrt{x}$. Entonces $\displaystyle\prod_{p\leq\sqrt{x}}\left(1-\frac{1}{p}\right)x$ es aproximadamente la cantidad de primos entre $\sqrt{x}$ y x.

La interpretación probabilística en muy delicada: Si vemos $\displaystyle\prod_{p\leq\sqrt{x}}\left(1-\frac{1}{p}\right)$ como la "probabilidad" de que un número no tenga divisores primos inferiores a x, entonces, por el teorema de los números primos, $\displaystyle\prod_{p\leq\sqrt{x}}\left(1-\frac{1}{p}\right)x \sim x/\ln(x)$. Pero esto no es correcto: El teorema de Mertens dice que

$$\prod_{p\leq\sqrt{x}}\left(1-\frac{1}{p}\right)x \ \sim\ \frac{2xe^{-\gamma}}{\ln(x)} \not\sim x/\ln(x)$$

La discrepancia la produce el hecho de que la divisibilidad por diferentes primos no constituyen " eventos suficientemente independientes" y tal vez el factor $e^{-\gamma}$ cuantifica esto en algún sentido. Otra manera de verlo es observar que el colado de Eratóstenes que usamos deja una fracción $1/\ln(x)$ de primos sin tocar, mientras que el colado aleatorio deja una fracción más grande, $1.123/\ln(x)$.

EJEMPLO 1.8 Si $x = 100000000000000$, $\pi(x) = 3204941750802$ y $\pi(\sqrt{x}) = 664579$. Los primos entre $\sqrt{x}$ y x son 3204941086223 mientras que

$$\prod_{p\leq\sqrt{x}}\left(1-\frac{1}{p}\right)x \sim \frac{2xe^{-\gamma}}{\ln(x)} = 3483410793219.25$$

1.2.4 El número de primos en una progresión aritmética.

Sean a y b números naturales y consideremos los enteros de la progresión aritmética $an+b,\;\; n = 0,1,2,\ldots$. De los números inferiores x en esta progresión, ¿cuántos son primos?. Si $\pi_{a,b}(x)$ denota la cantidad de primos $\leq x$ en esta sucesión, tenemos

Teorema 1.5 (Teorema de Dirichlet) *Si a y b son primos relativos entonces*

$$\lim_{x\to\infty}\frac{\pi_{a,b}(x)}{li(x)} = \frac{1}{\varphi(a)}$$

φ es la función de Euler,

$$\varphi(m) = \text{ número de enteros positivos } \leq m \text{ y coprimos con } m$$

En particular $\varphi(6) = 2$ y $\varphi(10) = 4$. De acuerdo con el teorema de Dirichlet, “en el infinito” (es decir tomando el límite), un 50% de primos están en cada una de las sucesiones $6n+1$ y $6n-1$ mientras que un 25% de primos se encuentra en cada una de las cuatro sucesiones $10n\pm1$ y $10n\pm3$. Se puede probar también que si $\text{mcd}\,(a,b) = 1$ entonces la sucesión $an+b$ tiene un número infinito de primos.

En realidad, las sucesiones $6n+1$ y $6n-1$ contienen todos los primos pues todo primo es de la forma $6k+1$ o $6k-1$. En efecto, cualquier natural es de la forma $6k+m$ con $m\in\{0,1,2,3,4,5\}$ (por ser “$\equiv \pmod 6$” una relación de equivalencia en $\mathbb{N}$, es decir parte $\mathbb{N}$ en seis clases). Ahora, todos los enteros $6k+m$ son claramente compuestos excepto para $m = 1$ y $m = 5$ por lo que si p es primo, entonces $p = 6k+1$ o $p = 6k+5 = 6q-1$ (si $q = k+1$), es decir p es de la forma $6k\pm1$.

1.2.5 Cantidad de factores primos de un número grande.

El teorema del límite central dice que si una población (continua o discreta) tiene media μ y varianza finita σ^2, la media muestral $\overline{X}$ tendrá una distribución que se aproxima a la normal.

Teorema 1.6 (Limite Central) *Si tenemos $X_1, X_2, \ldots, X_n$ variables aleatorias independientes, idénticamente distribuidas, con media μ y varianza σ^2, entonces, si n es suficientemente grande, la probabilidad de que $S_n = X_1 + X_2 + \ldots + X_n$ esté entre $n\mu + \alpha\sigma\sqrt{n}$ y $n\mu + \beta\sigma\sqrt{n}$ es*

$$\frac{1}{\sqrt{2\pi}} \int_{\alpha}^{\beta} e^{-t^2/2}\, dt$$

EJEMPLO 1.9 Si lanzamos una moneda limpia unas 10000 veces, uno esperaría que aproximadamente 5000 veces salga “cara”. Si denotamos con $X_i = 1$ el evento “en el lanzamiento i sale cara”, como la probabilidad que asumimos para el evento “sale cara” es $1/2$, entonces $n\mu = n \cdot 0.5 = 5000$ y $\sigma = \sqrt{n \cdot 0.25} = 5$. Luego, para calcular la probabilidad de que el número de caras esté entre 4850 y 5150, debemos calcular los límites α y β. Por razones de ajuste del caso discreto al caso continuo, se usa un factor de corrección de $1/2$. Resolviendo, $5000 + (\alpha)\sqrt{50} = 4850 - 0.5 \Longrightarrow \alpha = -3.01$ $5000 + (\alpha)\sqrt{50} = 5150 + 0.5 \Longrightarrow \beta = 3.01$

$$\frac{1}{\sqrt{2\pi}} \int_{-3.01}^{3.01} e^{-t^2/2}\, dt = 0.997388$$

Así, la probabilidad de que el número de caras esté entre 4850 y 5150 es de 0.997388

Si $\omega(n)$ denota la cantidad de factores primos de n, esta función se puede denotar como una suma de funciones $\rho_p(n)$, estadísticamente independientes, definidas por

$$\rho_p(n) = \begin{cases} 1 & \text{si} \quad p|n \\ 0 & \text{si} \quad p \nmid n \end{cases}$$

Esto sugiere que la distribución de los valores de $\omega(n)$ puede ser dada por la ley normal (con media $\ln\ln n$ y desviación estándar $\sqrt{\ln\ln n}$).

Mark Kac y Paul Erdös probaron que la densidad del conjunto de enteros n para el cual el número de divisores primos $\omega(n)$ está comprendido entre $\ln\ln n+\alpha\sqrt{\ln\ln n}$ y $\ln\ln n+\beta\sqrt{\ln\ln n}$, es

$$\frac{1}{\sqrt{2\pi}}\int_{\alpha}^{\beta} e^{-t^2/2}\,dt$$

es decir, el número de divisores primos está distribuido de acuerdo a la ley normal.

Teorema 1.7 *Denotamos con $N(x,a,b)$ la cantidad de enteros n en $\{3,4,\ldots,x\}$ para los cuales*

$$\alpha \leq \frac{\omega(n)-\ln\ln n}{\sqrt{\ln\ln n}} \leq \beta$$

Entonces, conforme $x\to\infty$,

$$N(x,a,b)=(x+o(x))\frac{1}{\sqrt{2\pi}}\int_{\alpha}^{\beta} e^{-t^2/2}\,dt$$

Para efectos prácticos, hacemos referencia a este teorema en estos términos

> Típicamente, el número de factores primos, inferiores a x, de un número n suficientemente grande es aproximadamente $\ln\ln x$.

1.3 Criba de Eratóstenes: Cómo colar números primos.

La criba[2] de Eratóstenes es un algoritmo que permite "colar" todos los números primos menores que un número natural dado n, eliminando los números compuestos de la lista $\{2,\ldots,n\}$. Es simple y razonablemente eficiente.

Primero tomamos una lista de números $\{2,3,\ldots,n\}$ y eliminamos de la lista los múltiplos de 2. Luego tomamos el primer entero después de 2 que no fue borrado (el 3) y eliminamos de la lista sus múltiplos, y así sucesivamente. Los números que permanecen en la lista son

---

[2] Criba, tamiz y zaranda son sinónimos. Una criba es un herramienta que consiste de un cedazo usada para limpiar el trigo u otras semillas, de impurezas. Esta acción de limpiar se le dice cribar o tamizar.

los primos $\{2,3,5,7,\ldots\}$.

EJEMPLO 1.10 Primos menores que $n=10$

Lista inicial	2	3	4	5	6	7	8	9	10
Eliminar múltiplos de 2	2	3	$\cancel{4}$	5	$\cancel{6}$	7	$\cancel{8}$	9	$\cancel{10}$
Resultado	2	3	5	7	9				
Eliminar múltiplos de 3	2	3	5	7	$\cancel{9}$				
Resultado	2	3	5	7					

Primer refinamiento: Tachar solo los impares

Excepto el $2,$ los pares no son primos, así que podríamos "tachar" solo sobre la lista de impares $\leq n$:

$$\{3,5,9,\ldots,\}=\left\{2i+3\ :\ \ i=0,1,\ldots\left\llbracket\frac{n-3}{2}\right\rrbracket\right\}$$

El último impar es n o $n-1.$ En cualquier caso, el último impar es $2\cdot\left\llbracket\frac{n-3}{2}\right\rrbracket+3$ pues,

Si n es impar, $n=2k+1$ y $\left\llbracket\frac{n-3}{2}\right\rrbracket=k-1\implies 2(k-1)+3=n.$

Si n es par, $n=2k$ y $\left\llbracket\frac{n-3}{2}\right\rrbracket=k-2\implies 2(k-2)+3=2k-1=n-1.$

Segundo refinamiento: Tachar de p_k^2 en adelante

En el paso $k-$ésimo hay que tachar los múltiplos del primo p_k desde p_k^2 en adelante.

Esto es así pues en los pasos anteriores se ya se tacharon $3\cdot p_k,\ 5\cdot p_k,\ldots,p_{k-1}\cdot p_k.$

Por ejemplo, cuando nos toca tachar los múltiplos del primo $7,$ ya se han eliminado los múltiplos de $2,3$ y $5,$ es decir, ya se han eliminado $2\cdot 7,\ 3\cdot 7,\ 4\cdot 7,\ 5\cdot 7$ y $6\cdot 7.$ Por eso iniciamos en $7^2.$

Tercer refinamiento: Tachar mientras $p_k^2\leq n$

En el paso $k-$ésimo hay que tachar los múltiplos del primo p_k solo si $p_k^2\leq n.$ En otro caso, nos detenemos ahí.

¿Porque?. En el paso $k-$ ésimo tachamos los múltiplos del primo p_k desde p_k^2 en adelante, así que si $p_k^2 > n$ ya no hay nada que tachar.

EJEMPLO 1.11 Encontrar los primos menores que $20.$ El proceso termina cuando el cuadrado del mayor número confirmado como primo es $< 20.$

1. La lista inicial es $\{2,3,5,7,9,11,13,15,17,19\}$
2. Como $3^2 \leq 20,$ tachamos los múltiplos de 3 desde $3^2 = 9$ en adelante:

$$\{2,3,5,7,\cancel{9},11,13,\cancel{15},17,19\}$$

3. Como $5^2 > 20$ el proceso termina aquí.
4. Primos $< 20:$ $\{2,3,5,7,11,13,17,19\}$

1.3.1 Algoritmo e implementación.

1. Como ya vimos, para colar los primos en el conjunto $\{2,3,\ldots,n\}$ solo consideramos los impares:

$$\left\{2i+3 \ : \quad i = 0,1,\ldots \left[\!\!\left[\frac{n-3}{2}\right]\!\!\right]\right\} = \{3,5,7,9,\ldots\}$$

2. Por cada primo $p = 2i+3$ (tal que $p^2 < n$), debemos eliminar los *múltiplos impares* de p menores que $n,$ a saber

$$(2k+1)p = (2k+1)(2i+3), \quad k = i+1, i+2, \ldots$$

Observe que si $k = i+1$ entonces el primer múltiplo en ser eliminado es $p^2 = (2i+3)(2i+3),$ como debe ser.

Esto nos dice que para implementar el algoritmo solo necesitamos un arreglo (booleano) de tamaño “`quo(n-3,2)`”. En Java se pone “`(n-3)/2`” y en VBA se pone “`(n-3)\2`”.

El arreglo lo llamamos `EsPrimo[i], i=0,1,...,(n-3)/2.`

Cada entrada del arreglo “`EsPrimo[i]`” indica si el número $2i+3$ es primo o no.

Por ejemplo

`EsPrimo[0] = true` pues $n=2\cdot 0+3=3$ es primo,

`EsPrimo[1] = true` pues $n=2\cdot 1+3=5$ es primo,

`EsPrimo[2] = true` pues $n=2\cdot 2+3=7$ es primo,

`EsPrimo[3] = false` pues $n=2\cdot 3+3=9$ no es primo.

Si el número $p=2i+3$ es primo entonces $i=(p-3)/2$ y

```
EsPrimo[(p-3)/2] = true.
```

Si sabemos que $p=2i+3$ es primo, debemos poner

```
EsPrimo[((2k+1)(2i+3) - 3)/2] = false
```

pues estas entradas representan a los múltiplos $(2k+1)(2i+3)$ de $p.$ Observe que cuando $i=0,1,2$ tachamos los múltiplos de $3,5$ y 7; cuando $i=3$ entonces $2i+3=9$ pero en este momento `esPrimo[3]=false` así que proseguimos con $i=4,$ es decir, proseguimos tachando los múltiplos de $11.$

En resumen: Antes de empezar a tachar los múltiplos de $p = 2i + 3$ debemos preguntar si `esPrimo[i]=true.`

Algoritmo 1.1: Criba de Eratóstenes

```
Entrada: $n \in \mathbb{N}$
Resultado: Primos entre $2$ y $n$
máx$= (n-3)/2$;
boolean esPrimo$[i]$,   $i = 1,2,...,$máx;
for $i = 1,2,...,máx$ do
    esPrimo$[i]$ =True;
$i = 0$;
while $(2i+3)(2i+3) \leq n$ do
    $k = i+1$;
    if esPrimo(i) then
        while $(2k+1)(2i+3) \leq n$ do
            esPrimo$[((2k+1)(2i+3)-3)/2]$ =False;
            $k = k+1$;
    $i = i+1$;
Imprimir;
for $j = 1,2,...,máx$ do
    if esPrimo$[j]$ =True then
        Imprima $j$
```

1.3.1.1 Implementación en Java. Vamos a agregar un método a nuestra clase "`Teoria_Numeros`". El método recibe el número natural $n > 2$ y devuelve un vector con los números primos $\leq n$. Para colar los números compuestos usamos un arreglo

```
boolean [] esPrimo = new boolean[(n-3)/2].
```

Al final llenamos un vector con los primos que quedan.

```
import java.math.BigInteger;
public class Teoria_Numeros
{   ...
 public static Vector HacerlistaPrimos(int n)
```

```
{
          Vector        salida = new Vector(1);
          int k   = 1;
          int max = (n-3)/2;
          boolean[]   esPrimo  = new boolean[max+1];

          for(int i = 0; i <= max; i++)
              esPrimo[i]=true;

          for(int i = 0; (2*i+3)*(2*i+3) <= n; i++)
          {
            k = i+1;
            if(esPrimo[i])
            {
              while( ((2*k+1)*(2*i+3)) <= n)
              {
                  esPrimo[((2*k+1)*(2*i+3)-3)/2]=false;
                  k++;
              }
            }
           }
           salida.addElement(new Integer(2));
           for(int i = 0; i <=max; i++)
           { if(esPrimo[i])
             salida.addElement(new Integer(2*i+3));
           }
           salida.trimToSize();
           return salida;
   }
   public static void main(String[] args)
   {
     System.out.println("\n\n");
    //-------------------------------------------------------------
     int    n = 100;
     Vector primos;
            primos = HacerlistaPrimos(n);
     //Cantidad de primos <= n
     System.out.println("Primos <="+ n+": "+primos.size()+"\n");
     //imprimir vector (lista de primos)
```

```
  for(int p = 1; p < primos.size(); p++)
  {
      Integer num = (Integer)primos.elementAt(p);
      System.out.println(""+(int)num.intValue());
  }
//------------------------------------------------------------------
  System.out.println("\n\n");
}}
```

1.3.1.2 *Uso de la memoria* En teoría, los arreglos pueden tener tamaño máximo `Integer.MAX_INT` $= 2^{31}-1 = 2\,147\,483\,647$ (pensemos también en la posibilidad de un arreglo multidimensional!). Pero en la práctica, el máximo tamaño del array depende del hardware de la computadora. El sistema le asigna una cantidad de memoria a cada aplicación; para valores grandes de n puede pasar que se nos agote la memoria (veremos el mensaje "`OutOfMemory Error`"). Podemos asignar una cantidad de memoria apropiada para el programa "cribaEratostenes.java" desde la línea de comandos, si n es muy grande. Por ejemplo, para calcular los primos menores que $n = 100\,000\,000,$ se puede usar la instrucción

```
C:\usrdir> java -Xmx1000m -Xms1000m Teoria_Numeros
```

suponiendo que el archivo "`Teoria_Numeros.java`" se encuentra en `C:\usrdir`.

Esta instrucción asigna al programa una memoria inicial (Xmx) de 1000 MB y una memoria máxima (Xms) de 1000 MB (siempre y cuando existan tales recursos de memoria en nuestro sistema).

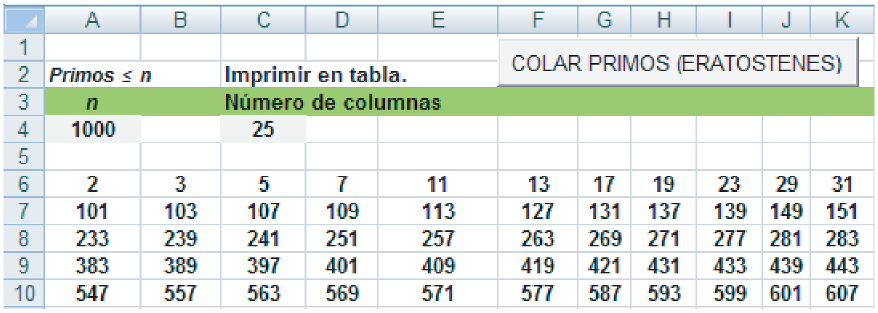
En todo caso hay que tener en cuenta los siguientes datos

n	Primos $\leq n$
10	4
100	25
1 000	168
10 000	1 229
100 000	9 592
1 000 000	78 498
10 000 000	664 579
100 000 000	5 761 455
1 000 000 000	50 847 534

10 000 000 000	455 052 511
100 000 000 000	4 118 054 813
1 000 000 000 000	37 607 912 018
10 000 000 000 000	346 065 536 839

1.3.1.3 Implementación en Excel. Para la implementación en Excel usamos un cuaderno como el de la figura (1.2).

El número n lo leemos en la celda $(4,1)$. El código VBA incluye una subrutina para imprimir en formato de tabla, con `ncols` columnas. Este último parámetro es opcional y tiene valor default 10. También incluye otra subrutina para limpiar las celdas para que no haya confusión entre los datos de uno y otro cálculo.



	A	B	C	D	E	F	G	H	I	J	K
1											
2	*Primos ≤ n*		Imprimir en tabla.								
3	*n*		Número de columnas								
4	1000		25								
5											
6	2	3	5	7	11	13	17	19	23	29	31
7	101	103	107	109	113	127	131	137	139	149	151
8	233	239	241	251	257	263	269	271	277	281	283
9	383	389	397	401	409	419	421	431	433	439	443
10	547	557	563	569	571	577	587	593	599	601	607

Figura 1.2 Primos $\leq n$.

Imprimir en formato de tabla

Para esto usamos la subrutina

```
Imprimir(ByRef Arr() As Long, fi, co, Optional ncols As Variant).
```

La impresión inicia en la celda “`(fi,co)`”. Para imprimir en formato de tabla usamos `Cells(fi + k, co + j)` con el número de columnas `j` variando de 0 a `ncols-1`. Para reiniciar `j` en cero actualizamos `j` con `j = j Mod ncols`. Para cambiar la fila usamos `k`. Esta variable aumenta en 1 cada vez que `j` llega a `ncols-1`. Esto se hace con división entera: `k = k + j \ (ncols - 1)`

Subrutina para borrar celdas

Para esto usamos la subrutina

`LimpiaCeldas(fi, co, ncols).`

Cuando hacemos cálculos de distinto tamaño es conveniente borrar las celdas de los cálculos anteriores para evitar confusiones. La subrutina inicia en la celda `(fi,co)` y borra `ncols` columnas a la derecha. Luego pasa a la siguiente fila y hace lo mismo. Prosigue de igual forma hasta que encuentre la celda `(fi+k,co)` vacía.

```
Option Explicit
    Private Sub CommandButton1_Click()
    Dim n, ncols
    n = Cells(4, 1)
    ncols = Cells(4, 3)
    Call Imprimir(ERATOSTENES(n), 6, 1, ncols)
End Sub

' Imprime arreglo en formato de tabla con "ncols" columnas,
' iniciando en la celda (fi,co)
Sub Imprimir(ByRef Arr() As Long, fi, co, Optional ncols As Variant)
Dim i, j, k
    ' Limpia celdas
    ' f        = fila en que inicia la limpieza
    ' co       = columna en q inicia la limpieza
    ' ncols    = nmero de columnas a borrar
Call LimpiaCeldas(fi, co, ncols)
If IsMissing(ncols) = True Then
            ncols = 10
End If
'Imprimir
j = 0
k = 0
For i = 0 To UBound(Arr)
    Cells(fi + k, co + j) = Arr(i)
    k = k + j \ (ncols - 1) 'k aumenta 1 cada vez que j llegue a ncols-1
    j = j + 1
    j = j Mod ncols         'j=0,1,2,...,ncols-1
```

```
Next i

End Sub


Function ERATOSTENES(n) As Long()
Dim i, j, k, pos, contaPrimos
Dim max As Long
Dim esPrimo() As Boolean
Dim Primos() As Long
max = (n - 3) \ 2 ' Divisin entera
ReDim esPrimo(max + 1)
ReDim Primos(max + 1)
For i = 0 To max
    esPrimo(i) = True
Next i
contaPrimos = 0
Primos(0) = 2 'contado el 2
j = 0
While (2 * j + 3) * (2 * j + 3) <= n
    k = j + 1
    If esPrimo(j) Then
        While (2 * k + 1) * (2 * j + 3) <= n
            pos = ((2 * k + 1) * (2 * j + 3) - 3) \ 2
            esPrimo(pos) = False
            k = k + 1
        Wend
    End If
j = j + 1
Wend

For i = 0 To max
   If esPrimo(i) Then
        contaPrimos = contaPrimos + 1 '3,5,...
        Primos(contaPrimos) = 2 * i + 3
    End If
Next i

ReDim Preserve Primos(contaPrimos) 'Cortamos el vector
ERATOSTENES = Primos()
End Function
```

```
Private Sub LimpiaCeldas(fi, co, nc)
Dim k, j
k = 0
While LenB(Cells(fi + k, co)) <> 0   ' celda no vac\'ia
      For j = 0 To nc
          Cells(fi + k, co + j) = ""  ' borra la fila hasta nc columnas
      Next j
      k = k + 1
Wend
End Sub
```

1.3.2 Primos entre m y n.

Para encontrar todos los primos entre m y n (con $m<n$) procedemos como si estuvieramos colando primos en la lista $\{2,3,...,n\}$, solo que eliminamos los múltiplos que están entre m y n: Eliminamos los múltiplos de los primos p para los cuales $p^2\leq n$ (o también $p\leq\sqrt{n}$), que están entre m y n.

Múltiplos de p entre m y n

Para los primos p inferiores a $\sqrt{n}$, buscamos el primer múltiplo de p entre m y n.

Si $m-1=pq+r,\ 0\leq r<p \ \Longrightarrow\ p(q+1)\geq m$

Así, los múltiplos de p mayores o iguales a m son

$$p(q+1),\ p(q+2),\ p(q+3),\ ...\ \text{ con }\ q=\text{quo}(m-1,p)$$

EJEMPLO 1.12 Para encontrar los primos entre $m=10$ y $n=30$, debemos eliminar los múltiplos de los primos $\leq\sqrt{30}\approx 5$. Es decir, los múltiplos de los primos $p=2,3,5$.

Como $10-1=2\cdot 4+1$, el 2 elimina los números $2(4+k)=8+2k,\ k\geq 1$; es decir $\{10,12,...,30\}$

Como $10-1=3\cdot 3+0$, el 3 elimina los números $3(3+k)=9+3k,\ k\geq 1$; es decir $\{12,15,18,21,24,27,30\}$

Como $10-1=5\cdot 1+4$, el 5 elimina los números $5(1+k)=5+5k,\ k\geq 1$; es decir $\{10,15,20,25.\}$

Finalmente nos quedan los primos $11, 13, 17, 19, 23, 29$.

1.3.2.1 Algoritmo. Como antes, solo consideramos los impares entre m y n. Si ponemos

$min = \text{quo}(m+1-3, 2)$ y $max = \text{quo}(n-3, 2)$

entonces $2 \cdot min + 3$ es el primer impar $\geq m$ y $2 \cdot max + 3$ es el primer impar $\leq n$. Así, los impares entre m y n son los elementos del conjunto $\{2 \cdot i + 3 \; : \; i = min, ..., max\}$

Como antes, usamos un arreglo booleano `esPrimo(i)` con $i = min, ..., max$. `esPrimo(i)` representa al número $2 \cdot i + 3$.

EJEMPLO 1.13 Si $m = 11$ y 20, $[\![(m+1-3)/2]\!] = 4$ y $[\![(n-3)/2]\!] = 8$. Luego $2 \cdot 4 + 3 = 11$ y $2 \cdot 8 + 3 = 19$.

Para aplicar el colado necesitamos los primos $\leq \sqrt{n}$. Esta lista de primos la obtenemos con la función `Eratostenes(isqrt(n))`. Aquí hacemos uso de la función `isqrt(n)` (algoritmo **??**).

Para cada primo p_i en la lista,

1. si $m \leq p_i^2$, tachamos los múltiplos impares de p_i como antes,

```
if $m \leq p_i^2$ then
    $k = (p_i - 1)/2$;
    while $(2k+1)p_i \leq n$ do
        esPrimo$[((2k+1)p_i - 3)/2]$ =False;
        $k = k + 1$;
```

Note que si $k = (p_i - 1)/2$ entonces $(2k+1)p_i = p_i^2$

2. si $p_i^2 < m,$ tachamos desde el primer múltiplo impar de p_i que supere m :

```
if $p_i^2 < m$ then
    $q = (m-1)/p$;
    $q_2 = \text{rem}(q,2)$;
    $k = q_2$;
    $mp = (2k+1-q_2+q)\cdot p_i$;
    while $mp \leq n$ do
        esPrimo[$(mp-3)/2$] =False;
        $k = k+1$;
        $mp = (2k+1-q_2+q)\cdot p_i$
```

Los múltiplos de p_i que superan m son $p_i(q+k)$ con $q = \text{quo}(m-1,p)$. De esta lista solo nos interesan los múltiplos impares. Esto requiere un pequeño análisis aritmético.

Como p_i es impar, $p_i(q+k)$ es impar solo si $q+k$ es impar. Poniendo $q_2 = \text{rem}(q,2)$ entonces $(2k+1-q_2\ +\ q)$ es impar si $k = q_2,\ q_2+1,\ldots$. En efecto,

$$2k+1-q_2\ +\ q = \begin{cases} 2k+1\ +\ q & \text{si} \quad q \quad \text{es par. Aquí} \quad k = q_2 = 0,1,\ldots \\ 2k\ +\ q & \text{si} \quad q \quad \text{es impar. Aquí} \quad k = q_2 = 1,2,\ldots \end{cases}$$

Luego, los múltiplos impares de p_i son los elementos del conjunto

$\{(2k+1-q_2\ +\ q)\cdot p\ \ :\ \ q_2 = \text{rem}(q,2)\ \ \text{y}\ \ k = q2, q2+1,\ldots\}$

La manera de tachar los múltiplos impares de p_i aparece arriba.

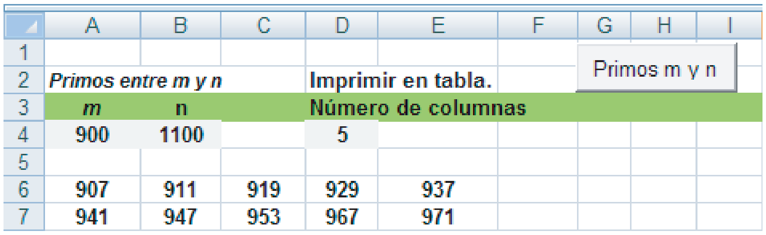
Ahora podemos armar el algoritmo completo.

Algoritmo 1.2: Colado de primos entre m y n.

```
Entrada: $n, m \in \mathbb{N}$ con $m < n$.
Resultado: Primos entre $m$ y $n$
Primo() = una lista de primos $\leq \sqrt{n}$;
$min = (m+1-3)/2$; $max = (n-3)/2$;
$esPrimo[i],\ i = min, \ldots, max$ ;
for $j = min, \ldots, max$ do
    $esPrimo[j]$ =True;
$np$ = cantidad de primos en la lista Primos;
Suponemos Primo(0) = 2;
for $i = 1, 2, \ldots, np$ do
    if $m \leq p_i^2$ then
        $k = (p_i - 1)/2$;
        while $(2k+1)p_i \leq n$ do
            esPrimo$[((2k+1)p_i - 3)/2]$ =False;
            $k = k+1$;
    if $p_i^2 < m$ then
        $q = (m-1)/p$;
        $q_2 = \text{rem}(q, 2)$;
        $k = q_2$;
        $mp = (2k+1-q_2+q) \cdot p_i$;
        while $mp \leq n$ do
            esPrimo$[(mp-3)/2]$ =False;
            $k = k+1$;
            $mp = (2k+1-q_2+q) \cdot p_i$
Imprimir;
for $j = min, \ldots, max$ do
    if $esPrimo[j]$ =$True$ then
        Imprima $2 * i + 3$
```

1.3.2.2 Implementación en Excel. Para la implementación en Excel usamos un cuaderno como el de la figura (1.3).

m y n los leemos en la celdas $(4,1)$, $(4,2)$. Como antes, el código VBA hace referencia a las subrutinas para imprimir en formato de tabla y limpiar las celdas (sección 1.3.1.3).



	A	B	C	D	E	F	G	H	I
1									
2	*Primos entre m y n*			Imprimir en tabla.					
3	*m*	n		Número de columnas					
4	900	1100		5					
5									
6	907	911	919	929	937				
7	941	947	953	967	971				

Figura 1.3 Primos $\leq n$.

En VBA Excel podemos declarar un arreglo que inicie en *min* y finalice en *max*, como el algoritmo. Por eso, la implementación es muy directa.

```
Option Explicit
Private Sub CommandButton1_Click()
Dim n, m, ncols
m = Cells(4, 1)
n = Cells(4, 2)
ncols = Cells(4, 4)
Call Imprimir(PrimosMN(m, n), 6, 1, ncols)
End Sub

Sub Imprimir(ByRef Arr() As Long, fi, co, Optional ncols As Variant)
...
End Sub

Function ERATOSTENES(n) As Long()
...
End Sub

Function isqrt(n) As Long
Dim xk, xkm1
If n = 1 Then
   xkm1 = 1
End If
```

```
If n > 1 Then
    xk = n
    xkm1 = n \ 2
    While xkm1 < xk
         xk = xkm1
         xkm1 = (xk + n \ xk) \ 2
    Wend
End If
isqrt = xkm1
End Function
' m < n
Function PrimosMN(m, n) As Long()
Dim i, j, k, pos, contaPrimos, mp, q, q2
Dim min, max
Dim esPrimo() As Boolean
Dim primo() As Long
Dim PrimosMaN() As Long

min = Int((m + 1 - 3) \ 2)
max = Int((n - 3) \ 2)

ReDim esPrimo(min To max)
ReDim PrimosMaN((n - m + 2) \ 2)
For i = min To max
    esPrimo(i) = True
Next i

primo = ERATOSTENES(isqrt(n))

For i = 1 To UBound(primo)          'primo(1)=3
    If m <= primo(i) * primo(i) Then
       k = (primo(i) - 1) \ 2
       While (2 * k + 1) * primo(i) <= n
       esPrimo(((2 * k + 1) * primo(i) - 3) \ 2) = False
       k = k + 1
       Wend
    End If
    If primo(i) * primo(i) < m Then
       q = (m - 1) \ primo(i)  'p(q+k)-> p*k
       q2 = q Mod 2
       k = q2
       mp = (2 * k + 1 - q2 + q) * primo(i)  'm\'ultiplos impares
```

```
        While mp <= n
            esPrimo((mp - 3) \ 2) = False
            k = k + 1
            mp = (2 * k + 1 - q2 + q) * primo(i)
       Wend
    End If
Next i

If m > 2 Then
    contaPrimos = 0
Else
    contaPrimos = 1
    PrimosMaN(0) = 2
End If

For i = min To max
   If esPrimo(i) Then
        PrimosMaN(contaPrimos) = 2 * i + 3
        contaPrimos = contaPrimos + 1 '3,5,...
    End If
Next i

If 1 <= contaPrimos Then
ReDim Preserve PrimosMaN(contaPrimos - 1)
Else
ReDim PrimosMaN(0)
End If

PrimosMN = PrimosMaN()
End Function
```

1.4 Factorización por ensayo y error.

1.4.1 Introducción

El método más sencillo de factorización (y muy útil) es el método de *factorización por ensayo y error* (FEE). Este método va probando con los posibles divisores de n hasta encontrar una factorización de este número.

En vez de probar con todos los posibles divisores de n (es decir, en vez de usar *fuerza bruta*) podemos hacer algunos refinamientos para lograr un algoritmo más eficiente en el sentido de reducir las pruebas a un conjunto de números más pequeño, en el que se encuentren los divisores pequeños de n.

1.4.2 Probando con una progresión aritmética.

Como estamos buscando factores pequeños de $n,$ podemos usar el siguiente teorema,

Teorema 1.8

Si $n \in \mathbb{Z}^+$ admite la factorización $n = ab,$ con $a, b \in \mathbb{Z}^+$ entonces $a \leq \sqrt{n}$ o $b \leq \sqrt{n}.$

Prueba. Procedemos por contradicción, si $a > \sqrt{n}$ y $b > \sqrt{n}$ entonces $ab > \sqrt{n}\sqrt{n} = n$ lo cual, por hipótesis, no es cierto.

Del teorema anterior se puede deducir que

- Si n no tiene factores d con $1 < d \leq \sqrt{n},$ entonces n es primo.

- Al menos uno de los factores de n es menor que $\sqrt{n}$ (no necesariamente todos). Por ejemplo $14 = 2 \cdot 7$ solo tiene un factor menor que $\sqrt{14} \approx 3.74166).$

De acuerdo al teorema fundamental de la aritmética, Cualquier número natural > 1 factoriza, de manera única (excepto por el orden) como producto de primos. Esto nos dice que la estrategia óptima de factorización sería probar con los primos menores que $\sqrt{n}.$ El problema es que si n es muy grande el primer problema sería que el cálculo de los primos de prueba duraría siglos (sin considerar los problemas de almacenar estos números).

Recientemente (2005) se factorizó un número de 200 cifras[3] (RSA-200). Se tardó cerca de 18 meses en completar la factorización con un esfuerzo computacional equivalente a 53 años de trabajo de un CPU 2.2 GHz Opteron.

1.4.3 Algoritmo.

Identificar si un número es primo es generalmente fácil, pero factorizar un número (grande) arbitrario no es sencillo. El método de factorización de un número N probando con divisores primos (“trial division”) consiste en probar dividir N con primos pequeños. Para esto se debe previamente almacenar una tabla suficientemente grande de números primos o generar la tabla cada vez. Como ya vimos en la criba de Eratóstenes, esta manera de proceder trae consigo problemas de tiempo y de memoria. En realidad es más ventajoso proceder de otra manera.

- Para hacer la pruebas de divisibilidad usamos los enteros 2, 3 y la sucesión $6k \pm 1,\ k = 1,2,\ldots$.

  Esta elección cubre todos los primos e incluye divisiones por algunos números compuestos (25,35,...) pero la implementación es sencilla y el programa suficientemente rápido (para números no muy grandes) que vale la pena permitirse estas divisiones inútiles.

- Las divisiones útiles son las divisiones por números primos, pues detectamos los factores que buscamos. Por el teorema de los números primos, hay $\pi(G) \approx G/\ln G$ números primos inferiores a $G,$ ésta sería la cantidad aproximada de divisiones útiles.

  Los números $2,\ 3$ y $6k \pm 1,\ k \in \mathbb{N}$ constituyen, hablando en grueso, una tercera parte de los naturales (note que $\mathbb{Z} = \bigcup \mathbb{Z}_6 = \bigcup \{\overline{0}, \overline{1}, \overline{2}, \overline{3}, \overline{4}, \overline{5}\}$, $\{\overline{1}, \overline{-1} = \overline{5}\}$ es una tercera parte). En $\{1,2,\ldots,G\}$ hay $\approx G/3$ de estos números. En estos $G/3$ números están los $\pi(G) \approx G/\ln G$ primos inferiores a $G,$ es decir, haríamos $\approx \dfrac{G/\ln G}{G/3} = 3/\ln G$ divisiones útiles.

---

[3] Se trata del caso más complicado, un número que factoriza como producto de dos primos (casi) del mismo tamaño.

Si probamos con todos los números, tendríamos que hacer $1/0.22 = 4.6$ más cálculos para obtener un 22% de divisiones útiles.

Cuando se juzga la rapidez de un programa se toma en cuenta el tiempo de corrida en el *peor caso* o se toma en cuenta el *tiempo promedio de corrida* (costo de corrida del programa si se aplica a muchos números). Como ya sabemos (por el Teorema de Mertens) hay un porcentaje muy pequeño de números impares sin divisores $\leq G$, así que en promedio, nuestra implementación terminará bastante antes de alcanzar el límite G (el "peor caso" no es muy frecuente) por lo que tendremos un programa con un comportamiento deseable.

Detalles de la implementación.

- Para la implementación necesitamos saber cómo generar los enteros de la forma $6k \pm 1$. Alternando el -1 y el 1 obtenemos la sucesión

$$5,\ 7,\ 11,\ 13,\ 17,\ 19,\ldots$$

que iniciando en 5, se obtiene alternando los sumandos 2 y 4. Formalmente, si $m_k = 6k-1$ y si $s_k = 6k+1$ entonces, podemos poner la sucesión como

$$7,\ 11,\ 13,\ldots,m_k,\ s_k,\ m_{k+1},\ s_{k+1},\ldots$$

Ahora, notemos que $s_k = m_k + 2$ y que $m_{k+1} = s_k + 4 = m_k + 6$. La sucesión es

$$7,\ 11,\ 13,\ldots,m_k,\ m_k+2,\ m_k+6,\ m_{k+1}+2,\ m_{k+1}+6,\ldots$$

En el programa debemos probar si el número es divisible por 2, por 3 y ejecutamos el ciclo

```
$p = 5$;
While $p \leq G$ Do{
Probar divisibilidad por $p$
Probar divisibilidad por $p+2$
$p = p+6$ }
```

- En cada paso debemos verificar si el divisor de prueba p alcanzó el límite Mín $\{G, \sqrt{N}\}$. Si se quiere evitar el cálculo de la raíz, se puede usar el hecho de que si $p > \sqrt{N}$ entonces $p > N/p$.

Algoritmo 1.3: Factorización por Ensayo y Error.

```
Entrada: $N \in \mathbb{N},\ G \leq \sqrt{N}$
Resultado: Un factor $p \leq G$ de $N$ si hubiera.
$p = 5$;
if $N$ es divisible por $2$ o $3$ then
    Imprimir factor;
else
    while $p \leq G$ do
        if $N$ es divisible por $p$ o $p+2$ then
            Imprimir factor;
            break;
        end
        $p = p + 6$
    end
end
```

1.4.4 Implementación en Java.

Creamos una clase que busca factores primos de un número N hasta un límite G. En el programa, $G =$ Mín $\{\sqrt{N}, G\}$.

Usamos un método `reducir(N,p)` que verifica si p es factor, si es así, continua dividiendo por p hasta que el residuo no sea cero. Retorna la parte de N que no ha sido factorizada.

El método `Factzar_Ensayo_Error(N, G)` llama al método `reducir(N,p)` para cada $p = 2,\ 3,\ 7,\ 11,\ 13, \ldots.$ hasta que se alcanza el límite G.

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

public class Ensayo_Error
{
    private Vector    salida = new Vector(1);
```

```
static BigInteger Ge     = new BigInteger("10000000");//10^7
BigInteger        UNO    = new BigInteger("1");
BigInteger        DOS    = new BigInteger("2");
BigInteger        TRES   = new BigInteger("3");
BigInteger        SEIS   = new BigInteger("4");
BigInteger        Nf;
int               pos    = 1; //posicin del exponente del factor

public Ensayo_Error(){}

public BigInteger reducir(BigInteger Ne, BigInteger p)
{
    int exp = 0, posAct = pos;
    BigInteger residuo;
    residuo =  Ne.mod(p);

    if(residuo.compareTo(BigInteger.ZERO)==0)
    {
      salida.addElement(p); //p es objeto BigInteger
      salida.addElement(BigInteger.ONE); //exponente
      pos = pos+2; //posicin del siguiente exponente (si hubiera)
    }

    while(residuo.compareTo(BigInteger.ZERO)==0)
    {
     Ne      =  Ne.divide(p); // Ne = Ne/p
     residuo =  Ne.mod(p);
     exp=exp+1;
     salida.set(posAct, new BigInteger(""+exp)); //p es objeto BigInteger
    }

    return Ne;
}//

public Vector Factzar_Ensayo_Error(BigInteger Ne, BigInteger limG)
{
    BigInteger p    = new BigInteger("5");

    Nf = Ne;
    Nf = reducir(Nf, DOS);
    Nf = reducir(Nf, TRES);

    while(p.compareTo(limG)<=0)
    {
      Nf= reducir(Nf, p);          //dividir por p
      Nf= reducir(Nf, p.add(DOS)); //dividir por p+2
      p = p.add(SEIS); //p=p+6
    }

    if(Nf.compareTo(BigInteger.ONE)>0)
    {
      salida.addElement(Nf); //p es objeto BigInteger
      salida.addElement(BigInteger.ONE); //exponente
    }
    return salida;
}
// Solo un argumento.
public Vector Factzar_Ensayo_Error(BigInteger Ne)
{
    BigInteger limG = Ge.min(raiz(Ne));
    return Factzar_Ensayo_Error(Ne, limG);
}

//raz cuadrada
public BigInteger raiz(BigInteger n)
{
                BigInteger xkm1 = n.divide(DOS);
                BigInteger xk   = n;

                  if(n.compareTo(BigInteger.ONE)< 0)
                    return xkm1=n;
                while(xk.add(xkm1.negate()).compareTo(BigInteger.ONE)>0)
                {
                    xk=xkm1;
                    xkm1=xkm1.add(n.divide(xkm1));
                    xkm1=xkm1.divide(DOS);
                }
                return xkm1;
```

```
     }
    //Imprimir
    public String print(Vector lista)
    {
       String tira="";
       for(int p = 0; p < lista.size(); p++)
       {
           if(p%2==0)     tira= tira+lista.elementAt(p);
           else           tira= tira+"^"+lista.elementAt(p)+" * ";
       }
       return tira.substring(0,tira.length()-3);
    }


   public static void main(String[] args)
   {
      BigInteger limG;
      BigInteger Nduro   = new BigInteger("2388005888439481");
      BigInteger N       = new BigInteger("27633027771706698949");
      Ensayo_Error Obj   = new Ensayo_Error();
      Vector   factores;

      factores = Obj.Factzar_Ensayo_Error(N); //factoriza

      //Imprimir vector de factores primos
      System.out.println("\n\n");
      System.out.println("N = "+N+"\n\n");
      System.out.println("Hay " +factores.size()/2+" factores primos <= " + Ge+"\n\n");
      System.out.println("N = "+Obj.print(factores)+"\n\n");
      System.out.println("\n\n");
   }
}
```

Al ejecutar este programa con $N = 367367653565289976655797,$ después de varios segundos la salida es

```
N = 27633027771706698949
Hay 3 factores primos <= 100000
N = 7^2 * 3671^3 * 408011^1
```

1.5 Método de factorización “rho” de Pollard.

1.5.1 Introducción.

En el método de factorización por ensayo y error, en su versión más cruda, probamos con todos los números entre 2 y $\sqrt{N}$ para hallar un factor de N. Si no lo hallamos, N es primo.

En vez de hacer estos $\approx \sqrt{N}$ pasos (en el peor caso), vamos a escoger una lista aleatoria de números, más pequeña que $\sqrt{N}$, y probar con ellos.

A menudo se construyen sucesiones *seudo-aleatorias* $x_0, x_1, x_2, \ldots$ usando una iteración de la forma $x_{i+1} = f(x_i) \pmod N$, con $x_0 = \text{random}(0, N-1)$. Entonces $\{x_0, x_1, \ldots\} \subseteq \mathbb{Z}_N$. Por lo tanto los x_i’s se empiezan a repetir en algún momento.

La idea es esta: Supongamos que ya calculamos la sucesión $x_0, x_1, x_2, \ldots$ y que es “suficientemente aleatoria”. Si p es un factor primo de N y si

$$\begin{cases} x_i \equiv x_j \pmod p \\ x_i \not\equiv x_j \pmod N \end{cases}$$

entonces, como $x_i - x_j = kp$, resulta que $\text{MCD}(x_i - x_j, N)$ es un factor no trivial de N.

Claro, no conocemos p, pero conocemos los x_i’s, así que podemos revelar la existencia de p con el cálculo del MCD: En la práctica se requiere comparar, de manera eficiente, los x_i con los x_j hasta revelar la presencia del factor p vía el cálculo del $\text{MCD}(x_i - x_j, N)$.

$$\begin{cases} x_i \equiv x_j \pmod p \\ \\ x_i \not\equiv x_j \pmod N \end{cases} \Longrightarrow \text{MCD}(x_i - x_j, N) \quad \text{es factor no trivial de} \quad N$$

Si $x_0, x_1, x_2, \ldots$ es “suficientemente aleatoria”, hay una probabilidad muy alta de que encontremos pronto una “repetición” del tipo $x_i \equiv x_j \pmod p$ antes de que esta repetición ocurra (mod N).

Antes de entrar en los detalles del algoritmo y su eficiencia, veamos un ejemplo.

EJEMPLO 1.14 Sea $N = 1387$. Para crear una sucesión "seudoaleatoria" usamos $f(x) = x^2 - 1$ y $x_1 = 2$. Luego,

$$\begin{aligned} x_0 &= 2 \\ x_{i+1} &= x_i^2 - 1 \pmod{N} \end{aligned}$$

es decir,

$$\begin{aligned} \{x_0, x_1, x_2, \ldots\} = \quad & \{2, 3, 8, 63, 1194, \\ & 1186, 177, 814, 996, 310, 396, 84, 120, 529, 1053, 595, 339, \\ & 1186, 177, 814, 996, 310, 396, 84, 120, 529, 1053, 595, 339, \ldots\} \end{aligned}$$

Luego, "por inspección" logramos ver que $1186 \not\equiv 8 \pmod{N}$ y luego usamos el detector de factores: $\text{MCD}(1186 - 8, N) = 19$. Y efectivamente, 19 es un factor de 1387. En este caso detectamos directamente un factor primo de N.

Por supuesto, no se trata de comparar todos los x_i's con los x_j's para $j < i$. El método de factorización "rho" de Pollard, en la variante de R. Brent, usa un algoritmo para detectar rápidamente un ciclo en una sucesión ([4]) y hacer solo unas cuantas comparaciones. Es decir, queremos detectar rápidamente $x_i \equiv x_j \pmod{p}$ usando la sucesión $x_{i+1} = f(x_i) \pmod{N}$ (que alcanza un ciclo un poco más tarde) y el test $\text{MCD}(x_i - x_j, N)$.

Típicamente necesitamos unas $O(\sqrt{p})$ operaciones. El argumento es heurístico y se expone más adelante. Básicamente lo que se muestra es que, como en el problema del cumpleaños, dos números x_i y x_j, tomados de manera aleatoria, son congruentes módulo p con probabilidad mayor que $1/2$, después de que hayan sido seleccionados unos $1.177\sqrt{p}$ números.

Aunque la sucesión $x_{i+1} = f(x_i) \pmod{N}$ cae en un ciclo en unas $O(\sqrt{N})$ operaciones, es muy probable que detectemos $x_i \equiv x_j \pmod{p}$ en unos $O(\sqrt{p})$ pasos. Si $p \approx \sqrt{N}$ entonces encontraríamos un factor de N en unos $O(N^{1/4})$ pasos. Esto nos dice que el algoritmo "rho" de Pollard factoriza N^2 con el mismo esfuerzo computacional con el que el método de ensayo y error factoriza N.

1.5.2 Algoritmo.

La algoritmo original de R. Brent compara x_{2^k-1} con x_j, donde $2^{k+1}-2^{k-1}\leq j\leq 2^{k+1}-1$. Los detalles de cómo esta manera de proceder detectan rápidamente un ciclo en una sucesión no se ven aquí pero pueden encontrarse en [4] y [22].

EJEMPLO 1.15 Sean $N=3968039,\ f(x)=x^2-1$ y $x_0=2$. Luego,

$$\begin{array}{lcl}
\text{MCD}(x_1-x_3,N) & = & 1\\
\text{MCD}(x_3-x_6,N) & = & 1\\
\text{MCD}(x_3-x_7,N) & = & 1\\
\text{MCD}(x_7-x_{12},N) & = & 1\\
\text{MCD}(x_7-x_{13},N) & = & 1\\
\text{MCD}(x_7-x_{14},N) & = & 1\\
\text{MCD}(x_7-x_{15},N) & = & 1\\
\vdots & \vdots & \vdots\\
\text{MCD}(x_{63}-x_{96},N) & = & 1\\
\text{MCD}(x_{63}-x_{97},N) & = & 1\\
\text{MCD}(x_{63}-x_{98},N) & = & 1\\
\text{MCD}(x_{63}-x_{99},N) & = & 1\\
\text{MCD}(x_{63}-x_{100},N) & = & 1\\
\text{MCD}(x_{63}-x_{101},N) & = & 1\\
\text{MCD}(x_{63}-x_{102},N) & = & 1\\
\text{MCD}(x_{63}-x_{103},N) & = & 1987
\end{array}$$

$N=1987\cdot 1997.$

En general, hay muchos casos en los que $\text{MCD}(x_i-x_j,N)=1$. En vez de calcular todos estos $\text{MCD}(z_1,N)$, $\text{MCD}(z_2,N),...,$ calculamos unos pocos $\text{MCD}(Q_k,N)$, donde $Q_k=\prod_{j=1}^{k} z_j\ (\text{mod}\,N)$. Brent sugiere escoger k entre $\ln N$ y $N^{1/4}$ pero lejos de cualquiera de los dos extremos ([4]). Riesel ([9]) sugiere tomar k como un múltiplo de 100.

EJEMPLO 1.16 Sean $N=3968039,\ f(x)=x^2-1$ y $x_0=2$. Luego, tomando $k=30$

$$\begin{aligned} Q_{30} = \prod_{j=1}^{30} z_j \pmod N = 3105033, \qquad \text{MCD}(Q_{30},N) &= 1 \\ Q_{60} = \prod_{j=31}^{60} z_j \pmod N = 782878, \qquad \text{MCD}(Q_{60},N) &= 1987 \end{aligned}$$

El algoritmo que vamos a describir aquí es otra variante del algoritmo de Brent ([5]) que es más sencillo de implementar.

Se calcula $\text{MCD}(x_i - x_j, N)$ para $i = 0,1,3,7,15,...$ y $j = i+1,...,2i+1$ hasta que, o $x_i = x_j \pmod N$ (en este caso se debe escoger una f diferente o un x_0 diferente) o que un factor no trivial de N sea encontrado.

Observe que si $i = 2^k - 1$ entonces $j = 2i+1 = 2^{k+1} - 1$, es decir el último j será el 'nuevo' i. Por tanto, en el algoritmo actualizamos x_i al final del `For`, haciendo la asignación $x_i = x_{2i+1} = x_j$.

Algoritmo 1.4: Método rho de Pollard (variante de R. Brent)

```
Entrada: $N \in \mathbb{N}$, $f$, $x_0$
Resultado: Un factor $p$ de $N$ o mensaje de falla.
salir=false;
$k = 0$;
$x_i = x_0$;
while salir=false do
    $i = 2^k - 1$;
    for $j = i+1, i+2, ..., 2i+1$ do
        $x_j = f(x_0) \pmod{N}$;
        if $x_i = x_j$ then
            salir=true;
            Imprimir "El método falló. Reintentar cambiando $f$ o $x_0$";
            Exit For;
        $g = \text{MCD}(x_i - x_j, N)$;
        if $1 < g < N$ then
            salir=true;
            Imprimir $N = N/g \cdot g$;
            Exit For;
        $x_0 = x_j$;
    $x_i = x_j$;
    $k++$;
```

1.5.3 Implementación en Java.

La implementación sigue paso a paso el algoritmo.

```
import java.math.BigInteger;
public class rhoPollard
{
rhoPollard(){}

public BigInteger f(BigInteger x)
{
    return x.multiply(x).add(BigInteger.ONE);//x^2+1
}

 public  void FactorPollard(BigInteger N)
 {
     int i, k;
     BigInteger xi,xj;
     BigInteger g   = BigInteger.ONE;
     BigInteger x0  = new BigInteger(""+2);;
     boolean salir  = false;

     k = 0;
     xi= x0;
     xj= x0;
     while(salir==false)
     { i=(int)(Math.pow(2,k)-1);
       for(int j=i+1; j<=2*i+1; j++)
       {
           xj=f(x0).mod(N);
           if(xi.compareTo(xj)==0)//si son iguales
           {salir=true;
            System.out.print("Fallo"+"\n\n");
            break;
           }
           g= N.gcd(xi.subtract(xj));
           if(g.compareTo(BigInteger.ONE)==1 && g.compareTo(N)==-1)//1<g<N
           {salir=true;
            System.out.print("Factor = "+g+"\n\n");
            break;
           }
           x0=xj;
       }
       xi=xj;
       k++;
     }
     System.out.print(N+" =  "+g+" . "+N.divide(g)+"\n\n");
  }

  public static void main(String[] args)
  {
   System.out.print("\n\n");
      rhoPollard obj = new rhoPollard();
      BigInteger N = new BigInteger("39680399966886876527");
      obj.FactorPollard(N);
   System.out.print("\n\n");
 }
}//
```

Sería bueno implementar una variante con el producto $Q_k = \prod_{j=1}^{k} z_j \pmod N$.

1.5.4 Complejidad.

En el método de Pollard-Brent $x_{i+1} = f(x_i) \pmod N$ entonces, si f entra en un ciclo, se mantiene en este ciclo. Esto es así pues si $f(x_i) = x_j \implies f(x_{i+1}) = f(f(x_i)) = f(x_j) = x_{j+1}$.

Si $a \neq 0, -2,$ se ha establecido de manera empírica (aunque no ha sido probado todavía), que esta es una sucesión de números suficientemente aleatoria que se vuelve periódica después de unos $O(\sqrt{p})$ pasos.

Lo de que se vuelva periódica, en algo parecido a $O(\sqrt{p})$ pasos, es fácil de entender si consideramos el *problema del cumpleaños* (“birthday problem”): ¿Cuántas personas se necesita seleccionar, de manera aleatoria, de tal manera que la probabilidad de que al menos dos de ellas cumplan años el mismo día, exceda $1/2$?

Si tomamos $n+1$ objetos (con reemplazo) de $N,$ la probabilidad de que *sean diferentes* es

$$Pr(n) = \left(1-\frac{1}{N}\right)\left(1-\frac{2}{N}\right)\cdots\left(1-\frac{n}{N}\right)$$

Puesto que $\ln(1-x) > -x,$ si x es suficientemente pequeño,

$$\ln Pr(n) \;>\; -\frac{1}{N}\sum_{k=1}^{n} k \;\sim\; -\frac{\frac{1}{2}n^2}{N}$$

Por tanto, para estar seguros que $Pr(n) \leq 1/2$ necesitamos

$$n > \sqrt{N}\cdot\sqrt{2\ln 2} \;\approx\; 1.1774\sqrt{N}.$$

Ahora, si no tomamos en cuenta que los años tienen distinto número de días (digamos que son de $N = 365$ días) y que hay meses en que hay más nacimientos que otros, el razonamiento anterior dice que si en un cuarto hay $n = 23$ personas, la probabilidad de que al menos dos coincidan en su fecha de nacimiento es más grande que $1/2$.

Ahora bien, ¿qué tan grande debe ser k de tal manera que al menos dos enteros, escogidos de manera aleatoria, sean congruentes (mod p) con probabilidad mayor que $1/2$?

Bueno, en este caso $N = p$ y k debe cumplir

$$Pr(k) = \left(1 - \frac{1}{p}\right)\left(1 - \frac{2}{p}\right)\cdots\left(1 - \frac{k-1}{p}\right) < \frac{1}{2}$$

Así, $\ln Pr(k) = -\dfrac{1}{p}\displaystyle\sum_{j=1}^{k-1} j \Longrightarrow Pr(k) \approx e^{-k(k-1)/2p}$. Luego, $Pr(k) \approx 1/2$ si $k \approx \sqrt{2p\ln 2} \approx 1.18\sqrt{p}$.

Bien, si $N = p \cdot m$ con p primo y $p \leq \sqrt{N}$ y seleccionamos de manera aleatoria algo más de $\sqrt{p}$ enteros x_i entonces la probabilidad de que $x_i = x_j \pmod{p}$ con $i \neq j$, es mayor que $1/2$.

1.6 Pruebas de Primalidad.

1.6.1 Introducción.

Para decidir si un número n pequeño es primo, podemos usar el método de ensayo y error para verificar que no tiene divisores primos inferiores a $\sqrt{n}$.

Para un número un poco más grande, la estrategia usual es primero verificar si tiene divisores primos pequeños, sino se usa el test para seudoprimos fuertes de Miller-Rabin con unas pocas bases p_i (con p_i primo) y usualmente se combina con el test de Lucas. Esta manera de proceder decide de manera correcta si un número es primo o no, hasta cierta cota 10^M. Es decir, la combinación de algoritmos decide de manera correcta si $n < 10^M$ Sino, decide de manera correcta solamente con una alta probabilidad y cabe la (remota) posibilidad de declarar un número compuesto como primo.

Aquí solo vamos a tratar rápidamente la prueba de Miller-Rabin.

1.6.2 Prueba de primalidad de Miller Rabin.

Iniciamos con test de primalidad de Fermat, por razones históricas. Esta prueba se basa el el teorema,

Teorema 1.9 **(Fermat)**
Sea p *primo. Si* $MCD(a,p)=1$ *entonces* $a^{p-1}\equiv 1\,(\bmod\, p)$.

Este teorema nos dice que si n es primo y a es un entero tal que $1\leq a\leq n-1$, entonces $a^{n-1}\equiv 1\,(\bmod\, n)$.

Por tanto, para probar que n es *compuesto* bastaría encontrar $1\leq a\leq n-1$ tal que $a^{n-1}\not\equiv 1\,(\bmod\, n)$.

Definición 1.2 *Sea* n *compuesto. Un entero* $1\leq a\leq n-1$ *para el que* $a^{n-1}\not\equiv 1\,(\bmod\, n)$, *se llama "testigo de Fermat" para* n.

Un testigo de Fermat para n sería un testigo de no-primalidad. De manera similar, un número $1\leq a\leq n-1$ para el que $a^{n-1}\equiv 1\,(\bmod\, n)$, apoya la posibilidad de que n sea primo,

Definición 1.3 *Sea* n *un entero compuesto y sea* a *un entero para el cual* $1\leq a\leq n-1$ *y* $a^{n-1}\equiv 1\,(\bmod\, n)$. *Entonces se dice que* n *es un seudoprimo respecto a la base* a. *Al entero* a *se le llama un "embaucador de Fermat" para* n.

Por ejemplo, $n=645=3\cdot 5\cdot 43$ es un seudoprimo en base 2 pues $2^{n-1}\equiv 1\,(\bmod\, n)$.

Es curioso que los seudoprimos en base 2 sean muy escasos. Por ejemplo, hay $882\,206\,716$ primos inferiores a 2×10^{10} y solo hay 19685 seudoprimos en base 2 inferiores a 2×10^{10}. Esto nos dice que la base 2 parece ser muy poco “embaucadora” en el sentido de que si tomamos un número grande n de manera aleatoria y si verificamos que $2^{n-1}\equiv 1\,(\mathrm{mod}\, n)$, entonces es muy probable que n sea primo. También los seudoprimos en base 3 son muy escasos y es altamente improbable que si tomamos un número grande n de manera aleatoria, este sea compuesto y que a la vez sea simultáneamente seudoprimo en base 2 y base 3.

Es decir, si un número n pasa los dos test $2^{n-1}\equiv 1\,(\mathrm{mod}\, n)$ y $3^{n-1}\equiv 1\,(\mathrm{mod}\, n)$; es muy probable que sea primo.

Sin embargo, hay enteros n compuestos para los cuales $a^{n-1}\equiv 1\,(\mathrm{mod}\, n)$ para todo a que cumpla $\mathrm{MCD}(a,n)=1$. A estos enteros se les llama números de Carmichael.

Por ejemplo, $n=561=3\cdot 11\cdot 17$ es número de Carmichael. Aunque este conjunto de números es infinito, son más bien raros (poco densos). En los primeros $100\,000\,000$ números naturales hay 2051 seudoprimos en base 2 y solo 252 números de Carmichael.

Nuestra situación es esta: Es poco probable que un número compuesto pase varios test de “primalidad” $a^{n-1}\equiv 1\,(\mathrm{mod}\, n)$ excepto los números de Carmichael, que son compuestos y pasan todos estos test.

Hay otro test, llamado “test fuerte de seudo-primalidad en base a” el cual los números de Carmichael no pasan. Además, si tomamos k números de manera aleatoria $a_1,a_2,...,a_k$ y si n pasa este test en cada una de las bases a_i, podemos decir que la probabilidad de que nos equivoquemos al declarar n como primo es menor que $1/4^k$. Por ejemplo, si $k=200$ la probabilidad de que nos equivoquemos es $<10^{-120}$

Teorema 1.10 *Sea* n *un primo impar y sea* $n-1=2^s r$ *con* r *impar. Sea* a *un entero tal que* $\mathrm{MCD}(a,n)=1$. *Entonces, o* $a^r\equiv 1(\mathrm{mod}\, n)$ *o* $a^{2^j r}\equiv -1(\mathrm{mod}\, n)$ *para algún* j, $0\leq j s-1$.

Con base en el teorema anterior, tenemos

Definición 1.4 *Sea* n *impar y compuesto y sea* $n-1=2^s r$ *con* r *impar. Sea* $1\leq a\leq n-1.$

(i) Si $a^r \not\equiv 1 \pmod{n}$ *y si* $a^{2^j r} \not\equiv -1 \pmod{n}$ *para* $0\leq j\leq s-1,$ *entonces* a *es llamado un testigo fuerte (de no-primalidad) de* $n.$

(ii) Si $a^r \equiv 1 \pmod{n}$ *y si* $a^{2^j r} \equiv -1 \pmod{n}$ *para* $0\leq j\leq s-1,$ *entonces* n *se dice un seudoprimo fuerte en la base* $a.$ *Al entero* a *se le llama "embaucador fuerte".*

Así, un seudoprimo fuerte n en base a es un número que actúa como un primo en el sentido del teorema 1.10.

Teorema 1.11 ***(Rabin)***

Si n *es un entero compuesto, a lo sumo* $\dfrac{1}{4}$ *de todos los números* $a,$ $1\leq a\leq n-1,$ *son embaucadores fuertes de* $n.$

Supongamos que tenemos un número compuesto $n.$ Tomamos k números $\{a_1,a_2,...,a_k\}$ de manera aleatoria y aplicamos el test fuerte de seudo-primalidad a n con cada uno de de estas bases $a_i.$ Entonces, hay menos que un chance en cuatro de que a_1 no sea testigo de no-primalidad de $n,$ y menos que un chance en cuatro de que a_2 no sea testigo de no-primalidad de $n,$ etc. Si n es primo, pasa el test para cualquier $a<n.$ Si cada a_i falla en probar que n es compuesto, entonces la probabilidad de equivocarnos al decir que n es primo es inferior a $\dfrac{1}{4^k}.$

1.6.3 Algoritmo.

Algoritmo 1.5: Miller-Rabin

```
Entrada: $n \geq 3$ y un parámetro de seguridad $t \geq 1$.
Resultado: “$n$ es primo” o “$n$ es compuesto”.
1  Calcule $r$ y $s$ tal que $n-1=2^s r$, $r$ impar;
2  for $i=1,2,\ldots,t$ do
3      $a=$Random$(2,n-2)$;
4      $y=a^r$ (mod $n$);
5      if $y \neq 1$ y $y \neq n-1$ then
6          $j=1$;
7          while $j \leq s-1$ y $y \neq n-1$ do
8              $y=y^2$ (mod $n$);
9              if $y=1$ then
10                 return “Compuesto”;
11             $j=j+1$;
12         if $y \neq n-1$ then
13             return “Compuesto”;
14 return “Primo”;
```

El algoritmo 1.5 verifica si en cada base a se satisface la definición 1.4. En la línea $9,$ si $y=1,$ entonces $a^{2^j r} \equiv 1$ (mod n). Puesto que este es el caso cuando $a^{2^{j-1} r} \not\equiv \pm 1$ (mod n) entonces n es compuesto. Esto es así pues si $x^2 \equiv y^2$ (mod n) pero si $x \not\equiv \pm y$ (mod n), entonces MCD$(x-y,n)$ es un factor no trivial de $n.$ En la línea $12,$ si $y \neq n-1,$ entonces a es un testigo fuerte de $n.$

Si el algoritmo 1.5 declara compuesto a n entonces n es definitivamente compuesto, por el teorema 1.10. Si n es primo, es declarado primo. Si n es compuesto, la probabilidad de que el algoritmo lo declare primo es inferior a $1/4^t$.

El algoritmo 1.5 requiere, para $n-1=2^j r$ con r impar, $t(2+j)\ln n$ pasos. t es el número de bases.

Una estrategia que se usa a veces es fijar las bases. Se toman como base algunos de los primeros primos en vez de tomarlas de manera aleatoria. El resultado importante aquí es este: Si $p_1,p_2,\ldots,p_t$ son los primeros t primos y si ψ_t es el más pequeño entero compuesto el cual es seudoprimo para todas las bases $p_1,p_2,\ldots,p_t,$ entonces el algoritmo de

Miller-Rabin, con las bases $p_1, p_2, ..., p_t$, siempre responde de manera correcta si $n < \psi_t$. Para $1 \leq t \leq 8$ tenemos

t	ψ_t
1	2047
2	1 373 653
3	25 326 001
4	3 215 031 751
5	2 152 302 898 747
6	3 474 749 660 383
7	341 550 071 728 321
8	341 550 071 728 321

1.6.4 Implementación en Java.

En la clase BigInteger de Java ya viene implementado el método `this.modPow(BigInteger r, BigInteger N)` para calcular $y = a^r \pmod N$. Para calcular r y s solo se divide $N-1$ por dos hasta que el residuo sea diferente de cero.

En esta implementación usamos los primeros ocho primos como bases. Así el algoritmo responde de manera totalmente correcta si $N < 341550071728321$.

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

public class Miller_Rabin
{
    public Miller_Rabin(){}

    public boolean esPrimoMR(BigInteger N)
    {
        //n>3 e impar. Respuesta 100% segura si N <341 550 071 728 321
        BigInteger  N1      = N.subtract(BigInteger.ONE);//N-1
        BigInteger  DOS     = new BigInteger("2");
        int[]       primo   = {2,3,5,7,11,13,17,19};
        int         s       = 0;
        boolean     esPrimo = true;
        BigInteger  a,r,y;
        int         j;
    //n-1 = 2^s r
    while(N1.remainder(DOS).compareTo(BigInteger.ZERO)==0)
    {    N1=N1.divide(DOS);
         s=s+1;
    }
    r  = N1;
    N1 = N.subtract(BigInteger.ONE);

    for(int i=0; i<=7; i++)
    {
        a = new BigInteger(""+primo[i]);
        y = a.modPow(r, N);
        if( y.compareTo(BigInteger.ONE)!=0 && y.compareTo(N1)!=0)
        {
           j=1;
           while(j<= s-1 && y.compareTo(N1)!=0 )
           {
              y = y.modPow(DOS, N);
              if(y.compareTo(BigInteger.ONE)==0) esPrimo=false;
              j++;
           }
           if(y.compareTo(N1)!=0) esPrimo = false;
        }

    }
   return esPrimo;
}

public static void main(String[] args)
{
  System.out.println("\n\n");
  BigInteger  N     = new BigInteger("6658378974");
  Miller_Rabin  obj = new Miller_Rabin();

  System.out.println(N+" es primo = "+obj.esPrimoMR(N)+"\n\n");

  System.out.println("\n\n");
}
```

```
}
```

Apéndice A

Notación O grande y algoritmos.

En esta primera parte vamos a establecer algunos resultados que nos servirán más adelante para hacer estimaciones que nos ayuden a analizar los algoritmos que nos ocupan. Como vamos a hablar de algunos tópicos de la teoría de números en términos de comportamiento promedio, necesitamos establecer algunas ideas y teoremas antes de entrar a la parte algorítmica.

A.1 Notación O grande

La notación O grande se usa para comparar funciones "complicadas" con funciones más familiares.

Por ejemplo, en teoría analítica de números, es frecuente ver el producto

$$\left(1-\frac{1}{2}\right)\left(1-\frac{1}{3}\right)\left(1-\frac{1}{5}\right)\cdots = \prod_{\substack{p\leq x,\\ p \text{ primo}}}\left(1-\frac{1}{p}\right)$$

*

Este producto nos da una estimación de la proporción (o densidad) de enteros positivos sin factores primos inferiores a x (en un conjunto $\{1,2,...,n\}$). Si x es muy grande, se vuelve complicado calcular este producto porque no es fácil obtener todos los primos que uno quiera en un tiempo razonable. En vez de eso, tenemos la fórmula de Mertens (1874)

$$\prod_{\substack{p\le x,\\ p \text{ primo}}} \left(1-\frac{1}{p}\right) = \frac{e^{-\gamma}}{\log x} + O\left(1/\log^2(x)\right)$$

Esta fórmula dice que el producto es comparable a la función $e^{-\gamma}/\log x$, con la cual nos podemos sentir más cómodos. Aquí $e^{-\gamma} \approx 0.561459$ y $O\left(1/\log^2(x)\right)$, la estimación del "error" en esta comparación, indica una función inferior a un múltiplo de $1/\log^2(x)$ si x es suficientemente grande.

Definición A.1 *Si* $h(x) > 0$ *para toda* $x \ge a$, *escribimos*

$$f(x) = O(h(x)) \quad \textit{si existe } \; C \; \textit{ tal que } \; |f(x)| \le C\,h(x) \; \textit{ para toda } \; x \ge a. \tag{A.1}$$

Escribimos

$$f(x) = g(x) + O(h(x))$$

si existe C *tal que* $|f(x) - g(x)| \le C\,h(x)$ *siempre que* $x \ge a$

También podemos pensar en "$O(h(x))$" como una función que es dominada por $h(x)$ a partir de algún $x \ge a$.

Definición A.2 *Si*

$$\lim_{x\to\infty} \frac{f(x)}{g(x)} = 1$$

decimos que f *es asintóticamente igual a* g *conforme* $x \to \infty$ *y escribimos,*

$$f(x) \sim g(x) \quad \textit{conforme} \quad x \to \infty$$

En los siguientes ejemplos se exponen algunos resultados y algunos métodos que vamos a usar más adelante con la intención de que las pruebas queden de tamaño aceptable.

EJEMPLO A.1 Si $h(x)>0$ para toda $x\geq a$ y si h es acotada, entonces h es $O(1)$. En particular $\operatorname{sen}(x)=O(1)$

EJEMPLO A.2 $\sqrt{x}=O\left(\dfrac{x}{\ln^2(x)}\right)$

Lo que hacemos es relacionar $\sqrt{x}$ con $\ln(x)$ usando (la expansión en serie de) e^x. Sea $y=\ln(x)$. Luego $e^y=\sum_{k=0}^{\infty}y^k/k!\geq y^4/4!$, así que

$$\begin{aligned}24e^y\geq y^4 \quad &\Longrightarrow \quad \ln^4(x)\leq 24x\\ &\Longrightarrow \quad \ln^2(x)\sqrt{x}\leq\sqrt{24}x\\ &\Longrightarrow \quad \sqrt{x}\leq\sqrt{24}\,\frac{x}{\ln^2(x)}\end{aligned}$$

EJEMPLO A.3 Si $n,d\neq 0\in\mathbb{N}$ entonces $[n/d]=n/d+O(1)$. Aquí, $[x]$ denota la parte entera de x.

En efecto, por el algoritmo de la división, existe $k,r\in\mathbb{Z}$ tal que $n=k\cdot d+r$ con $0\leq r<d$ o también $\dfrac{n}{d}=k+\dfrac{r}{d}$. Luego, $\left[\dfrac{n}{d}\right]=k=\dfrac{n-r}{d}$.

Ahora, $\left|\left[\dfrac{n}{d}\right]-\dfrac{n}{d}\right|=\dfrac{r}{d}<1$ para cada $n\geq 0$. Así, tenemos $[n/d]=n/d+O(1)$, tomando $C=1$.

EJEMPLO A.4 Aunque la serie armónica $\displaystyle\sum_{k=1}^{\infty}\frac{1}{k}$ es divergente, la función $H_n=\displaystyle\sum_{k=1}^{n}\frac{1}{k}$ es muy útil en teoría analítica de números. La función $\tau(n)$ cuenta cuántos divisores tiene n. Vamos a mostrar que $\displaystyle\sum_{k=1}^{n}\tau(k)=nH(n)+O(n)$ y entonces, $\displaystyle\sum_{k=1}^{n}\tau(k)\ =\ n\ln(n)+O(n)$.

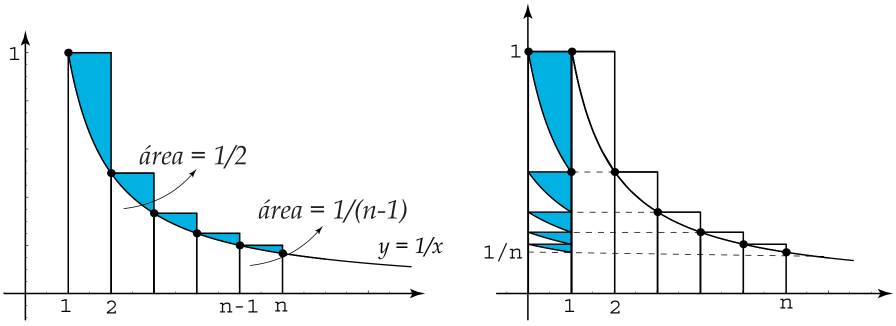
Primero, vamos a mostrar, usando argumentos geométricos, que existe una número real γ, llamada *constante de Euler*, tal que

$$H_n=\ln(n)+\gamma+O(1/n)\,.$$

Prueba. Hay que mostrar que $\exists C$ tal que $0 < H_n - \ln(n) - \gamma < C \cdot 1/n$ para $n > n_0$.

Usando integral de Riemann,

$$\sum_{k=1}^{n-1} \frac{1}{k} = \int_1^n \frac{1}{x}\,dx + E_n \quad \text{i.e.} \quad H_{n-1} = \ln(n) + E_n$$



(a) *(b)*

Figura A.1 Comparando el área $\ln(n)$ con la suma H_n.

Geométricamente, H_{n-1} corresponde a la suma de las áreas de los rectángulos desde 1 hasta n y E_n la suma de las áreas de las porciones de los rectángulos sobre la curva $y = 1/x$.

En el gráfico (b) de la figura A.1 vemos que $E_n \leq 1$ para toda $n \geq 1$, así que E_n es una función de n, que se mantiene acotada y es creciente, por lo tanto esta función tiene un límite, el cual vamos a denotar con γ. Así, $\lim_{n\to\infty} E_n = \gamma$. En particular, para cada n fijo, $\gamma > E_n$.

Como $\gamma - E_n$ corresponde a la suma (infinita) de las áreas de las regiones sombreadas en la figura A.2, se establece la desigualdad

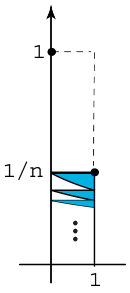


Figura A.2 $\gamma - E_n$.

$$\gamma - E_n < 1/n$$

de donde

$$0 < \gamma - (H_{n-1} - \ln(n)) < 1/n.$$

Ahora restamos $1/n$ a ambos lados para hacer que aparezca H_n, tenemos

$$\frac{1}{n} > H_n - \ln(n) - \gamma > 0$$

que era lo que queríamos demostrar.

Aunque en la demostración se establece $H_n - \ln(n) - \gamma < 1/n$, la estimación del error $O(1/n)$ corresponde a una función dominada por un múltiplo de $1/n$. Veamos ahora algunos cálculos que pretender evidenciar el significado de $O(1/n)$.

n	H_n	$\ln(n)$	$\lvert H_n - \ln(n) - \gamma \rvert$	$1/n$
170000	12.62077232	12.62076938	$2.94117358 \times 10^{-6}$	$5.88235294 \times 10^{-6}$
180000	12.67793057	12.67792779	$2.77777520 \times 10^{-6}$	$5.55555555 \times 10^{-6}$
190000	12.73199764	12.73199501	$2.63157663 \times 10^{-6}$	$5.26315789 \times 10^{-6}$
200000	12.78329081	12.78328831	$2.49999791 \times 10^{-6}$	$5. \times 10^{-6}$

Observando las dos últimas columnas se puede establecer una mejor estimación del error con $\dfrac{1}{2n}$ y todavía mejor con $\dfrac{1}{2n} - \dfrac{1}{12n^2}$!

n	H_n	$\ln(n)+\gamma+\frac{1}{2n}-\frac{1}{12n^2}$
100000	12.090146129863427	12.090146129863427
150000	12.495609571309556	12.495609571309554
200000	12.783290810429621	12.783290810429623

También, de estas tablas se puede obtener la aproximación $\gamma \approx 0.577216$

Segundo, vamos a mostrar que $\sum_{k=1}^{n}\tau(k)=nH(n)+O(n)$ y que $\sum_{k=1}^{n}\tau(k) \;=\; n\ln(n)+O(n).$ Podemos poner $\tau(k)$ como una suma que corre sobre los divisores de $k,$ $\tau(k)=\sum_{d|k}1.$ Luego,

$$\sum_{k=1}^{n}\tau(k)=\sum_{k=1}^{n}\sum_{d|k}1$$

La idea es usar argumentos de divisibilidad para usar la expansión del ejemplo A.3. Si $d|k$ entonces $k=d\cdot c\leq n.$ Esto nos dice que el conjunto de todos los divisores positivos de los números k inferiores o iguales a $n,$ se puede describir como el conjunto de todos los pares (c,d) con la propiedad $cd\leq n$ (por supuesto, se puede hacer una demostración formal probando la doble implicación "$\Longleftrightarrow$").

Ahora, $cd\leq n \iff d\leq n \wedge c\leq n/d.$ Entonces podemos escribir,

$$\sum_{k=1}^{n}\tau(k)=\sum_{\substack{c,d\\ cd\leq n}}1=\sum_{d\leq n}\;\sum_{c\leq n/d}1$$

La suma $\sum_{c\leq n/d}1$ corre sobre los enteros positivos menores o iguales que $n/d.$ Esto nos da $[n/d]$ sumandos, i.e. $\sum_{c\leq n/d}1=[n/d].$ Finalmente, usando el ejemplo A.3,

$$\begin{aligned}
\sum_{k=1}^{n} \tau(k) &= \sum_{d\leq n} [n/d] \\
&= \sum_{d\leq n} \{n/d + O(1)\} \\
&= \sum_{d\leq n} n/d \;+\; \sum_{d\leq n} O(1) \\
&= n\sum_{d\leq n} 1/d \;+\; \sum_{d\leq n} O(1) \\
&= nH_n + O(n)
\end{aligned}$$

En los ejercicios se pide mostrar, usando la figura A.1, que $H_n = \log(n) + O(1)$. Usando este hecho,

$$\sum_{k=1}^{n} \tau(k) = nH_n + O(n) = n\{\ln(n) + O(1)\} + O(n) = n\ln(n) + O(n).$$

(Los pequeños detalles que faltan se completan en los ejercicios)

EJERCICIOS

A.1 Probar que $\sum_{d\leq n} O(1) = O(n)$

A.2 Probar que $nO(1) + O(n) = O(n)$

A.3 Usar la figura A.1 para probar que $H_n = \log(n) + O(1)$.

A.4 Probar que $\dfrac{H_n - \log(n)}{\gamma} = 1 + O(1/n)$

A.5 Probar que $\sqrt{n}H_n = \sqrt{n}\log(n) + O(\sqrt{n})$

A.6 Probar que $H_n - \log(n) \sim \gamma$

A.7 Probar que $H_n \sim \log(n)$

A.2 Estimación de la complejidad computacional de un algoritmo.

La teoría de la complejidad estudia la cantidad de pasos necesarios para resolver un problema dado. Lo que nos interesa aquí es cómo el número de pasos crece en función del tamaño de la entrada (“input”), despreciando detalles de hardware.

Tiempo de corrida.

El tiempo de corrida de un algoritmo es, simplificando, el número de pasos que se ejecutan al recibir una entrada de tamaño n. Cada paso i tiene un costo c_i distinto, pero hacemos caso omiso de este hecho y solo contamos el número de pasos.

La complejidad $T(n)$ de un algoritmo es el número de pasos necesario para resolver un problema de tamaño (input) n. Un algoritmo es de orden a lo sumo $g(n)$ si existen constantes c y M tales que,

$$T(n) < c\,g(n), \quad \forall n > M$$

En este caso escribimos $T(n) = O(g(n))$. Esta definición dice que esencialmente T no crece más rápido que g. El interés, por supuesto, funciones g que crecimiento “lento”.

Clases de complejidad.

Las funciones g usadas para medir complejidad computacional se dividen en diferentes *clases de complejidad.*

Si un algoritmo es de orden $O(\ln n)$ tiene *complejidad logarítmica.* Esto indica que es un algoritmo muy eficiente pues si pasamos de un input de tamaño n a otro de tamaño $2n$, el algoritmo hace pocos pasos adicionales pues $T(2n) \leq c\ln 2n = c\ln 2\, +\, c\ln n \leq c(1+\ln n)$.

Otra importante clase de algoritmos, son los de orden $O(n\ln n)$. Muchos algoritmos de ordenamiento son de esta clase.

Los algoritmos de orden $O(n^k)$ se dicen de *complejidad polinomial.* Hay muchos algoritmos importantes de orden $O(n^2)$, $O(n^3)$. Si el exponente es muy grande, el algoritmo usualmente se vuelve ineficiente.

Si g es exponencial, decimos que el algoritmo tiene *complejidad exponencial*. La mayoría de estos algoritmos no son prácticos.

EJEMPLO A.5

- Si $T(n) = n^3 + 4$ entonces $T(n) = O(n^3)$ pues $n^3 + 4 \leq cn^3$ si $c > 1$.

- Si $T(n) = n^5 + 4n + \ln(n) + 5$ entonces $T(n) = O(n^5)$.

  Para ver esto, solo es necesario observar que n^5 *domina* a los otros sumandos para n suficientemente grande.

  En particular, $n^5 \geq \ln(n)$ si $n \geq 1$: Sea $h(n) = n^5 - \ln(n)$. Entonces $h'(n) = 5n^4 - 1/n \geq 0$ si $n \geq 1$ entonces h es creciente. Luego $h(n) \geq h(1) \geq 0$ si $n \geq 1$.

EJEMPLO A.6 Consideremos el siguiente fragmento de código,

```
while (N > 1)
{
    N = N / 2;
}
```

En este ejemplo, cada iteración requiere una comparación ‘`N>1`’ y una división ‘`N/2`’. Obviamos el costo de estas operaciones y contamos el número de pasos.

Si la entrada es un número real n entonces la variable N almacenará los valores $\{n, n/2,\ n/2^2,\ n/2^3, \ldots\}$. En el $k-$ésimo paso, la variable N almacena el número $n/2^{k-1}$ y el ciclo se detiene si $n/2^{k-1} < 1$, es decir, el ciclo se detiene en el momento que $k - 1 > \lg(n)$. Así que se ejecutan aproximadamente $2\lg(n) + 2$ pasos para un input de “tamaño” n. Por lo tanto es el tiempo de corrida es $T(n) = O(\lg(n))$

EJEMPLO A.7 En el siguiente código, contamos los ceros en un arreglo `a` de tamaño N.

```
int count = 0;
for (int i = 0; i < N; i++)
    if (a[i] == 0) count++;\\
```

Los pasos son

Declaración de variables	2
Asignación de variables	2
Comparaciones `i<N`	$N+1$
Comparaciones `a[i] == 0`	N
Accesos al array `a[]`	N
Incrementos	$\leq 2N$
Total	$4N+5 \leq T(N) \leq 5N+5$

Observe que el incremento `i++` se realiza N veces mientras que el incremento `count++` se realiza como máximo N veces (solo si el arreglo `a[]` tiene todas las entradas nulas).

El tiempo de corrida es $T(n) = O(n)$.

EJEMPLO A.8 En el siguiente código, contamos las entradas `(i,j)` tal que $a[i]+a[j]=0$. `a[]` es un arreglo de tamaño N.

```
int count = 0;
    for (int i = 0; i < N; i++)
        for (int j = i+1; j < N; j++)
            if (a[i] + a[j] == 0) count++;
```

Para el conteo, usamos la identidad $\sum_{i=1}^{N} i = 1/2N(N+1)$

Declaración de variables	$N+2$
Asignación de variables	$N+2$
Comparaciones `i<N, J<N`	$1/2(N+1)(N+2)$
Comparaciones `a[i] + a[j] == 0`	$1/2N(N-1)$
Accesos al array `a[]`	$N(N-1)$
Incrementos (máximo N^2, mínimo N^2-N)	$\leq N^2$

$5+N+3N^2 \leq T(N) \leq 3N^2+2N+5$

El tiempo de corrida es $T(n) = O(n^2)$

EJEMPLO A.9 1. Si $f,g:\mathbb{N}\longrightarrow\mathbb{R}^+$, muestre que $\text{Máx}\{f(n),g(n)\}=O(f(n)+g(n))$.

Solución.

Hay que demostrar que existe c y M tal que $\text{Máx}\{f(n),g(n)\}\leq c\cdot(f(n)+g(n)),\ \forall n\geq M$.

Por definición de máximo, $\text{Máx}\{f(n),g(n)\}\leq f(n)+g(n)$. Así, la definición de O-grande se cumple para $c=1$ y $M=1$,

$$\text{Máx}\{f(n),g(n)\}\leq 1\cdot(f(n)+g(n))\quad\forall n\geq 1.$$

2. Muestre que $2^{n+1}=O(2^n)$ pero $2^{2n}\neq O(2^n)$

Solución.

A.) Hay que demostrar que existe c y M tal que $2^{n+1}\leq c\cdot 2^n\ \ \forall n\geq M$.

$2^{n+1}=O(2^n)$ pues $2^{n+1}\leq 2\cdot 2^n\ \ \forall n$.

B.) Por contradicción. $2^{2n}\neq O(2^n)$ pues si $2^{2n}\leq c\,2^n\implies 2^n\leq c$ y esto no puede ser pues 2^n no es acotada superiormente (2^n es creciente).

3. Muestre que $f\in O(g)$ no implica necesariamente que $g\in O(f)$

Solución.

Un ejemplo es suficiente.

Por ejemplo, $n=O(n^2)$ pero $n^2\neq O(n)$ pues $f(n)=n$ no es acotada.

4. Si $f(n)\geq 1$ y $\lg[g(n)]\geq 1$ entonces muestre que si $f\in O(g)\implies \lg[f]\in O(\lg[g])$

Solución.

Hay que demostrar que existe c y M tal que $\lg f(n) \leq c_1 \cdot \lg g(n),\ \forall\, n \geq M$.

$f \in O(g) \implies f(n) \leq c\,g(n),\ \forall n \geq M$. Luego, $\lg f(n) \leq \lg(c\,g(n)) = \lg c + \lg g(n)$ por ser lg una función creciente.

Como $\lg[g(n)] \geq 1 \implies \lg c \ \leq\ \lg c \cdot \lg g(n)$, entonces

$$\lg f(n) \leq \lg c + \lg g(n) \leq (\lg c + 1) \lg g(n), \quad \forall n \geq M.$$

Así que basta tomar $c_1 = \lg c + 1$ para que se cumpla la definición de O-grande.

5. Calcule el tiempo de corrida del siguiente programa

```
public class ThreeSum {
    // retorna el n\'umero de distintos (i, j, k)
    // tal que (a[i] + a[j] + a[k] == 0)
public static int count(int[] a) {
int N = a.length;
int cnt = 0;
    for (int i = 0; i < N; i++)
        for (int j = i+1; j < N; j++)
            for (int k = j+1; k < N; k++)
                if (a[i] + a[j] + a[k] == 0) cnt++;
return cnt;
}}
```

Solución.

Debemos contar los pasos que hace el programa.

El término dominante en el tiempo de corrida es la cantidad de accesos a la instrucción ‘`if (a[i] + a[j] + a[k] == 0) cnt++;`’. Contar este número de accesos será suficiente para establecer el tiempo de corrida.

En el conteo que sigue usamos las fórmulas

$$\begin{aligned}\sum_{i=1}^{M} i &= \frac{1}{2}M(M+1) \qquad \text{(A.2)}\\ \sum_{i=1}^{M} i^2 &= \frac{1}{6}M(M+1)(2M+1). \qquad \text{(A.3)}\end{aligned}$$

Para contar cuántas veces se ejecuta el `if` del tercer `for`, observemos que este `if` solo se ejecuta (pasa el test `k < N`) si $j \leq N-2$, es decir, si $i \leq N-3$.

i	j	Veces que se ejecuta el `if` del 3er `for`
0	1 to N	$\sum_{k=2}^{N-1}(N-k) = 1+2+\ldots+N-2$
1	2 to N	$\sum_{k=3}^{N-1}(N-k) = 1+2+\ldots+N-3$
$\vdots$	$\vdots$	$\vdots$
$N-3$	$N-2$ to N	1
$N-2$	$N-1$ to N	No se ejecuta

Así, el `if` del tercer `for` se ejecuta

$$\begin{aligned}\sum_{j=2}^{N}(1+2+\ldots+N-j) &= \sum_{j=2}^{N}(1/2(N-j)(N-j+1)) \quad \text{por (A.2)}\\ &= \sum_{j=2}^{N}\left(\frac{j^2}{2}-(N+1/2)j+\frac{N^2}{2}+\frac{N}{2}\right)\\ &= \frac{N^3}{6}-\frac{N^2}{2}+\frac{N}{3} \quad \text{usando (A.2) y (A.3)}\end{aligned}$$

Finalmente, $T(n) = O(n^3)$.

Nota: Otra forma de contar es observando que el programa recorre todos los tripletes, con componentes distintas, (i, j, k) de un conjunto de N elementos (el arreglo `a[]`), es decir,

$$\binom{N}{3} = 1/6N(N-1)(N-2) = \frac{N^3}{6} - \frac{N^2}{2} + \frac{N}{3}$$

Tamaño del input en teoría de números.

El tamaño del "input" depende de la clase de problema que estemos analizando. En los algoritmos de búsqueda y ordenamiento el tamaño de la entrada es el número de datos. En teoría algorítmica de números, el tamaño de la entrada es el número de dígitos del número de entrada y la complejidad se mide en términos de cantidad de operaciones de bits.

Número de dígitos.

La representación en base $b = 2$ de un entero n es $(d_{k-1}d_{k-2}\cdots d_0)_2$ donde

$$n = d_{k-1}2^{k-1} + d_{k-2}2^{k-2} + \cdots + d_0\,2^0,\;\; d_i \in \{0,1\}$$

EJEMPLO A.10 $2^{10} = 1024 = (10000000000)_2$

Si $2^{k-1} \leq n \leq 2^k$ entonces n tiene k dígitos en base $b = 2$.

EJEMPLO A.11 $2^{10} \leq 2^{10} \leq 2^{11}$, así $n = 2^{10}$ tiene 11 dígitos en base 2.

$2^7 = 128 \leq 201 \leq 256 = 2^8$, así $n = 201$ tiene 8 dígitos en base 2. En efecto, $201 = (11001001)_2$

El número k de dígitos, en base $b = 2$, de un número $n \neq 0$ se puede calcular con la fórmula

$$k = [\log_2|n|] + 1 = \left[\frac{\ln|n|}{\ln(2)}\right] + 1$$

EJEMPLO A.12 Si $n=2^{10}$ entonces $k=\left[\log_2(2^{10})\right]+1=10+1=11$

Si $n=201$ entonces $k=[\log_2(201)]+1=[7.65105...]+1=8$

Recordemos que acostumbra usar "$\lg n$" en vez de $\log_2(n)$. En términos de "$O-$grande", el número de bits de n es $[\lg(n)]+1=O(\lg n)=O(\ln n)$. Se acostumbra decir "el número de bits de n es $O(\ln n)$".

La sumar dos números de tamaño $\ln n$ requiere $O(\ln n)$ operaciones de bits y la multiplicación y la división $O(\ln n)^2$.

Complejidad polinomial en teoría de números

EJEMPLO A.13

- Supongamos que un número n se representa con β bits, es decir, $\beta=\lfloor \lg n\rfloor+1$ o $n=O(2^{\log n})$.

  Si un algoritmo que recibe un entero n, tiene complejidad $O(n)$ *en términos del número de operaciones aritméticas*, entonces, si por cada operación aritmética se hacen $O(\ln n)^2$ operaciones de bits, el algoritmo tiene complejidad

  $$\begin{aligned} O(n) &= O(n(\ln n)^2) \\ &= O\left(2^{\ln n}(\ln n)^2\right) \end{aligned}$$

  en términos de operaciones de bits.

  Es decir, un algoritmo con complejidad polinomial en términos del número de operaciones aritméticas, tiene complejidad exponencial en términos de operaciones de bit.

- El algoritmo de Euclides para calcular $\text{MCD}(a,b)$ con $a<b$ requiere $O(\ln a)$ operaciones aritméticas (pues un teorema de Lamé (1844) establece que el número de

divisiones necesarias para calcular el MCD(a,b) es a lo sumo cinco veces el número de dígitos decimales de a, es decir $O(\log_{10} a)$). Esto corresponde a $O(\ln a)^3$ en términos de operaciones de bits (asumiendo como antes que divisiones y multiplicaciones necesitan $O(\ln n)^2$ operaciones de bit).

Apéndice B

Implementaciones en VBA para Excel.

B.1 Introducción.

Para hacer las implementaciones en VBA para Excel necesitamos Xnumbers, un complemento (gratuito) para VBA y VB. Xnumbers nos permite manejar números grandes de hasta 200 dígitos. XNumbers se puede obtener en

`http://digilander.libero.it/foxes/SoftwareDownload.htm`

Aquí usamos un “dll”, `Xnumbers.dll`. Para usarlo y hacer el cuaderno Excel portable, debemos proceder como sigue,

1. Ponemos `Xnumbers.dll` en al misma carpeta del cuaderno Excel,
2. En el editor de Visual Basic, hacemos una referencia (Herramientas-Referencias-Examinar). Con esto ya podemos usar las funciones de este paquete.

*

3. Para hacer el cuaderno portable (para que funcione en cualquier PC) insertamos un módulo en `ThisWorkBook` y pegamos el código

```
Option Explicit

Sub Installation_Procedure()
' Installation Procedure
' v.  6/10/2004
Dim myTitle As String
    myTitle = "XNUMBERS"
    'Activate the error handler
    On Error Resume Next

    'Check if Excel allows to make changes to VBA project
    Excel_Security_Check
    If Err <> 0 Then GoTo Error_handler

    'Remove old reference to this VBA project (if any)
    VBA_Link_Remove "DLLXnumbers"

    'Add the reference to this VBA project
    VBA_Link_Add "xnumbers.dll"
    If Err <> 0 Then GoTo Error_handler

    'Check if the ActiveX works
    ActiveX_test "xnumbers.dll"
    If Err <> 0 Then
        Err.Clear
        'The activeX may be to register. Do it.
        DLL_Register "xnumbers.dll"
        If Err <> 0 Then GoTo Error_handler
        MsgBox "Xnumbers.dll registering...", vbInformation
        'Repeat the check
        ActiveX_test "xnumbers.dll"
        If Err <> 0 Then GoTo Error_handler
    End If

Exit Sub
```

```
Error_handler:
    'Something has gone wrong. Show a message
    MsgBox Err.Description, vbCritical, myTitle
    Exit Sub
End Sub

Sub VBA_Link_Add(DLLname)
'Links a DLL library to an XLA project
Dim LibFile, Msg
    LibFile = ThisWorkbook.Path & "\" & DLLname
    If Dir(LibFile) <> "" Then
        ThisWorkbook.VBProject.References.AddFromFile LibFile
    Else
        Msg = "Unable to find " & LibFile
        Err.Raise vbObjectError + 513, , Msg
    End If
End Sub

Sub VBA_Link_Remove(FileLinked)
'Removes the Links of a DLL library from a XLA project
On Error Resume Next
    With ThisWorkbook.VBProject
        .References.Remove .References(FileLinked)
    End With
End Sub

Sub Excel_Security_Check()
'Shows a warning if Excel not allowes to change a XLA project
Dim Msg As String, myTitle As String
myTitle = "XNUMBERS addin"
If Excel_VBA_Protection Then
    Msg = "Your Excel security restriction does not allow to install this addin."
    Err.Raise vbObjectError + 513, , Msg
End If
End Sub

Private Function Excel_VBA_Protection() As Boolean
' Checks if Excel has the VBA protection. Returns true/false
Dim tmp
```

```
On Error Resume Next
tmp = ThisWorkbook.VBProject.Name
If Err <> 0 Then
    Excel_VBA_Protection = True
Else
    Excel_VBA_Protection = False
End If
End Function

Sub ActiveX_test(ActiveXname)
' Checks if the Xnumbers ActiveX works
Dim Msg, Xnum As New Xnumbers
On Error GoTo Error_handler
    With Xnum

    End With
Exit Sub
Error_handler:
    Msg = Err.Description & " <" & ActiveXname & ">"
    Err.Raise vbObjectError + 513, , Msg
End Sub

Private Sub DLL_Register(DLLname, Optional UnReg = False)
'Tries to register the Xnumbers ActiveX
Dim DLLfile, Msg, Save_path, cmd_line
    Save_path = CurDir
    Path_Change ThisWorkbook.Path
    If Dir(DLLname) <> "" Then
    If UnReg = False Then
        cmd_line = "REGSVR32 /s " + DLLname     'register silent
    Else
        cmd_line = "REGSVR32 /u /s " + DLLname  'unregister silent
    End If
        Shell cmd_line
    Else
        Msg = "Unable to find " & DLLfile
        Err.Raise vbObjectError + 513, , Msg
    End If
    Path_Change Save_path
```

```
End Sub

Sub Path_Change(myPath)
'change global path (drive+path)
Dim myDrive
    myDrive = Left(myPath, 1)
    ChDrive myDrive
    ChDir myPath
End Sub

'-------- test routines  ----------------
Sub DLL_Register_test()
    DLL_Register "xnumbers.dll"
End Sub

Sub DLL_UnRegister_test()
    DLL_Register "xnumbers.dll", True
End Sub

Sub VBA_Link_Remove_test()
    VBA_Link_Remove "DLLXnumbers"
End Sub
'-----------------------------------------
```

Adicionalmente, deberá permitir macros (nivel de protección bajo) y en el menú “Herramientas - Opciones - Seguridad - Seguridad de Macros - Fuentes de Confianza” deberá habilitar la opción “Confiar en el acceso a proyectos Visual Basic”.

Si usamos Excel, debemos tener el cuidado de leer e imprimir estos números en celdas con formato de texto. Esta propiedad puede ser establecida desde el mismo código VBA.

B.2 Algoritmos Básicos.

Para los algoritmos de este trabajo, necesitamos el MCD, cálculo de residuos, potencias módulo m e inversos multiplicativos en $\mathbb{Z}_m$. El complemento XNumbers para VB y para VBA para Excel, viene con `gcd` y `xPowMod`. El cálculo de residuos módulo m se puede hacer con `xPowMod.` Los inversos requieren implementar el algoritmo extendido de Euclides pues `xPowMod` no permite potencias negativas.

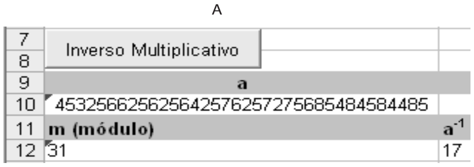
B.2.1 Cálculo de inversos multiplicativos en $\mathbb{Z}_m$

Sea $a \in \mathbb{Z}_m$. Si a y m son primos relativos, a^{-1} existe. En este caso, existen s,t tal que $sa+tm=1$. Usualmente s,t se calculam usando el algoritmo extendido de Euclides. Luego $a^{-1}=s \mod m$. El algoritmo es como sigue,

Algoritmo B.1: Inverso Multiplicativo mod m.

```
Entrada: $a \in \mathbb{Z}_m$
Resultado: $a^{-1} \bmod m$, si existe.
Calcular $x,t$ tal que $xa+tm = \text{MCD}(a,m)$;
if $\text{MCD}(a,m) > 1$ then
    $a^{-1} \bmod m$ no existe
else
    return $s \mod m$
```

Para hacer una implementación en Excel, hacemos un cuaderno con el número a en la celda `A10`. Imprimos en la celda `B12`. El módulo lo leemos en la celda `A12.`



	A	
7		
8		
9	**a**	
10	453256625625642576257275685484584485	
11	**m (módulo)**	**a^{-1}**
12	31	17

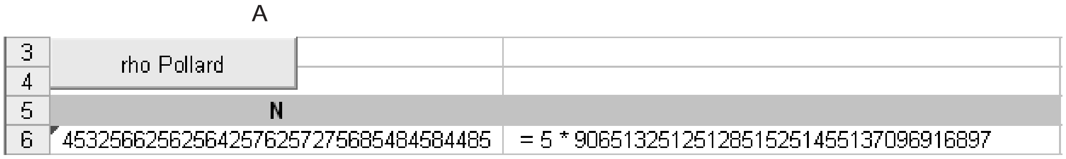
```
'BOTON
Private Sub CommandButton2_Click()
Call invMult
End Sub

'Necesitamos una funci\'on signo
Function MPSgn(x)
Dim MP As New Xnumbers
Dim salida
salida = 1
```

```
 If MP.xComp(x) = 1 Then
 salida = 1
  Else: salida = -1
 End If
 MPSgn = salida
End Function


Sub invMult()
Dim MP As New Xnumbers
Dim a, m, c, c1, d, d1, d2, q, r, x, t
MP.DigitsMax = 100
'Entran a y m, sale a^-1 mod m
a  = Cells(10, 1)
m  = Cells(12, 1)
'algoritmo extendido de Euclides
c  = MP.xAbs(a)
d  = MP.xAbs(m)
c1 = "1"
d1 = "0"
c2 = "0"
d2 = "1"

While MP.xComp(d) <> 0
q  = MP.xDivInt(c, d)
r  = MP.xSub(c, MP.xMult(q, d))
r1 = MP.xSub(c1, MP.xMult(q, d1))
r2 = MP.xSub(c2, MP.xMult(q, d2))
c  = d
c1 = d1
c2 = d2
d  = r
d1 = r1
d2 = r2
Wend
x = MP.xDivInt(c1, MPSgn(a) * MPSgn(c))
Cells(12, 2).NumberFormat = "@" 'pasar a formato texto
If mcd > 1 Then
```

```
      Cells(12, 2) = "Inverso no existe"
Else: Cells(12, 2) = MP.xPowMod(x, 1, m) 'Escribe el n\'umero
End If
Set MP = Nothing 'destruir el objeto.
End Sub
```

B.2.2 Algoritmo rho de Pollard.

El número N queremos factorizar, lo leemos en la celda (en formato texto) `A6`. La factorización la imprimimos en la celda `B6`



N

453256625625642576257275685484584485 = 5 * 90651325125128515251455137096916897

La subrutina VBA es

```
'BOTON
Private Sub CommandButton1_Click()
Call rhoPollard
End Sub



Sub rhoPollard()
Dim MP As New Xnumbers
Dim salir As Boolean
Dim n, nc, i, k, xj, x0, g
MP.DigitsMax = 100
n = Cells(6, 1)
k = 0
x0 = 2
xi = x0
salir = False
While salir = False
```

```
    i = 2 ^ k - 1
    For j = i + 1 To 2 * i + 1
        xj = MP.xPowMod(MP.xSub(MP.xPow(x0, 2), 1), 1, n) 'x^2-1
        If MP.xComp(xi, xj) = 0 Then
            salir = True
            Cells(6, 2) = "  El m\'etodo fall\'o, cambie f o x0"
        Exit For
        End If
        g = MP.xAbs(MP.xGCD(MP.xSub(xi, xj), n)) 'MCD(xi-xj,n)
        If MP.xComp(1, g) = -1 And MP.xComp(g, n) = -1 Then
            salir = True
            Cells(6, 2).NumberFormat = "@"  'los XNumbers son texto
            Cells(6, 2) = "  = " + g + " * " + MP.xDivInt(n, g)
            Exit For
        End If
        x0 = xj
    Next j
    xi = xj
    k = k + 1
Wend
Set MP = Nothing
End Sub
```

Bibliografía

[1] M. Kac. *Statistical Independence in Probability, Analysis and Number Theory.* Wiley, New York, 1959.

[2] N. Koblitz *A course in number theory and cryptography.* 2ed., Springer,1994.

[3] G.H. Hardy, J.E. Littlewood. *An Introduction to Theory of Numbers.* Oxford Univ. Press. 1938.

[4] R. Brent. “An Improved Monte Carlo Factorization Algorithm.” BIT 20 (1980), 176-184. (`http://wwwmaths.anu.edu.au/~brent/pub/pubsall.html`).

[5] R. Brent, J. M. Pollard. “Factorization of the Eighth Fermat Number.” Mathematics of Computation, vol 36, n 154 (1981), 627-630.
(`http://wwwmaths.anu.edu.au/~brent/pub/pubsall.html`).

[6] Harold M. Edwards. *Riemann’s Zeta Function.* Dover Publications Inc. 2001.

[7] P.L. Chebyshev. “The Theory of Probability”. Translated by Oscar Sheynin (www.sheynin.de) 2004. Versión en internet:
`http://www.sheynin.de/download/4_Chebyschev.pdf`. Consultada Diciembre 16, 2006.

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

[9] Hans Riesel. *Prime Numbers and Computer Methods for Factorization.* Springer; 2 edition. 1994.

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

[11] J. Stopple. *A Primer of Analytic Number Theory. From Pythagoras to Riemann.*
Cambridge. 2003.

[12] RSA, Inc. `http://www.rsasecurity.com/`. Consultada Noviembre 11, 2006.

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

[14] ArjenK. Lenstra. “Integer Factoring”.
`http://modular.fas.harvard.edu/edu/Fall2001/124/misc/`
Consultada: Octubre, 2006.

[15] P. Montgomery. “Speeding the Pollard and Elliptic Curve Method”.
Mathematics of Computation. Vol 48, Issue 177. Jan 1987. 243-264.
`http://modular.fas.harvard.edu/edu/Fall2001/124/misc/`
Consultada: Octubre, 2006.

[16] Joachim von zur Gathen, Jürgen Gerhard. “*Modern Computer Algebra*”. Cambridge University Press, 2003.

[17] Maurice Mignotte. “*Mathematics for Computer Algebra*”. Springer,1992.

[18] A. Menezes, P. van Oorschot, S. Vanstone. *Handbook of Applied Cryptography.* Vanstone, CRC Press, 1996.
`(www.cacr.math.uwaterloo.ca/hac)`

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

[20] J.Stopple. *A primer of Analytic Number Theory.* Cambridge, 2003.

[21] G. Tenenbaum. *Introduction to Analytic and Probabilistic Number Theory*. Cambridge Studies in Advanced Mathematics.1995.

[22] S. Y. Yan. *Number Theory for Computing.* 2nd edition. Springer. 2001.

Sección Tecnologías de Internet
Revista digital Matemática, Educación e Internet (www.cidse.itcr.ac.cr/revistamate/). Vol. 9, N^o 2. 2009

Criba de Eratóstenes: Cómo colar números primos. Implementación en Java y VBA para Excel.

Walter Mora F.
wmora2@yahoo.com.mx
Escuela de Matemática
Instituto Tecnológico de Costa Rica

Introducción

La Criba de Eratóstenes es un algoritmo que permite hallar todos los números primos menores que un número natural dado n eliminando los números compuestos de la lista $\{2,3,...,n\}$. Es simple y razonablemente eficiente. En este trabajo se presenta un algoritmo (explicado en detalle) y la respectiva implementación. Al final se explica como manejar la memoria para el caso de números grandes.

Palabras claves: Números primos, algoritmo, criba de Eratóstenes.

1.1 Criba de Eratóstenes: Cómo colar números primos.

La criba[1] de Eratóstenes es un algoritmo que permite "colar" todos los números primos menores que un número natural dado n, eliminando los números compuestos de la lista $\{2,...,n\}$. Es simple y razonablemente eficiente.

---

[1] Criba, tamiz y zaranda son sinónimos. Una criba es un herramienta que consiste de un cedazo usada para limpiar el trigo u otras semillas, de impurezas. Esta acción de limpiar se le dice cribar o tamizar.

Primero tomamos una lista de números $\{2,3,...,n\}$ y eliminamos de la lista los múltiplos de $2.$ Luego tomamos el primer entero después de 2 que no fue borrado (el 3) y eliminamos de la lista sus múltiplos, y así sucesivamente. Los números que permanecen en la lista son los primos $\{2,3,5,7,...\}.$

EJEMPLO 1.1 Primos menores que $n=10$

Lista inicial	2	3	4	5	6	7	8	9	10
Eliminar múltiplos de 2	2	3	$\cancel{4}$	5	$\cancel{6}$	7	$\cancel{8}$	9	$\cancel{10}$
Resultado	2	3	5	7	9				
Eliminar múltiplos de 3	2	3	5	7	$\cancel{9}$				
Resultado	2	3	5	7					

Primer refinamiento: Tachar solo los impares

Excepto el $2,$ los pares no son primos, así que podríamos "tachar" solo sobre la lista de impares $\leq n$:

$$\{3,5,9,...,\}=\left\{2i+3\;:\quad i=0,1,...\left\llbracket\frac{n-3}{2}\right\rrbracket\right\}$$

El último impar es n o $n-1.$ En cualquier caso, el último impar es $2\cdot\left\llbracket\frac{n-3}{2}\right\rrbracket+3$ pues,

Si n es impar, $n=2k+1$ y $\left\llbracket\frac{n-3}{2}\right\rrbracket=k-1\implies 2(k-1)+3=n.$

Si n es par, $n=2k$ y $\left\llbracket\frac{n-3}{2}\right\rrbracket=k-2\implies 2(k-2)+3=2k-1=n-1.$

Segundo refinamiento: Tachar de p_k^2 en adelante

En el paso $k-$ ésimo hay que tachar los múltiplos del primo p_k desde p_k^2 en adelante.

Esto es así pues en los pasos anteriores se ya se tacharon $3\cdot p_k,\ 5\cdot p_k,...,p_{k-1}\cdot p_k.$

Por ejemplo, cuando nos toca tachar los múltiplos del primo $7,$ ya se han eliminado los múltiplos de $2,3$ y $5,$ es decir, ya se han eliminado $2\cdot 7,\ 3\cdot 7,\ 4\cdot 7,\ 5\cdot 7$

y $6 \cdot 7.$ Por eso iniciamos en 7^2.

Tercer refinamiento: Tachar mientras $p_k^2 \leq n$

En el paso $k-$ ésimo hay que tachar los múltiplos del primo p_k solo si $p_k^2 \leq n$. En otro caso, nos detenemos ahí.

¿Porque?. En el paso $k-$ ésimo tachamos los múltiplos del primo p_k desde p_k^2 en adelante, así que si $p_k^2 > n$ ya no hay nada que tachar.

EJEMPLO 1.2 Encontrar los primos menores que 20. El proceso termina cuando el cuadrado del mayor número confirmado como primo es < 20.

1. La lista inicial es $\{2, 3, 5, 7, 9, 11, 13, 15, 17, 19\}$
2. Como $3^2 \leq 20$, tachamos los múltiplos de 3 desde $3^2 = 9$ en adelante:

$$\{2, 3, 5, 7, \cancel{9}, 11, 13, \cancel{15}, 17, 19\}$$

3. Como $5^2 > 20$ el proceso termina aquí.
4. Primos $< 20:$ $\{2, 3, 5, 7, 11, 13, 17, 19\}$

1.1.1 Algoritmo e implementación.

1. Como ya vimos, para colar los primos en el conjunto $\{2, 3, ..., n\}$ solo consideramos los impares:

$$\left\{2i+3\ :\quad i=0,1,...\left[\!\!\left[\frac{n-3}{2}\right]\!\!\right]\right\}=\{3,5,7,9,...\}$$

2. Por cada primo $p=2i+3$ (tal que $p^2<n$), debemos eliminar los *múltiplos impares* de p menores que $n,$ a saber

   $$(2k+1)p=(2k+1)(2i+3),\quad k=i+1,i+2,...$$

   Observe que si $k=i+1$ entonces el primer múltiplo en ser eliminado es $p^2=(2i+3)(2i+3),$ como debe ser.

Esto nos dice que para implementar el algoritmo solo necesitamos un arreglo (booleano) de tamaño "`quo(n-3,2)`". En Java se pone "`(n-3)/2`" y en VBA se pone "`(n-3)\2`".

El arreglo lo llamamos `EsPrimo[i], i=0,1,...,(n-3)/2.`

Cada entrada del arreglo "`EsPrimo[i]`" indica si el número $2i+3$ es primo o no.

Por ejemplo

`EsPrimo[0] = true` pues $n=2\cdot 0+3=3$ es primo,

`EsPrimo[1] = true` pues $n=2\cdot 1+3=5$ es primo,

`EsPrimo[2] = true` pues $n=2\cdot 2+3=7$ es primo,

`EsPrimo[3] = false` pues $n=2\cdot 3+3=9$ no es primo.

Si el número $p=2i+3$ es primo entonces $i=(p-3)/2$ y

```
EsPrimo[(p-3)/2] = true.
```

Si sabemos que $p = 2i + 3$ es primo, debemos poner

```
EsPrimo[((2k+1)(2i+3) - 3)/2] = false
```

pues estas entradas representan a los múltiplos $(2k+1)(2i+3)$ de p. Observe que cuando $i = 0,1,2$ tachamos los múltiplos de $3,5$ y 7; cuando $i = 3$ entonces $2i+3 = 9$ pero en este momento `esPrimo[3]=false` así que proseguimos con $i = 4$, es decir, proseguimos tachando los múltiplos de 11.

En resumen: Antes de empezar a tachar los múltiplos de $p = 2i + 3$ debemos preguntar si `esPrimo[i]=true.`

Algoritmo 1.1: Criba de Eratóstenes

```
Entrada: $n \in \mathbb{N}$
Resultado: Primos entre $2$ y $n$
máx = $(n-3)/2$;
boolean esPrimo[$i$],   $i = 1,2,...,$máx;
for $i = 1,2,..., máx$ do
    esPrimo[$i$] =True;
$i = 0$;
while $(2i+3)(2i+3) \leq n$ do
    $k = i+1$;
    if esPrimo(i) then
        while $(2k+1)(2i+3) \leq n$ do
            esPrimo[$((2k+1)(2i+3)-3)/2$] =False;
            $k = k+1$;
    $i = i+1$;
Imprimir;
for $j = 1,2,..., máx$ do
    if esPrimo[$j$] =True then
        Imprima $j$
```

1.1.1.1 Implementación en Java. Vamos a agregar un método a nuestra clase "`Teoria_Numeros`". El método recibe el número natural $n > 2$ y devuelve un vector con los números primos $\leq n$. Para colar los números compuestos usamos un arreglo

```
boolean [] esPrimo = new boolean[(n-3)/2].
```

Al final llenamos un vector con los primos que quedan.

```
import java.math.BigInteger;
public class Teoria_Numeros
{   ...
 public static Vector HacerlistaPrimos(int n)
 {
            Vector        salida = new Vector(1);
            int k   = 1;
            int max = (n-3)/2;
            boolean[]   esPrimo  = new boolean[max+1];

            for(int i = 0; i <= max; i++)
                esPrimo[i]=true;

            for(int i = 0; (2*i+3)*(2*i+3) <= n; i++)
            {
              k = i+1;
              if(esPrimo[i])
              {
                while( ((2*k+1)*(2*i+3)) <= n)
                {
                    esPrimo[((2*k+1)*(2*i+3)-3)/2]=false;
                    k++;
                }
              }
             }
             salida.addElement(new Integer(2));
             for(int i = 0; i <=max; i++)
             { if(esPrimo[i])
               salida.addElement(new Integer(2*i+3));
             }
             salida.trimToSize();
```

```
         return salida;
}
public static void main(String[] args)
{
  System.out.println("\n\n");
 //-------------------------------------------------------------------
  int    n = 100;
  Vector primos;
         primos = HacerlistaPrimos(n);
  //Cantidad de primos <= n
  System.out.println("Primos <="+ n+": "+primos.size()+"\n");
  //imprimir vector (lista de primos)
  for(int p = 1; p < primos.size(); p++)
  {
      Integer num = (Integer)primos.elementAt(p);
      System.out.println(""+(int)num.intValue());
  }
//-------------------------------------------------------------------
  System.out.println("\n\n");
}}
```

1.1.1.2 Uso de la memoria En teoría, los arreglos pueden tener tamaño máximo `Integer.MAX_INT` $= 2^{31} - 1 = 2\,147\,483\,647$ (pensemos también en la posibilidad de un arreglo multidimensional!). Pero en la práctica, el máximo tamaño del array depende del hardware de la computadora. El sistema le asigna una cantidad de memoria a cada aplicación; para valores grandes de n puede pasar que se nos agote la memoria (veremos el mensaje "`OutOfMemory Error`"). Podemos asignar una cantidad de memoria apropiada para el programa "cribaEratostenes.java" desde la línea de comandos, si n es muy grande. Por ejemplo, para calcular los primos menores que $n = 100\,000\,000$, se puede usar la instrucción

```
C:\usrdir> java -Xmx1000m -Xms1000m Teoria_Numeros
```

suponiendo que el archivo "`Teoria_Numeros.java`" se encuentra en `C:\usrdir.`

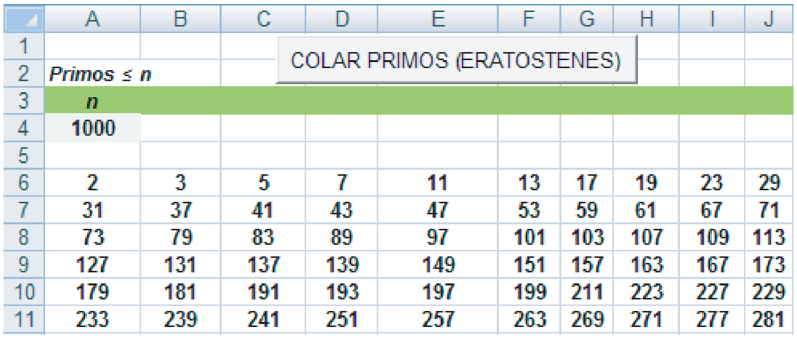
Esta instrucción asigna al programa una memoria inicial (Xmx) de 1000 MB y una memoria máxima (Xms) de 1000 MB (siempre y cuando existan tales recursos de memoria en nuestro sistema).

En todo caso hay que tener en cuenta los siguientes datos

n	Primos $\leq n$
10	4
100	25
1 000	168
10 000	1 229
100 000	9 592
1 000 000	78 498
10 000 000	664 579
100 000 000	5 761 455
1 000 000 000	50 847 534
10 000 000 000	455 052 511
100 000 000 000	4 118 054 813
1 000 000 000 000	37 607 912 018
10 000 000 000 000	346 065 536 839

1.1.1.3 *Implementación en Excel.* Para la implementación en Excel usamos un cuaderno como el de la figura (1.1).

El número n lo leemos en la celda $(4,1)$. El código VBA incluye una subrutina para imprimir en formato de tabla, con `ncols` columnas. Este último parámetro es opcional y tiene valor default 10. También incluye otra subrutina para limpiar las celdas para que no haya confusión entre los datos de uno y otro cálculo.



	A	B	C	D	E	F	G	H	I	J
1										
2	*Primos ≤ n*									
3	*n*									
4	1000									
5										
6	2	3	5	7	11	13	17	19	23	29
7	31	37	41	43	47	53	59	61	67	71
8	73	79	83	89	97	101	103	107	109	113
9	127	131	137	139	149	151	157	163	167	173
10	179	181	191	193	197	199	211	223	227	229
11	233	239	241	251	257	263	269	271	277	281

Figura 1.1 Primos $\leq n$.

Imprimir en formato de tabla

Para esto usamos la subrutina

```
Imprimir(ByRef Arr() As Long, fi, co, Optional ncols As
                        Variant).
```

La impresión inicia en la celda “`(fi,co)`”. Para imprimir en formato de tabla usamos `Cells(fi + k, co + j)` con el número de columnas `j` variando de 0 a `ncols-1.` Para reiniciar `j` en cero actualizamos `j` con `j = j Mod ncols.` Para cambiar la fila usamos `k.` Esta variable aumenta en 1 cada vez que `j` llega a `ncols-1.` Esto se hace con división entera: `k = k + j \ (ncols - 1)`

Subrutina para borrar celdas

Para esto usamos la subrutina

```
LimpiaCeldas(fi, co, ncols).
```

Cuando hacemos cálculos de distinto tamaño es conveniente borrar las celdas de los cálculos anteriores para evitar confusiones. La subrutina inicia en la celda `(fi,co)` y borra `ncols` columnas a la derecha. Luego pasa a la siguiente fila y hace lo mismo. Prosigue de igual forma hasta que encuentre la celda `(fi+k,co)` vacía.

```
Option Explicit
    Private Sub CommandButton1_Click()
    Dim n, ncols
    n = Cells(4, 1)
    ncols = Cells(4, 3)
    Call Imprimir(ERATOSTENES(n), 6, 1, ncols)
End Sub

’ Imprime arreglo en formato de tabla con "ncols" columnas,
’ iniciando en la celda (fi,co)
Sub Imprimir(ByRef Arr() As Long, fi, co, Optional ncols As Variant)
Dim i, j, k
    ’ Limpia celdas
    ’ f        = fila en que inicia la limpieza
    ’ co       = columna en q inicia la limpieza
    ’ ncols    = número de columnas a borrar
Call LimpiaCeldas(fi, co, ncols)
If IsMissing(ncols) = True Then
            ncols = 10
End If
’Imprimir
j = 0
k = 0
For i = 0 To UBound(Arr)
    Cells(fi + k, co + j) = Arr(i)
    k = k + j \ (ncols - 1) ’k aumenta 1 cada vez que j llegue a ncols-1
    j = j + 1
    j = j Mod ncols         ’j=0,1,2,...,ncols-1
Next i

End Sub


Function ERATOSTENES(n) As Long()
Dim i, j, k, pos, contaPrimos
Dim max As Long
Dim esPrimo() As Boolean
Dim Primos() As Long
max = (n - 3) \ 2 ’ División entera
ReDim esPrimo(max + 1)
ReDim Primos(max + 1)
For i = 0 To max
    esPrimo(i) = True
Next i
contaPrimos = 0
Primos(0) = 2 ’contado el 2
j = 0
While (2 * j + 3) * (2 * j + 3) <= n
    k = j + 1
    If esPrimo(j) Then
        While (2 * k + 1) * (2 * j + 3) <= n
            pos = ((2 * k + 1) * (2 * j + 3) - 3) \ 2
            esPrimo(pos) = False
            k = k + 1
        Wend
    End If
j = j + 1
Wend

For i = 0 To max
   If esPrimo(i) Then
        contaPrimos = contaPrimos + 1 ’3,5,...
        Primos(contaPrimos) = 2 * i + 3
    End If
Next i

ReDim Preserve Primos(contaPrimos) ’Cortamos el vector
ERATOSTENES = Primos()
End Function


Private Sub LimpiaCeldas(fi, co, nc)
Dim k, j
k = 0
While LenB(Cells(fi + k, co)) <> 0   ’ celda no vac\’ia
      For j = 0 To nc
          Cells(fi + k, co + j) = ""  ’ borra la fila hasta nc columnas
      Next j
      k = k + 1
```

```
Wend
End Sub
```

1.1.2 Primos entre m y n.

Para encontrar todos los primos entre m y n (con $m < n$) procedemos como si estuvieramos colando primos en la lista $\{2, 3, ..., n\}$, solo que eliminamos los múltiplos que están entre m y n : Eliminamos los múltiplos de los primos p para los cuales $p^2 \leq n$ (o también $p \leq \sqrt{n}$), que están entre m y n.

Múltiplos de p entre m y n

Para los primos p inferiores a $\sqrt{n}$, buscamos el primer múltiplo de p entre m y n.

Si $m - 1 = pq + r,\ 0 \leq r < p \quad \Longrightarrow \quad p(q+1) \geq m$

Así, los múltiplos de p mayores o iguales a m son

$$p(q+1),\ p(q+2),\ p(q+3),\ ... \quad \text{con} \quad q = \text{quo}(m-1, p)$$

EJEMPLO 1.3 Para encontrar los primos entre $m = 10$ y $n = 30$, debemos eliminar los múltiplos de los primos $\leq \sqrt{30} \approx 5$. Es decir, los múltiplos de los primos $p = 2, 3, 5$.

Como $10 - 1 = 2 \cdot 4 + 1$, el 2 elimina los números $2(4+k) = 8 + 2k,\ k \geq 1$; es decir $\{10, 12, ..., 30\}$

Como $10 - 1 = 3 \cdot 3 + 0$, el 3 elimina los números $3(3+k) = 9 + 3k,\ k \geq 1$; es decir $\{12, 15, 18, 21, 24, 27, 30\}$

Como $10-1=5\cdot 1+4$, el 5 elimina los números $5(1+k)=5+5k,\ k\geq 1$; es decir $\{10,15,20,25.\}$

Finalmente nos quedan los primos $11,13,17,19,23,29$.

1.1.2.1 *Algoritmo.* Como antes, solo consideramos los impares entre m y n. Si ponemos

$min=\mathrm{quo}(m+1-3,2)$ y $max=\mathrm{quo}(n-3,2)$

entonces $2\cdot min+3$ es el primer impar $\geq m$ y $2\cdot max+3$ es el primer impar $\leq n$. Así, los impares entre m y n son los elementos del conjunto $\{2\cdot i+3\ :\ i=min,...,max\}$

Como antes, usamos un arreglo booleano `esPrimo(i)` con $i=min,...,max$. `esPrimo(i)` representa al número $2\cdot i+3$.

EJEMPLO 1.4 Si $m=11$ y 20, $[\![(m+1-3)/2]\!]=4$ y $[\![(n-3)/2]\!]=8$. Luego $2\cdot 4+3=11$ y $2\cdot 8+3=19$.

Para aplicar el colado necesitamos los primos $\leq\sqrt{n}$. Esta lista de primos la obtenemos con la función `Eratostenes(isqrt(n)).` Aquí hacemos uso de la función `isqrt(n)` (algoritmo **??**).

Para cada primo p_i en la lista,

1. si $m\leq p_i^2$, tachamos los múltiplos impares de p_i como antes,

```
if $m \leq p_i^2$ then
    $k = (p_i - 1)/2$;
    while $(2k+1)p_i \leq n$ do
        esPrimo$[((2k+1)p_i - 3)/2]$ =False;
        $k = k+1$;
```

Note que si $k = (p_i - 1)/2$ entonces $(2k+1)p_i = p_i^2$

2. si $p_i^2 < m,$ tachamos desde el primer múltiplo impar de p_i que supere m :

Los múltiplos de p_i que superan m son $p_i(q+k)$ con $q = \text{quo}(m-1, p)$. De esta lista solo nos interesan los múltiplos impares. Esto requiere un pequeño análisis aritmético.

Como p_i es impar, $p_i(q+k)$ es impar solo si $q+k$ es impar. Poniendo $q_2 = \text{rem}(q, 2)$ entonces $(2k+1-q_2\ +\ q)$ es impar si $k = q_2,\ q_2+1, \ldots$. En efecto,

$$2k+1-q_2\ +\ q = \begin{cases} 2k+1\ +\ q & \text{si} \quad q \quad \text{es par. Aquí} \quad k = q_2 = 0, 1, ... \\ 2k\ +\ q & \text{si} \quad q \quad \text{es impar. Aquí} \quad k = q_2 = 1, 2, ... \end{cases}$$

Luego, los múltiplos impares de p_i son los elementos del conjunto

$$\{(2k+1-q_2\ +\ q)\cdot p \quad : \quad q_2 = \text{rem}(q,2) \quad \text{y} \quad k = q2, q2+1, ...\}$$

La manera de tachar los múltiplos impares de p_i es

```
if $p_i^2 < m$ then
    $q = (m-1)/p$ ;
    $q_2 = \mathrm{rem}(q,2)$ ;
    $k = q_2$ ;
    $mp = (2k+1-q_2+q)\cdot p_i$ ;
    while $mp \leq n$ do
        esPrimo[$(mp-3)/2$] =False;
        $k = k+1$;
        $mp = (2k+1-q_2+q)\cdot p_i$
```

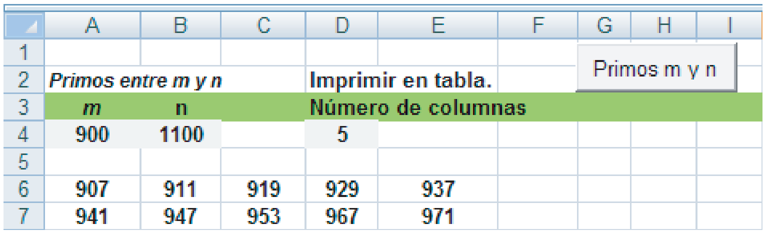
Ahora podemos armar el algoritmo completo.

```
Algoritmo 1.2: Colado de primos entre $m$ y $n$.
Entrada: $n, m \in \mathbb{N}$ con $m < n$.
Resultado: Primos entre $m$ y $n$
Primo() $=$ una lista de primos $\leq \sqrt{n}$;
$min = (m+1-3)/2$; $max = (n-3)/2$;
$esPrimo[i],\ i = min, ..., max$;
for $j = min, ..., max$ do
    $esPrimo[j]$ =True;
$np$ $=$ cantidad de primos en la lista Primos;
Suponemos Primo$(0) = 2$;
for $i = 1, 2, ..., np$ do
    if $m \leq p_i^2$ then
        $k = (p_i - 1)/2$;
        while $(2k+1)p_i \leq n$ do
            esPrimo$[((2k+1)p_i - 3)/2]$ =False;
            $k = k+1$;
    if $p_i^2 < m$ then
        $q = (m-1)/p$;
        $q_2 = \text{rem}(q, 2)$;
        $k = q_2$;
        $mp = (2k+1-q_2+q) \cdot p_i$;
        while $mp \leq n$ do
            esPrimo$[(mp-3)/2]$ =False;
            $k = k+1$;
            $mp = (2k+1-q_2+q) \cdot p_i$
Imprimir;
for $j = min, ..., max$ do
    if $esPrimo[j] = True$ then
        Imprima $2 * i + 3$
```

1.1.2.2 Implementación en Excel.

Para la implementación en Excel usamos un cuaderno como el de la figura (1.2).

m y n los leemos en la celdas $(4,1)$, $(4,2)$. Como antes, el código VBA hace referencia a las subrutinas para imprimir en formato de tabla y limpiar las celdas (sección 1.1.1.3).



	A	B	C	D	E	F	G	H	I
1									
2	*Primos entre m y n*			Imprimir en tabla.					
3	*m*	n		Número de columnas					
4	900	1100		5					
5									
6	907	911	919	929	937				
7	941	947	953	967	971				

Figura 1.2 Primos $\leq n$.

En VBA Excel podemos declarar un arreglo que inicie en *min* y finalice en *max*, como el algoritmo. Por eso, la implementación es muy directa.

```
Option Explicit
Private Sub CommandButton1_Click()
Dim n, m, ncols
m = Cells(4, 1)
n = Cells(4, 2)
ncols = Cells(4, 4)
Call Imprimir(PrimosMN(m, n), 6, 1, ncols)
End Sub

Sub Imprimir(ByRef Arr() As Long, fi, co, Optional ncols As Variant)
...
End Sub

Function ERATOSTENES(n) As Long()
...
End Sub

Function isqrt(n) As Long
Dim xk, xkm1
If n = 1 Then
   xkm1 = 1
End If
If n > 1 Then
```

```
    xk = n
    xkm1 = n \ 2
    While xkm1 < xk
         xk = xkm1
         xkm1 = (xk + n \ xk) \ 2
    Wend
End If
isqrt = xkm1
End Function
’ m < n
Function PrimosMN(m, n) As Long()
Dim i, j, k, pos, contaPrimos, mp, q, q2
Dim min, max
Dim esPrimo() As Boolean
Dim primo() As Long
Dim PrimosMaN() As Long

min = Int((m + 1 - 3) \ 2)
max = Int((n - 3) \ 2)

ReDim esPrimo(min To max)
ReDim PrimosMaN((n - m + 2) \ 2)
For i = min To max
    esPrimo(i) = True
Next i

primo = ERATOSTENES(isqrt(n))

For i = 1 To UBound(primo)          ’primo(1)=3
    If m <= primo(i) * primo(i) Then
       k = (primo(i) - 1) \ 2
       While (2 * k + 1) * primo(i) <= n
       esPrimo(((2 * k + 1) * primo(i) - 3) \ 2) = False
       k = k + 1
       Wend
    End If
    If primo(i) * primo(i) < m Then
       q = (m - 1) \ primo(i)  ’p(q+k)-> p*k
       q2 = q Mod 2
       k = q2
       mp = (2 * k + 1 - q2 + q) * primo(i)  ’m\’ultiplos impares
        While mp <= n
```

```
            esPrimo((mp - 3) \ 2) = False
            k = k + 1
            mp = (2 * k + 1 - q2 + q) * primo(i)
        Wend
    End If
Next i

If m > 2 Then
    contaPrimos = 0
Else
    contaPrimos = 1
    PrimosMaN(0) = 2
End If

For i = min To max
   If esPrimo(i) Then
        PrimosMaN(contaPrimos) = 2 * i + 3
        contaPrimos = contaPrimos + 1 '3,5,...
    End If
Next i

If 1 <= contaPrimos Then
ReDim Preserve PrimosMaN(contaPrimos - 1)
Else
ReDim PrimosMaN(0)
End If

PrimosMN = PrimosMaN()
End Function
```

Bibliografía

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

[2] Hans Riesel. *Prime Numbers and Computer Methods for Factorization.* Springer; 2 edition. 1994.

[3] Paulo Ribenboim. *The New Book of Prime Number Records.* 3rd ed. Springer. 1995.

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

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 "·", 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 "·".
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:

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

donde p es primo.

- La aritmética módulo p es muy sencilla. Las operaciones con "+" y "·" 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 $a \equiv b \pmod{p/\text{mcd}(c,p)}$.

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 "·", tal que $(R, +)$ es un grupo abeliano, "·" 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 $v(p) = \text{grado}\, p$.
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,

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

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 $r_1(x)$,
$$\begin{array}{l|l} x^5 - 32 & x^3 - 8 \\ \cline{2-2} \cdots & x^2 \\ \hline \text{residuo: } 8x^2 - 32 & \end{array}$$

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

Ahora, dividimos $r_1(x)$ por $r_2(x)$,
$$\begin{array}{l|l} x^3 - 8 & 8x^2 - 32 \\ \cline{2-2} \cdots & x/8 \\ \hline \text{residuo: } 4x - 8 & \end{array}$$

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

Ahora, dividimos $r_2(x)$ por $r_3(x)$,
$$\begin{array}{l|l} 8x^2 - 32 & 4x - 8 \\ \cline{2-2} \cdots & 2x + 4 \\ \hline \text{residuo: } 0 & \end{array}$$

e) $r_4(x) = 0$.

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

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	12593338795500743100931141992187500

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: $c(x) = \mathrm{MCD}\,(a(x), b(x))$

```
$c(x) = \mathrm{pp}(a(x))$ ;
$d(x) = \mathrm{pp}(b(x))$ ;
while $d(x) \neq 0$ do
    $r(x) = \mathrm{prem}(c(x), d(x))$ ;
    $c(x) = d(x)$ ;
    $d(x) = \mathrm{pp}(r(x))$ ;
$\lambda = \mathrm{MCD}(\mathrm{cont}(a(x)), \mathrm{cont}(b(x)))$ ;
$g(x) = \lambda\, c(x)$ ;
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 $\begin{array}{l|l} yx^2+y^2x & yx^2+(y^2+1)x+y \\ \cline{2-2} -yx^2-(y^2+1)x-y & 1 \\ \hline \text{residuo: } -x-y & \end{array}$

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 $\begin{array}{l|l} yx^2+(y^2+1)x+y & x+y \\ \cline{2-2} -yx^2-y^2x & yx+1 \\ \hline x+y & \\ -x-y & \\ \hline \text{residuo: } 0 & \end{array}$

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 $\varrho = \text{Máx}_{1\leq i\leq k}\,\{\text{grado}\, r_{i-1}(x) - \text{grado}\, r_i(x)\}$ 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

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

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 $r_i(x)$, $\delta_i = \text{grado}\, r_{i-1}(x) - \text{grado} r_i(x)$ 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 $\psi_i = (-\text{cp}r_{i-1})^{\delta_{i-1}}\psi_{i-1}^{1-\delta_{i-1}}$ 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 $\textbf{quo}(a,b)$ para indicar el cociente de la división usual. Hay que notar que $\textbf{pquo}(r_{i-1}(x), r_i(x)) = \textbf{quo}(\alpha_i\, r_{i-1}(x), r_i(x))$

Algoritmo 1.4.1: Algoritmo PRS Subresultante.

Entrada: Polinomios $a(x), b(x) \in D[x],$ $\text{grado}\, a(x) \geq \text{grado}\, b(x),$ D DFU.

Salida: $c(x) = \text{MCD}\,(a(x), b(x))$

```
$r_0 = a(x)\,;$
$r_1 = b(x)\,;$
$deg_0 = \text{grado}(r_0),\ deg_1 = \text{grado}(r_1),\ cp_0 =$ 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
    $c = \alpha_1\, r_0,\ q = \text{quo}(c, r_1),\ r_0 = r_1 w\ r_1 = \text{quo}(c - q \cdot r_1, \beta_1)\,;$
    $deg_0 = \text{grado}(r_0),\ deg_1 = \text{grado}(r_1)\,;$
    $cp_0 =$ 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
        $\psi_1 = \text{quo}(-cp_0^{\delta_0}, \psi_0^{\delta_0-1})\,;$
    else
        $\psi_1 = -cp_0^{\delta_0} \cdot \psi_0$
    $\beta_1 = -cp_0 \cdot \psi_1^{\delta_1}\,;$
Normalizar salida;
$r_0 =$ MCD $(\text{cont}(a(x)), \text{cont}(b(x))) \cdot \text{pp}(r_0);$
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, $\text{rem}\,(a, b)$ 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 $-p/2 < u \leq p/2$
$u = \text{rem}(m, p)$ ;
if $u > p/2$ then
    $u = u - p$
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 $\xi^{n+1} > 2|u_{\text{max}}|$, donde $u_{\text{max}} = \max_e |u_e|$.

- $a(x) = \sum\limits_e u_e x^e = \sum\limits_e \left(\sum\limits_{i=0}^{n} u_{e,i}\xi^i \right) x^e = u_0(x) + u_1(x)\xi + ... + u_n(x)\xi^n.$
- 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 $-\xi/2 < u_i \leq \xi/2$.

```
$e = \gamma$;
$i = 0$;
while $e \neq 0$ do
    $u_i = \phi_\xi(e)$;
    $e = (e - u_i)/\xi$;
    $i = i + 1$;
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* $\xi \geq 2 \cdot \text{Mín}\{\|A\|_\infty, ||B||_\infty\} + 2$ *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* $pp(\overline{G}) \,|\, A$ *y* $pp(\overline{G}) \,|\, B$ *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 $\gamma = \text{MCD}\,(A, B) \in \mathbb{Z}$
3  $\xi = 2 \cdot \text{Mín}\{||A||_\infty, ||B||_\infty\} + 2$ ;
4  $i = 0$ ;
5  while $i < 7$ do
6      if length($\xi$) · máx{grado $A$, grado $B$, } > 5000 then
7          return $-1$ //$MCD \geq 0$, $-1$ se usa como indicador de fallo
8      $\gamma = \text{MCDHEU}(\phi_{(x-\xi)}(A), \phi_{(x-\xi)}(B))$ //llamada recursiva;
9      if $\gamma \neq -1$ then
10         Generar $G$ a partir de la expansión $\xi-$ádica de $\gamma$
11     //División en $\mathbb{Q}[x]$;
12     if $G|A$ y $G|B$ then
13         return pp($G$)
14     Crear un nuevo punto de evaluación;
15     $\xi = \text{quo}(\xi \times 73794,\ 27011)$
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

$$\begin{aligned}40 &= 4+1\cdot 36^1\\ G_2 &= 4+y\end{aligned}$$

- 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

$$\begin{aligned}4+y &= y\cdot 4^0+1\cdot 4^1\\ G_1 &= y+1\cdot x\end{aligned}$$

- 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.

- mcd$(78,32)=2$. En efecto;

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

- 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,

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

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

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

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

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

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 $g = sa + tb$.
$c = n(a),\ d = n(b)$;
$c_1 = 1,\ d_1 = 0$;
$c_2 = 0,\ d_2 = 1$;
while $d! = 0$ do
    $q = \text{quo}(c,d),\ r = c - qd$;
    $r_1 = c_1 - qd_1,\ r_2 = c_2 - qd_2$;
    $c = d,\ c_1 = d_1,\ c_2 = d_2$;
    $d = r,\ d_1 = r_1,\ d_2 = r_2$;
MCD $= n(c)$;
$s = c_1/(u(a) * u(c)), t = c_2/(u(b) * u(c))$;
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 $a(x) = \displaystyle\sum_{i=0}^{n} a_i x^i$ 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.

Memorias del Seminario de Teoría de Números y al Álgebra Computacional
Módulo I

Instituto Tecnológico de Costa Rica
Escuela de Matemática

2007

Índice general

Capítulo 1

¿Cómo evaluar expresiones matemáticas en el computador?

MSc. Alexander Borbón Alpízar
Escuela de Matemática
Instituto Tecnológico de Costa Rica.
aborbon@itcr.ac.cr

abstract

En este artículo se muestra una forma de programar un evaluador de expresiones matemáticas en JAVA. El programa se construye paso a paso y se explican detalladamente las partes más importantes del mismo. El evaluador consta de dos partes o módulos, el primero se encarga de convertir la expresión digitada a notación postfija que es más sencilla para el computador; el segundo es el que evalúa la expresión que se obtuvo en un valor específico. Para poder comprender y reescribir este programa se necesita tener conocimientos básicos en la programación en JAVA, sin embargo, se explicará el uso de varias primitivas utilizadas y de algunos conceptos básicos de programación.

1.1. Introducción

El objetivo principal de este artículo es mostrar una manera de programar un evaluador de expresiones matemáticas en JAVA. En el artículo se explicará tanto la teoría matemática sobre la evaluación de expresiones como la creación del programa mismo.

El evaluador se realiza en JAVA [4], ya que es un lenguaje de programación muy accesible (es "open source", que significa que puede ser utilizado de forma gratuita) y permite acoplarse fácilmente con un navegador de Internet. Además, este programa tiene clases con muchas funciones que permiten trabajar con texto, lo que hace más sencillo el trabajo. Sin embargo, las ideas pueden ser trasladadas a otros lenguajes de programación (de hecho, al final del artículo se encuentra la versión en JAVA y un programa similar para Visual Basic).

El evaluador consta de dos módulos, el primero se encarga de revisar que la expresión esté bien digitada y la "traduce" a una expresión más simple para que la computadora pueda "entenderla". En este caso, la expresión se pasará a notación postfija[1], esta notación lo que hace es escribir el operador después de los números que opera; por ejemplo, una expresión como $x + 1$ el programa la traduciría como "x 1 +".

---

[1] Conocida también como notación polaca invertida.

El segundo módulo lo que hace es evaluar un valor dado en la expresión en notación postfija (ya sea la última que se digitó o cualquier expresión en notación postfija). Este módulo también tomará en cuenta cuando se evalúa en un valor en donde la expresión no tiene sentido en los números reales (división por cero o raíz par de un número negativo); en estos casos se lanzará una excepción (un error no muy grave o una conclusión anormal en JAVA).

Para la programación de estos módulos se va a suponer que la ecuación que digita el usuario está bien escrita y en la última sección se explicará la idea general para poder verificarlo.

Este artículo está dirigido a profesores de matemáticas y programadores que quieran aprender a realizar un evaluador de expresiones matemáticas para sus proyectos o que les gustaría conocer un poco sobre la técnica que hay atrás de la programación de un evaluador.

Aunque en este momento el programador ansioso debe querer iniciar programando de una vez, se necesita primero comprender bien la teoría básica, así que antes de empezar a trabajar en JAVA se va a explicar la notación postfija, cómo se evalúa en ella y la procedencia de los operadores.

1.2. Notación Postfija

El primer módulo del programa se encargará de traducir la expresión que digita el usuario (en notación infija) en una expresión más simple, esta segunda expresión se dice que está en notación postfija; esta forma de escribir una expresión matemática tiene la ventaja que se evalúa de forma lineal por lo que es más sencillo para una computadora "entenderlo".

Al inicio talvez se ve un poco complejo, pero no es así; lo que hace el lenguaje es colocar primero los números con los que va a operar y luego escribe la operación, por ejemplo $a+b$ se escribiría "a b +"; $(5-8)*4$ se escribiría "5 8 - 4 *", otros ejemplos son:

Notación infija	Notación postfija
$\text{sen}(x)$	x sen
$1+3*4$	1 3 4 * +
$\text{sen}(9+x)$	9 x + sen

Notación infija	Notación postfija
$3*6$	3 6 *
$4-\text{sen}(x)$	4 x sen -
$\text{sen}(x^2)+4*x$	x 2 ˆ sen 4 x * +

Para evaluar una expresión en notación infija en un valor específico para x se deben seguir las reglas matemáticas para la prioridad de las operaciones:

1. Las potencias tienen prioridad sobre cualquier operación
2. La multiplicación y la división tienen prioridad sobre la suma y la resta.
3. Si se presenta un paréntesis, se deben realizar primero las operaciones dentro de éste. Si hay un paréntesis dentro de otro tiene prioridad el paréntesis interno.

Por el contrario, en la notación postfija siempre se trabaja de izquierda a derecha; note además que la notación postfija no tiene paréntesis por la forma lineal en que se lee. Comparemos un ejemplo en donde se evalúa un valor en una expresión utilizando estas dos notaciones.

1.2.1. Evaluación en notación infija y postfija

Tomemos un ejemplo sencillo como $4+x^3$ cuando $x=2$, esta expresión se escribe en la notaciones anteriores de la siguiente manera:

Notación infija: 4 + x ˆ 3
Notación postfija: 4 x 3 ˆ +

Para evaluar $4 + x^3$ en notación infija primero se debe tomar el 2 (en vez de la x), elevarlo al cubo y luego sumarle 4; se nota que esto no está en forma lineal, es decir, se debe ir primero a la segunda parte de la expresión, evaluarla y luego sumarle al resultado el cuatro.

Es muy complejo hacer que un programa evalue de esta forma. Si la expresión “$4 + x^3$” se traduce a la forma “4 x 3 ˆ +”, entonces se convierte en una expresión más sencilla; para evaluar esta expresión en $x = 2$, el algoritmo es leer el texto de izquierda a derecha, si se encuentra un número lo apila[2] y las operaciones se irán realizando conforme aparezcan.

Así, en el ejemplo “4 x 3 ˆ +” se toma el 4 y se mete en una pila de números, el segundo valor que se toma es la x, en vez de ésta introducimos el 2 (que es el valor que estamos evaluando), luego se toma el 3 y se mete nuevamente en la pila de números, es decir, tenemos una pila de números como sigue

3
2
4

Ahora sigue una potencia, por lo que se toman los dos últimos números de la pila (recuerde que tomamos los dos de arriba: 2 y 3) y se realiza la potencia $2^3 = 8$, así, el 2 y el 3 se sustituyen en la pila por 8, se obtiene la pila

8
4

Por último, sigue un signo de suma, éste también se realiza con dos números, por lo que se toman el 4 y el 8, el resultado es $4 + 8 = 12$ y en la pila queda un 12; como ya se acabó la expresión entonces el resultado es 12.

Aunque a primera vista esta forma de evaluar parece más compleja, para un programa no lo es, ya que se siguen los pasos en forma lineal (la expresión se lee de izquierda a derecha), la algoritmo estaría compuesto por tres reglas:

1. Si lo que sigue en la expresión es un número, se agrega a la pila de números.
2. Si sigue una operación que ocupa dos números (como la suma y la resta) se sacan los dos últimos números de la pila, se realiza la operación y se introduce el resultado en la pila.
3. Si es una función que ocupa un solo número (como seno o coseno) entonces se saca un número de la pila, se evalúa y se guarda el resultado.

Con este ejemplo se observa que evaluar una expresión en notación postfija es más sencillo que hacerlo en notación infija, por lo que lo primero que se hará el programa es “traducir” la expresión que digite el usuario a dicha notación. Veamos primero la teoría sobre cómo se hará esto.

1.3. Traducción a notación postfija

Para la traducción a notación postfija se utilizarán dos pilas, una en donde se guardarán los números y otra para los operadores y los paréntesis[3]. Aunque se dice que una de las pilas maneja números, esto no es

---

[2] En computación, existe una estructura de datos que utiliza la metáfora de una pila de objetos, por ejemplo, piense en una pila de documentos, si uno agrega un nuevo documento lo coloca encima de la pila y si uno toma un documento lo toma de arriba de la pila; es decir, una pila es una estructura de datos en donde el último objeto en entrar es el primero en salir. Como referencia ver [1].

[3] Existe un algoritmo para este programa que utiliza una sola pila, sin embargo, consideramos que es más sencillo y didáctico con dos.

muy cierto, en realidad las dos pilas se trabajarán con texto (*String*), esto permite concatenar varias tiras fácilmente y se pueden manejar expresiones que no son números como si lo fueran.

Para la traducción del lenguaje natural a notación postfija es fundamental manejar la prioridad de las operaciones y el uso de paréntesis.

Como se dijo al inicio del artículo, para evaluar una expresión matemática se deben seguir las siguientes reglas:

1. Primero se evalúan las potencias.
2. A continuación siguen las multiplicaciones, las divisiones y el resto de la división entera (%).
3. Por último se realizan las sumas y las restas.
4. Si hay paréntesis, se hace primero la expresión que está dentro del paréntesis interno.

Por esto, para la prioridad se les dará a las operaciones los siguientes valores:

Operador	Prioridad
+, -	0
*, /, %	1
ˆ	2

Por lo que en nuestro programa ocupamos una función como la que sigue:

```
private int prioridad(char s) {
   if (s=='+' || s=='-')
      return 0;
   else if (s=='*' || s=='/' || s=='%')
      return 1;
   else if (s=='^')
      return 2;

   return -1;
} //Fin de la funcion prioridad
```

Esta función recibe un caracter *s*, dependiendo de la operación que represente este caracter se devuelve el valor de la prioridad correspondiente, si recibe un operador que no corresponde con las operaciones se devuelve -1.

Otro punto importante es que todas las funciones que reciben un parámetro (seno, coseno, tangente,...) se manejan como un paréntesis que abre "(", el usuario debe digitarlas con ese paréntesis, es decir: "sen(", "cos(", "tan(", ... Se manejan así porque cuando se digita el paréntesis que cierra ")", este paréntesis saca todo lo que hay en la pila hasta que encuentra el de apertura o una función (lo que quiere decir que en términos prácticos funcionan igual con una ligera diferencia que luego se verá).

A continuación se muestran varios ejemplos para observar el algoritmo que se debe seguir para traducir una expresión de notación infija a postfija.

1.3.1. Una expresión simple

Como primer ejemplo se va a traducir una expresión simple como 3*x+4, la prioridad indica que primero se tiene que hacer la multiplicación y luego la suma; para esto, recuerde que la función le asigna prioridad 0 a la suma y 1 a la multiplicación y se dará la regla que una prioridad inferior saca de la pila cualquier operación con prioridad igual o superior a ella.

Dado esto, se pueden notar que:

1. Los operadores “+” y “-” son los que tienen menor prioridad (0), por lo que siempre sacarán todos los operadores precedentes.
2. Los operadores “*”, “/” y “%” sacan a la potencia “ˆ” y a ellos mismos.
3. El operador “ˆ” es el que tiene mayor prioridad, no saca a nadie, ni siquiera a sí mismo ya que la prioridad para realizar las potencias es de derecha a izquierda (contrario a todas las demás operaciones matemáticas), es decir: $2^{3^4} = 2^{\left(3^4\right)}$ o de manera escrita 2ˆ3ˆ4=2ˆ(3ˆ4)

Por lo tanto, se van a sacar los elementos de la expresión 3*x+4; primero se toma el 3 y se introduce en la pila de números, luego se toma el * y se mete en la pila de operadores, se obtiene

Números	Operadores
3	*

Luego se saca la x, ésta se debe manejar como un número pues cuando se evalúe así será, esta se mete en la pila de números

Números	Operadores
x	*
3	

Ahora se debe sacar un + que se tendría que introducir en la pila de operadores, pero tiene prioridad cero que es menor que la prioridad de la multiplicación (que es 1), por lo que la multiplicación se debe hacer antes que la suma (la multiplicación tienen prioridad sobre la suma). Para este caso se sacan dos números de la pila y los “multiplicamos” obteniendo “3 x *” (recuerde que en realidad manejamos texto, cuando se dice que lo multiplicamos se quiere decir que se escribe la multiplicación en notación postfija). El resultado se introduce en la pila como un número (ahora se maneja como si toda esta expresión fuera un número porque es como si ya se hubiera evaluado) y la suma se mete en la pila de operadores, se obtienen las pilas

Números	Operadores
3 x *	+

Ahora se saca el número 4 y se mete en la pila de números

Números	Operadores
4	+
3 x *	

Aquí ya se acaba la expresión por lo que se saca todo lo que queda en la pila. Para cada operador se deben tomar los valores necesarios de la pila de números; en este caso sólo hay una suma y dos números, al realizar la operación se sacan los dos números y se escribe en postfijo “3 x * 4 +” pues se colocan los dos números primero y luego la operación.

1.3.2. Una expresión con funciones

Se tomará como segundo ejemplo una expresión con más elementos, por ejemplo:

$$1+3*\tan(2*(1+x)-1)$$

En este caso se toma el 1, el + y el 3

Números
3
1

Operadores
+

Sigue la multiplicación, pero esta sólo saca la potencia y las de su mismo nivel por lo que no saca la suma, simplemente se agrega a los operadores

Números
3
1

Operadores
*
+

Sigue la función tangente que se maneja dentro de los operadores como un paréntesis que abre (en la pila le quitamos el paréntesis que en postfijo no se necesita)

Números
3
1

Operadores
tan
*
+

Ahora se deben incluir el 2 y el *. A la hora de incluir *, tangente no funciona como un operador sino como el inicio de otra expresión por lo que * no lo puede sacar; aquí no hay problema con la función prioridad porque a tan le asignaría un -1 que no lo saca nadie. También se agrega el paréntesis de apertura

Números
2
3
1

Operadores
(
*
tan
*
+

Ahora incluimos el 1, el + y la x

Números
x
1
2
3
1

Operadores
+
(
*
tan
*
+

Viene el cierre de paréntesis, este sacaría todos los elementos hasta que haya una apertura (un paréntesis o una función). En este caso solo debe sacar el + y se quita el paréntesis de la pila.

Números
1 x +
2
3
1

Operadores
*
tan
*
+

Sigue un - que tiene menor prioridad que el *, por lo que tenemos que sacar la multiplicación antes de meter el menos.

Números
2 1 x + *
3
1

Operadores
-
tan
*
+

Sigue un uno y luego un cierre de paréntesis que sacaría hasta tangente. En este caso, se debe sacar el menos que queda; la tangente, contrario al paréntesis de apertura, se debe agregar al final del texto para que se evalúe en la expresión.

Números
2 1 x + * 1 - tan
3
1

Operadores
*
+

Aquí se acaba la expresión por lo que se debe sacar todo lo que queda, obteniendo como resultado “1 3 2 1 x + * 1 - tan * +”

1.4. Programa que traduce de notación infija a postfija

Recuerde que para esta parte del programa se va a suponer que la expresión que digita el usuario no tiene errores, luego se darán consejos para detectarlos.

En los ejemplos de la sección anterior se observó que para realizar la traducción de la expresión se necesitan dos pilas de *Strings* (texto), una para los números y otra para los operadores. JAVA ya posee una clase que maneja pilas, esta clase se llama *Stack* que se encuentra dentro del paquete *java.util*, lo primero que tenemos que hacer es llamar a esta librería y hacer nuestra clase *Parseador*[4]; el inicio de nuestro programa se debe ver como:

```
import java.util.*;

public class Parseador{
...
}//fin de Parseador
```

Para crear una nueva pila se debe definir como un nuevo objeto:

```
Stack nuevaPila = new Stack();
```

La clase *Stack* se maneja con objetos (introduce objetos y saca objetos); para introducir un nuevo objeto dentro de la pila se utiliza la instrucción

```
nuevaPila.push(Objeto);
```

---

[4] Aquí se usa la palabra Parseador como un nombre simbólico para el programa, en realidad esta palabra no existe en español sino que se hace referencia a la palabra parser en inglés que tiene un significado similar a “traductor”, por supuesto el usuario puede utilizar la palabra que guste.

Para sacar un objeto de la pila (recuerde que una pila saca el último objeto que se introdujo) utilizamos

```
nuevaPila.pop();
```

Para "mirar" un objeto de la pila sin sacarlo se usa

```
nuevaPila.peek()
```

Además se puede preguntar si la pila está vacía con la instrucción

```
nuevaPila.empty()
```

que devuelve *True* si está vacía o *False* si no lo está.

Para empezar con la clase *Parseador*, definimos la variable global *ultimaParseada* como sigue:

```
private String ultimaParseada;
```

Esta guarda un registro de la última expresión parseada (o traducida) en notación postfija, la expresión se guarda por si alguien quiere evaluar sin tener que dar la expresión en donde se evalúa.

El constructor de nuestra clase lo único que hace es poner *ultimaParseada* en 0.

```
public Parseador(){
   ultimaParseada="0";
}
```

La función parsear se define de tal forma que recibe un texto con la expresión en notación infija y devuelve otro texto con la expresión en notación postfija. La función lanza una excepción (*SintaxException*) si encuentra que la expresión está mal digitada.

```
public String parsear(String expresion) throws SintaxException{
   Stack PilaNumeros=new Stack(); //Pila de números
   Stack PilaOperadores= new Stack(); //Pila de operadores
   String fragmento;
   int pos=0, tamano=0;
   byte cont=1;
   final String funciones[]={"1 2 3 4 5 6 7 8 9 0 ( ) x e + - * / ^ %",
         "pi",
         "ln(",
         "log( abs( sen( sin( cos( tan( sec( csc( cot( sgn(",
         "rnd() asen( asin( acos( atan( asec( acsc( acot( senh( sinh( cosh( tanh(
              sech( csch( coth( sqrt(",
         "round( asenh( acosh( atanh( asech( acsch( acoth("};
   final private String parentesis="( ln log abs sen sin cos tan sec csc cot asen asin
              acos atan asec acsc acot senh sinh cosh tanh sech csch coth sqrt round";
   final private String operadoresBinarios="+ - * / ^ %";

   //La expresión sin espacios ni mayúsculas.
   String expr=quitaEspacios(expresion.toLowerCase());
```

La función necesita dos pilas: *PilaNumeros* y *PilaOperadores*.

La variable *fragmento* se encargará de guardar el fragmento de texto (*String*) que se esté utilizando en el momento (ya sea un número, un operador, una función, etc.). La variable *pos* va a marcar la posición del caracter que se está procesando actualmente en el *String*.

La variable *funciones* contiene un arreglo de textos (*String*), en la primera posición tiene todas las expresiones de un caracter que se aceptarán, en la segunda posición están las expresiones de dos caracteres y así hasta llegar a la posición seis.

La variable *parentesis* contiene a todas las expresiones que funcionarán como paréntesis de apertura.

En *operadoresBinarios* se guardan todos los operadores que reciben dos parámetros.

Todas estas definiciones se hacen como texto (*String*) para después poder comparar si la expresión que el usuario digitó concuerda con alguno de ellos.

Se define también el *String* en donde se guarda la expresión sin espacios ni mayúsculas (*expr*); la función *toLowerCase()* ya está implementada en JAVA en la clase *String* mientras que *quitaEspacios* es una función que se define de la siguiente manera

```
private String quitaEspacios(String expresion){
   String unspacedString = "";  //Variable donde guarda la función

   //Le quita los espacios a la expresión que leyó
   for(int i = 0; i < expresion.length(); i++){
      if(expresion.charAt(i) != ' ')
         unspacedString += expresion.charAt(i);
   }//for

   return unspacedString;
}//quitaEspacios
```

Esta función va tomando cada uno de los caracteres de la expresión, si el caracter no es un espacio lo agrega al texto sin espacios *unspacedString*.

La excepción que lanza el programa también se define como una clase privada

```
private class SintaxException extends ArithmeticException{
   public SintaxException(){
      super("Error de sintaxis en el polinomio");
   }

   public SintaxException(String e){
      super(e);
   }
}
```

Volviendo a la función parsear, lo que seguiría es realizar un ciclo mientras no se acabe la expresión, es decir

```
   try{
      while(pos<expr.length()){
```

Lo que se haría dentro del *while* es ver si lo que sigue en la expresión es algo válido y tomar decisiones dependiendo si es un número, un operador, un paréntesis o una función.

El código:

```
   tamano=0;
   cont=1;
   while (tamano==0 && cont<=6){
      if(pos+cont<=expr.length() &&
           funciones[cont-1].indexOf(expr.substring(pos, pos+cont))!=-1){
         tamano=cont;
      }
      cont++;
   }
```

Hace que el contador vaya de 1 a 6 que es la máxima cantidad de caracteres que tiene una función y se inicializa *tamano* en cero. Luego se pregunta si la posición actual (*pos*) más el contador (*cont*) es menor de la longitud del texto y si el fragmento de texto que sigue está en alguna de las funciones, si esto pasa el siguiente texto que se tiene que procesar es de tamaño *cont*.

Ahora se van tomando algunos casos con respecto al tamaño encontrado.

```
if (tamano==0){
   ultimaParseada="0";
   throw new SintaxException("Error en la expresión");
```

Si *tamano* continúa siendo cero quiere decir que el fragmento de texto que sigue no coincidió con ninguna función ni con algo válido por lo que se lanza una excepción y se pone la última expresión parseada en 0.

```
}else if(tamano==1){
```

Pero si el tamaño es uno tenemos varias opciones, la primera es que sea un número

```
if(isNum(expr.substring(pos,pos+tamano))){
   fragmento="";
do{
   fragmento=fragmento+expr.charAt(pos);
   pos++;
}while(pos<expr.length() && (isNum(expr.substring(pos,pos+tamano)) ||
      expr.charAt(pos) == '.' || expr.charAt(pos) == ','));
   try{
      Double.parseDouble(fragmento);
   }catch(NumberFormatException e){
      ultimaParseada="0";
      throw new SintaxException("Número mal digitado");
   }
   PilaNumeros.push(new String(fragmento));
   pos--;
```

En la primera línea, para preguntar si el caracter es un número se utiliza la función *isNum* definida por

```
private boolean isNum(char s) {
   if (s >= '0' && (s <= '9'))
      return true;
   else
      return false;
} //Fin de la funcion isNum
```

Que devuelve *True* si el caracter que recibe es un número (está entre 0 y 9) o *False* si no lo es.

Si el caracter actual es un número existe un problema, no se sabe cuántos dígitos tenga este número, por lo que se sacan todos los caracteres que sigan que sean números, puntos o comas.

En la variable *fragmento* se va a guardar el número, por lo que al inicio se debe vaciar (*fragmento=" "*). Luego se hace un ciclo mientras no se acabe la expresión y el caracter que sigue sea un número, un punto o una coma: *while(pos<expr.length() && (isNum(expr.substring(pos,pos+tamano)) || expr.charAt(pos) == '.' || expr.charAt(pos) == ','))*{ .

Luego, la variable *fragmento* se trata de convertir a *double* y se mete en la pila de números; si no la puede convertir el programa lanza una excepción en el bloque *try-catch*.

De esta manera se maneja un número cualquiera. Ahora la segunda posibilidad con tamaño uno es que el caracter sea x o e. Estos dos casos se manejan como un número que se mete en la pila.

```
   else if (expr.charAt(pos)=='x' || expr.charAt(pos)=='e'){
      PilaNumeros.push(expr.substring(pos,pos+tamano));
```

Si el caracter que sigue es uno de los operadores +, *, / y % (el menos '-' se maneja igual, pero se recomienda ponerlo en un caso aparte para manejar posteriormente los menos unarios) se deben sacar todos los operadores con prioridad mayor o igual a ellos y meter el operador en la pila

```
   }else if (expr.charAt(pos)=='+' || expr.charAt(pos)=='*' ||
         expr.charAt(pos)=='/' || expr.charAt(pos)=='%'){
      sacaOperadores(PilaNumeros, PilaOperadores, expr.substring(pos,pos+tamano));
```

En este caso se declara una función (ya que se utilizará mucho al manejar detalles posteriores) que realice el trabajo de sacar operadores de la pila, esta función se define de la siguiente manera

```
private void sacaOperadores(Stack PilaNumeros, Stack PilaOperadores, String operador){
   final String parentesis="( ln log abs sen sin cos tan sec csc cot sgn asen asin
         acos atan asec acsc acot senh sinh cosh tanh sech csch coth sqrt round asenh
         asinh acosh atanh asech acsch acoth";
   while(!PilaOperadores.empty() &&
         parentesis.indexOf((String)PilaOperadores.peek( ))==-1 &&
         ((String)PilaOperadores.peek()).length()==1 &&
         prioridad(((String)PilaOperadores.peek()).charAt(0))>=
         prioridad(operador.charAt(0))){
      sacaOperador(PilaNumeros, PilaOperadores);
   }
   PilaOperadores.push(operador);
}
```

Aquí se vuelve a declarar la variable *parentesis* para el *while*. En este *while* se indica que se deben sacar operadores mientras haya algo en la pila, lo que siga en la pila no sea un paréntesis, sea de un solo caracter y cuya prioridad sea mayor o igual a la prioridad del operador que se está procesando en ese momento; luego se guarda el operador en la pila correspondiente.

Observe que el *while* llama a *sacaOperador* que es una función definida posteriormente cuyo código es

```
private void sacaOperador(Stack Numeros, Stack operadores) throws EmptyStackException{
   String operador, a, b;
   final String operadoresBinarios="+ - * / ^ %";
   try{
      operador=(String)operadores.pop();

      if(operadoresBinarios.indexOf(operador)!=-1){
         b=(String)Numeros.pop();
         a=(String)Numeros.pop();
         Numeros.push(new String(a+" "+b+" "+operador));
      }else{
         a=(String)Numeros.pop();
         Numeros.push(new String(a+" "+operador));
      }
   }catch(EmptyStackException e){
      throw e;
   }
}//sacaOperador
```

Esta función se encarga de sacar dos números de la pila correspondiente, esto si es un operador binario o un número si es unario; luego introduce el nuevo número en notación postfija en la pila. Esta función recibe las dos pilas y no devuelve nada, si se le acaban los elementos a alguna de las pilas se lanza una excepción. Esta es la última función externa que se va a necesitar para traducir la expresión.

Con la potencia (ˆ) es más sencillo ya que este operador no saca a nadie, el código quedaría de la siguiente manera.

```
   }else if (expr.charAt(pos)=='^'){
      PilaOperadores.push(new String("^"));
```

En este caso, simplemente se mete el operador a su pila correspondiente.

Si el caracter fuera un paréntesis de apertura simplemente se mete en la pila de operadores.

```
   }else if (expr.charAt(pos)=='('){
      PilaOperadores.push(new String("("));
```

Y si el caracter es un paréntesis de cierre se deben sacar operadores hasta encontrar una apertura de paréntesis en la pila.

```
   }else if (expr.charAt(pos)==')'){
      while(!PilaOperadores.empty() &&
            parentesis.indexOf(((String) PilaOperadores.peek()))==-1){
         sacaOperador(PilaNumeros, PilaOperadores);
      }
      if(!((String)PilaOperadores.peek()).equals("(")){
         PilaNumeros.push(new String(((String)PilaNumeros.pop()) + " " +
               PilaOperadores.pop()));
      }else{
         PilaOperadores.pop();
      }
   }
```

Si el paréntesis de apertura no era el caracter "(" (es decir, era una función) entonces la concatena al final del texto en la notación postfija, si era un paréntesis simplemente lo desecha (recuerde que en notación postfija no hay paréntesis).

Aquí se terminan de procesar todos los elementos posibles de un solo caracter, ahora se pasará al caso de dos o más caracteres

```
   }else if(tamano>=2){
      fragmento=expr.substring(pos,pos+tamano);
      if(fragmento.equals("pi")){
         PilaNumeros.push(fragmento);
      }else if(fragmento.equals("rnd()")){
         PilaNumeros.push("rnd");
      }else{
         PilaOperadores.push(fragmento.substring(0,fragmento.length()-1));
      }
   }
   pos+=tamano;
}//Fin del while
```

En este caso se toma la expresión en *fragmento*, si *fragmento* es igual a "*pi*", lo metemos en la pila de números; lo mismo hacemos si es *rnd()*.

Cualquier otra posibilidad de tamaño mayor que dos es que sea una función por lo que se mete en la pila de operadores quitándole el paréntesis. Al final se aumenta la posición de acuerdo al tamaño del texto procesado para pasar al siguiente caracter en la expresión.

Ya cuando se acabó la expresión se debe procesar todo lo que quedó en las pilas en el proceso final, esto se hace con un while que saque todos los operadores que quedaron

```
      //Procesa al final
      while(!PilaOperadores.empty()){
         sacaOperador(PilaNumeros, PilaOperadores);
      }

   }catch(EmptyStackException e){
      ultimaParseada="0";
      throw new SintaxException("Expresión mal digitada");
   }

   ultimaParseada=((String)PilaNumeros.pop());

   if(!PilaNumeros.empty()){
      ultimaParseada="0";
      throw new SintaxException("Error en la expresión");
   }

   return ultimaParseada;
}//Parsear
```

Si hubo algún error con las pilas en el proceso se lanza una excepción y se pone a *ultimaParseada* en cero. Al final se devuelve *ultimaParseada*.

Con esto ya acabamos el programa que convierte una expresión matemática del lenguaje natural a la notación postfija.

1.5. Evaluación de expresiones postfijas

Ahora que se tiene el traductor de expresiones en notación postfija, lo que hace falta es el segundo módulo, es decir, el que evalúa una expresión en notación postfija en un número dado.

Al igual que para el módulo anterior, se va a iniciar mostrando varios ejemplos que realizan esta evaluación para observar cuál es el procedimiento que se debe seguir.

1.5.1. Evaluación en notación postfija: primer ejemplo

En este caso vamos a iniciar evaluando en la expresión “x 2 ˆ sen 4 x * +” cuando $x = 3$. Recuerde que para evaluar en postfijo se leía la expresión de izquierda a derecha y se seguían tres reglas a saber:

1. Si lo que sigue en la expresión es un número, se agrega a la pila de números.
2. Si sigue una operación que ocupa dos números (como la suma y la resta) se sacan los dos últimos números de la pila, se realiza la operación y se introduce en la pila el resultado.
3. Si es una función que ocupa un solo número (como seno o coseno) entonces se saca un número de la pila, se evalúa y se guarda el resultado.

Por lo tanto en este caso se inicia tomando la x y se introduce un 3 en la pila (es el valor que se evalúa en vez de la x), luego se toma el 2 y también se introduce en la pila obteniendo

2
3

Luego sigue un ˆ por lo que se toman los dos valores y se obtiene $3^2 = 9$, nos queda una pila de un elemento

9

Ahora sigue sen, esta función se realiza sobre un solo número (el único que hay en la pila), se obtiene $\text{sen}(9) = 0{,}412118485241757$, este es el nuevo valor en la pila

0.412118485241757

Sigue agregar a la pila un 4 y un 3 (en vez de la x)

3
4
0.412118485241757

Sigue una multiplicación, por lo que tomamos el 4 y el 3 de la pila y le metemos $4 \cdot 3 = 12$ a la pila, nos queda

12
0.412118485241757

Por último, queda una suma, es decir $0{,}412118485241757 + 12 = 12{,}412118485241757$ que es el resultado final.

1.5.2. Evaluación en notación postfija: segundo ejemplo

Un segundo ejemplo antes de programar el módulo será evaluar la expresión

$$\operatorname{asen}\left(\tan\left(\ln\left(x+\pi\right)\right)\right)+x^{x^2+3\cdot x+5}$$

que en notación postfija se escribe “x pi + ln tan asen x x 2 ˆ 3 x * + 5 + ˆ +” cuando $x=-1$. Se toma la x, es decir, se introduce el -1 en la pila, luego pi

3.14159265358979
-1

Ahora sigue la suma de estos términos

2.14159265358979

Ahora se le saca logaritmo natural a este valor

0.761549782880893

Sigue el cálculo de la tangente

0.953405606022648

Ahora se calcula el arcoseno

1.26433002209559

Sigue introducir dos veces -1 (en vez de x) en la pila y un dos

2
-1
-1
1.26433002209559

Ahora sigue una exponente, $(-1)^2$=1

1
-1
1.26433002209559

Se agrega un 3 y un -1

-1
3
1
-1
1.26433002209559

Se multiplica $3\cdot -1=-3$

-3
1
-1
1.26433002209559

Se suma $1 + -3 = -2$

-2
-1
1.26433002209559

Se agrega un 5

5
-2
-1
1.26433002209559

Se suma, $-2 + 5 = 3$

3
-1
1.26433002209559

Sigue un exponente $(-1)^3 = -1$

-1
1.26433002209559

Por último, se suman $1{,}26433002209559 + -1 = 0{,}26433002209559$, que es el resultado final.

1.6. Función que evalúa expresiones

Para hacer este módulo se inicia declarando la función para evaluar, la cual recibe la expresión en notación postfija y el número que se va a evaluar (que es un *double*); la función devuelve el resultado de la evaluación (que también es un *double*). En este caso la función se llamará *f* para simular que se evalúa en una función *f(x)*. Esta función lanzará una excepción (*ArithmeticException*) si encuentra un error en la expresión.

```
public double f(String expresionParseada, double x) throws ArithmeticException{
   Stack pilaEvaluar = new Stack(); //Pila de double’s para evaluar
   double a, b;
   StringTokenizer tokens=new StringTokenizer(expresionParseada);
   String tokenActual;
   ...
```

En esta función se declaran algunas variables. En un principio se declara la pila que guardará los números (*pilaEvaluar*), luego se declaran dos números a y b en donde se guardarán los elementos que se sacan de la pila para ser operados.

El *StringTokenizer* es otra clase que se define en la librería *java.util*, ésta toma un texto y lo separa en unidades lexicográficas (llamadas lexemas o “tokens” en inglés), cada palabra separada por un espacio es un lexema; por ejemplo, el texto “2 3 + sen” tiene cuatro lexemas: “2”, “3”, “+” y “sen”. El *tokenActual* guarda la unidad lexicográfica que se procesa en un momento dado.

Para saber si hay más lexemas en una expresión se utiliza *tokens.hasMoreTokens()* que devuelve *True* o *False* y para sacar el siguiente lexema se usa *tokens.nextToken()*. Así, lo que falta en nuestra función es ir pasando por los lexemas e ir haciendo cálculos o guardando números según corresponda, el código sería el siguiente

```
try{
   while(tokens.hasMoreTokens()){
      tokenActual=tokens.nextToken();
      ...
   }//while
}catch(EmptyStackException e){
   throw new ArithmeticException("Expresión mal parseada");
}catch(NumberFormatException e){
   throw new ArithmeticException("Expresión mal digitada");
}catch(ArithmeticException e){
   throw new ArithmeticException("Valor no real en la expresión");
}

a=((Double)pilaEvaluar.pop()).doubleValue();

if(!pilaEvaluar.empty())
   throw new ArithmeticException("Expresión mal digitada");

return a;

}//funcion f
```

Los bloques *try-catch-throw* son los que lanzan las excepciones si en algún momento se acabaron los elementos en la pila (*EmptyStackException*), si hubo algún número que no pudo entender (*NumberFormatException*) o si hubo algún cálculo que no corresponde a un número real (*ArithmeticException*).

El proceso de analizar lexemas se repetirá mientras hallan más lexemas *while(tokens.hasMoreTokens())*{ y se saca el siguiente lexema con la instrucción *tokenActual=tokens.nextToken();*.

Al final del proceso, sacamos el último valor de la pila con la instrucción *a=((Double)pilaEvaluar.pop()).doubleValue();* lo que hace es sacar el elemento con *pilaEvaluar.pop()*, luego lo convierte a un objeto *Double* (recuerde que la pila trabaja con objetos) y, por último, calcula su valor double primitivo (con *doubleValue()*).

Si este no era el último valor en la pila, quiere decir que hubo un error al evaluar (faltaron operadores) y se lanza una excepción con el bloque

```
if(!pilaEvaluar.empty())
   throw new ArithmeticException("Expresión mal digitada");
```

Si no hubo ningún error entonces devuelve el valor encontrado.

Ahora veamos el bloque de código que hace falta dentro del *while*.

Si el lexema actual es alguno de los números “e”, “pi” o “x”, simplemente lo metemos en la pila, tomemos por ejemplo el número “e”:

```
if(tokenActual.equals("e")){
   pilaEvaluar.push(new Double(Math.E));
```

Al meter el número *Math.E* en la pila (este número es un *double*) se debe introducir como un objeto *Double*, por eso se utiliza el código *new Double*.

Cuando se tiene que meter "x" lo que se hace es introducir en la pila el valor que recibe la función. En todos los siguientes casos de este *if* utilizamos *elseif*; en este caso se utilizó *if* por ser el primero.

Por otro lado, si el lexema siguiente es un operador que recibe dos números, entonces primero se saca los dos números de la pila, y se guarda el resultado de la operación en la pila, tomando como ejemplo la suma.

```
}else if(tokenActual.equals("+")){
   b=((Double)pilaEvaluar.pop()).doubleValue();
   a=((Double)pilaEvaluar.pop()).doubleValue();
   pilaEvaluar.push(new Double(a+b));
```

De igual manera se hace para funciones con un solo número, como logaritmo.

```
}else if(tokenActual.equals("ln")){
   a=((Double)pilaEvaluar.pop()).doubleValue();
   pilaEvaluar.push(new Double(Math.log(a)));
```

Por último, si no fue nada de lo anterior, la única posibilidad que queda es que sea un número, este caso se toma el lexema y se trata de convertir en un *double* con *Double.parseDouble(tokenActual)*, si no puede automáticamente se lanza la excepción; el número se convierte en un nuevo objeto *Double* y se mete en la pila. El código quedaría

```
}else{
   pilaEvaluar.push(new Double(Double.parseDouble( tokenActual)));
}
```

Agregando los casos correspondientes para cada una de las funciones que se admiten, ya se tiene un evaluador de expresiones en notación postfija funcional.

En el programa de ejemplo que se muestra al final del artículo, se admite lo siguiente:

- Números "especiales": e, pi, x.
- Operadores: +, -, *, /, %, ^.
- Funciones: ln, log, abs, sen, sin, cos, tan, sec, csc, cot, sgn, asen, asin, acos, atan, asec, acsc, acot, senh, sinh, cosh, tanh, sech, sech, coth, sqrt, asenh, asinh, acosh, atanh, asech, acsch, acoth, round.
- Números reales, por ejemplo 2,15, -20,5, etc.

El evaluador se puede modificar para poder admitir la variable y o la z por si queremos evaluar funciones de varias variables, además se pueden agregar otras funciones con modificaciones mínimas (otra variable se manejaría igual que la x, para otra función habría que manejarla como las demás funciones con su cálculo correspondiente).

Este evaluador también lo sobrecargamos para que pueda admitir la forma *f(x)* que evaluaría la función en la última expresión parseada (recuerde que la última expresión parseada se guardaba en el módulo anterior en una variable global llamada *ultimaParseada*).

```
public double f(double x) throws ArithmeticException{
   try{
      return f(ultimaParseada,x);
   }catch(ArithmeticException e){
      throw e;
   }
}//Fin de la funcion f
```

Esta función se puede probar introduciendo cualquier expresión en notación postfija y un número para evaluar.

Uniendo estos dos programas, se tiene una poderosa herramienta para evaluar expresiones y funciones matemáticas.

Para poder usar esta clase dentro de un programa se debe crear un objeto (*Obj*) de tipo *Parseador*.

```
Parseador miparser = new Parseador();
```

Para parsear una expresión *expr* se escribe *miparser.parsear(expr)*, la función devuelve un *String* con *expr* en notación postfija, además el programa también guarda de manera automática la última expresión parseada. Para evaluar el número x en la expresión se utilizar *miparser.f(x)* para evaluar en la última expresión o se puede pasar una expresión en notación postfija escribiendo *miparser.f(exprEnPostfija, x)*.

Así, por ejemplo, el siguiente es el código de un applet[5] en donde se utiliza la clase *Parseador*

```
import java.applet.*;
import java.awt.*;

public class PruebaParseador extends java.applet.Applet {
  //Constructor del parseador
  Parseador miparser=new Parseador();
  //Expresión a parsear
  String expresion=new String();
  //Valor en el que se va a evaluar
  double valor=0;
  //Textfield donde se digita la expresión a parsear
  TextField inputexpresion = new TextField("x + 5");
  //Textfield donde se digita el valor a evaluar en la expresión
  TextField inputvalor = new TextField("0",5);
  //Botón para evaluar
  Button boton= new Button("Evaluar la expresión");
  //Resultado de parsear la expresión
  TextField outputparseo = new TextField("          ");
  //Resultado de la evaluación en la expresión
  TextField outputevaluar = new TextField("         ");
  //Label donde se dan los errores
  Label info = new Label("Información en extremo importante           ", Label.CENTER);

  public void init(){ //Todo se pone en el applet
    add(inputexpresion);
```

---

[5] Una aplicación hecha para poner en una página de Internet

```
    add(inputvalor);
    add(boton);
    add(outputparseo);
    add(outputevaluar);
    add(info);
  }//init

  public boolean action(Event evt, Object arg){
    if (evt.target instanceof Button){ //Si se apretó el botón
      try{
        info.setText(""); //Se pone el Label de los errores vacío
        expresion=inputexpresion.getText(); //Se lee la expresión
        //Se lee el valor a evaluar
        valor=Double.valueOf(inputvalor.getText()).doubleValue();
        //Se parsea la expresión
        outputparseo.setText(miparser.parsear(expresion));
        //Se evalúa el valor y se redondea
        outputevaluar.setText(""+redondeo(miparser.f(valor),5));
      }catch(Exception e){ //Si hubo error lo pone en el Label correspondiente
        info.setText(e.toString());
      }
    }//if del botón
    return true;
  }//action

  /*
  *Se redondea un número con los decimales dados
  */
  private double redondeo(double numero, int decimales){
    return ((double)Math.round(numero*Math.pow(10,decimales)))/Math.pow(10,decimales);
  }

}//PolCero
```

El applet se puede ver bajar y ver funcionando en la sección correspondiente a los programas, si se copia el texto se debe pegar en un archivo que se llame PruebaParseador.java y se debe tener una página HTML de prueba para verlo, la más sencilla es una página que tenga por código:

```
<HTML>
<HEAD>
</HEAD>
<BODY>
<CENTER>
<APPLET
  code  = "PruebaParseador.class"
  width = "500"
  height    = "300"
  >
</APPLET>
```

```
</CENTER>
</BODY>
</HTML>
```

Este applet tiene dos cajas de texto (*TextField*), una es donde se digita la expresión a evaluar (*inputexpresion*) y la otra es donde se digita el valor a evaluar(*inputvalor*), por defecto estas cajas iniciar con ´´x+5” y “0” respectivamente. El applet tiene un botón (*boton*), al ser presionado, se lee lo que el usuario escribió en las cajas de texto y se escribe el resultado de parsear la expresión con el texto *outputparseo.setText(miparser.parsear(expresion));*, observe que el código que hace la traducción es *miparser.parsear(expresion)* y el resultado se escribe en la etiqueta *outputparseo*; algo similar se hace con el resultado de la evaluación, sólo que antes se redondea el resultado con el código *redondeo(miparser.f(valor),5)*

1.7. Comentarios finales

Para finalizar, se comentará un poco las ideas para verificar si una expresión que digitó el usuario está bien escrita, lo cual no verificamos en el artículo pero sí está en el programa final. Para esto, lo que se hizo fue declarar una variable global *anterior* que se encarga de guardar un número dependiendo si lo último que tradujo en una expresión fue un número, un operador, etc. de la siguiente manera:

Valor	Última subexpresión traducida	Significa
0	nada	nada
1	Número, pi, e, x	Números
2	+, -, *, /, ˆ, %	Operadores
3	(, sen(, cos(, etc.	Paréntesis de apertura
4	)	Paréntesis de cierre

De esta forma se deben cumplir las siguientes reglas:

- Si no se había traducido nada (*anterior=0*) entonces se puede admitir cualquier cosa menos (+ * / ˆ %).
- Si lo anterior fue un número puede seguir cualquier cosa.
- Si lo anterior fue un operador puede seguir cualquier cosa menos otro operador (con excepción de -).
- Si lo anterior fue un paréntesis de apertura puede seguir cualquier cosa menos (+ * / ˆ %)
- Si lo anterior fue un cierre de paréntesis debe seguir un operador, un número (en cuyo caso hay una multiplicación oculta), un paréntesis de apertura (también hay una multiplicación oculta) u otro paréntesis de cierre, estas multiplicaciones ocultas deben ser agregadas antes de poner el número o el paréntesis.
- Para un menos unario se debe agregar un -1 en la pila de números y un “por” en la de operadores (sacando los correspondientes operadores con mayor prioridad); el menos unario se da si no había nada anterior al menos o si era otro operador o un paréntesis de apertura.

El código completo con su correspondiente archivo, el archivo de prueba y la página HTML se encuentran al final de este artículo junto con el código similar en un módulo de Visual Basic que puede ser usado en cualquier programa de este lenguaje.

Otro programa recomendado que es similar al que se desarrolló en el artículo y que puede ser estudiado, bajado con su código completo y utilizado en otros programas libremente es: JEP[5]

Bibliografía

[1] Aho, Alfred; Hopcroft, John; Ullman, Jeffrey.(1983) Data structures and algorithms. Massachusetts : Addison-Wesley.

[2] Deitel H. y Deitel P. (1998) *Cómo programar en Java*. [Traducción del libro *Java how to program*] Primera edición. Prentice-Hall, México.

[3] Murillo, M.; Soto, A. y Alfredo, J. (2000) *Matemática básica con aplicaciones*. EUNED. San José, C.R.

[4] Sitio Web de JAVA: www.java.sun.com

[5] Sitio Web de JEP - Java Expresion Parser: http://www.singularsys.com/jep

Capítulo 2

Multiplicación de Polinomios con la Transformada Rápida de Fourier (FFT).

MSc. Geovanni Figueroa Mata
Escuela de Matemática
Instituto Tecnológico de Costa Rica.
gfigueroa@itcr.ac.cr

abstract

Se analiza un algoritmo para la multiplicación de polinomios el cual usa la transformada rápida de Fourier (FFT) para lograr una complejidad de $\Theta(nlog(n))$. Los algoritmos y cálculos se desarrollan con el software `Mathematica`.

2.1. Introducción

En el algoritmo clásico la multiplicación de polinomios se realiza término a término. Esto es, dados dos polinomios

$$p(x) = \sum_{i=0}^{m} a_i x^i$$

de grado m y

$$q(x) = \sum_{j=0}^{n} b_j x^j$$

de grado n, el producto de estos dos polinomios será un polinomio de grado $m+n$ dado por:

$$p(x) \cdot q(x) = \sum_{i=0}^{m} \sum_{j=0}^{n} a_i b_j x^{i+j}$$

Este algoritmo requiere de $(n+1)(m+1)$ multiplicaciones con lo cual su complejidad es de $\Theta(m \cdot n)$. Así, para la mayoría de las situaciones comunes este algoritmo resulta adecuado. Sin embargo, su desempeño se ve disminuido cuando el grado de los polinomios es muy grandes.

En lo que sigue nuestro objetivo será estudiar una técnica más eficiente para la multiplicación de polinomios, para lograr esto, dicha técnica usa la **transformada rápida de Fourier**.

2.2. Una primera aproximación

La estrategia general del método mejorado para la multiplicación de polinomios que vamos a analizar se resume en: `evaluar`, `multiplicar` e `interpolar`.

Esta estrategia se apoya en el siguiente teorema de interpolación, el cual afirma que un polinomio de grado n está completamente determinado por su valor en $n+1$ puntos diferentes.

Teorema 1 *Si* $x_0, x_1, \ldots, x_n$ *son* $n+1$ *puntos y* $f(x)$ *es una función que pasa por estos puntos, entonces existe un polinomio único* $p(x)$ *de grado a lo más* n *con la propiedad de que* $p(x_k) = f(x_k)$ *para* $k = 0, 1, \ldots, n$. *Este polinomio se conoce como el polinomio de interpolación de Lagrange y está dado por:*

$$p(x) = \sum_{k=0}^{n} f(x_k) l_{n,k}(x)$$

donde

$$l_{n,k}(x) = \prod_{\substack{i=0 \\ i \neq k}}^{n} \frac{x - x_i}{x_k - x_i}$$

En este sentido, al multiplicar dos polinomios de grado m y n respectivamente obtenemos un polinomio de grado $m \cdot n$; observe que si podemos determinar los valores de este polinomio en $m \cdot n + 1$ puntos, entonces éste queda completamente determinado. Pero esto es posible, pues podemos evaluar cada uno de los dos polinomios en estos $m \cdot n + 1$ y al multiplicar estos valores obtenemos los $m \cdot n + 1$ valores necesarios para determinar el producto de los polinomios.

Por ejemplo, dados los polinomios: $P_1(x) = 1 + x$ y $P_2(x) = 1 + x + x^2$ vamos a calcular el producto $P_1(x) \cdot P_2(x)$, con esta estrategia. Para desarrollar los cálculos e implementar los algoritmos necesarios vamos a usar el software `Mathematica` e identificaremos el código por medio del símbolo 🗎.

Primero debemos definir los polinomios.

🗎

```
P1[x_]=1+x;
P2[x_]=1+x+x^2;
```

Ahora podemos ejecutar cada uno de los pasos de la estrategia de multiplicación.

2.2.1. Primer paso: evaluar

El proceso inicia evaluando cada polinomio en un conjunto de puntos diferentes, el cual seleccionamos arbitrariamente. En este caso necesitamos cuatro puntos, pues el polinomio resultante (producto) será de grado tres.

```
n=4;   (* n = grado + 1 *)
l={};
i=0;
While[i< n,
  num=Random[Integer,{-5,5}];
  If [!MemberQ[l,num],
  AppendTo[l,num];i+=1];
];
l
```

Al ejecutar las líneas anteriores obtenemos el conjunto de puntos deseado.

```
{5,-5,2,-2}
```

El siguiente fragmento de código evalua cada uno de los polinomios en el conjunto de puntos elegido.

```
lP1={};
lP2={};
For [i=1,i<=  n,i++,
AppendTo[lP1,P1[l[[i]]]];
AppendTo[lP2,P2[l[[i]]]];
]
Print["Evaluando en P1[x] obtenemos: ", lP1];
Print["Evaluando en P2[x] obtenemos: ", lP2];
```

Al ejecutar el código anterior obtenemos la evaluación de los polinomios en el conjunto de puntos elegido.

```
Evaluando en P1[x] obtenemos: {6,-4,3,-1}
Evaluando en P2[x] obtenemos: {31,21,7,3}
```

2.2.2. Segundo paso: multiplicar

Ahora procedemos a calcular los valores del polinomio resultante evaluado en la lista de puntos elegida, es decir, calculamos $p_1(x_i) \cdot p_2(x_i)$.

```
Print ["Lista de valores de cada polinomio"];
Print [lP1];
Print [lP2];
Print ["Lista de valores del producto"];
lm=lP1*lP2
```

Con esto, obtenemos la lista de valores para el producto.

```
Lista de valores de cada polinomio
{6,-4,3,-1}
{31,21,7,3}
Lista de valores del producto
{186,-84,21,-3}
```

2.2.3. Tercer paso: Interpolar

Hasta aquí hemos construido un conjunto de valores para el polinomio producto, lo que resta es interpolar para obtener dicho polinomio.

```
puntos={};
For [d=1,d<=4, d++,
AppendTo[puntos,{l[[d]],lm[[d]]}]
]
puntos
```

Así, el conjunto de puntos para interpolar el polinomio producto es:

```
{{5,186},{-5,-84},{2,21},{-2,-3}}
```

Ahora debemos calcular los factores

$$l_k(x) = \prod_{i=0\ i\neq k}^{n} \frac{x - x_i}{x_k - x_i}$$

del polinomio de interpolación de Lagrange.

```
L[k_,n_]:=Module[{},
expresion=1;
For [i=1,i<=n+1,i++,
If [i!= k,
expresion=(x-l[[i]])/(l[[k]]-l[[i]])*expresion;
];(*end if*)
];(*end for*)
expresion
](*end*)
```

El siguiente paso es calcular el polinomio de interpolación de Lagrange.

```
p[n_]:=Sum[ lm[[k]]*L[k,n],{k,1,n+1}];
Lpolinomio=p[n-1]
```

De donde obtenemos el polinomio que resulta de multiplicar los polinomios dados.

```
-(2/5) (-2-x)(2-x)(5-x)-
1/28   (2-x)(5-x)(5+x) +
1/4    (5-x)(2+x)(5+x) +
31/35  (-2+x)(2+x)(5+x)
```

Simplificándolo obtenemos:

```
Expand[Lpolinomio]

1+2 x+2 x^2+x^3
```

Fácilmente podemos comprobar que el resultado es el correcto.

2.3. Mejorando el algoritmo

Como ha sido desarrollada la estrategia anterior no resulta ser un algoritmo atractivo para la multiplicación de polinomios, pues las mejores técnicas para la evaluación de polinomios (método de Horner) e interpolación (fórmula de Lagrange) necesitan de n^2 multiplicaciones. Sin embargo, es razonable esperar que la evaluación e interpolación sean más fáciles de realizar para cierto conjunto de puntos que para otros.

Resulta que el conjunto de puntos más adecuado para realizar la evaluación e interpolación de polinomios es el conjunto de las *raíces complejas de la unidad*, con esto y una manera más eficiente de evaluar el polinomio lograremos mejorar significativamente el algoritmo.

2.3.1. Raíces de la unidad

Las raíces n-ésimas de la unidad se obtienen al resolver la ecuación $z^n = 1$ y están dadas por:

$$z_n^k = \text{Cos}\left(\frac{2\pi k}{n}\right) + i\ \text{Sen}\left(\frac{2\pi k}{n}\right), \qquad \text{con } k = 0, 1, \cdots, n-1$$

El siguiente fragmento de código calcule las raíces n-ésimas de la unidad.

```
raiz[n_]:=Module[{l={}},
For [k=0,k<=n-1,k++,
zk=Cos[(2*k*Pi)/n]+i*Sin[(2*k*Pi)/n];
AppendTo[l,zk];
];   (*end for*)
l
]    (*end module*)
```

Por ejemplo, las raíces cuartas de la unidad están dadas por:

```
n=4;
raicescuartas=raiz[4]

{1,i,-1,-i}
```

Observe que en este caso $z_4^1 = i$ es la raíz cuarta principal de la unidad, pues $\left(z_4^0\right)^0 = 1$, $\left(z_4^1\right)^1 = z_4^1 = i$ (la raíz principal), $\left(z_4^2\right)^2 = -1$ y $\left(z_4^3\right)^3 = -i$.

Además se tienen los siguientes resultados, los cuales son válidos en general:

- $z_n^0 = 1$, z_n^1 es la raíz principal de la unidad.
- Las raíces de la unidad son: $\left(z_n^1\right)^0$, $\left(z_n^1\right)^1$, $\left(z_n^1\right)^2$,..., $\left(z_n^1\right)^{n-1}$.
- Si n es par $z_n^{n/2} = -1$, pues $z_n^{n/2} = \text{Cos}(\pi) + i\text{Sen}(\pi) = -1$.
- Como

$$z_4^{4/2} = z_4^2 = -1 = -z_4^0$$

$$z_4^3 = \left(z_4^1\right)^3 = \left(z_4^1\right)^2 z_4^1 = z_4^2 z_4^1 = -z_4^1$$

  podemos reescribir las raíces cuartas de la unidad como $\left\{z_4^0, z_4^1, -z_4^0, -z_4^1\right\}$.

Esto quiere decir que las raíces octavas de la unidad podrían escribirse como:

$$\left\{z_8^0, z_8^1, z_8^2, z_8^3, -z_8^0, -z_8^1, -z_8^2, -z_8^3\right\}$$

Note que al tomar el cuadrado de cada una de las raíces cuartas de la unidad obtenemos:

$$\left\{\left(z_4^0\right)^2, \left(z_4^1\right)^2, \left(z_4^0\right)^2, \left(z_4^1\right)^2\right\} = \left\{z_2^0, z_2^1, z_2^0, z_2^1\right\}$$

es decir, dos copias de las raíces dobles de la unidad. Si hacemos esto con las raíces octavas de la unidad obtenemos dos copias de las raíces cuartas de la unidad. Esto es cierto en general y nos permitirá realizar la evaluación de un polinomio en las raíces complejas de la unidad de una forma más eficiente.

2.3.2. Evaluando un polinomio

Observe que un polinomio como el siguiente

$$p_3(x) = a_0 + a_1x + a_2x^2 + a_3x^3$$

se puede reescribir como:

$$p_3(x) = a_0 + a_1x + a_2x^2 + a_3x^3 = a_0 + a_2x^2 + x\left(a_1 + a_3x^2\right) = p_e\left(x^2\right) + p_0\left(x^2\right)$$

donde

$$p_e(x) = a_0 + a_2x \qquad p_0(x) = a_1 + a_3x$$

De esta forma si queremos evaluar $p_3(x)$ (con 4 coeficientes) en las raíces cuartas de la unidad, necesitamos evaluar dos polinomios (con 2 coeficientes) en las raíces dobles de la unidad, lo que se traduce en un ahorro significativo.

En resumen, la evaluación de $p_3(x)$ en las raíces cuartas de la unidad se realiza de la siguiente forma:

$$\begin{array}{rcl}
p_3\left(z_4^0\right) & = & p_e\left(z_2^0\right) + z_4^0 p_0\left(z_2^0\right) \\
p_3\left(z_4^1\right) & = & p_e\left(z_2^1\right) + z_4^1 p_0\left(z_2^1\right) \\
p_3\left(z_4^2\right) & = & p_e\left(z_2^0\right) - z_4^0 p_0\left(z_2^0\right) \\
p_3\left(z_4^3\right) & = & p_e\left(z_2^1\right) - z_4^1 p_0\left(z_2^1\right)
\end{array}$$

En general, para evaluar un polinomio $p(x)$ en las n-ésimas raíces de la unidad, se evalúan recursivamente $p_e(x)$ y $p_0(x)$ en las $\frac{n}{2}$ $-$ ésimas raíces de la unidad. Esto se detiene cuando $n = 2$ y se evalúa $a_0 + a_1x$ en 1 y -1, con los resultados $a_0 + a_1$ y $a_0 - a_1$ (esto para n par).

Continuando con el ejemplo, vamos a evaluar los polinimios $p_1(x) = 1 + x$ y $p_2(x) = 1 + x + x^2$ en las raíces cuartas de la unidad, pero usando lo anterior.

Primero definamos de nuevo los polinomios.

```
P4[x_]:=1+x;
P5[x_]:=1+x+x^2;
```

Ahora calculemos las raíces cuartas de la unidad.

```
w4:=raicescuartas;
Print["Raices cuartas de la unidad w4 = ",w4];

Raices cuartas de la unidad w4 = {1,i,-1,-i}
```

Calculamos las raíces dobles de la unidad.

```
w2=Take[w22,Length[w22]/2];
Print["Raices dobles de la unidad w2 = ",w2];
Print["Cuadrado de las raices cuartas (w4)^2 =",(w4)^2];

Raices dobles de la unidad w2 = {1,-1}
Cuadrado de las raices cuartas (w4)^2 = {1,-1,1,-1}
```

Los polinomios $p_e(x)$ y $p_0(x)$ para el polinomio $p_4(x)$ son:

```
Pe4[x_]:=1;
P04[x_]:=1;
```

Evaluamos $p_4(x)$ en las raíces cuartas de la unidad.

```
r41=Pe4[w2[[1]]]+w4[[1]]*P04[w2[[1]]];
r42=Pe4[w2[[2]]]+w4[[2]]*P04[w2[[2]]];
r43=Pe4[w2[[1]]]-w4[[1]]*P04[w2[[1]]];
r44=Pe4[w2[[2]]]-w4[[2]]*P04[w2[[2]]];
rP4={r41,r42,r43,r44};
Print["Al evaluar P_4 en las raices
cuartas de la unidad obetenemos: ", rP4]

Al evaluar P_4 en las raices cuartas
de la unidad obetenemos: {2,1+i,0,1-i}
```

Los polinomios $p_e(x)$ y $p_0(x)$ para el polinomio $p_5(x)$ son:

```
Pe5[x_]:=1+x;
P05[x_]:=1;
```

Análogamente procedemos a evaluar $p_5(x)$ en las raíces cuartas de la unidad.

```
r51=Pe5[w2[[1]]]+w4[[1]]*P05[w2[[1]]];
r52=Pe5[w2[[2]]]+w4[[2]]*P05[w2[[2]]];
r53=Pe5[w2[[1]]]-w4[[1]]*P05[w2[[1]]];
r54=Pe5[w2[[2]]]-w4[[2]]*P05[w2[[2]]];
rP5={r51,r52,r53,r54};
Print["Al evaluar P_5 en las raices
cuartas de la unidad obetenemos: ", rP5]
```

```
Al evaluar P_5 en las raíces cuartas
de la unidad obetenemos: {3,i,1,-i}
```

2.3.3. Interpolando el producto

Ahora que ya tenemos una forma rápida de evaluar polinomios es un conjunto espeficífico de puntos, todo lo que necesitamos es una forma también rápida de interpolar los polinomios en esos mismos puntos, y como consecuencia obtendremos un método eficiente de multiplicación de polinomios. Sorprendentemente, para las raíces n-ésima de la unidad, al ejecutar la evaluación en un conjunto adecuado de puntos se tiene la interpolación, es decir, podemos usar exactamente el mismo algoritmo de evaluación para interpolación.

Esto se resumen en el siguiente teorema de la inversión para la transformada discreta de Fourier el cual nos dice como calcular los coeficientes del polinomio resultante.

Teorema 2 (de la inversión) *Si $r(x) = p(x)q(x)$ y $s(x)$ es el polinomio cuyos coeficientes s_i están dados por $p\left(z_n^i\right) q\left(z_n^i\right) = r\left(z_n^i\right)$, entonces para hallar los coeficientes del polinomio de interpolación $r(x)$ tenemos la siguiente relación*

$$s\left(z_n^{-t}\right) = nr_t \Longrightarrow r_t = \frac{s\left(z_n^{-t}\right)}{n}$$

Observe que el conjunto de puntos mágico para la interpolación esta formado por las inversas de las raíces n-ésimas de la unidad.

Iniciemos calculando el conjunto de puntos para interpolar el producto de los polinomios, es decir, $p\left(z_n^i\right) q\left(z_n^i\right) = r\left(z_n^i\right)$.

```
r={};
For [i=1,i<=Length[rP4],i++,
AppendTo[r,rP4[[i]]*rP5[[i]]]
]
Print["r = ",r]
```

```
r = {6,-1+i,0,-1-i}
```

Ahora con estos puntos debemos construimos el polinomio $s(x)$:

```
S[x_]=r[[1]] +r[[2]] x +r[[3]] x^2+r[[4]] x^3;
```

Calculemos el inverso de cada una de las raíces cuartas de la unidad.

```
Print["Raices cuartas de la unidad w_4 = ",w4]
wInv=1/w4;
Print["Inversas de las raices de la unidad
w_4^{-1}  = ",wInv]

Raices cuartas de la unidad w_4 = {1,i,-1,-i}

Inversas de las raices de la unidad w_4^{-1}
= {1,-i,-1,i}
```

Lo que nos queda por hacer para obtener el polinomio $r(x)$ es evaluar las raíces inversas de la unidad en el polinomio $s(x)$.

```
coeficiente0=S[wInv[[1]]];
coeficiente1=S[wInv[[2]]];
coeficiente2=S[wInv[[3]]];
coeficiente3=S[wInv[[4]]];
PQ[x_]=coeficiente0+coeficiente1*x+
coeficiente2*x^2+coeficiente3*x^3;
Print["Asi, tenemos que P_4(x)*P_5(x) = "
,Expand[PQ[x]/4]]

Asi, tenemos que P_4(x)*P_5(x) = 1+2 x+2x^2+x^3
```

Observe que los coeficientes del polinomio resultante quedan multiplicados por n, por esta razón en el caso anterior fue necesario dividir por 4.

2.4. Implementando el algoritmo

Ahora ya tenemos todas las piezas del algoritmo para la multiplicación de polinomios El esquema general es:

- **Evaluar:** los polinomios a multiplicar en las raíces n-ésimas de la unidad.
- **Multiplicar:** los valores obtenidos anteriormente para obtener los puntos de interpolación del producto.
- **Interpolar:** para encontrar el resultado evaluando el polinomio definido por los valores obtenidos en las raíces n-ésimas de la unidad.

El siguiente fragmento de código calcula la transformada rápida de Fourier (FFT), que en el fondo es el algoritmo que se describío para la evaluación de un polinomios en las raíces n-ésimas de la unidad. Recibe como parámetros de entrada el grado del polinomio, una lista que corresponde a sus coeficientes y la raíz

n-ésima principal de la unidad, entonces evalúa el polinomio en todas las raíces n-ésimas de la unidad y devuelve una lista con estos valores.

```
FFT[n_,lista_,w_]:=Module[{pares={},impares={},ans={},
 bans={},cans={}},
 ans=Table[0,{i,n}];
If[n==1, ans[[1]]=lista[[1]];
  else, t=1/2*n;
  For [i=1,i<=n/2,i++,
    c=AppendTo[pares,lista[[2*i]]];
    b=AppendTo[impares,lista[[2*i-1]]];
    ];
  b2=b;
  c2=c;
  bans=FFT[t,b,w^2];
  c2={};
  pares={};
  t=1/2*n;
  For [i=1,i<=n/2,i++,
   c2=AppendTo[pares,lista[[2*i]]];];
   cans=FFT[t,c2,w^2
   ];
   alpha=1;
   For[i=1,i<=n/2,i++,
     ans[[i]]=bans[[i]]+alpha*cans[[i]];
     ans[[i+n/2]]=bans[[i]]-alpha*cans[[i]];
     alpha=alpha*w;];
 ];(*end if*)
 ans
]      (*end module*)

raiz[n_]:=Module[{l={}},
For [k=0,k<=n-1,k++,
zi=Cos[(2*k*Pi)/n]+I*Sin[(2*k*Pi)/n];
AppendTo[l,zi];
];     (*end for*)
l
]
raizprincipal[m_]:=raiz[m][[2]]
```

Ejemplo

Considere el polinomio $p(x) = 1 + 2x + 3x^2$ de grado 8 (los restantes coeficientes son cero), con la raíz octava principal $1 + i\sqrt{2}$, entonces al evaluarlo en las raíces octavas de la unidad obtenemos la siguiente lista de valores:

```
FFT[8,{1,2,3,0,0,0,0,0},(1+i/Sqrt[2]]

{6,(1+3i)+(1+i)Sqrt[2],-2+2i,(1-3i)-(1-i)Sqrt[2],
2,(1+3i)-(1+i)Sqrt[2],-2-2i,(1-3i)+(1-i)Sqrt[2]}
```

El siguiente fragmento de código permite escribir una lista de coeficientes, que representan un polinomio, a su forma usual $a_i x^i$.

```
EscribirPolinomio[lista_]:=Module[{},
pol=lista[[Length[lista]]];
k=lista;
For[i=2,i<=Length[k],i++,
pol=x*pol+k[[Length[k]-i+1]];
];(*end for*)
pol
]
```

Ejemplo

La lista $\{1,2,3,0,0,0,0,0\}$ represenata al polinomio $p(x) = 1 + 2x + 3x^2$, como podemos comprobar:

```
Expand[EscribirPolinomio[{1,2,3,0,0,0,0,0}]]

1+2 x+3 x^2
```

El siguiente algoritmo realiza la multiplicación de dos polinomios $p(x)$, $q(x)$ siempre y cuando el grado menos 1 del polinomio resultante sea una potencia de 2. Esta restricción se realizó para facilitar la implementación del algoritmo.

El algoritmo recibe dos listas con los coeficientes de los polinomios y calcula su producto.

```
multiplica[listap_, listaq_]:=Module[{p={},
q={},r={}},
 gp=Length[listap]-1;
 gq=Length[listaq]-1;
 Print["p(x) = ",
  Expand[EscribirPolinomio[listap]]];
 Print["q(x) = ",
  Expand[EscribirPolinomio[listaq]]];
 n=gp+gq+1;
 Print["Grado de p(x)q(x) = ",n-1];
 If[IntegerQ[Log[2,n]],
   p=Table[0,{i,n}];
   For[i=1,i<=Length[listap],i++,
     p[[i]]=listap[[i]] ];(*end for*)
     q=Table[0,{i,n}];
     For[i=1,i<=Length[listaq],i++,
     q[[i]]=listaq[[i]]
   ];
  (*evaluando con FFT los polinomios*)
 w=raizprincipal[n];
 pe=FFT[n,p,w];
 qe=FFT[n,q,w];
 (*calculando e interpolando pq*)
 r=pe*qe;
 re=FFT[n,r,w];
 For [i=2,i<=Length[re]-1,i++,
  t=re[[i]];
  re[[i]]=re[[i+1]];
  re[[i+1]]=t;
 ];   (*end for*)
 resultado=Round[re/n];
 Print[resultado];
 Print["p(x)q(x) = ",
   Expand[EscribirPolinomio[resultado]]];
  else,
   Print["Error con los grados de los polinomio"];
 ];   (*end if*)
 ]
```

Ejemplo 1:

Multipliquemos los polinomios $p(x) = 1 + x$ y $q(x) = 1 + x + x^2$. Note que el grado del polinomio resultante es $1 + 2 = 3 = 4 - 1 = 2^2 - 1$.

```
multiplica[{1,1},{1,1,1}]

p(x) = 1+x
q(x) = 1+x+x^2
Grado de p(x)q(x) = 3
{1,2,2,1}
p(x)q(x) = 1+2 x+2 x^2+x^3
```

Ejemplo 2:

Multipliquemos los polinomios

$$p(x) = 1 + 2x + 2x^2 + x^3 + x^4 + x^5$$

y

$$q(x) = 1 + x + x^2 + x^3 + x^4 + x^5 +^{x} 6 + x^7 + x^8 + x^9 + x^{10}$$

Observe que en este caso el grado del polinomio resultante es $5+10=15=16-1=2^4-1$.

```
multiplica[{1,2,2,1,1,1},{1,1,1,1,1,1,1,1,1,1,1}]

p(x) = 1 + 2x + 2x^2 + x^3 + x^4 + x^5
q(x) = 1 + x + x^2 + x^3 + x^4 + x^5 + x^6 +
x^7 + x^8 + x^9 + x^10
Grado de p(x)q(x) = 15
{1,2,3,5,7,8,8,8,8,8,8,7,6,5,3,1}
p(x)q(x) = 1 + 2x + 3x^2 + 5x^3 + 7x^4 +
8x^5 + 8x^6 + 8x^7 + 8x^8 + 8x^9 + 8x^10 +
7x^11 + 6x^12 + 5x^13 + 3x^14 + x^15
```

Ejemplo 3:
Multipliquemos los polinomios

$$p(x)=1+x+x^2+x^3$$

y

$$q(x)=1+x+x^2+x^3+x^4$$

Observe que en este caso el grado del polinomio resultante es $3+4=7=8-1=2^3-1$.

```
multiplica[{1,1,1,1},{1,1,1,1,1}]

p(x) = 1 + x + x^2 + x^3
q(x) = 1 + x + x^2 + x^3 + x^4
Grado de p(x)q(x) = 7
{1,2,3,4,4,3,2,1}
p(x)q(x) = 1 + 2x + 3x^2 + 4x^3 + 4x^4 + 3x^5 + 2x^6 + x^7
```

2.5. Resultados importantes

El algoritmo descrito para la evaluación e interpolación de polinomios en las raíces n-ésimas de la unidad se conoce como la transformada rápida de Fourier (FFT), en el fondo es un algoritmo para calcular eficientemente la transformada discreta de Fourier (DFT). La aparición de este algoritmo significó un gran avance en el análisis matemático y la informática.

Transformada discreta de Fourier

Suponga que $n>1$ es una potencia de 2 y sea ω una constante tal que $\omega^{n/2}=-1$. Considere la n-tupla $a=(a_0,a_1,...,a_{n-1})$ esta define de forma natural el polinomio $p_a(x)=a_0+a_1x+...+a_{n-1}x^{n-1}$, entonces la transformada discreta de Fourier de a con respecto a ω es la n-tupla:

$$\mathrm{DFT}_\omega(a)=\left(p_a(1),p_a(\omega),p_a\left(\omega^2\right),...,p_a\left(\omega^{n-1}\right)\right)$$

La transformada discreta de Fourier proporciona un método para transformar un polinomio de su representación convencional con n coeficientes a su representación en términos de sus valores en las raíces de la unidad. Esta conversión del polinomio se realiza en $\Theta(n^2)$ operaciones aritméticas y el procedimiento de cálculo recursivo descrito (transformada rápida de Fourier) lo permite realizar en $\Theta(n\log(n))$ operaciones aritmética. Este método fue descubierto por Jim Cooley y John Tukey en 1965.

`Mathematica` tiene algortimos numéricos para calcular la transformada discreta de Fourier y su inversa, con estos fácilmente podemos calcular la multiplicación de dos polinomios siguiendo la estrategia descrita.

Ejemplo

Para multiplicar los polinomios $p(x) = 1 + x + x^2$ y $q(x) = 1 - x + 2x^2$, primero debemos hallar su representación en términos de las raíces de la unidad.

```
Fourier[{1,1,1,0,0},FourierParameters->{1,1}];
l1=Chop[%];
Print["Polinomio P(x): ",l1];
Fourier[{1,-1,2,0,0},FourierParameters->{1,1}];
l2=Chop[%];
Print["Polinomio Q(x): ",l2];
```

```
Polinomio P(x): {3., 0.5+1.53884 i,
0.5-0.363271 i, 0.5+0.363271 i, 0.5-1.53884 i}

Polinomio Q(x): {2.,-0.927051+0.224514 i,
2.42705-2.4899 i, 2.42705 + 2.4899 i,
-0.927051-0.224514 i}
```

Ahora calculemos la lista de puntos para interpolar.

```
l3=Chop[l1*l2];
Print["Polinomio R(x): ",l3];
```

```
Polinomio R(x): {6.,-0.809017-1.31433 i,
0.309017-2.12663 i,0.309017+2.12663 i,
-0.809017+1.31433 i}
```

Por último interpolamos, usando la transformada inversa de Fourier.

```
l4=InverseFourier[l3,FourierParameters->{1,1}];
Print["Polinomio R(x): ",l4];
Print["Polinomio R(x): ",Chop[l4]]
```

```
Polinomio R(x): {1.,1.3314*10^-16,2.,1.,2.}
Polinomio R(x): {1.,0,2.,1.,2.}
```

De aquí obtenemos que el polinomio resultante está dado por:

$$r(x) = 1 + 2x^2 + x^3 + 2x^4$$

lo cual podemos corroborar fácilmente.

```
Expand[(1+x+x^2) (1-x+2x^2)]
```

```
1+2 x^2+x^3+2 x^4
```

Bibliografía

[1] Baase, Sara. Computer Algorithms: Introduction to Design and Analysis, Addison-Wesley, New York, 1988.

[2] Brassard, Gilles; Bratley Paul. Algorithmics Theory and Pratice, Prentice-Hall, New Jersey, 1990.

[3] Sedgewick, Robert. Algorithms in $C++$, Addison-Wesley, Massachusetts, 1992.

Capítulo 3

Óptimo Sobre un Conjunto No–Convexo

Luis Ernesto Carrera
Escuela de Matemática
Instituto Tecnológico de Costa Rica

Feliú Davino Sagols
Cinvestav, México.

3.1. Introducción

En el contexto del **XI Taller de Modelación Matemática** realizado en el mes de julio del año 2006 en el Centro de Investigación y Estudios Avanzados del Instituto Politécnico Nacional, en México, D.F., se presentó, entre otros, un problema de optimización combinatoria relacionado con la industria editorial.

El problema fue propuesto por el Dr. David Romero, y una simplificación de dicho problema (PS) es la siguiente:

Dados n, t y d_i $(i = 1, \ldots, n)$, $2 \cdot t \geq n$, se desea producir d_i impresiones en papel de la imagen i, para $i = 1, \ldots, n$, a costo mínimo.

Se tienen 2 planchas de impresión en las que se pueden colocar t negativos. En la j–ésima plancha, $j = 1, 2$, se colocan $s_{i,j}$ negativos de la imagen i para $i = 1, \ldots, n$, y se realizan b_j impresiones de la misma.

El problema consiste en determinar $s_{i,j}$ $(i = 1, \ldots, n$ y $j = 1, 2)$ y b_j $(j = 1, 2)$, tales que minimicen $b_1 + b_2$ sujeto a que $b_1 \cdot s_{i,1} + b_2 \cdot s_{i,2} \geq d_i$, $\forall i = 1, \ldots, n$.

En el proceso de encontrar una solución a dicho problema, se utiliza un algoritmo que encuentra la solución óptima de una función lineal $ax + by$ con $(x, y) \in \mathbb{N}^2$ en cualquier región no–convexa con ciertas propiedades de monotonía. Dicho algoritmo es el que se presenta en este trabajo.

La estrategia utilizada en el algoritmo es una estrategia de enumeración parcial, tomando en cuenta las características del conjunto de soluciones factibles.

3.2. Definición del problema general

Nos dedicamos primero a caracterizar el conjunto de soluciones factibles:

Definición 1 *Se dice que un conjunto de puntos* $E \in \mathbb{N}^2$ *pertenece a la clase de conjuntos* $\mathcal{S}_x^2$ *si dado que el punto* $(x, y) \in E$, *entonces para todo* $x' \geq x$, $x' \in \mathbb{N}$, *existe* $y' \in \mathbb{N}$, $y' \leq y$ *tal que* $(x', y') \in E$.

Basados en la definición anterior, donde un ejemplo de un conjunto que pertenece a la clase de conjuntos $\mathcal{S}_x^2$ se muestra en la Figura 3.1, se presenta el problema general (PG):

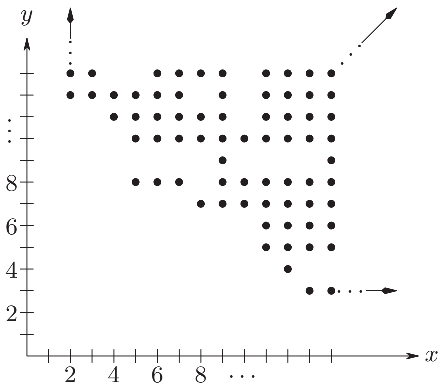


Figura 3.1: Conjunto de puntos que pertenece a la clase de conjuntos $\mathcal{S}_x^2$.

Dados $E \in \mathcal{S}_x^2$ y $a, b \in I\!R^+$, encontrar $(x^*, y^*) \in E$ tal que $ax^* + by^* = \text{mín}\{ax + by \,|\, (x, y) \in E\}$.

Se introduce ahora la siguiente notación, donde el conjunto E es un elemento arbitrario de $\mathcal{S}_x^2$:

1. $x_{\min} \overset{\text{def}}{=} \text{mín}\{x \,|\, \exists y \in I\!N \text{ tal que } (x, y) \in E\}$
2. $\forall \chi \in I\!N,\ \chi \geq x_{\min},\ y_{[\chi]} \overset{\text{def}}{=} \text{mín}\{y \,|\, (\chi, y) \in E\}$
3. $\forall \chi \in I\!N,\ \chi \geq x_{\min},\ x_{[\chi]} \overset{\text{def}}{=} \text{mín}\{x \,|\, (x, y_{[\chi]}) \in E\}$.

Por definición de $\mathcal{S}_x^2$, dado que debe existir algún y para el cual $(x_{\min}, y) \in E$, se puede asegurar que para todo $\chi \geq x_{\min}$ existe y' tal que $(\chi, y') \in E$.

Ahora se presenta el resultado en el que se basa la ejecución del algoritmo:

Proposiciónn 1 *Sean $E \in \mathcal{S}_x^2$ y $\chi \geq x_{\min}$. Se cumple entonces que para todo $x \in I\!N$, $x_{\min} \leq x \leq \chi$, si $(x, y) \in E$ entonces $y \geq y_{[x]}$.*

Demostración 1 *Sea $(x', y') \in E$, $x' \geq x_{\min}$ y $y' = y_{[x']}$. Obsérvese que si existiera $(x, y) \in E$ con $x_{\min} \leq x \leq x'$ y $y < y'$, entonces por la Definición 1, debe existir (x', y'') tal que $y'' \leq y$, lo cual contradeciría la minimalidad de y'. Se sigue el resultado.*

3.3. Algoritmo para el problema general

Se asume que para cualquier problema:

1. Es posible determinar el valor de $x_{\min}$
2. Es posible determinar algún punto $(x^*, y^*) \in E$
3. Es posible construir las siguientes funciones:
    a. $f : \{x_{\min}, x_{\min} + 1, \ldots\} \longrightarrow I\!N$
    $f(\chi) = y_{[\chi]}$
    b. $g : \{x_{\min}, x_{\min} + 1 \ldots\} \longrightarrow I\!N \times I\!N$
    $g(\chi) = (x_{[\chi]}, y_{[\chi]})$.

La idea del algoritmo está basada en limitar la búsqueda del óptimo en un subconjunto de puntos de $\mathbb{N}\times\mathbb{N}$, en el cual se podría encontrar la solución óptima al problema; dicho subconjunto se va a ir reduciendo de manera iterativa hasta quedar con un subconjunto vacío. Lo importante en este proceso es que no es necesario determinar todos los mínimos locales de la región factible.

Se supone que al inicio se tiene algún punto $(x^*, y^*) \in E$ (por ejemplo podría ser el punto $(x_{\min}, y_{[x_{\min}]})$). Estamos interesados en todos los puntos $(x, y) \in E$ tales que $ax + by < ax^* + by^*$, pues uno de ellos debe ser la solución que buscamos. Puede suceder sin embargo que no exista ningún $(x, y) \in E$ tal que $ax + by < ax^* + by^*$, en cuyo caso ya se tiene la solución.

Se sabe además que cualquier punto $(x, y) \in E$ debe cumplir que $x \geq x_{\min}$ e $y \geq 1$. Así se tiene la región R definida por el conjunto de puntos $(x, y) \in \mathbb{N} \times \mathbb{N}$, tales que $ax + by \leq ax^* + by^* - 1$, $x \geq x_{\min}$ e $y \geq 1$, como se muestra en la Figura 3.2.

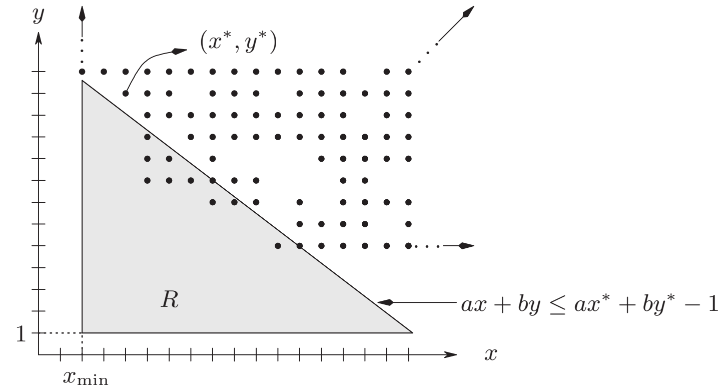


Figura 3.2: Dado un punto $(x^*, y^*) \in E$, región R donde podría encontrarse un punto $(x, y) \in E$ tal que $ax + by < ax^* + by^*$. Es importante recordar que el conjunto E es desconocido.

Se comienza la búsqueda en el extremo derecho de dicha región, determinado por la intersección de las rectas $ax + by = ax^* + by^* - 1$ e $y = 1$, de donde el valor de la abscisa de dicho extremo está dado por:

$$\varphi = \lfloor x^* + [b(y^* - 1) - 1]/a \rfloor \,.$$

Se encuentra el punto $(x_{[\varphi]}, y_{[\varphi]})$, como se muestra en la Figura 3.3. Si sucediera que $ax_{[\varphi]}+by_{[\varphi]} < ax^*+by^*$, entonces se actualiza el punto $(x^*, y^*) \leftarrow (x_{[\varphi]}, y_{[\varphi]})$.

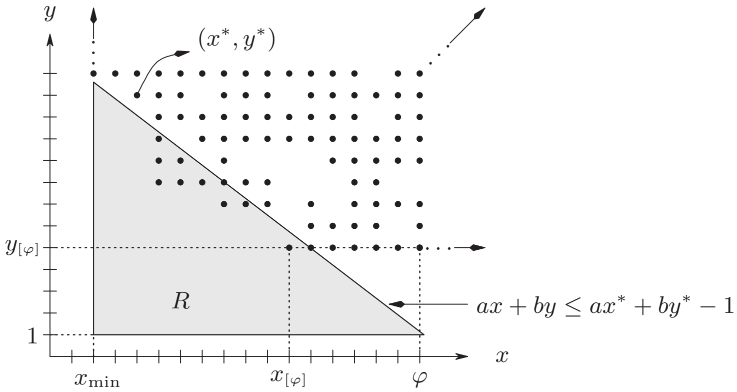


Figura 3.3: Buscando un punto que sea solución del Problema PG en el extremo derecho de la región R para la región factible $E \in \mathcal{S}^2$.

Se tiene entonces que no existe ningún punto $(x, y) \in E$ con $x > \varphi$ tal que $ax + by < ax^* + by^*$, ya que cualquier punto con $x > \varphi$ se encuentra fuera de la región R. Además no existe ningún punto $(x, y) \in E$,

$y < y_{[\varphi]}$, tal que $ax + by < ax^* + by^*$, los que están fuera de la región por no poder mejorar la solución, y dentro de la región R debido a que por la Proposición 1, para todo punto $(x, y) \in E$ tal que $x \leq \varphi$ debe cumplirse que $y \geq y_{[\varphi]}$.

Además, por la posibilidad de actualización al punto (x^*, y^*) debe cumplirse que $x_{[\varphi]} + y_{[\varphi]} \geq x^* + y^*$.

Por lo anterior se puede construir una nueva región R definida por el conjunto de puntos $(x, y) \in I\!N \times I\!N$ tales que $x \geq x_{\min}$, $ax + by < ax^* + by^*$ e $y \geq y_{[\varphi]}$, como se muestra en la Figura 3.4. Luego basta repetir el procedimiento de encontrar un valor φ de la abscisa para el extremo de la nueva región, el valor del punto $(x_{[\varphi]}, y_{[\varphi]})$ e ir actualizando el punto (x^*, y^*) cuando sea del caso, hasta que la región R sea un conjunto vacío.

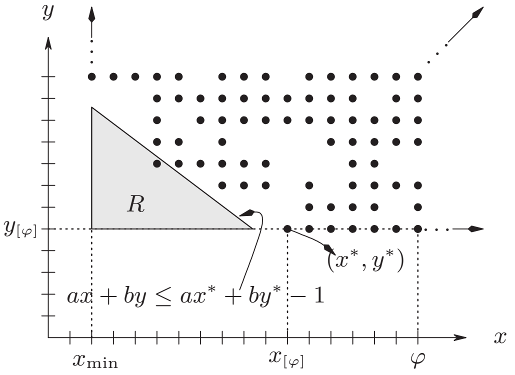


Figura 3.4: Dados los puntos $(x^*, y^*) \in \mathbb{I}_E$ y $(x_{[\varphi]}, y_{[\varphi]}) \in \mathbb{I}_E$, y dado que para todo punto $(x, y) \in \mathbb{I}_E$ tal que $x \geq x_{[\varphi]}$ se cumple que $x + y \geq x^* + y^*$, región R donde se puede encontrar un punto (x', y'), tal que $(x', y') \in \mathbb{I}_E$ y $x' + y' = \text{mín}\{x + y \mid (x, y) \in E\}$.

Se presenta entonces el algoritmo que resuelve el Problema PG. La entrada del algoritmo es el valor de $x_{\min}$ para el conjunto E y un punto $(x', y') \in E$, mientras que la salida del algoritmo es un punto (x^*, y^*) tal que $ax^* + bx^* = \text{mín}\{ax + by \mid (x, y) \in E\}$

Algoritmo 1 $(x_{\min},\ (x', y'))$

```
$(x^*, y^*) \leftarrow (x', y')$
$\sigma^* \leftarrow ax^* + by^*$
$\varphi \leftarrow \lfloor (\sigma^* - 1 - b)/a \rfloor$
mientras $\varphi \geq x_{\min}$
    $(x_{[\varphi]}, y_{[\varphi]}) \leftarrow g(\varphi)$
    $\sigma \leftarrow ax_{[\varphi]} + by_{[\varphi]}$
    si $\sigma < \sigma^*$
        $(x^*, y^*) \leftarrow (x_{[\varphi]}, y_{[\varphi]})$
        $\sigma^* \leftarrow \sigma$
    $\varphi \leftarrow \lfloor (\sigma^* - 1 - by_{[\varphi]})/a \rfloor$
regresa $(x^*, y^*)$
```

3.4. Variantes al Algoritmo 1

En esta sección se muestran dos variantes al algoritmo OptimoE, las cuales mejoran en una constante el tiempo de ejecución del algoritmo. La primera sigue asegurando la optimalidad de la solución, mientras que la segunda sacrifica, dentro de un intervalo dado, dicha optimalidad, aunque mejora de tiempo sustancial el tiempo de ejecución.

3.4.1. Cota inferior en un punto

Podría suceder que en el Algoritmo 1, el costo de evaluar $g(\varphi)$ en la instrucción 11 sea muy grande, pero que se disponga de una función L que aproxime inferiormente el valor de $y_{[\varphi]}$, cuyo costo de evaluación sea menor.

Recordemos que se había construido la región de búsqueda definida por el conjunto de puntos $(x, y) \in I\!N \times I\!N$ tales que $x \geq x_{\min}$, $y \geq y'$ y $ax + by < ax^* + by^*$. Dado el valor de la abscisa del extremo derecho de la región φ, determinado por la intersección de las rectas $ax + by = ax^* + by^* - 1$ y $y = y'$, para todo valor de $y \leq y_{[x]}$ se pueden presentar dos casos:

1. $y > y'$, con lo cual se puede redefinir la región de búsqueda R, asignando $y' \leftarrow y$, caso que se muestra en la Figura 3.5

2. $y \leq y'$, con lo cual no es posible redefinir la región de búsqueda.

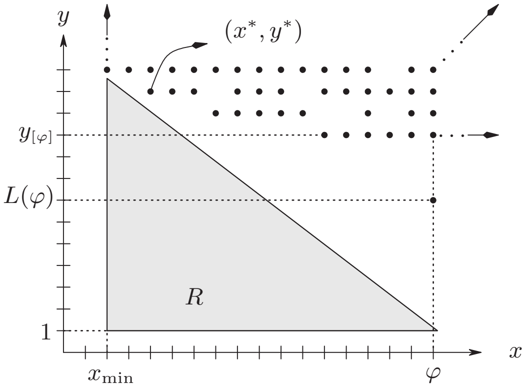


Figura 3.5: Se muestra uno de los 2 casos posibles para la cota inferior de $y_{[\varphi]}$ mediante $L(\varphi)$, en el contexto de buscar la solución en una región R.

Definición 2 *Sea L una función tal que:*

$$L : \{x_{\min}, x_{\min} + 1, \dots\} \longrightarrow I\!N$$
$$L(\lambda, \varphi) = \rho$$

Se dice entonces que L pertenece a la clase de funciones $\mathcal{L}$, si para todo $\varphi \geq x_{\min}$ se cumple que $\rho \leq y_{[\varphi]}$.

La idea de utilizar aproximaciones para reducir la región triangular de búsqueda se tomó de [5], en donde se utiliza en un algoritmo que encuentra el óptimo para el problema de la multi-mochila (multi-knapsack problem).

Se define una clase de funciones $\mathcal{L}$, ya que pueden existir varias funciones que aproximen el valor deseado. Mientras mejor sea la aproximación, mejor el desempeño del algoritmo al utilizar la función.

3.5. Acotando la solución del algoritmo con un ϵ

La segunda variante, es que tal vez se desea sacrificar la optimalidad de la solución a cambio de un mejor rendimiento del Algoritmo 1. Así, se introduce la variable ϵ, que es una cota superior de la distancia entre el valor de la función a optimizar evaluada en el punto óptimo y el valor de la función a optimizar evaluada en el punto regresado por el algoritmo, es decir, si el algoritmo regresa el punto (x', y'), entonces:

$$ax' + by' \leq ax^* + by^* + \epsilon$$

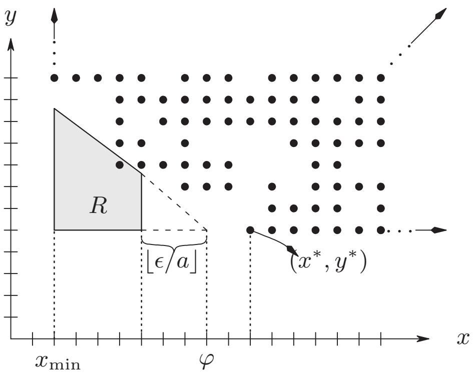


Figura 3.6: Se muestra la región donde se busca el óptimo cuando se mueve el extremo de la región en la que se busca en un ϵ. La diferencia entre $ax' + by'$ y el mínimo valor posible para la suma de las coordenadas de cualquier punto en la región eliminada es de ϵ.

donde (x^*, y^*) es una solución óptima.

Lo anterior se logra haciendo aumentando el salto calculado en la instrucción 10 del Algoritmo 1 en $\lceil\epsilon/a\rceil$ unidades, como se muestra en la Figura 3.6.

3.5.1. Implementación de las variantes

A continuacón se implementan las variantes mostradas en el Algoritmo 1, donde se incluye el valor de ϵ:

Algoritmo 2 $(x_{\min},\ (x', y'),\ \epsilon)$

```
$(x^*, y^*) \leftarrow (x', y')$
$\sigma^* \leftarrow ax^* + by^*$
$\varphi \leftarrow \lfloor(\sigma^* - 1 - b)/a\rfloor - \lfloor\epsilon/a\rfloor$
mientras $\varphi \geq x_{\min}$
    $y' \leftarrow L(\varphi)$
    si $(a\varphi + by' > \sigma^*)$
        $\varphi \leftarrow \lfloor(\sigma^* - 1 - by')/a\rfloor - \lfloor\epsilon/a\rfloor$
        continuar
    $(x_{[\varphi]}, y_{[\varphi]}) \leftarrow g(\varphi)$
    $\sigma \leftarrow ax_{[\varphi]} + by_{[\varphi]}$
    si $\sigma < \sigma^*$
        $(x^*, y^*) \leftarrow (x_{[\varphi]}, y_{[\varphi]})$
        $\sigma^* \leftarrow \sigma$
    $\varphi \leftarrow \lfloor(\sigma^* - 1 - by_{[\varphi]})/a\rfloor - \lfloor\epsilon/a\rfloor$
regresa $(x^*, y^*)$
```

3.6. Conclusiones

Para resolver el problema PS fue necesario generar todas las permutaciones del vector $\mathbf{s_1} = [s_{1,1}, s_{2,1}, \ldots, s_{n,1}]$, tales que $s_{i,1} = 0, \ldots, t$ y $s_{1,1} + \cdots + s_{n,1} = t$. En cada permutación, la región factible pertenece a la clase de conjuntos S_x^2, por lo que se aplicaba el Algoritmo 2.

Además fue posible extender el problema PG de $j = 2$ a $j = m$ dimensiones en aquellos casos donde es posible reducir conjuntos en $I\!N^m$ a conjuntos en S_x^2. En el problema PS eso fue posible, construyendo para todo conjunto en $I\!N^m$, en el cual se debía optimizar la función lineal $a_1x_1 + \cdots + a_nx_n$, un conjunto $Z^2 = \{(x_1, a_2x_2 + \cdots + a_nx_n)\}$.

Aunque sigue siendo un algoritmo de tiempo exponencial, haciendo una enumeración exhaustiva de la región factible permitía resolver en tiempo razonable solamente problemas para $j = 2$. Con nuestra implementación fue posible resolver en tiempo razonable problemas para $j = 4$ y $j = 5$.

En caso de que no fuera posible encontrar la función $g(x)$, el algoritmo se puede implementar sin ninguna diferencia. En la resolución del problema PS se utilizó una función que encontraba un valor x en $x_{[\varphi]} \leq x \leq \varphi$.

Bibliografía

[1] Carrera–Retana L. E.; *Algoritmo para encontrar el óptimo a una simplificación de un problema combinatorio presente en la industria editorial.* Tesis de maestría, Cinvestav, México D. F., 2007.

[2] Lawler E. L., Bell M. D.; *A method for solving discrete optimization problems*, Operations Research **14**, No. 6 (1966).

[3] Papadimitriou C. H., Steiglitz K.; *Combinatorial Optimization: Algorithms and Complexity.* Prentice–Hall, New Jersey, 1982.

[4] Preparata F. P, Shamos M. I.; *Computational geometry.* Springer–Verlag, New York, USA, 1985.

[5] Sun X. L., Wang F. L., Li D.; *Exact Algorithm for Concave Knapsack Problems: Linear underestimation and Partition Method*, Journal of Global Optimization **33**, No. 1 (2005), 15–30.

Capítulo 4

Factorización de Enteros.

Cindy Calderón
Centro de Recursos Virtuales - CRV
Escuela de Matemática
Instituto Tecnológico de Costa Rica.

Filánder Sequeira
Centro de Recursos Virtuales - CRV
Escuela de Matemática
Instituto Tecnológico de Costa Rica.

Walter Mora F.
Centro de Recursos Virtuales - CRV
Escuela de Matemática
Instituto Tecnológico de Costa Rica.

4.1. Residuos Cuadráticos

4.1.1. Preliminares

En esta sección introducimos la teoría de residuos cuadráticos. Esta teoría tiene aplicaciones en factorización de enteros y reconocimiento de primos (entre otras cosas).

Definición 1

Si $mcd(a,n)=1$ y si la congruencia

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

tiene solución, entonces a se llama *residuo cuadrático* de n.

▷ Ejemplo 1

1. Los residuos $0 \bmod p$ $3 \bmod 9$ son cuadrados pero no son residuo cuadráticos pues esto se necesita que $mcd(a,n)=1$

- Como $(\pm 1)^2 \equiv 1 \pmod{7}$, $(\pm 2)^2 \equiv 4 \pmod{7}$, $(\pm 3)^2 \equiv 2 \pmod{7}$, entonces $a = 1,2,4$ son residuos cuadráticos módulo 7.
- $a = 3,5,6$ no son residuos cuadráticos módulo 7.

- $a = 4$ no es residuo cuadrático módulo 12 pues $mcd(4, 12) > 1$.

Con el Criterio de Euler podremos responder a la pregunta ¿Cuáles a son residuos cuadráticos módulo p? y con la Ley de Reciprocidad Cuadrática podremos responder una pregunta más complicada ¿Para cuáles primos p es a un residuo cuadrático?.

Por ahora recordemos que el símbolo de congruencia "$\equiv$" funciona como un símbolo de igualdad (módulo n) con la ley de cancelación

$$ac \equiv bc \pmod{n} \implies a \equiv b \pmod{n} \quad \text{si} \quad \text{mcd}(c, n) = 1.$$

por eso decimos que a es un residuo cuadrático (o un cuadrado) módulo n.

El primer teorema importante establece la relación entre los residuos cuadráticos de n y sus factores primos

Teorema 1

1. Sea p un primo impar y $mcd(a, p) = 1$. Entonces a es residuo cuadrático de p^α, $(\alpha \in \mathbb{N})$ si y sólo si a es residuo cuadrático de p.
2. Sea $n = p_1^{\alpha_1} p_2^{\alpha_2} \cdots p_k^{\alpha_k}$. Entonces a es residuo cuadrático de n si y sólo si a es residuo cuadrático de cada uno de los factores p_1, p_2, ..., p_k
3. Si a es impar
   i. a es residuo cuadrático módulo 2.
   ii. a es residuo cuadrático módulo 4 si y sólo si $a \equiv 1 \pmod{4}$.
   iii. a es residuo cuadrático módulo 2^α $(\alpha \geq 3)$, si y sólo si $a \equiv 1 \pmod{8}$.

Prueba.

4.1.2. Símbolo de Legendre, Criterio de Euler y la Ley de Reciprocidad Cuadrática.

Para introducir las reglas de cálculo para decidir si a es residuo cuadrático módulo p necesitamos el símbolo de Legendre

Definición 2

Si p es primo impar, el símbolo de Legendre $\left(\dfrac{a}{p}\right)$ se define como

$$\left(\frac{a}{p}\right) = \begin{cases} 1 & \text{si} \quad a \text{ es residuo cuadrático módulo } p \\ -1 & \text{si} \quad a \text{ no es residuo cuadrático módulo } p \end{cases}$$

Podemos definir formalmente el símbolo de Legendre, usando el grupo

$$(\mathbb{Z}/p\mathbb{Z})^* = \{a \in \mathbb{Z}/p\mathbb{Z} \mid mcd(a,p) = 1\}$$

de unidades de $\mathbb{Z}/p\mathbb{Z}$, como el epimorfismo $\psi : (\mathbb{Z}/p\mathbb{Z})^* \longrightarrow \{-1, 1\}$ tal que $\psi(a) = \left(\dfrac{a}{p}\right)$.

Teorema 2

Sean a, b enteros y p primo impar y $mcd(p, ab) = 1$. Entonces

1. $\left(\dfrac{a^2}{p}\right) = 1$
2. Si $a \equiv b \pmod{p}$ entonces $\left(\dfrac{a}{p}\right) = \left(\dfrac{b}{p}\right)$
3. $\left(\dfrac{ab}{p}\right) = \left(\dfrac{a}{p}\right)\left(\dfrac{b}{p}\right)$
4. **(Criterio de Euler)** Si $mcd(a,p) = 1$ y si p es un primo impar entonces

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

o lo que es lo mismo

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

5. $\left(\dfrac{-1}{p}\right) = (-1)^{(p-1)/2}$
6. $\left(\dfrac{2}{p}\right) = (-1)^{(p^2-1)/8}$
7. **(Ley de Reciprocidad Cuadrática)** Si p y q son primos impares

$$\left(\frac{p}{q}\right) = \left(\frac{q}{p}\right)(-1)^{[(q-1)/2][(p-1)/2]}$$

Prueba.

Observe que de la regla 3 y *bajo las condiciones del teorema,* podemos decir que

- el producto de dos residuos cuadráticos es un residuo cuadrático
- el producto de dos residuos no cuadráticos es un residuo cuadrático
- el producto de un residuo no cuadrático por un residuo cuadrático es un residuo no cuadrático.

Fuera de las condiciones del teorema, si a y b no son residuos cuadráticos módulo n entonces, en general, no podemos decir nada acerca del producto.

▷ Ejemplo 2

1. ¿Es 3 residuo cuadrático del primo 726377359? Veamos

$$\begin{aligned}\left(\frac{3}{726377359}\right) &= (-1)^{[(3-1)/2][(726377359-1)/2]}\left(\frac{726377359}{3}\right) && \text{(Regla 7)}\\ &= (-1)\left(\frac{1}{3}\right) = -1 && \text{(Reglas 2, 3 y 5)}\end{aligned}$$

2. ¿Es 69 residuo cuadrático del primo 389? Veamos

$$\begin{aligned}\left(\frac{69}{389}\right) &= \left(\frac{3}{389}\right)\left(\frac{23}{389}\right) && \text{(Regla 3)}\\ &= \left(\frac{389}{3}\right)\left(\frac{389}{23}\right) && \text{(Regla 7)}\\ &= \left(\frac{2}{3}\right)\left(\frac{-2}{23}\right) && \text{(Regla 2)}\\ &= \left(\frac{2}{3}\right)\left[\left(\frac{-1}{23}\right)\left(\frac{2}{23}\right)\right] && \text{(Regla 3)}\\ &= -1\cdot\left[(-1)^{(23-1)/2}\cdot 1\right] = 1 && \text{(Reglas 5 y 6)}\end{aligned}$$

   por tanto 3 no es residuo cuadrático módulo 726377359.

3. **(Criterio de Euler)** Sea $p = 7$. El Criterio de Euler predice que los residuos cuadráticos módulo 7 satisfacen $a^{(7-1)/2} \equiv 1 \pmod 7$ i.e. $a^3 \equiv 1 \pmod 7$. Esto es muy conveniente porque entonces no se requiere factorizar a, como se hizo en el ejemplo anterior. Hagamos un cálculo de verificación. $(\mathbb{Z}/7\mathbb{Z})^* = \{1,2,3,4,5,6\}$. y entonces los cuadrados módulo 7 son $\{1,2,4\}$. Ahora para cada

$a \in (\mathbb{Z}/7\mathbb{Z})^*$ calculamos $a^{(p-1)/2}$ mód $7 = a^3$ mód 7

$$\begin{aligned} 1^3 &= 1 \\ 2^3 &= 1 \\ 4^3 &= 1 \end{aligned}$$

luego observamos que los a tales que $a^3 = 1$ son $\{1, 2, 4\}$ como se esperaba.

4. **(Criterio de Euler)** ¿Es 3 residuo cuadrático del primo 726377359? Usando el computador podemos comprobar que

$$3^{(p-1)/2} \equiv -1 \pmod{726377359}$$

por tanto 3 no es residuo cuadrático módulo 726377359.

5. $\left(\dfrac{123}{4567}\right) = -1$

Para efectos de factorización de enteros debemos ver algunas reglas de cálculo de manera más detallada.

Teorema 3

Si p es primo impar entonces

1. $\left(\dfrac{-1}{p}\right) = (-1)^{(p-1)/2} = \begin{cases} 1 & \text{si} \quad p = 4k+1 \\ -1 & \text{si} \quad p = 4k-1 \end{cases}$
2. $\left(\dfrac{2}{p}\right) = (-1)^{(p^2-1)/8} = \begin{cases} 1 & \text{si} \quad p = 8k \pm 1 \\ -1 & \text{si} \quad p = 8k \pm 3 \end{cases}$
3. $\left(\dfrac{5}{p}\right) = 1$ si $p = 20k \pm 1$ o $p = 20 \pm 9$
4. Si $p = 2k+1$ es primo impar, para cualquier $a \in (\mathbb{Z}/p\mathbb{Z})^*$ se tiene que [9]

$$a^k = \begin{cases} 1 & \text{si} \quad a \quad \text{es residuo cuadrático de} \quad p \\ -1 & \text{si} \quad a \quad \text{no es residuo cuadrático de} \quad p \end{cases}$$

Relacionado con este tema, no es raro encontrar tablas de progresiones aritméticas que contienen los factores primos para números de cierta forma. Por ejemplo

Números de la forma	Posibles Factores Primos
n^2+1	$2, 4k+1$
n^2-2	$2,\ 8k+1,\ 8k+7$
n^2+2	$2,\ 8k+1,\ 8k+3$
n^2-3	$2,\ 3,\ 12k+1,\ 12k+11$
n^2+3	$2,\ 3,\ 12k+1,\ 12k+7$
n^2-5	$2,\ 5,\ 10k+1,\ 10k+9$
n^2+5	$2,\ 5,\ 20k+1,\ 20k+3,\ 20k+7,\ 20k+9$
n^2-6	$2,\ 3,\ 24k+1,\ 24k+5,\ 24k+19,\ 24k+23$
n^2+6	$2,\ 3,\ 24k+1,\ 24k+5,\ 24k+7,\ 24k+11$

4.2. Números Primos. Factorización de Enteros

Factorizar un número positivo n significa encontrar enteros $a>1$ y $b>1$ tales que $n=ab$. Este producto es llamada una factorización de n. Los enteros positivos que se pueden factorizar se llaman *compuestos* y los enteros $n>1$ que no son compuestos, se llaman *primos* [1]. Un número es primo si y sólo si sus únicos divisores naturales son 1 y él mismo. Los primeros números primos son

$$2, 3, 5, 7, 11, 13, ...$$

Vamos a usar inducción y la Identidad de Bézout para probar el llamado Teorema Fundamental de la Aritmética: todo número natural se puede factorizar como un producto de primos de manera única.

En este contexto, si n es primo convenimos en que n tiene una factorización prima trivial (él mismo). La *factorización prima* del número compuesto n es $n=p_1^{r_1}\cdot p_2^{r_2}\cdots p_k^{r_k}$ donde cada p_i es primo y cada $r_i\in\mathbb{Z}$. Por ejemplo, $12=2^2\cdot 3$.

Primero vamos a considerar la factorización básica

Lema 1 (Estrategia Básica de Factorización)

Si $n\in\mathbb{Z}^+$ admite la factorización $n=ab$, con $a,b\in\mathbb{Z}$ entonces $a\leq\sqrt{n}$ o $b\leq\sqrt{n}$.

Prueba. Procedemos por contradicción, si $a>\sqrt{n}$ y $b>\sqrt{n}$ entonces $ab>\sqrt{n}\sqrt{n}=n$ lo cual, por hipótesis, no es cierto.

---

[1] El nombre "número primo" viene, según Jámblico (Iamblichus), de que estos son los primeros números que aparecen en la sucesión que va quedando al aplicar el algoritmo de Eratóstenes (conocido como "Criba de Eratóstenes")

Del lema anterior se puede deducir que

- Si n es un número compuesto entonces uno de sus factores d cumple $1 \leq d \leq \sqrt{n}$

- Si n no tiene factores d con $1 < d \leq \sqrt{n}$, entonces n es primo.

Esta es la manera más sencilla para factorizar números relativamente pequeños. Para números grandes se usan métodos (un poco) más eficientes entre los que destaca uno basado en curvas elípticas y otro conocido como “Number Field Sieve” (NFS) [10].

▷ Ejemplo 3

- Uno de los factores de 243, si hay, debe ser un número entero entre 1 y $\sqrt{243} \approx 15{,}58$. Para encontrar este factor deberíamos verificar los números $\{2, 3, ..., 15\}$. Como 243 es impar, solo deberíamos verificar los números impares de esta lista. Iniciamos probando con 3. Y este número divide a 243, $243 = 3 * 81$. Observe que $81 > \sqrt{243} \approx 15{,}58$!

- Uno de los factores de 277, si hay, debe ser un número entero entre 1 y $\sqrt{277} \approx 16{,}6$. Para encontrar este factor deberíamos verificar los números $\{2, 3, ..., 16\}$. Como 277 es impar, solo deberíamos verificar los números impares de esta lista. En este caso ningún número de la lista divide a 277 así que este número es primo.

- $14 = 2 \cdot 7$ ($\sqrt{14} \approx 3{,}74166$).

El problema de factorizar un número se considera un problema difícil (excepto tal vez para los enteros con factores pequeños). Hay una relación estrecha entre factorización y criptografía que hace que mucha gente este interesada en evaluar en la práctica la complejidad del problema de factorización de enteros que tienen divisores muy grandes. Aunque hay métodos de factorización enormemente más eficientes que el único método que hemos descrito, eso no significa que (hasta ahora) se pueda factorizar cualquier número en un tiempo razonable.

- En el año 2005, usando métodos avanzados de factorización (y con cinco meses de de cálculo en varias computadoras a la vez) se logró la factorización del número

$$
\begin{array}{l}
310741824049004372135075003588856793003734602284 27\\
2754572016194882320644051808150455634682967172328 6\\
7824379162728380334154710731085019195485290073377 2\\
482278352574238645401469173660247765234 6609
\end{array}
$$

Los factores son

1634733645809253848443133883865090859841783670033092312181110852389333100104508151212118167511579

y

1900871281664822113126851573935413975471896789968515493666638539088027103802104498957191261465571

El número, conocido como RSA-704,

74037563479561712828046796097429573142593188889231289084936232638972765034028266276891996419625117843995894330502127585370118968098286733173273108930900552505116877063299072396380786710086096962537934650563796359

no ha sido factorizado aún. RSA Security Inc. pagaba hasta el 2007, \$36000 dólares al que lo logrará factorizar. De hecho, hay números con más dígitos (y con premios más grandes) [8].

Vamos a ver ahora algunos teoremas acerca de números primos y algunos métodos de factorización.

Teorema 4

Cualquier número natural >1 factoriza como producto de primos.

Prueba. Ya sabemos que $n=2$ y $n=3$ son primos y que $n=4=2\cdot 2$. Supongamos que $n>1$ es compuesto, sea $n=ab$ con $1<a<n$ y $1<b<n$. Supongamos por inducción que $a=p_1\cdot p_2\cdots p_r$ y $b=q_1\cdot q_2\cdots q_s$ con p_i,q_i primos. Entonces $n=ab=p_1\cdot p_2\cdots p_r\cdot q_1\cdot q_2\cdots q_s$ que es un producto de primos.

La prueba del Teorema Fundamental de la Aritmética depende del siguiente lema.

Lema 2

Si p es primo y p divide a ab entonces $p|a$ o $p|b$.

Prueba. Recordemos que, usando la Identidad de Bézout demostramos que si $a|bc$ y $\text{mcd}(a,b)=1$ entonces $a|c$. Si p divide a ab entonces o $p|a$ o $\text{mcd}(p,a)=1$. En el primer caso, $p|a$. Si se da el segundo caso, entonces $p|b$.

Teorema 5 (Teorema Fundamental de la Aritmética)

Cualquier número natural ≥ 2 factoriza, de manera única, como un producto de primos.

Prueba. Por el teorema anterior, solo debemos demostrar la unicidad. Supongamos que el resultado es cierto para todos los números $< a$. Ahora suponemos que $a = p_1 \cdot p_2 \cdots p_m = q_1 \cdot q_2 \cdots q_n$ son dos factorizaciones de a en producto de primos.

Si $a = p_1$ es primo, $n = m$ y $p_1 = q_1$ pues un primo no factoriza como producto de dos o más primos.

Si a no es primo, como $p_1 \cdot p_2 \cdots p_m = q_1 \cdot q_2 \cdots q_n$ entonces, por el lema anterior, $p_1|q_j$ para algún j, es decir $p_1 = q_j$ por ser ambos primos. Por lo tanto

$$\frac{a}{p_1} = p_2 \cdots p_m = q_1 \cdot q_2 \cdots q_{j-1} \cdot q_{j+1} \cdots q_n$$

Como $\dfrac{a}{p_1}$ es un entero $< a$ entonces, por hipótesis de inducción, estas dos factorizaciones de $\dfrac{a}{p_1}$ son la misma, es decir los conjuntos $\{p_2, p_3, ..., p_m\}$ y $\{q_1, ..., q_{j-1}, q_{j+1}, ..., a_n\}$ son iguales. Como $p_1 = q_j$ entonces

$$\{p_1, p_2, p_3, ..., p_m\} \;=\; \{q_1, ..., q_{j-1}, q_j, q_{j+1}, ..., a_n\}$$

que es lo que se quería mostrar.

Si conocemos la factorización prima de dos o más números entonces podemos calcular rápidamente el máximo común divisor y le mínimo común múltiplo. Si

$$a = p_1^{r_1} \cdot p_2^{r_2} \cdots p_n^{r_n} \quad \text{y} \quad b = q_1^{s_1} \cdot q_2^{s_2} \cdots q_m^{s_m}$$

reescribimos a y b usando todos los factores $\{p_1, p_2, ...p_n, q_1, q_2, ..., q_m\} = \{c_1, c_2, ..., c_t\}$

$$a = c_1^{d_1} \cdots c_t^{d_t}$$
$$b = c_1^{e_1} \cdots c_t^{e_t}$$

donde algunos de los exponentes d_i o e_i son cero.

Entonces,

el máximo común divisor es $d = c_1^{f_1} \cdot c_2^{f_2} \cdots c_t^{f_t}$ donde f_i es el mínimo exponente en ambas factorizaciones, i.e, si $c_i = p_j = q_k$ entonces $f_i = \text{Mín}\{r_j, s_k\}$

el mínimo común múltiplo es $d = c_1^{g_1} \cdot c_2^{g_2} \cdots c_t^{g_t}$ donde g_i es el máximo exponente en ambas factorizaciones, i.e, si $c_i = p_j = q_k$ entonces $g_i = \text{Máx}\{r_j, s_k\}$

▷ Ejemplo 4

Si $x = 2^3 \cdot 3^2 \cdot 5^2$, $y = 2 \cdot 5^4 \cdot 7^3$ y $z = 2^2 \cdot 3 \cdot 5^3 \cdot 11$, reescribimos estos números como

$$\begin{aligned} x &= 2^3 \cdot 3^2 \cdot 5^2 \cdot 7^0 \cdot 11^0 \\ y &= 2^1 \cdot 3^0 \cdot 5^4 \cdot 7^3 \cdot 11^0 \\ z &= 2^2 \cdot 3^1 \cdot 5^3 \cdot 7^0 \cdot 11^1 \end{aligned}$$

entonces $\text{mcd}(x,y,z) = 2 \cdot 5^2$ y $\text{mcm}(x,y,z) = 2^3 \cdot 5^4 \cdot 7^3 \cdot 11^1$

Como método de cálculo general, este procedimiento no es eficiente pues obtener la factorización prima de un número grande es un problema complejo.

Teorema 6

Hay un número infinito de números primos.

Prueba. . La prueba de Euclides es como sigue. Si hubiera un número finito de primos, digamos $p_1, p_2, ...p_r$, el número $m = p_1 \cdot p_2 \cdots p_r + 1$ debería tener un factor primo, digamos p_j. Como $p_j | m$ y $p_j | (p_1 \cdot p_2 \cdots p_r)$ entonces $p_j | (m - p_1 \cdot p_2 \cdots p_r)$ y entonces $p_j | 1$ que es imposible. Así, el factor primo no es un p_j sino un nuevo primo, lo cual contradice la suposición de finitud.

Note que el número $m = p_1 \cdot p_2 \cdots p_r + 1$ es primo para algunos conjuntos finitos de primos, por ejemplo

$m = 2 \cdot 3 \cdot 5 \cdot 7 \cdot 11 \ + \ 1 = 2311$ es primo.

$m = 2 \cdot 3 \cdot 5 \cdot 7 \cdot 11 \cdot 13 \ + \ 1 = 30031$ no es primo.

4.2.1. Criba de Eratóstenes

La criba[2] de Eratóstenes es un algoritmo que permite hallar todos los números primos menores que un número natural dado n eliminando los números compuestos de la lista $\{2,...,n\}$. Es simple y razonablemente eficiente.

Primero tomamos una lista de números $\{2,3,...,n\}$ y eliminamos de la lista los múltiplos de 2. Luego tomamos el primer entero después de 2 que no fue borrado (el 3) y eliminamos de la lista sus múltiplos, y así sucesivamente. Los números que permanecen en la lista son los primos $\{2,3,5,7,...\}$.

Recordemos que un número n o es primo o es divisible por un primo $\leq \sqrt{n}$. Por lo tanto, si un número en el conjunto $\{2,3,...,n\}$ no es divisible por un primo $\leq \sqrt{n}$ entonces es primo. En lo que sigue en vez de usar $d \leq \sqrt{n}$ usamos la expresión equivalente (en este contexto) $d^2 \leq n$.

▷ Ejemplo 5

Encontrar los primos menores que 30. El proceso termina cuando el cuadrado del mayor número confirmado como primo es < 30.

La lista es

```
2   3   4   5   6   7   8   9   10  11  12  13  14  15  16  17
18  19  20  21  22  23  24  25  26  27  28  29  30
```

2 es primo, eliminamos todos los múltiplos de 2. La lista actualizada es

```
2 3   5   7   9     11     13     15     17
  19  21  23  25    27     29
```

El siguiente número (después del 2) es el 3. Como $3^2 < 30$, las cosas no terminan aquí. Eliminamos sus múltiplos.

La lista actualizada es

```
2 3  5   7          11     13           17
  19       23    25         29
```

El siguiente número (después del 3) es el 5. Como $5^2 < 30$, las cosas no terminan aquí. Eliminamos sus múltiplos.

La lista actualizada es

---

[2] Criba, tamiz y zaranda son sinónimos. Una criba es un herramienta que consiste de un cedazo usada para limpiar el trigo u otras semillas, de impurezas. Esta acción de limpiar se le dice cribar o tamizar.

```
2 3  5   7        11    13          17
  19     23             29
```

El siguiente número (después del 5) es el 7. Como $7^2 > 30$, El proceso terminó.

ALGORITMO VA AQUI

Detalles.

- Los múltiplos del $k-$ ésimo primo p_k se eliminan desde p_k^2 en adelante. Esto es así pues cuando llegamos a p_k ya se han eliminado $2\cdot p_k$, $3\cdot p_k$, ..., $p_{k-1}\cdot p_k$. Por ejemplo, cuando llegamos al primo 7, ya se han eliminado los múltiplos de $2, 3$ y 5, es decir $2\cdot 7$, $3\cdot 7$, $4\cdot 7$, $5\cdot 7$ y $6\cdot 7$. Por eso iniciamos en 7^2.

  Hasta aquí, la implementación podría hacerse con un arreglo booleano de tamaño n. El código Java sería

```
int n = 100000000;
boolean[] esPrimo = new boolean[n + 1];
for (int i = 2; i <= n; i++)
    esPrimo[i] = true;
for (int i = 2; i*i <= n; i++)
{
    if (esPrimo[i])
    {
        for (int j = i; i*j <= n; j++)
            esPrimo[i*j] = false;
    }
}
```

- Como los pares no son primos (excepto el 2), solo deberíamos analizar los números impares [3]

$$\left\{2i+3 \ : \quad i = 0, 1, ... \left\lfloor \frac{n-3}{2} \right\rfloor \right\} = \{3, 5, 7, 9, ...\}$$

  Por cada primo $p = 2i+3$ (tal que $p^2 < n$) debemos eliminar los múltiplos impares de p menores que n, a saber $(2k+1)p = (2k+1)(2i+3)$, $k = i+1, i+2, ...$ (observe que si $k = i+1$ entonces el primer múltiplo en ser eliminado es $p^2 = (2i+3)(2i+3)$).

---

[3] Si $n = 2k+1$ es impar, $\left\lfloor \frac{n-3}{2} \right\rfloor = k-1 \in \mathbb{N}$. Si $n = 2k$ es par, $\left\lfloor \frac{n-3}{2} \right\rfloor = k-2$

Esto nos dice que para implementar el algoritmo solo necesitamos un arreglo (booleano) de tamaño "`(n-3)/2`"[4]. El arreglo es `EsPrimo[i], i=0,1,...,(n-3)/2.` Cada entrada `EsPrimo[i]` indica si el número primo $2i+3$ es primo o no. Así, si el número $s = 2i+3$ es primo entonces `EsPrimo[(s-3)/2]=true.` Por tanto, lo único que debemos hacer es inicializar el arreglo con todos sus valores en `true` y luego asignar el valor `false` a las entradas que representan a los números $(2k+1)(2i+3), \quad k = i+1, i+2, ...$ es decir, debemos hacer la asignación

```
EsPrimo[((2k+1)(2i+3) - 3)/2] = false
```

Eso es todo. En seudocódigo

```
//Entrada: n
//Salida : primos <= n

  max = (n-3)/2
  esPrimo[i] arreglo booleano, i=0,...max;
  for i=0 hasta max
      esPrimo[i]=true

  i=0
  While (2*i+3)*(2*i+3) <= n
  {
      k=i+1
      While (2*k+1)*(2*i+3) <= n
      {
        esPrimo[((2*k+1)*(2*i+3)-3)/2]=false;
        k=k+1
      }
      i=i+1
  }
 //imprimir
 imprima 2
 for i=0 hasta max
     if esPrimo[i]= true
       imprima 2*i+3
```

Implementación en Java.

Vamos a implementar una clase que recibe el número natural $n > 2$ y devuelve un vector con los números primos $\leq n$. Para colar los números compuestos usamos un arreglo

---

[4] En Java esta división es "entera"

```
boolean [] esPrimo = new boolean[(n-3)/2].
```

Al final llenamos un vector con los primos que quedan.

```
/*********************************************************
* Si estamos en la carpeta C:\eratostenes>                *
* Compilar:      C:\eratostenes> javac cribaEratostenes.java *
* Ejecutar:      C:\eratostenes> java cribaEratostenes       *
* Ver el comentario sobre el uso de la memoria para          *
* números grandes (más adelante)                             *
**********************************************************/

import java.util.Vector;

public class cribaEratostenes
{
     //constructor
     cribaEratostenes(){ }

     //método que calcula la lista
     Vector HacerlistaPrimos(int n)
     {
        Vector        salida = new Vector(1);
        int k   = 1;
        int max = (n-3)/2;
        boolean[]   esPrimo  = new boolean[max+1];

        for(int i = 0; i <= max; i++)
            esPrimo[i]=true;

        for(int i = 0; (2*i+3)*(2*i+3) <= n; i++)
        {
          k = i+1;

          while( ((2*k+1)*(2*i+3)) <= n)
          {
            esPrimo[((2*k+1)*(2*i+3)-3)/2]=false;
            k++;
          }
         }

         salida.addElement(new Integer(2));

         for(int i = 0; i <=max; i++)
         { if(esPrimo[i])
           salida.addElement(new Integer(2*i+3));
         }
         salida.trimToSize();
         return salida;
     }

    //Cómo usar esta clase?
    public static void main(String[] args)
    {
        int n = 100;
        Vector lista_de_primos;
        cribaEratostenes  criba = new  cribaEratostenes();

        //usar el método HacerlistaPrimos(n) vía el objeto criba
        lista_de_primos = criba.HacerlistaPrimos(n);

        //Primos <= n
       System.out.println("Primos <="+ n+": "+lista_de_primos.size()+"\n");
        //imprimir vector
       for(int p = 1; p < lista_de_primos.size(); p++)
       {
         Integer num = (Integer)lista_de_primos.elementAt(p);
         System.out.println(""+(int)num.intValue());
       }
    }
}
```

Uso de la Memoria

En teoría, los arreglos pueden tener tamaño máximo `Integer.MAX_INT` $= 2^{31}-1 = 2\,147\,483\,647$ (pensemos también en la posibilidad de un arreglo multidimensional!). Pero en la práctica, el máximo tamaño del array depende del hardware de la computadora. El sistema le asigna una cantidad de memoria a cada aplicación; para valores grandes de n puede pasar que se nos agote la memoria (veremos el mensaje "`OutOfMemory Error`"). Podemos asignar una cantidad de memoria apropiada para el programa "cribaEratostenes.java", si n es muy alto, desde la línea de comandos. Por ejemplo, para calcular los primos menores que $n = 100\,000\,000$, se puede usar la instrucción

```
C:\eratostenes> java -Xmx1000m -Xms1000m cribaEratostenes
```

Esta instrucción asigna al programa una memoria inicial (Xmx) de 1000 MB y una memoria máxima (Xms) de 1000 MB (siempre y cuando existan tales recursos de memoria en nuestro sistema).

En todo caso hay que tener en cuenta los siguientes datos

```
                n     Primos <= n
---------------------------------
               10               4
              100              25
            1 000             168
```

10 000	1 229
100 000	9 592
1 000 000	78 498
10 000 000	664 579
100 000 000	5 761 455
1 000 000 000	50 847 534
10 000 000 000	455 052 511
100 000 000 000	4 118 054 813
1 000 000 000 000	37 607 912 018
10 000 000 000 000	346 065 536 839

Primos entre m y n.

Para encontrar todos los primos entre m y n, eliminamos todos los múltiplos de los primos $\leq \sqrt{n}$. Para hacer esto, notemos que si p es un número primo entonces los múltiplos de p más grandes que m son

$$m-1-r+p,\ \ m-1-r+2p,\ \ m-1-r+3p,\ ...$$

siendo $m-1=pq+r$ con $0\leq r<p$. Esto es así porque como $p>r$ entonces $-r+kp>1$ y además $m-1-r+kp=p(q+k)$.

▷ Ejemplo 6

Encontrar los primos entre que 10 y 30. Primero enumeramos de $m=10$ a $n=30$.

```
10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
```

Debemos eliminar los múltiplos de los primos $\leq \sqrt{30}\approx 5$. Es decir los primos $p=2,3,4,5$.

Eliminamos los múltiplos de 2. La lista queda así

```
11      13      15      17      19      21      23      25      27      29
```

Luego eliminamos los múltiplos de 3 en la lista. Como $10-1=3\cdot 3+0$ entonces el múltiplo de 3 más pequeño entre 10 y 30 es $10-1-0+3=12$. Ahora vamos de tres en tres, iniciando en 12, eliminando los múltiplos de 3 de esta lista (es decir sacamos $\{12,15,18,21,24,27,30\}$). La lista queda así

```
11      13              17      19                      23      25              29
```

Luego eliminamos los múltiplos de 5 en la lista. Como $10-1=5\cdot 1+4$ entonces el múltiplo de 5 más pequeño entre 10 y 30 es $10-1-4+5=10$. Ahora vamos de cinco en cinco, iniciando en 10, eliminando los múltiplos de 5 de esta lista (es decir sacamos $\{10,15,20,25\}$). La lista queda así

```
11  13        17  19          23              29
```

Y todo termina aquí pues el siguiente primo es $7 > \sqrt{30}$.

Para que la criba opere entre m y n necesitamos tener un arreglo con todos los primos $\leq \sqrt{n}$.

Ejercicios 1

1. Implementar un programa para calcular los primos entre m y n.

4.2.2. Teorema de los Números Primos. Primos en un Intervalo

En esta sección veremos algunos teoremas clásicos acerca de la distribución de primos. Las pruebas de los teoremas aparecen en libros avanzados (ver por ejemplo [1] y [2]), así que solo haremos una breve discusión acerca del alcance y significado de estos teoremas. Esto será suficiente para comprender algunas decisiones a la hora de implementar algunos algoritmos que aparecen más adelante.

Com ya vimos, el número de primos $\leq x$ se denota $\pi(x)$. Existen fórmulas "sencillas" para aproximar asintóticamente $\pi(x)$. Gauss y Legendre formularon las primeras conjeturas, basadas en métodos empíricos, acerca de la distribución de los primos. Aunque esta es una gran historia[5], un resumen algo brusco es este: ellos se ocuparon, en ciertos intervalos de la forma $[0{,}95 \cdot 10^n, 1{,}05 \cdot 10^n]$ y $[10^n, 10^n + 150000]$, de calcular la proporción de números primos. Calculando valores para 10^n y 10^{2n} notaron que la densidad de primos se reduce a la mitad cuando tomamos el cuadrado de x. Una función que modela este comportamiento es $1/\ln(x)$ pues $1/\ln(x^2) = 0{,}5/\ln(x)$.

En 1849 Gauss escribió a Encke

> "Siendo un muchacho consideré el problema (de cuántos primos hay hasta un número dado), en 1792 o 1793, y encontré que la densidad de primos alrededor de x es $1/\log x$, así que el número de primos hasta una cota x dada es aproximadamente $\int \frac{dn}{\log n}$"

Esta es la tabla que Gauss envió a Encke (los errores de conteo, una vez corregidos, dan un resultado a favor de Gauss).

---

[5] Ver [6]

x	Primos $< x$	$\int \frac{dn}{\log n}$	Diferencia
500000	41556	41606.4	50.4
1000000	78501	78627.5	126.5
1500000	114112	114263.1	151.1
2000000	148883	149054.8	171.8
2500000	183016	183245.0	229.0
3000000	216745	216970.6	225.6

Gauss no indica qué exactamente quiere decir con el símbolo $\int \frac{dn}{\log n}$ pero, de acuerdo a la tabla anterior, parece ser $\int_2^x \frac{dn}{\log n}$.

La tabla corregida (sin decimales) es

x	Primos $< x$	$\int_2^x \frac{dt}{\log t}$	Diferencia
500000	41538	41605.	68
1000000	78498	78626.	129
1500000	114155	114262.	108
2000000	148933	149054.	121
2500000	183072	183244.	172
3000000	216816	216970.	154

Legendre publicó en su *Teoría de Números* ($\sim$ 1800) una fórmula empírica para $\pi(x)$ más o menos parecida a la de Gauss. Asumiendo que la densidad promedio de primos es de la forma

$$\frac{1}{A_1 x^{m_1} + A_2 x^{m_2} + ...} \quad (*)$$

con $m_1 > m_2 > ...,$ Legendre deduce que la densidad debe ser de la forma[6]

$$\frac{1}{A \log x + B}$$

y determina A y B de manera empírica: $A = 1$ y $B = 1{,}08366$.

Más tarde, Chebyshev probó que

---

[6] basado resultados obtenidos por Euler.

i. la mejor aproximación de la forma (*) se obtiene con $B = 1$ (aunque para efectos de comportamiento asintótico usamos $B = 0$)

ii. el error relativo en la aproximación

$$\pi(x) \sim \int_2^x \frac{dt}{\ln t}$$

es menos del 11 % para x suficientemente grande, es decir

$$0{,}89 \cdot \int_2^x \frac{dt}{\ln t} < \pi(x) < 1{,}10555 \cdot \int_2^x \frac{dt}{\ln t}, \quad x \ \text{ suficientemente grande.}$$

y en 1962 Rosser y Schoenfeld probaron

$$\frac{x}{\ln x} \leq \pi(x), \quad x \geq 17$$

y más generalmente

$$\frac{x}{\ln x}\left(1 + \frac{1}{2\ln x}\right) \leq \pi(x) \leq \frac{x}{\ln x}\left(1 + \frac{3}{2\ln x}\right) \quad x \geq 59$$

En la siguiente tabla se muestran unos cálculos usando la integral logarítmica $\mathrm{li}(x) = \displaystyle\int_0^x \frac{dx}{\ln x}$ y $x/\ln(x)$.

x	$\pi(x)$	$\mathrm{li}(x)$	$x/\ln(x)$	$\mathrm{li}(x) - \pi(x)$	$\pi(x) - x/\ln(x)$
20000000	1270607	1270905.	1189680.	298.	80926.
40000000	2433654	2434016.	2285141.	362.	148512.
60000000	3562115	3562683.	3350110.	568.	212004.
80000000	4669382	4670090.	4396199.	708.	273182.
100000000	5761455	5762209.	5428681.	754.	332773.
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
10^{18}	24 739 954 287 740 860	...	...	21 949 555	612 483 070 893 536

Notación

En lo que sigue, "$f(x) \sim g(x), \quad x \longrightarrow \infty$" significa que el *error relativo* [7] se hace arbitrariamente pequeño conforme x se hace muy grande o, lo que es lo mismo, que

$$\lim_{x\to\infty} \frac{f(x)}{g(x)} = 1$$

En 1896, los matemáticos Hadamard y de la Vallée-Poussin de manera independiente probaron el

Teorema 7 **(Teorema de los Números Primos)**

$\pi(x)$ es asintóticamente igual a $\operatorname{li}(x)$, es decir

$$\lim_{x\to\infty} \frac{\pi(x)}{\operatorname{li}(x)} = 1$$

¿Qué dice este teorema?. Primero tenemos que ubicarnos en la naturaleza de las aproximaciones que aquí se manejan.

- El teorema dice que el *error relativo* $\left|\frac{\pi(x)}{\operatorname{li}(x)} - 1\right| = \left|\frac{li(x)-\pi(x)}{\operatorname{li}(x)}\right| \longrightarrow 0$ o también $\left|\frac{li(x)-\pi(x)}{\pi(x)}\right| \longrightarrow 0$ conforme $x \longrightarrow \infty$, aunque el error absoluto $|li(x)-\pi(x)|$ puede ser bastante grande.

  Por ejemplo, si $x = 10^{18}$ entonces

  $\pi(x) = 24\ 739\ 954\ 287\ 740\ 860$

  $\operatorname{li}(x) \ - \ \pi(x) = 21\ 949\ 555$

  error relativo: $\dfrac{li(x)-\pi(x)}{\pi(x)} = 8{,}87211 \times 10^{-7}$

  En realidad se puede establecer que

$$\lim_{x\to\infty} \frac{\operatorname{li}(x)}{x/\ln(x)} = \lim_{x\to\infty} \frac{\operatorname{li}(x)}{x/(\ln(x)-1)} = \lim_{x\to\infty} \frac{\pi(x)}{x/\ln(x)} = 1$$

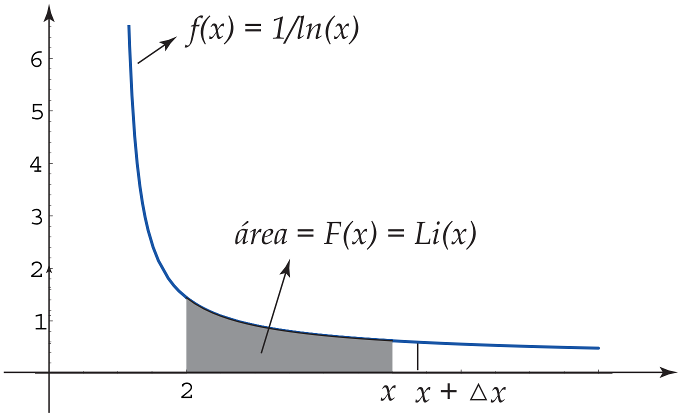
---

[7] Si $\widehat{s}$ es una aproximación de s, el error relativo $\dfrac{\widehat{s}-s}{s}$ mide el porcentaje de error de la aproximación. Por ejemplo, si $s = 0{,}000012$ y $\widehat{s} = 0{,}000009$, el error absoluto es $|\widehat{s}-s| = 0{,}000003$ pero el error relativo es $\dfrac{\widehat{s}-s}{s} = 0{,}25$, es decir un 25 %. Si $s = 1\ 000\ 000$ y $\widehat{s} = 999\ 999$, el error absoluto es $|\widehat{s}-s| = 1$ pero el error relativo es $\dfrac{\widehat{s}-s}{s} = 10^{-6}$.

- Es muy útil entender este teorema en términos "probabilísticos", aunque no tengamos propiamente una función de densidad.

  El Teorema de los Números Primos dice que un entero aleatoriamente escogido ('cercano' a x), es primo con probabilidad $\dfrac{1}{\ln x}$ o, como se dice a veces, la densidad *promedio* de primos cercanos a x es *aproximadamente* $1/\ln x$.

  Para ser más precisos, si $\triangle x$ es tal que $\triangle x + x$ es asintóticamente igual a x, la proporción de primos en el intervalo $[x, \triangle x + x]$ es asintóticamente igual a $1/\ln x$, es decir si t es un número que está a una distancia $\leq \triangle x$ ("pequeña respecto" a x) de x, entonces la probabilidad de que t sea primo se aproxima a $1/\ln x$ conforme $x \longrightarrow \infty$.



  Esta última afirmación implica que los primos se "arralan" conforme x crece. Visto de otra forma, los números grandes tienen una probabilidad alta de ser compuestos pues habría muchos primos $\leq \sqrt{x}$ que podrían ser divisores.

  Por ejemplo, sea $\triangle x = \sqrt{x}$ (x domina a $\sqrt{x}$ si x se hace grande).

  La proporción de números en $[x, \triangle x + x]$ que son primos es $Pr_1 = \dfrac{\pi(\triangle x + x) - \pi(x)}{\triangle x}$. Esta es la probabilidad de que un número escogido aleatoriamente en $[x, \triangle x + x]$ sea primo.

  La probabilidad aproximada es $Pr_2 = 1/\ln(x)$.

x	$\triangle x$	Primos en $[x, \triangle x + x]$	Pr_2	Pr_1
1000	100	4	0.4	0.2171
1 000 000	1000	75	0.0723824	0.075
10 000 000	3162.28	197	0.062042	0.0622969
100 000 000	10000	551	0.0551	0.0542868

4.2.3. Teorema de Mertens. Proporción de Números sin Factores Primos $\leq G$.

La probabilidad de que $x \in \{1,2,...,n\}$ sea divisible por p es aproximadamente $\dfrac{1}{p}$. Por ejemplo, la probabilidad de que $x \in \{1,2,3,...,10\}$, sea divisible por 5 es $\dfrac{2}{10} = \dfrac{1}{5} = 0{,}2$; pero la probabilidad de que $x \in \{1,2,3,...,11\}$, sea divisible por 5 es $\dfrac{2}{11} = 0{,}181818...$ Si 15, la probabilidad vuelve a ser $1/5$.

Luego, $1-1/p$ es aproximadamente la probabilidad de que x *no* sea divisible por p.

Aproximadamente la mitad de los números enteros en $\{1,2,...,n\}$ son divisibles por 2. Decimos "aproximadamente" pues, por ejemplo, entre 1 y 11 solo hay cinco pares y entonces $5/11 \approx 0{,}45$.

De manera similar, $\left(1-\dfrac{1}{2}\right)\left(1-\dfrac{1}{3}\right)$ es aproximadamente la probabilidad de que un número entero en $\{1,2,...,n\}$ no sea divisible ni por 2 ni por 3. Decimos "aproximadamente" pues esta afirmación solo es exacta para múltiplos de 2 y 3.

En general,

$$\left(1-\frac{1}{2}\right)\left(1-\frac{1}{3}\right)\cdots\left(1-\frac{1}{p_k}\right)$$

es la cantidad *aproximada* de números en $\{1,2,...,n\}$ que *no* son divisibles por ningún primo entre 2 y $\leq p_k$ (el $k-$ésimo primo).

Por ejemplo

- Entre en $\{1,2,...,100\}$ hay 78 números que son divisibles por 2, 3, 5 o 7. La proporción es $\dfrac{78}{100} = 0{,}78$.

  Entre en $\{1,2,...,500\}$ hay 385 números que son divisibles por 2, 3, 5 o 7. La proporción es $\dfrac{385}{500} = 0{,}77$

  Ahora

  $$\left(1-\frac{1}{2}\right)\left(1-\frac{1}{3}\right)\left(1-\frac{1}{5}\right)\left(1-\frac{1}{7}\right) = 0{,}228571$$

  lo que indica que aproximadamente el 77% ($1-0{,}228571 = 0{,}771429$) de los números entre 1 y n son divisibles por $2, 3, 5$ o 7.

  Por otra parte, en $\{1,2,...,100\}$ hay 80 números que son divisibles por 2, 3, 5, 7, 11 o 13. La proporción es $\dfrac{80}{100} = 0{,}80$ y en $\{1,2,...,12564544\}$ hay 10154562 números que son divisibles por 2, 3, 5, 7, 11 o 13. La proporción es $\dfrac{10154562}{12564544} = 0{,}808192.$

Ahora

$$\left(1-\frac{1}{2}\right)\left(1-\frac{1}{3}\right)\cdots\left(1-\frac{1}{13}\right) = 0{,}191808$$

esto indica que hay aproximadamente un 20 % de los números entre $\{1,...,n\}$ *no* divisibles por 2, 3, 5,7,11 o 13.

- Para $p_{1229} = 9973$

$$\left(1-\frac{1}{2}\right)\left(1-\frac{1}{3}\right)\cdots\left(1-\frac{1}{p_{1229}}\right) = 0{,}0608847$$

esto indica que aproximadamente el 94 % de los números entre $\{1,2,...,n\}$ son divisibles por un primo entre 2 y 9973 y aproximadamente un 6 % *no* son divisibles.

No se puede considerar como eventos independientes la divisibilidad por diferentes primos. Por ejemplo, si $X_2 =$ "x es divisible por 2" y $X_3 =$ "x es divisible por 3" entonces si $n = 13$ tenemos

$Pr[X_2] = 6/13$

$Pr[X_3] = 4/13$

$Pr[X_2 \wedge X_3] = 2/13 \neq Pr[X_2]Pr[X_3] = 24/169$

en cambio, X_2 y X_3 son independientes si $n = 12$. Debido a esto es que es erróneo esperar que, como la probabilidad de que x sea primo es la probabilidad de que x no sea divisible por ningún primo $p \leq \sqrt{x}$, el producto $\displaystyle\prod_{2\leq p\leq\sqrt{x}}\left(1-\frac{1}{p}\right)$ sea asintóticamente igual a $1/\ln(x)$. El resultado correcto se establece en el siguiente teorema

Teorema 8 **(Teorema de Mertens)**

$$\prod_{\substack{2\leq p\leq x,\\ p\text{ primo}}}\left(1-\frac{1}{p}\right) \sim \frac{e^{-\gamma}}{\ln(x)} \approx \frac{0{,}5615}{\ln(x)} \quad \text{si } x \longrightarrow \infty \tag{4.2.1}$$

γ es la constante de Euler

Sustituyendo x por $x^{0,5}$ encontramos que

$$\prod_{\substack{2\leq p\leq\sqrt{x},\\ p \text{ primo}}} \left(1-\frac{1}{p}\right) \sim \frac{2e^{-\gamma}}{\ln(x)} \approx \frac{1{,}12292}{\ln(x)} \quad \text{si } x \longrightarrow \infty$$

También, multiplicando (4.2.1) por $2,$ la fórmula

$$\prod_{\substack{3\leq p,\\ p \text{ primo}}}^{G} \left(1-\frac{1}{p}\right) \sim \frac{2e^{-\gamma}}{\ln(G)} \approx \frac{1{,}12292}{\ln(G)}$$

nos daría la proporción de números impares que no tienen un factor primo $\leq G.$

G	Proporción approx de impares sin factores $\leq G.$
100	0.243839
1000	0.162559
10000	0.121919
100000	0.0975355
1000000	0.0812796
10000000	0.0696682
100000000	0.0609597
1000000000	0.0541864
10000000000	0.0487678

4.2.4. Teorema de Dirichlet. El Número de Primos en una Progresión Aritmética.

Sean a y b números naturales y consideremos los enteros de la progresión aritmética $an+b,\quad n=0,1,2,\ldots.$ ¿Cuántos de estos números $\leq x$ son primos?. Si $\pi_{a,b}(x)$ denota la cantidad de primos $\leq x$ en esta sucesión, entonces

Teorema 9 (Teorema de Dirichlet)

Si a y b son primos relativos entonces

$$\lim_{x\to\infty} \frac{\pi_{a,b}(x)}{\operatorname{li}(x)} = \frac{1}{\varphi(a)}$$

φ es la función de Euler,

$$\varphi(m) = \text{ número de enteros positivos } \leq m \text{ y coprimos con } m$$

En particular $\varphi(6) = 2$ y $\varphi(10) = 4.$ De acuerdo con el teorema de Dirichlet, "en el infinito" (es decir tomando el límite), un 50 % de primos están en cada una de las sucesiones $6n + 1$ y $6n - 1$ mientras que un 25 % de primos se encuentra en cada una de las cuatro sucesiones $10n \pm 1$ y $10n \pm 3.$ Se puede probar también que si mcd $(a, b) = 1$ entonces la sucesión $an + b$ tiene un número infinito de primos.

En realidad, las sucesiones $6n + 1$ y $6n - 1$ contienen todos los primos pues todo primo es de la forma $6k + 1$ o $6k - 1.$ En efecto, cualquier natural es de la forma $6k + m$ con $m \in \{0, 1, 2, 3, 4, 5\}$ (por ser "$\equiv \pmod 6$" una relación de equivalencia en $\mathbb{N}$). Ahora, todos los enteros $6k + m$ son claramente compuestos excepto para $m = 1$ y $m = 5$ por lo que si p es primo, entonces $p = 6k+1$ o $p = 6k+5 = 6q-1$ (si $q = k + 1$), es decir p es de la forma $6k \pm 1.$

4.2.5. Factorización de un Número por Ensayo y Error.

Identificar si un número es primo es generalmente fácil, pero factorizar un número (grande) arbitrario no es sencillo. Recientemente (2005) se factorizó un número de de 200 cifras[8] (RSA-200). Se tardó cerca de 18 meses en completar la factorización con un esfuerzo computacional equivalente a 53 años de trabajo de un CPU 2.2 GHz Opteron. El método de factorización de un número N probando con divisores primos ("trial division") consiste en probar dividir N con primos pequeños. Para esto se debe previamente almacenar una tabla suficientemente grande de números primos o generar la tabla cada vez. Como ya vimos en la criba de Eratóstenes, esta manera de proceder nos trae consigo problemas de tiempo y de memoria. En realidad es más ventajoso proceder de otra manera.

- Para hacer la pruebas de divisibilidad usamos los enteros 2, 3 y la sucesión $6k \pm 1, \;\; k = 1, 2, \ldots.$

  Esta elección cubre todos los primos e incluye divisiones por algunos números compuestos (25,35,...) pero la implementación es tan sencilla (y el programa suficientemente rápido) que vale la pena permitirse estas divisiones inútiles.

- En general, debemos decidir un límite G en la búsqueda de divisores. Si se divide únicamente por divisores primos $\leq G,$ se harían $\pi(G) \approx G/\ln G$ divisiones. Si se divide por 2, 3 y $6k \pm 1$ se harían aproximadamente $G/3$ divisiones[9] de las cuales $\dfrac{G/3}{G/\ln G} = 3/\ln G$ son divisiones útiles. Si $G = 10^6$, tendríamos $\approx$ 22 % divisiones útiles. En este caso, un ciclo probando divisiones por primos únicamente es $\approx 1/0{,}22 = 4{,}6$ veces más lento[10].

---

[8] Se trata del caso más complicado, un número que factoriza como producto de dos primos (casi) del mismo tamaño.

[9] Pues los números naturales ($\leq G$) son de la forma $6k + m$ con $m \in \{0, 1, ..., 5\}$ y solo estamos considerando $m = 1, 5,$ es decir una tercera parte.

[10] Aún si se almacena previamente una tabla de primos en forma compacta, esto consume tiempo [6]

Cuando se juzga la rapidez de un programa se toma en cuenta el tiempo de corrida en el *peor caso* o se toma en cuenta el *tiempo promedio de corrida* (costo de corrida del programa si se aplica a muchos números). Como ya sabemos (por el Teorema de Mertens) hay un porcentaje muy pequeño de números impares sin divisores $\leq G,$ así que en promedio, nuestra implementación terminará bastante antes de alcanzar el límite G (el "peor caso" no es muy frecuente) por lo que tendremos un programa con un comportamiento deseable.

Detalles de la implementación.

- Para la implementación necesitamos saber cómo generar los enteros de la forma $6k \pm 1.$ Alternando el -1 y el 1 obtenemos la sucesión

  $$5,\ 7,\ 11,\ 13,\ 17,\ 19, ...$$

  que iniciando en $5,$ se obtiene alternando los sumandos 2 y $4.$ Formalmente, si iniciamos con el número $m_k = 6k-1$ y si $s_k = 6k+1$ entonces un pequeño cálculo nos lleva a

  $$\begin{aligned} &s_k = m_k + 2 \\ &m_{k+1} = s_k + 4 \quad (\text{y también } m_{k+1} = m_k + 6\,) \end{aligned}$$

  Luego la sucesión se obtiene iniciando en $m_1 = 5$ y alternando los sumandos 2 y $4.$

  En la implementación, usamos una variable `p` y hacemos la asignación inicial `p=5`; en el ciclo hacemos dos divisiones de prueba, una con `p=p+2` y luego otra con `p=p+6`.

- En cada paso debemos verificar si el divisor de prueba `p` alcanzó el límite $\text{Mín}\,\{G, \sqrt{N}\}.$ Si se quiere evitar el cálculo de la raíz, se puede usar el hecho de que si $p > \sqrt{N}$ entonces $p > N/p.$

  En la implementación Java usaremos la clase `sqrt` (para BigInteger) que ya hemos implementado en el capítulo dedicado a este lenguaje.

- Si se tiene éxito con un número de prueba p el nuevo límite será $\text{Mín}\,\{G, \sqrt{N/p}\}.$
  pues redefinimos N como $N/p.$

- Para no pasar por alto factores primos de multiplicidad $> 1,$ debemos probar cada vez de nuevo los factores primos encontrados (como divisores de N) hasta que éstos ya no dividan lo que queda de $N.$

Implementación en Java.

Creamos una clase que busca factores primos de un número N hasta un límite G. Para hacer esto creamos dos métodos `reducir` y `factorizar`. El método `reducir(N,p)` verifica si N es divisible por p. Si p es divisor de N, se almacena en un vector, N se redefine cada vez como N/p y se continua dividiendo por p hasta que el residuo sea cero. Observe que antes de que se pruebe con un número compuesto (una división inútil) ya se ha probado reducir con los primos de su factorización, así que estos números no entran en la lista de factores.

El método `factorizar(N,G)` llama a `reducir(N,p)` con $p = 2, 3, 2k \pm 1, \quad k = 1, 2...$ hasta que se alcance el límite de pruebas G.

En esta implementación usamos $G = \sqrt{N}$. El código para escoger el mínimo entre ambos está presente pero deshabilitado. También llamamos la clase `sqrt` para BigInteger que implementamos previamente en el capítulo 1.

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

public class TrialDivision
{
//Recibe N y G. Default es G=Sqrt(N)
//Retorna un vector con los factores de N
private Vector listaFactores;
BigInteger  Ge      = BigInteger.ONE;
BigInteger  Nf      = new BigInteger("1");
Vector      salida  = new Vector(1);
BigInteger  plimite = BigInteger.ZERO;
BigInteger  UNO     = new BigInteger("1");
BigInteger  DOS     = new BigInteger("2");
BigInteger  TRES    = new BigInteger("3");
BigInteger  SEIS    = new BigInteger("6");
//clase sqrt tiene el  método "raiz" para BigInteger
sqrt eval           = new sqrt();
//var solo para efectos de impresión
static BigInteger  limIni  = BigInteger.ZERO;

public TrialDivision(BigInteger N)
{
         Nf=N;
        listaFactores=Factorizar(Nf,Ge);
}


public TrialDivision(BigInteger N, BigInteger G)
{
        Nf=N;
        listaFactores=Factorizar(Nf,G);
}

// uso: Nf= reducir(Nf,p)
public BigInteger reducir(BigInteger Ne, BigInteger p)
{
      //divide Ne por p repetidamente.
      BigInteger[] dr = new BigInteger[2];
      int residuo=0;

      if(p.compareTo(UNO)> 0)
      {
        dr      =  Ne.divideAndRemainder(p);
        residuo =  dr[1].compareTo(BigInteger.ZERO);
        while(residuo==0)
        {
         Ne=dr[0]; // Ne = Ne/p
         dr      =  Ne.divideAndRemainder(p);
         residuo =  dr[1].compareTo(BigInteger.ZERO);
         salida.addElement(p); //p es objeto BigInteger
        }
      }//if

      plimite = eval.raiz(Nf);

      // Habilitar límite = Mín {G,Sqrt(Nf)}
   /*
      if(plimite.compareTo(Ge)>=0 )
        plimite=Ge;
   */
        return Ne;
}//
public Vector Factorizar(BigInteger Ne, BigInteger lG)
{
        BigInteger p     = new BigInteger("5");
        BigInteger[] ndr = new BigInteger[2];
                 //solo para efectos de impresión
                 limIni = eval.raiz(Ne);
        Ge=lG;
        Nf = Ne;
        Nf = reducir(Nf, DOS);
        Nf = reducir(Nf, TRES);

        while(p.compareTo(plimite)<=0)
        {
             ndr =  Nf.divideAndRemainder(p);
             if(ndr[1].compareTo(BigInteger.ZERO)==0)
                Nf= reducir(Nf, p);         //dividir por p
             ndr =  Nf.divideAndRemainder(p.add(DOS));
             if(ndr[1].compareTo(BigInteger.ZERO)==0)
                Nf= reducir(Nf, p.add(DOS)); //dividir por p+2
             p = p.add(SEIS);
        }
        if(p.compareTo(eval.raiz(Nf))>0)
           reducir(Nf, Nf);
        return salida;
}

public Vector toVector()
{
   return listaFactores;
}

 // Cómo usar la clase.
 public static void main(String[] args)
 {
 BigInteger N   = new BigInteger("367367653565289976655797");
 BigInteger Pba = new BigInteger("1");
 Vector factores;

 //Factorización completa
 TrialDivision  Vfactores = new  TrialDivision(N);
 // Factores primos con límite  G = 1000
 //TrialDivision  Vfactores = new  TrialDivision(N, new BigInteger("1000"));

 factores = Vfactores.toVector();

 //Imprimir vector de factores primos
 System.out.println("N = "+N);
 System.out.println("Factores Primos <= " + limIni);
 for(int p = 0; p < factores.size(); p++)
    System.out.print(""+factores.elementAt(p)+",  ");

 //Imprimir factorización completa si el límite es sqrt(N)
 System.out.println("\n\n");
 System.out.print("N = ");
 for(int p = 0; p < factores.size(); p++)
 {
   System.out.print(""+factores.elementAt(p)+"*");
   Pba=Pba.multiply((BigInteger)factores.elementAt(p));
 }
  System.out.print("\n\n");
  System.out.println("p_1p_2...p_k = "+Pba);
 }//main
}//
```

Al ejecutar este programa con $N = 3673676535652899766557 97$, después de varios segundos la salida es

```
N = 367367653565289976655797
Factores Primos <= 606108615320
13,  1051,  4969,  22606601,  239359451,

N = 13*1051*4969*22606601*239359451

p_1p_2...p_k = 367367653565289976655797
```

4.3. El Método “rho” de Pollard.

Uno de los problemas más grandes relacionados con la computación matemática, es encontrar los divisores primos de un número natural, sin tener que realizar muchas divisiones a este número. El método “Trial Division”(factorización por ensayo y error) para factorización, trabaja bien buscando pequeños factores primos de enteros, pero no trabaja bien con enteros grandes. Sin embargo, hay que tomar en cuenta que la mayoría de los métodos de factorización dependen de algunas propiedades especiales para garantizar el éxito.

En 1975, John Pollard propuso un método muy eficaz, hoy conocido como “rho” de Pollard, para encontrar un factor pequeño, diferente de uno, de un número entero grande n. Este método trabaja sobre una sucesión de x_i definida a partir de una función previamente determinada, donde cada x_i pertenece a una de las clases residuales del número que se quiere factorizar.

Ejemplo: Se quiere factorizar $n = 527$. Usamos la función $f(n) = n^2 + 1$, y definimos la sucesión de x_i de la siguiente manera: $x_0 = 0$ y $x_{i+1} = f(x_i) \pmod{n}$, $i \geq 0$. Los primeros 22 términos de la sucesión son:

$$\begin{array}{r} 0, 1, 2, 5, \\ 26, 150, 367, 305, 274, 243, \\ 26, 150, 367, 305, 274, 243, \\ 26, 150, 367, 305, 274, 243, \\ \vdots \end{array}$$

Como se puede notar, hay un patrón que se repite después del término x_4.

Observe que:

$$\begin{array}{rcl} i = 1, \quad \text{MCD}(x_2 - x_1, n) & = & \text{MCD}(2 - 1, 527) = 1 \\ i = 2, \quad \text{MCD}(x_4 - x_2, n) & = & \text{MCD}(26 - 2, 527) = 1 \\ i = 3, \quad \text{MCD}(x_6 - x_3, n) & = & \text{MCD}(367 - 5, 527) = 1 \\ i = 4, \quad \text{MCD}(x_8 - x_4, n) & = & \text{MCD}(274 - 26, 527) = 31 \end{array}$$

Además, $527 = 17 \cdot 31$. Con lo que se tiene que un factor de n aparece justo cuando inicia el patrón de repetición, en x_4.

4.3.1. Para familiarizarnos...

Suponga que tiene un dado de 20 lados, con las caras numeradas del 1 al 20. Al lanzar el dado repetidamente, se obtiene una secuencia de enteros $x_0, x_1, x_2, \ldots$ en un rango $1 \leq x_i \leq 20$. Eventualmente, encontrará un valor que ya habías obtenido antes en la secuencia, es decir, se obtienen un entero k tal que:

- $x_0, x_1, x_2, \ldots, x_{k-1}$ son todos distintos, pero
- $x_k = x_j$ para algún $0 \leq j < k$.

Ahora lo que se quiere es determinar ¿qué tan largo es k, en promedio?. En otras palabras, se quiere determinar ¿cuántos lanzamientos hay antes que aparezca la primer repetición?, y esto se logra determinando la varianza para la cual los j primeros lanzamientos sean todos distintos, así en el lanzamiento $j+1$ aparecerá la primer repetición.

Sabemos que el número de muestras posibles después de j lanzamientos esta dado por la distribución 20^j.

Es claro que si $j = 1$, es decir, con un solo lanzamiento hay un 100 % de probabilidades de que los j primeros lanzamientos sean todos distintos. En este caso la probabilidad debe ser 1, lo podemos comprobar algebraicamente:

$$\frac{20}{20^1} = \frac{20!}{20^1 19!} = 1$$

Ahora se puede notar que:

1. Si se tienen dos lanzamientos en los cuales los resultados son todos distintos, entonces el número de maneras de obtener esos dos lanzamientos es:

   $$20 \cdot 19$$

   Y la probabilidad de obtener ese resultado sería:

   $$\frac{20 \cdot 19}{20^2} = \frac{20!}{20^2 18!}$$

2. Si se tienen tres lanzamientos en los cuales los resultados son todos distintos, entonces el número de maneras de obtener esos tres lanzamientos es:

   $$20 \cdot 19 \cdot 18$$

   Y la probabilidad de obtener ese resultado sería:

   $$\frac{20 \cdot 19 \cdot 18}{20^3} = \frac{20!}{20^3 17!}$$

3. Si se tienen cuatro lanzamientos en los cuales los resultados son todos distintos, entonces el número de maneras de obtener esos cuatro lanzamientos es:

   $$20 \cdot 19 \cdot 18 \cdot 17$$

Y la probabilidad de obtener ese resultado sería:

$$\frac{20 \cdot 19 \cdot 18 \cdot 17}{20^4} = \frac{20!}{20^4 16!}$$

$$\vdots$$

Recuerde que lo que se quiere es determinar cuando los lanzamientos dejan de ser todos distintos, así en el peor de los casos, si se tienen 20 lanzamientos de los cuales todos son diferentes, entonces el número de maneras de obtener esos resultados es:

$$20 \cdot 19 \cdot 18 \cdot 17 \cdot ... \cdot 1$$

Y la probabilidad estaría dada por:

$$\frac{20 \cdot 19 \cdot 18 \cdot 17 \cdot ... \cdot 1}{20^{20}} = \frac{20!}{20^{20}}$$

Así la varianza que se obtengan lanzamientos sin repetición esta dada por:

$$\frac{20!}{20^1 19!} + \frac{20!}{20^2 18!} + \frac{20!}{20^3 17!} + \frac{20!}{20^4 16!} + ... + \frac{20!}{20^{20}} =$$

$$\frac{20!}{20^1(20-1)!} + \frac{20!}{20^2(20-2)!} + \frac{20!}{20^3(20-3)!} + \frac{20!}{20^4(20-4)!} + ... + \frac{20!}{20^{20}(20-20)!} =$$

$$\sum_{j=1}^{20} \frac{20!}{20^j(20-j)!}$$

Donde j es el número de lanzamientos que se realizan.

Con lo que se tiene que el valor promedio de k es $5,29$. Lo cual indica que, en promedio, los primeros cinco lanzamientos serán todos distintos y la primera repetición aparecerá cerca del sexto lanzamiento.

Ahora, reemplace el dado de 20 lados por uno de n lados, y vuélvase a hacer la misma interrogante: ¿cuál es la varianza con la cual los j primeros lanzamientos sean todos distintos?.

En general, se tiene en promedio que:

$$k = \sum_{j=1}^{n} \frac{n!}{n^j(n-j)!}$$

4.3.2. Método de Factorización: Rho de Pollard

El método Rho de Pollard sigue un patrón semejante al ejemplo anterior en la búsqueda de los factores primos de un número compuesto dado. Trabajando sobre las clases residuales del número se puede asegurar que en algún momento se obtendrá al menos un par de números que pertenezca a la misma clase.

En general, el método Rho de Pollard usa valores de una sucesión y por medio del Máximo Común Divisor, de la resta de dos valores de la sucesión (estos valores son determinados de manera conveniente) y n, busca el menor factor primo de n.

Utiliza una sucesión de la forma:

$$x_0 = 0$$

$$x_k \equiv x_{k-1}^2 + 1 \mod n;\ n \geq 1,$$

donde n es el número que se va factorizar. Eventualmente, en algún momento los valores de x_i empezarán a repetirse.

Pollard notó que si el modulo estaba dado por un número menor que n entonces iba a haber menos congruencias, es decir, la secuencia se iba a empezar a repetir relativamente más rápido. Si p es el divisor más pequeño de n, diferente de uno, entonces p es pequeño comparado con n, suponiendo que n es un número compuesto. Es claro que habrán relativamente menos congruencias modulo p que modulo n. Es decir, si determinamos una nueva sucesión de y_i de la forma:

$$y_i \equiv x_i \mod p$$

entonces los valores de y_i se empezarán a repetir más rápido que los valores de x_i. Note que probablemente existirán enteros y_i y y_j, con $i < j$, tales que son congruentes modulo p pero no modulo n.

Ahora bien, si $p \,|\, (y_j - y_i) \wedge n \nmid (y_j - y_i)$, entonces $\text{mcd}(y_j - y_i,\, n)$ es diferente de uno, probablemente p pues $p \,|\, n$. No obstante, dicho divisor p de n aún no es conocido, esa es precisamente la misión del método.

Pollard toma la idea del método de Floyd para descubrir un ciclo a lo largo de la sucesión de y_i, de la cual solo toma los casos en los que se cumple que $y_i \equiv y_{2i} \mod p$ y aparitr de estos valores es que busca el mcd de tal forma que sea diferente de uno. Esto disminuye la cantidad de calculos y acelera el proceso.

Note que los valores de y_i pueden estar dentro de los valores de x_i pero no al contrario, es decir, no todos los valores de x_i están dentro de los valores de y_i.

4.3.3. Método rho de Pollard

Suponga que n es un número entero compuesto. Definimos la secuencia $x_0, x_1, x_2, \ldots$ de enteros x_i en el rango $0 \leq x_i < n$ recursivamente como se sigue: $x_0 = 0$, y para $k > 0$ se tiene:

$$x_{k+1} \equiv x_k^2 + 1 \mod n$$

Observe que dado $n = 20$ se tiene:

$$\begin{gathered}
x_0 = 0 \\
x_1 = 0^2 + 1 = 1 \,(\text{mod } 20) \\
x_2 = 1^2 + 1 = 2 \,(\text{mod } 20) \\
x_3 = 2^2 + 1 = 5 \,(\text{mod } 20) \\
x_4 = 5^2 + 1 = 6 \,(\text{mod } 20) \\
x_5 = 6^2 + 1 = 17 \,(\text{mod } 20) \\
x_6 = 17^2 + 1 = 10 \,(\text{mod } 20) \\
\mathbf{x_7 = 10^2 + 1 = 1 \,(mod\ 20)} \\
\mathbf{x_8 = 1^2 + 1 = 2 \,(mod\ 20)} \\
\vdots
\end{gathered}$$

Además, note que la primera repetición aparece después del término x_6.

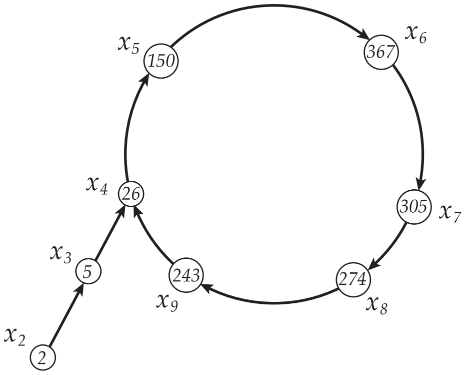
Ejemplo: Suponga que $n = 527$. Entonces, la secuencia estaría dada por:

$$\begin{gathered}
x_0 = 0\\
x_1 = 0^2 + 1 = 1 \pmod{527}\\
x_2 = 1^2 + 1 = 2 \pmod{527}\\
x_3 = 2^2 + 1 = 5 \pmod{527}\\
x_4 = 5^2 + 1 = 26 \pmod{527}\\
x_5 = 26^2 + 1 = 150 \pmod{527}\\
x_6 = 150^2 + 1 = 22501 = 367 \pmod{527}\\
x_7 = 367^2 + 1 = 134690 = 305 \pmod{527}\\
x_8 = 305^2 + 1 = 93026 = 276 \pmod{527}\\
x_9 = 276^2 + 1 = 76177 = 243 \pmod{527}\\
\mathbf{x_{10}} = \mathbf{243^2} + \mathbf{1} = \mathbf{59050} = \mathbf{26}\,(\mathbf{mod\ 527})\\
\mathbf{x_{11}} = \mathbf{26^2} + \mathbf{1} = \mathbf{150}\,(\mathbf{mod\ 527})\\
\vdots
\end{gathered}$$

Es decir, la secuencia sería:

$$0, 1, 2, 5, 26, 150, 367, 305, 274, 243, \mathbf{26}, \mathbf{150}, \mathbf{367}, \ldots$$

Eventualmente, la secuencia deberá empezar a repetirse, esto se ilustra a continuación:



Note que la longitud de la parte inicial de la secuencia (la parte antes de la primera repetición) se comporta de manera similar a una secuencia de random.

Sin embargo, algunas veces las secuencias de random resultan ser un poco extensas. Para esto se crea una nueva secuencia de y_i, al parecer un poco más corta, de la siguiente forma:

Sea p el divisor primo más pequeño de n, diferente de uno, así $p \leq \sqrt{n}$ y definimos la secuencia $y_0, y_1, y_2, \ldots$ de enteros tales que $y_i = x_i \mod p$, con $0 \leq y_i < p$. Entonces:

$$\begin{aligned}
y_{k+1} &\equiv x_{k+1} && \mod p\\
&\equiv x_k^2 + 1 && \mod p\\
&\equiv y_k^2 + 1 && \mod p
\end{aligned}$$

Con lo que se tiene que y_i satisface una recurrencia similar a la de x_i, así $y_0 = 0$ y para $k > 0$ tenemos $y_{k+1} \equiv y_k^2 + 1 \mod p$.

Retomando el ejemplo anterior, tenemos que el factor primo más pequeño de 527 es 17. Así, la secuencia y_i estaría dada por:

$$\begin{gathered}
y_0 = 0 \\
y_1 = 0^2 + 1 = 1 \,(\text{mod } 17) \\
y_2 = 1^2 + 1 = 2 \,(\text{mod } 17) \\
y_3 = 2^2 + 1 = 5 \,(\text{mod } 17) \\
y_4 = 5^2 + 1 = 9 \,(\text{mod } 17) \\
y_5 = 9^2 + 1 = 82 = 14 \,(\text{mod } 17) \\
y_6 = 10^2 + 1 = 101 = 16 \,(\text{mod } 17) \\
\mathbf{y_7 = 16^2 + 1 = 257 = 2 \,(mod\ 17)} \\
\mathbf{y_8 = 2^2 + 1 = 5 \,(mod\ 17)} \\
\vdots
\end{gathered}$$

Es decir, la secuencia sería:

$$0, 1, 2, 5, 9, 14, 10, 16, \mathbf{2}, \mathbf{5}, \mathbf{9}, \mathbf{14}, \mathbf{10}, \ldots$$

La validez del proceso no esta en conocer cual es el valor p, si no en poder detectar cuando la secuencia de y_i empieza a repetirse.

Pero el proceso no termina ahí. Para poder mostrar la efectividad y validez del proceso se necesitan algunos resultados importantes que se enunciarán a continuación:

I. Como $y_i \equiv x_i \mod p$ entonces, para algunos índices i y k, se tiene que:

$$\begin{aligned}
y_i \equiv y_k \mod p \quad &\Rightarrow \quad x_i \equiv x_k \mod p \\
&\Rightarrow \quad p \,|\, (x_k - x_i)
\end{aligned}$$

Como p divide a n, entonces tenemos que:

$$p \,|\, \text{mcd}\ (x_k - x_i, n)$$

En particular, se tiene que:

$$x_i \equiv x_k \mod p \Rightarrow \text{mcd}\ (x_k - x_i, n) \neq 1$$

Justificación:

- Sabemos que $\text{mcd}(a, b) = 1$ si y solo si $a \wedge b$ son primos relativos o si $a = 1 \vee b = 1$.
- Como $x_i \equiv x_k \mod p$ entonces $x_i \wedge x_k$ no pueden ser consecutivos, con lo que se tiene que $x_k - x_i \neq 1$. Además, recordemos que n es un número compuesto, por lo tanto, $n \neq 1$.
- Recordemos que p es el menor factor primo de n, con lo que se tiene que $p \,|n$, además, tenemos que $p \,|\, (x_k - x_i)$.

- Con lo que se tiene que $n \neq 1 \wedge (x_k - x_i) \neq 1$ y que $n \wedge x_k - x_i$ no son primos relativos.
- Por lo tanto, $\text{mcd}\ (x_k - x_i, n) \neq 1$.

II. Siempre que $x_k \neq x_j$ el mcd $(x_k - x_j, n)$ debe ser un factor de n diferente de uno y n.
Sin embargo, lo anterior necesita una idea más para hacer que el método Rho de Pollard sea práctico.
Suponga que desea factorizar n. Entonces:

- Calcule la secuencia $x_0, x_1, \ldots$ como se hizo anteriormente.
- Para x_k calcule $g = \text{mcd}\ (x_k - x_i, n)$ para todo $0 \leq i < k$. Eventualmente, usted deberá buscar valores de i y k para los cuales $g \neq 1$, con lo que se tendría que $\text{mcd}\ (x_k - x_i, n)$ es un factor de n diferente de uno.

En el ejemplo anterior, $n = 527$, tenemos que el $\text{mcd}\ (x_5 - x_4, 527) = 31$. Por lo tanto, obtenemos una factorización de 527: $17 \cdot 31$.

III. Suponga que:

- $y_i \equiv y_k \mod p$, con $0 \leq i < k$.
- m el múltiplo positivo más pequeño de $k - i$ para cualquier $m \geq i$.

Entonces:

- $m \leq k$ y $y_m \equiv y_{2m} \mod p$.
- $\text{mcd}\ (x_{2m} - x_m, n) \neq 1$ es un factor de n diferente de uno..

Demostración:

Sabemos que el múltiplo más pequeño de un número es él mismo, con lo que se tiene que el múltiplo más pequeño de $k - i$ es $k - i$. Así

$$m = k - i \Rightarrow -i = m - k$$

Además, note que:

$$\begin{aligned} m \equiv m \mod (k-i) &\Rightarrow (m-k) \mod (k-i) = -k \\ &\Rightarrow -i \mod (k-i) = -k \end{aligned}$$

1. Si $i > 0$ entonces:

$$\begin{aligned} & m = k - i \\ \Rightarrow \quad & m = |i - k| \\ \Rightarrow \quad & m = |i + (-i \mod (k-i))| \\ < \quad & i + (k - i) = k \end{aligned}$$

2. Si $i = 0$ entonces:

$$m = k - i = k$$

$$\therefore\ m \leq k \text{ en cualquier caso}$$

Falta mostrar que $y_m \equiv y_{2m} \mod p$, como $m = k-i$ entonces sería lo mismo mostrar que $y_m \equiv y_{m+(k-i)}$ mod p, para esto usaremos inducción sobre m.

1. Para $m = 0$ en claro que $y_0 \equiv y_{0+0} \mod p$.

2. Asumamos validez para m, es decir, $y_m \equiv y_{m+(k+i)} \mod p$.

3. Debemos mostrar validez para $m+1$, es decir $y_{m+1} \equiv y_{m+1+(k-i)} \mod p$:

Recordemos que:

$$y_{m+1} \equiv y_m^2 + 1 \mod p$$

Con lo que se tiene que:

$$\begin{aligned} y_{m+1} \equiv y_m^2 + 1 \mod p \quad &\Rightarrow \quad y_{m+1} \equiv y_{m+(k-i)}^2 + 1 \mod p \text{ (Hipótesis de Inducción)} \\ &\Rightarrow \quad y_{m+1} \equiv y_{m+(k-i)}^2 + 1 \mod p \\ &\Rightarrow \quad y_{m+1} \equiv y_{m+(k-i)+1} \mod p \\ &\Rightarrow \quad y_{m+1} \equiv y_{m+1+(k-i)} \mod p \end{aligned}$$

$$\therefore y_m \equiv y_{m+(k-i)} \mod p$$

Es decir:

$$y_m \equiv y_{2m} \mod p$$

Ahora bien, como $x_i \equiv y_i$ mod p entonces se cumple que $x_m \equiv x_{2m}$ mod p. Con lo que se tiene que:

$$p\,|(x_{2m} - x_m) \Rightarrow x_{2m} - x_m = sp;\ s \in \mathbb{Z}$$

Recordemos que p es el divisor primo más pequeño de n, diferente de uno, con lo que podemos asegurar que $\text{mcd}(x_{2m} - x_m, n) \neq 1$ es un factor de n diferente de uno.

4.3.4. El algoritmo Rho de Pollard

Supongamos que n es un entero compuesto que a usted le gustaría factorizar. Sabiendo que $x_0 = 0$. Ahora, para cada $k \geq 1$:

1. Calculemos x_k por medio de la fórmula:

$$x_k \equiv (x_{k-1}^2 + 1) \mod n$$

2. Calculemos z_{2k} utilizando la siguiente iteración:

$$\begin{aligned} z_{2k} &\equiv x_{2k} \mod n \\ &\equiv x_{2k-1}^2 + 1 \mod n \\ &\equiv (x_{2k-2}^2 + 1 \mod n)^2 + 1 \mod n \end{aligned}$$

3. Calcule $g = \text{ mcd } (z_{2k} - x_k, n)$. Si $g \neq 1$ entonces ya encontramos un factor de n diferente de uno: g. De lo contrario, continuamos buscando con el siguiente k.

Cuando al algoritmo termina, en el paso anterior (3), g será un factor de n diferente de uno.

Continuando con el ejemplo anterior, $n = 527$, obtenemos los siguientes resultados:

k	x_k	z_{2k}	$\text{mcd}(z_{2k}-x_k, 527)$
0	0	0	527
1	1	2	1
2	2	26	1
3	5	367	1
4	26	274	31

Así, el método Rho de Pollard ha descubierto a 31 como factor de n diferente de n.

Algoritmo 4.3.1: *Rho*(n)

```
Data: $\mathbf{n} \in \mathbb{N}$
Result: $Rho(\mathbf{n})$
$\mathbf{x} \leftarrow 2, \mathbf{z} \leftarrow 2, \mathbf{g} \leftarrow 1$;
while $\mathbf{g} = 1$ do
    $\mathbf{x} \leftarrow (\mathbf{x}^2 + 1)$ Mod $\mathbf{n}$;
    $\mathbf{z} \leftarrow \left[\left[(\mathbf{z}^2 + 1) \text{ Mod } \mathbf{n}\right]^2 + 1\right]$ Mod $\mathbf{n}$;
    $\mathbf{g} \leftarrow$ Mcd $(|\mathbf{x} - \mathbf{z}|, \mathbf{n})$;
if $\mathbf{g} = \mathbf{n}$ then
    Error
return $\mathbf{g}$;
```

4.3.5. ¿Cuándo falla el método Rho de Pollard?

Es claro que el algoritmo anterior debe terminar eventualmente. Pero es posible que sí $z_{2k} = x_k$ pues la primera vez el mcd será diferente de uno (será n), entonces el factor retornado es simplemente n.

Cuando esto ocurre, uno puede volver a iniciar el método Pollard Rho:

1. Usando otro valor inicial. Por ejemplo: $x_0 = 2$ en lugar de $x_0 = 0$.
2. También podemos usar una iteración diferente. Por ejemplo: $x \mapsto x^2 + 2$ ó $x \mapsto x^3 + 1$ en lugar de $x \mapsto x^2 + 1$.

Usualmente se utiliza la iteración $x \mapsto x^2 + c$, para un c elegido al azar en el rango $0 \leq c \leq n$.

Ejemplo: Cuando $n = 1241$, obtenemos la siguiente secuencia de x_i:

$$0, 1, 2, 5, 26, 677, 401, 713, 801, \mathbf{5}, \mathbf{26}, \mathbf{677}, \mathbf{401}, ...$$

Ahora, observe los resultados en la siguiente tabla:

k	x_k	z_{2k}	$\text{mcd}(z_{2k}-x_k, 1241)$
0	0	0	1241
1	1	2	1
2	2	26	1
3	5	401	1
4	26	801	1
5	677	26	1
6	401	401	1241

Así, Rho de Pollard falla. Pues nos estaría diciendo que el único factor de 1241 diferente de uno es 1241, es decir, que 1241 es un número primo lo cual no es verdadero.

Pero, veamos que sucede si corremos el Rho de Pollard con la iteración:

$$x_{k+1} = (x_k^2 + 2) \bmod n$$

La secuencia de x_i sería:

$$0, 2, 6, 38, 205, 1074, 589, ...$$

Y tendríamos la siguiente tabla:

k	x_k	z_{2k}	$\mathrm{mcd}(z_{2k} - x_k, 1241)$
0	0	0	1241
1	1	6	1
2	6	205	1
3	38	589	1
4	205	1	17

Rápidamente el método encuentra que 17 es un factor de 1241.

4.3.6. Problema del método Rho de Pollard

Cuando k es grande, hacer los cálculos $\mathrm{MCD}(x_k - x_j, n)$, $j < k$; tiene un costo alto, ya que, como el cálculo del MCD requiere $O(\log_2 n)^3$ operaciones de bits. Por ejemplo, si $n = 10^{1000}$ (mil dígitos), el cálculo de *cada* $\mathrm{MCD}(x_k - x_j, n)$ requiere aproximadamente de $3{,}5 \times 10^{10}$ operaciones de bits. La acumulación de tantas operaciones hace que la búsqueda empiece a tardar una cantidad de tiempo nada razonable.

4.3.7. Variante de Richard Brent.

Cuando k es grande, hacer los cálculos $\mathrm{MCD}(x_k - x_j, n)$, $j < k$; tiene un costo alto, ya que, como el cálculo del MCD requiere $O(\log_2 n)^3$ operaciones de bits. Por ejemplo, si $n = 10^{1000}$ (mil dígitos), el cálculo de *cada* $\mathrm{MCD}(x_k - x_j, n)$ requiere aproximadamente de $3{,}5 \times 10^{10}$ operaciones de bits. La acumulación de tantas operaciones hace que la búsqueda empiece a tardar una cantidad de tiempo nada razonable.

En 1980, Richard Brent publicó una variante más rápida del algoritmo de Rho. Él utilizó las mismas ideas que Pollard, pero él utilizó un método diferente de detección del ciclo que era más rápido que el algoritmo original de John Pollard.

La idea es hacer un solo cálculo del MCD para cada k.

Lo que hay que observar es que, una vez que hay un x_{k_0} y un x_{j_0} tales que $x_{k_0} \equiv x_{j_0} \pmod{p}$ para algún divisor p de n, entonces $x_k \equiv x_j \pmod{p}$ para k, j tales que $k - j = k_0 - j_0$.

Para ver esto, se considera $k = k_0 + m$, $j = j_0 + m$.

Bien, la variante de Brent calcula sucesivamente x_k, y para cada k procede como sigue:

Supongamos que $2^h \leq k \leq 2^{h+1}$, es decir k tiene $(h+1)$ bits. Sea j el más grande entero de h bits, $j = 2^h - 1$. Comparamos x_k con este x_j particular, es decir, calculamos $\mathrm{MCD}(x_k - x_j, n)$. Si tenemos éxito paramos ,y si no, nos movemos al siguiente $k + 1$.

4.3.8. ¿Por qué funciona?

Supongamos que estamos analizando el caso k y que ya pasó $x_{k_0} \equiv x_{j_0} \,(\text{mód}\, r)$ con $r|n,$ para algún $k_0 < k.$ Ahora, $\text{MCD}(x_k - x_j, n) > 1$ sucede si $x_k \equiv x_j \,(\text{mód}\, r),$ es decir $k - j = k_0 - j_0.$ ¿Cómo debería ser j (el único hipotéticamente desconocido, aunque no conocemos k_0 ni j_0)?. Pongamos $k = j + (k_0 - j_0)$ y supongamos que k_0 tiene $(h+1)$ bits. Brent toma $j = 2^{h+1} - 1.$ Entonces j es el más grande entero de $h+1$ bits (ver la sección final del este capítulo) y k tendría $h+2$ bits. La regla *limpia* es: si k tiene $(h+1)$ bits, sea $j = 2^h - 1,$ compare x_k con este particular $x_j.$ Notemos además que $k < 2^{h+2} = 4 \cdot 2^h \leq 4k_0.$

La ventaja del método de Variante de R. Brent es que solo calculamos un MCD para cada $k.$ La desventaja es que no vamos a detectar la primera vez que haya un par $x_{k_0},\ x_{j_0}$ con $j_0 < k_0$; tal que $\text{MCD}(x_{k_0} - x_{j_0}, n) > 1,$ sino que lo detectaríamos un poco más tarde.

Ejemplo: Consideremos de nuevo el problema de factorizar $n = 1387.$ Usamos $f(n) = n^2+1$ y digamos que $x_0 = 1194.$ Usando la variante de Brent tenemos,

$$\begin{aligned}
i = 1, \quad \text{MCD}(x_1 - x_0, n) &= \text{MCD}(1188 - 1194, n) = 1\\
i = 2, \quad \text{MCD}(x_2 - x_0, n) &= \text{MCD}(766 - 1194, n) = 1\\
i = 3, \quad \text{MCD}(x_3 - x_2, n) &= \text{MCD}(56 - 766, n) = 1\\
&\vdots \quad \vdots\\
i = 12, \quad \text{MCD}(x_{12} - x_6, n) &= \text{MCD}(829 - 26, n) = 73, \text{éxito!}
\end{aligned}$$

Con el método sin modificar necesitaríamos 16 cálculos, pero ahora sólo se necesitan $12.$ En general, la variante de Brent es aproximadamente 24 % más rápida.

4.3.9. Algoritmo del método Rho de Brent

```
Algoritmo 4.3.2: $VarianteRho(\mathbf{n})$
  Data: $\mathbf{n} \in \mathbb{N}$
  Result: $VarianteRho(\mathbf{n})$
  $\mathbf{x} \leftarrow 0$, $\mathbf{v} \leftarrow [\mathbf{x}]$, $\mathbf{i} \leftarrow 1$, $\mathbf{k} \leftarrow 1$, $\mathbf{g} \leftarrow 1$
  for $\mathbf{i} \leq 100$ do
      $\mathbf{x} \leftarrow (\mathbf{x}^2 + 1)$ Mod n
      AppendTo(v, x)
  for $\mathbf{k} \leq$ Len(v) do
      $\mathbf{h} \leftarrow$ Len(CambiarBase(k, 2))
      $\mathbf{h} \leftarrow 2^{\mathbf{h}-1}$ - 1
      $\mathbf{g} \leftarrow$ Mcd(v[k ] - v[h ], n)
      if $\mathbf{g} \neq 1$ then
          $\mathbf{k} \leftarrow$ Len(v)
  if $\mathbf{g} = \mathbf{n}$ then
      Error
  return g;
```

4.4. Tiempo de ejecución términos de operaciones bits.

4.4.1. Número de dígitos.

La representación en base $b = 2$ de un entero n es $(d_{k-1}d_{k-2}\cdots d_0)_2$ donde

$$n = d_{k-1}2^{k-1} + d_{k-2}2^{k-2} + \cdots + d_0\, 2^0,\ \ d_i \in \{0,1\}$$

Ejemplo

$2^{10} = 1024 = (10000000000)_2$

☐

Si $2^{k-1} \leq n \leq 2^k$ entonces n tiene k dígitos en base $b = 2$.

Ejemplo

$2^{10} \leq 2^{10} \leq 2^{11}$, así $n = 2^{10}$ tiene 11 dígitos en base 2.

$2^7 = 128 \leq 201 \leq 256 = 2^8$, así $n = 201$ tiene 8 dígitos en base 2. En efecto, $201 = (11001001)_2$

☐

• El número k de dígitos, en base $b = 2$, de un número $n \neq 0$ se puede calcular con la fórmula

$$k = [\log_2 |n|] + 1 = \left[\frac{\ln|n|}{\ln(2)}\right] + 1$$

Ejemplo

Si $n = 2^{10}$ entonces $k = \left[\log_2(2^{10})\right] + 1 = 10 + 1 = 11$

Si $n = 201$ entonces $k = [\log_2(201)] + 1 = [7.65105...] + 1 = 8$

☐

• Se acostumbra usar "$\lg n$" en vez de $\log_2(n)$.

4.4.2. Operaciones de Bits ("bit operations")

"bit" es una abreviación de "dígito binario". Sumar dos bits en una suma que involucra dos números de k bits, requiere de algunas reglas. Tal y como es usual en la escuela primaria, cada vez que sumamos dos bits en la misma posición, debemos ver si “llevamos algo”.

- Una operación de bit es sumar dos bits de acuerdo con estas reglas de la suma.

Ejemplo

Sumar $10 = (1010)_2$ y $2 = (0010)_2$ *requiere cuatro operaciones de bits*. En efecto, sumamos como usual, bit a bit, de derecha a izquierda.

Paso 1.) Iniciamos con $0 + 0 = 0$,

Paso 2.) $1 + 1 = 0$ y “llevamos” 1,

Paso 3.) $0 + 0 = 0$ más el uno que llevamos, da 1;

Paso 4.) finalmente, $1 + 0 = 1$.

$$\begin{array}{cccccc} & 1 & & & & \longleftarrow \text{“llevamos”} \\ 1 & 0 & 1 & 0 & & \\ 0 & 0 & 1 & 0 & & \\ \hline 1 & 1 & 0 & 0 & & \end{array}$$

$\square$

4.4.3. Estimación de la complejidad de un algoritmo.

En teoría computacional de números, la complejidad de un algoritmo no se mide en términos del número de operaciones aritméticas que se deben ejecutar pues, por ejemplo, no es lo mismo sumar dos números de un dígito que dos números de varios cientos de dígitos. La complejidad se mide en términos del número de operaciones de bit para ejecutar el algoritmo. Para poder describir adecuadamente la complejidad, usamos la notación “O” (conocida como “$O-$grande”).

Definición.
Sean f y g funciones de variable real, positivas. Entonces, escribimos $f(n) = O(g(n))$ si existe una constante $c \in \mathbb{R}^+$ tal que $f(n) \leq c\, g(n)$ para n suficientemente grande.

En la definición anterior, g es una función que juega el papel de punto de comparación como un objeto más familiar y sencillo que *acota superiormente* a f.

Ejemplo

- Si $f(n) = n^3 + 4$ entonces $f(n) = O(n^3)$ pues eventualmente $n^3 + 4 \leq c\,n^3$ con $c = 2$

- Si $f(n) = n^5 + 4n + \ln(n) + 5$ entonces $f(n) = O(n^5)$.

  Para ver esto, solo es necesario observar que n^5 es un función creciente en $\mathbb{R}^+$ y por tanto *domina* a los otros sumandos para n suficientemente grande. En particular, podemos probar que $n^5 \geq \ln(n)$ si $n \geq 1$: esto es lo mismo que decir que $n^5 - \ln(n) \geq 0$ si $n \geq 1$. Sea $h(n) = n^5 - \ln(n)$. Entonces $h'(n) = 5n^4 - 1/n \geq 0$ si $n \geq 1$. Así h es creciente en $[1, \infty[$. Luego $h(n) \geq h(1) \geq 0$ si $n \geq 1$.

- En términos de "$O-$grande", el número de bits de n es $[\lg(n)] + 1 = O(\lg n)$. Se acostumbra decir "el número de bits de n es $O(\lg n)$".

- La suma, con el método usual, de dos números de k bits requiere $O(k) = O(\lg(n))$ operaciones de bit (si $k = [\lg(n)] + 1$).
- La multiplicación de dos números de k bits, usando el método que aprendimos en primaria para multiplicar, requiere $O(k^2) = O(k)^2 = O(\lg n)^2$ operaciones de bits.

  Para números con varios cientos de dígitos, la multiplicación es más eficiente con la transformada rápida de Fourier, en este caso se requiere $O(\lg(n) \lg\lg(n) \lg\lg\lg(n))$ operaciones de bit.

□

4.4.4. Complejidad polinomial.

Definición.
Un algoritmo que recibe como entrada los enteros $n_1, n_2, ..., n_k$, se dice que tiene complejidad polinomial (o que "corre" en tiempo polinomial), *en términos de operaciones de bit,* si su tiempo de corrida es $O(P(\lg n_1 + \lg n_2 + ... + \lg n_k))$ donde P es un polinomio.

Un tiempo polinomial indica que el algoritmo corre en tiempo manejable.

Ejemplo

- Si un algoritmo que recibe un entero n, tiene complejidad (polinomial) $O(n)$ *en términos del número de operaciones aritméticas,* en términos de operaciones de bit, el algoritmo tiene complejidad

$$\begin{aligned} O(n) &= O(n(\lg n)^2) \\ &= O\left(2^{\lg n}(\lg n)^2\right) \end{aligned}$$

si asumimos que, en el peor caso, cada operación aritmética necesita $O(\lg n)^2$ operaciones de bit. Es decir, un algoritmo con complejidad polinomial en términos del número de operaciones aritméticas tiene complejidad exponencial en términos de operaciones de bit (es decir respecto a la unidad de medida $\lg n$).

- El algoritmo de Euclides, para calcular $\mathrm{MCD}(a,b)$ con $a<b$, requiere $O(\lg a)$ operaciones aritméticas (pues un teorema de Lamé (1844) establece que el número de divisiones necesarias para calcular el $\mathrm{MCD}(a,b)$ es a lo sumo cinco veces el número de dígitos decimales de a, es decir $O(\log_{10} a)$). Esto corresponde a $O(\lg\, a)^3$ en términos de operaciones de bits (asumiendo como antes que divisiones y multiplicaciones necesitan $O(\lg n)^2$ operaciones de bit).

□

Si consideramos el problema de ordenar una lista de enteros, la complejidad medida en términos de operaciones aritméticas refleja correctamente el tiempo de corrida (depende de la cantidad de enteros por ordenar). La mayoría de problemas en teoría computacional de números requiere medir la complejidad en términos de operaciones de bits, es decir depende no tanto del número de enteros sino más bien del tamaño, como por ejemplo factorizar un entero grande.

Bibliografía

[1] G.H. Hardy, J.E. Littlewood. *An Introduction to Theory of Numbers.* Oxford Univ. Press. 1938.

[2] Harold M. Edwards. *Riemann's Zeta Function.* Dover Publications Inc. 2001.

[3] P.L. Chebyshev."The Theory of Probability". Translated by Oscar Sheynin (www.sheynin.de) 2004. Versión en internet: `http://www.sheynin.de/download/4_Chebyschev.pdf`. Consultada Diciembre 16, 2006.

[4] Hausmann, B. "A new simplification of Kronecker's method of factorization of polynomials". American Mathematical Monthly 47 (1937).

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

[6] Hans Riesel. *Prime Numbers and Computer Methods for Factorization.* Springer; 2 edition. 1994.

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

[8] RSA, Inc. `http://www.rsasecurity.com/rsalabs/node.asp?id=2093`. Consultada Noviembre 11, 2006.

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

[10] ArjenK. Lenstra." Integer Factoring". `http://modular.fas.harvard.edu/edu/Fall2001/124/misc/arjen_lenstra_factoring.pdf` Consultada: Octubre,2006.

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

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

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

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

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

[16] Conte,S. de Boor, C. *Análisis Numérico Elemental.* 2da. Edición. McGraw-Hill, 1985.

[17] Gallian, J. *Contemporary Abstract Algebra.* Houghton Mifflin Co., 2006.

Capítulo 5

Cantidad de Primos Menor que un Número Dado.

Manuel Alfaro A.
Escuela de Matemática
Instituto Tecnológico de Costa Rica.

Dado un número real x , el problema que estudiaremos consiste en calcular: $\pi(x)$, que representa la cardinalidad del conjunto $\{\ p : primo \leq x\}$, primos menores o iguales a x. Para llevar a cabo esta labor existen varias fórmulas, dentro de éstas algunas son algo complejas. En el presente documento vamos a revisar algunas de estas fórmulas como la fórmula de Legendre, la de Meissel y la de Lehmer. Iniciamos con la fórmula de Legendre que es la más simple de todas, pero desafortunadamente, es la que consume mayor labor de cálculo.

5.0.5. Fórmula de Legendre

Esta fórmula es muy sencilla pero poco eficiente cuando se trata de valores grandes para x , y se deriva de la siguiente idea :

$$1+ \text{ los números primos en } [1,x] = \text{ los números enteros en } [1,x] - \text{ los números compuestos en } [1,x]$$

que proviene del principio de inclusióny exclusión,y se puede expresar en una forma más precisa del siguiente modo:

$$1+\pi(x)=\pi\left(\sqrt{x}\right)+\lfloor x\rfloor-\sum_{p_i\leq\sqrt{x}}\left\lfloor\frac{x}{p_i}\right\rfloor+\sum_{p_i<p_j\leq\sqrt{x}}\left\lfloor\frac{x}{p_ip_j}\right\rfloor-\sum_{p_i<p_j<p_k\leq\sqrt{x}}\left\lfloor\frac{x}{p_ip_jp_k}\right\rfloor+\cdots \tag{5.0.1}$$

Donde $\lfloor x\rfloor$ denota la parte entera del número real x y además, nótese que la expresión $\left\lfloor\frac{x}{p}\right\rfloor$ enumera todos los enteros divisibles por p que pertenecen al intervalo $[1,x]$, pues $\left\lfloor\frac{x}{p}\right\rfloor = n$ significa que $np\leq x<(n+1)\,p$ y por tanto hay exactamente n múltiplos positivos de p que son menores o iguales a x.

Ahora, dado que todo número compuesto en el intervalo $[1,x]$ tiene al menos un factor primo menor o igualque $\sqrt{x}$, hay un total de $\displaystyle\sum_{p_i\leq\sqrt{x}}\left\lfloor\frac{x}{p_i}\right\rfloor$ múltiplos de primos p con $p\leq\sqrt{x}$. Pero, esta suma considera los mismos primos, $p_i=1\cdot p_i$, como compuestos y por tanto los elimina, lo que justifica agregar el término $\pi\left(\sqrt{x}\right)$ en la fórmula (**??**) , para así reponerlos.

Por otro lado, como existen algunos números compuestos en $[1,x]$ que son múltiplos a la vez de dos primos distintos $p_i,p_j\ \leq\ \sqrt{x}$, y en consecuencia son considerados dos veces en $\displaystyle\sum_{p_i\leq\sqrt{x}}\left\lfloor\frac{x}{p_i}\right\rfloor$, por lo tanto debemos agregar el término siguiente: $\displaystyle\sum_{p_i<p_j\leq\sqrt{x}}\left\lfloor\frac{x}{p_ip_j}\right\rfloor$ en la fórmula (**??**).

Igualmente en esta nueva suma los enteros divisibles por tres diferentes primos $p_i<p_j<p_k\leq\sqrt{x}$ ya no serían quitados del todo y por esta razón se deben eliminar por medio del término $\displaystyle\sum_{p_i<p_j<p_k\leq\sqrt{x}}\left\lfloor\frac{x}{p_ip_jp_k}\right\rfloor$ y así sucesivamente.

Para entender mejor esta fórmula desarrollemos un ejemplo muy sencillo:

$$\begin{aligned}
\pi\left(100\right) &= \pi\left(10\right)+\lfloor 100\rfloor-\left\lfloor\tfrac{100}{2}\right\rfloor-\left\lfloor\tfrac{100}{3}\right\rfloor-\left\lfloor\tfrac{100}{5}\right\rfloor-\left\lfloor\tfrac{100}{7}\right\rfloor+\\
&+\left\lfloor\tfrac{100}{2\cdot3}\right\rfloor+\left\lfloor\tfrac{100}{2\cdot5}\right\rfloor+\left\lfloor\tfrac{100}{2\cdot7}\right\rfloor+\ \left\lfloor\tfrac{100}{3\cdot5}\right\rfloor+\left\lfloor\tfrac{100}{3\cdot7}\right\rfloor+\left\lfloor\tfrac{100}{5\cdot7}\right\rfloor+\\
&-\left\lfloor\tfrac{100}{2\cdot3\cdot5}\right\rfloor-\left\lfloor\tfrac{100}{2\cdot3\cdot7}\right\rfloor-\left\lfloor\tfrac{100}{2\cdot5\cdot7}\right\rfloor-\left\lfloor\tfrac{100}{3\cdot5\cdot7}\right\rfloor+\left\lfloor\tfrac{100}{2\cdot3\cdot5\cdot7}\right\rfloor-1\\
&= 4+100-50-33-20-14+16+10+7+6+4+2-3-2-1-0+0-1\\
&= 25
\end{aligned}$$

Así que $\pi\left(100\right)=25$. Desafortunadamente la fórmula de Legendre no es práctica para calcular $\pi\left(x\right)$ para valores muy grandes de x . El siguiente progreso importante en el cálculo de $\pi\left(x\right)$ fue hecho por Meissel, quien le hizo una modificación eficiente a la fórmula de Legendre y halló los valores de $\pi\left(10^7\right)$, $\pi\left(10^8\right)$ y $\pi\left(10^9\right)$. Esta fórmula la presentamos a continuación.

5.0.6. Fórmula de Meissel

La fórmula se basa en el análisis detallado de la expresión $P_l(x,a)$ en el intervalo $[1,x]$, que representa la cantidad de enteros que pertenecen a $[1,x]$ que pueden escribirse como producto de l factores primos todos mayores que p_a , donde estos puede que no sean necesariamente distintos. De este modo si consideramos los primeros a primos: $p_1=2$, $p_2=3,\ldots,p_a$, entonces tenemos para los distintos valores de l las expresiones siguientes:

1. $P_1(x,a)=\pi\left(x\right)-a$ y cuenta los primos p tales que $p_a<p\leq x$, es decir los primos mayores a p_a
2. $P_2(x,a)$ representa los enteros n que se pueden escribirse como producto de dos primos mayores a p_a , osea $n=p_ip_j\ \leq\ x$, donde $a+1\leq i\leq j$.
3. Análogamente $P_3(x,a)$ representa los enteros $n=p_ip_j\ p_k\leq\ x$, con $a+1\leq i\leq j\leq k$

4. y así sucesivamente.

Utilizando estas nuevas expresiones podemos expresar todos los enteros en $[1,x]$, de la siguiente manera:

$$1+\overbrace{\sum_{1\leq i\leq a}\lfloor\tfrac{x}{p_i}\rfloor-\sum_{1\leq i<j\leq a}\lfloor\tfrac{x}{p_ip_j}\rfloor+\sum_{1\leq i<j<k\leq a}\lfloor\tfrac{x}{p_ip_jp_k}\rfloor-\cdots}^{\{p_1,\,p_2,...,\,p_a\}+\text{ números con al menos un factor primo }\ p\,\leq\,p_a}+\overbrace{\pi\,(x)-a}^{P_1(x,a)}+P_2(x,a)+\ P_3(x,a)+\cdots=\lfloor x\rfloor \quad (5.0.2)$$

Donde el número de términos $P_l(x,a)$ que se deben considerar depende del valor de a que se elija, por ejemplo si escogemos a de modo que $p_{a+1}>\sqrt{x}\ \wedge\ p_a\leq\sqrt{x}$, entonces se tendría que $P_l(x,a)=0$, para toda $l\geq 2$, dado que $p_ip_j\cdots>\sqrt{x}\sqrt{x}\cdots\geq x$; y en este caso la fórmula anterior coincide con la fórmula original de Legendre.

Por otro lado si a es escogido de modo que cumpla la condición $\sqrt{[}3]x<p_{a+1}\leq\sqrt{x}$, entonces se tiene que $P_2(x,a)\neq 0$, dado que al menos $p_{a+1}^2\leq\ x$ y también se tiene que $P_l(x,a)=0$, para toda $l\geq 3$. Reformulando, el cálculo de $P_2(x,a)$ se tendría que:

$P_2(x,a)=$ enteros de la forma $p_{a+1}p_j\ \leq\ x$, con $a+1\leq j$

$+$ enteros de la forma $p_{a+2}p_j\ \leq\ x$, con $a+2\leq j$

$+\ \cdots$

Y por lo tanto podemos reescribir el término $P_2(x,a)$ del siguiente modo:

$$P_2(x,a)=\left[\pi\left(\tfrac{x}{p_{a+1}}\right)-a\right]+\left[\pi\left(\tfrac{x}{p_{a+2}}\right)-(a+1)\right]+\cdots$$
$$=\sum\left\{\pi\left(\tfrac{x}{p_i}\right)-(i-1)\right\},\ \text{para}\ \ p_a<p_i\leq\sqrt{x}\ \ (\text{si}\ \ p_i>\sqrt{x}\ ,\ \text{entonces}\ \ p_ip_j\ \not\leq\ x)$$

Ahora, supongamos que escogemos $a<\pi\left(\sqrt{x}\right)$ y el valor $b=\pi\left(\sqrt{x}\right)$, es decir b cumple que $p_{b+1}>\sqrt{x}\ \wedge\ p_b\leq\sqrt{x}$, entonces tenemos el siguiente resultado:

$$P_2(x,a)=\sum_{i=a+1}^{b}\left\{\pi\left(\frac{x}{p_i}\right)-(i-1)\right\}=-\frac{(b-a)\,(b+a-1)}{2}+\sum_{i=a+1}^{b}\pi\left(\frac{x}{p_i}\right) \quad (5.0.3)$$

Además, si escogemos $a=\pi\left(\sqrt{[}3]x\right)=c$, esto con el propósito de hacer $P_l(x,a)=0$, para toda $l\geq 3$, y sustituimos el resultado anterior en la igualdad (**??**), para luego de despejar $\pi\,(x)$ y con esto obtener la fórmula de Meissel :

$$\pi\,(x)\ =\lfloor x\rfloor-\sum_{i=1}^{c}\lfloor\frac{x}{p_i}\rfloor+\sum_{1\leq i<j\leq c}\lfloor\frac{x}{p_ip_j}\rfloor-\cdots+\frac{(b+c-2)\,(b-c+1)}{2}-\sum_{i=c+1}^{b}\pi\left(\frac{x}{p_i}\right) \quad (5.0.4)$$

Para poner en práctica esta fórmula consideremos el siguiente ejemplo:

Sea $x=64$, entonces a debe satisfacer $\sqrt{[}3]64<p_{a+1}\leq\sqrt{64}$, es decir $4<p_{a+1}\leq 8$, y por ende $a=2$ y así mismo $c=2$. Luego, $b=\pi\left(\sqrt{64}\right)$, así que $b=4$ y posteriormente aplicamos la fórmula de Meissel:

$$\pi\,(64)=\lfloor 64\rfloor-\sum_{i=1}^{2}\lfloor\tfrac{64}{p_i}\rfloor+\sum_{1\leq i<j\leq 2}\lfloor\tfrac{64}{p_ip_j}\rfloor-\cdots+\tfrac{(4+2-2)(4-2+1)}{2}-\sum_{i=2+1}^{4}\pi\left(\tfrac{64}{p_i}\right)$$
$$=64-(32+21)+10+6-(5+4)=80-62=18$$

Por tanto $\pi\,(64)=18$, que es correcto pues:

hay $\{2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,,59,61\}$, 18 primos ≤ 64

5.0.7. Fórmula de Lehmer

En la deducción de la fórmula de Meissel se usó la igualdad (**??**) en la que se reescribió la expresión $P_2(x,a)$, única que queda bajo condiciones apropiadas y finalmente se despejó el término $\pi\,(x)$. Ahora, si llevamos a un paso más esta misma igualdad, imponiendo nuevas condiciones, llegamos a la fórmula de Lehmer. Para obtenerla debemos analizar el siguiente término, es decir $P_3(x,a)$ que puede describirse, de manera similar a como se hizo con $P_2(x,a)$, como sigue:

$$\begin{aligned} P_3(x,a) = & \text{ enteros de la forma } p_{a+1}p_jp_k \leq x \text{ ,con } a+1 \leq j \leq k \\ & + \text{enteros de la forma } p_{a+2}p_jp_k \leq x \text{ , con } a+2 \leq j \leq k \\ & + \cdots \\ & = P_2\left(\frac{x}{p_{a+1}}, a\right) + P_2\left(\frac{x}{p_{a+2}}, a\right) + \cdots = \sum_{i>a} P_2\left(\frac{x}{p_i}, a\right) \end{aligned}$$

Se además denotamos $b_i = \pi\left(\sqrt{\frac{x}{p_i}}\right)$ y usando (**??**) se tiene que:

$$P_3(x,a) = \sum_{i>a} P_2\left(\frac{x}{p_i}, a\right) = \sum_{i=a+1}^{c}\sum_{j=i}^{b_i}\left\{\pi\left(\frac{x}{p_ip_j}\right) - (j-1)\right\}$$

donde se asume que $a < c = \pi\left(\sqrt{[3]x}\right)$ para que en todos contenga términos positivos y si elegimos $a = \pi\left(\sqrt{[4]x}\right)$ y $b = \pi\left(\sqrt{x}\right)$, obtenemos la fórmula de Lehmer buscada:

$$\begin{aligned} \pi\,(x) = & \lfloor x \rfloor - \sum_{i=1}^{a} \lfloor \frac{x}{p_i} \rfloor + \sum_{1 \leq i < j \leq a} \lfloor \frac{x}{p_ip_j} \rfloor - \cdots + \frac{(b+a-2)\,(b-a+1)}{2} + \\ & - \sum_{i=a+1}^{b} \pi\left(\frac{x}{p_i}\right) - \sum_{i=a+1}^{c}\sum_{j=i}^{b_i}\left\{\pi\left(\frac{x}{p_ip_j}\right) - (j-1)\right\} \end{aligned}$$

5.1. Cálculos

La parte que consume mayor labor de cálculo en las fórmulas de Lehmer y Meissel es todavía el cálculo de la suma de Legendre:

$$\phi\,(x,a) = \lfloor x \rfloor - \sum \left\lfloor \frac{x}{p_i} \right\rfloor + \sum \left\lfloor \frac{x}{p_ip_j} \right\rfloor - \sum \left\lfloor \frac{x}{p_ip_jp_k} \right\rfloor + \cdots \tag{5.1.5}$$

que cuenta los números positivos $\leq x$ que no sean divisbles por los primeros a primos: $p_1,\, p_2, \ldots, p_a$. En cualquiera de las fórmulas vistas el su cálculo se hace tedioso para valores grandes de x y para simplificar estos cálculos se usarán algunas técnicas y fórmulas que se describen a continuacón:

1. $\phi\,(x,a) = \phi\,(x,a-1) - \phi\left(\dfrac{x}{p_a}, a-1\right)$

   Esta fórmula expresa el hecho de que los enteros no divisibles por cualquiera de los primos

   $p_1,\, p_2, \ldots, p_a$ son precisamente aquellos que no son divisibles por $p_1,\, p_2, \ldots, p_{a-1}$ excepto aquellos que sean divisibles por p_a. La idea es usar esta fórmula repetidamente hasta llegar a $\phi\,(y,k)$ para algún valor de k razonablemente grande que además sea fácil de calcular.

2. Si definimos $m_k = p_1\, p_2 \cdots p_k$, podemos probar la siguiente identidad:

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

Para demostrar esta igualdad, partimos de lo siguiente:

$\phi\left(m_k, k\right) = \phi\left(m_k, k-1\right) - \phi\left(\dfrac{m_k}{p_k}, k-1\right)$, esto usando la fórmula del punto anterior, luego,

$\phi\left(m_k, k\right) = p_k \cdot \phi\left(m_{k-1}, k-1\right) - \phi\left(m_{k-1}, k-1\right)$, y de aquí que:

$\phi\left(m_k, k\right) = \left(p_k - 1\right) \cdot \phi\left(m_{k-1}, k-1\right)$, obteniendo así una fórmula recursiva y aplicándola reiteradamente obtenemos lo siguiente:

$\phi\left(m_k, k\right) = \left(p_k - 1\right)\left(p_{k-1} - 1\right)\left(p_{k-2} - 1\right) \cdots\ \phi\left(m_1, 1\right)$ y como $\phi\left(m_1, 1\right) = 1 = p_1 - 1$, entonces se tiene el resultado deseado.

3. Otro resultado que se puede obtener es el siguiente:

$\phi\left(s \cdot m_k + t, a\right) = s \cdot \phi\left(m_k, a\right) + \phi\left(t, a\right)$ donde $a \leq k$

Lo anterior se puede demostrar del siguiente modo:

$$\begin{aligned}
\phi\left(s \cdot m_k + t, a\right) &= \left\lfloor s \cdot m_k + t \right\rfloor - \sum \left\lfloor \frac{s \cdot m_k + t}{p_i} \right\rfloor + \sum \left\lfloor \frac{s \cdot m_k + t}{p_i p_j} \right\rfloor - \sum \left\lfloor \frac{s \cdot m_k + t}{p_i p_j p_k} \right\rfloor + \cdots \\
&= s m_k + \lfloor t \rfloor - \sum \left(\frac{s\ m_k}{p_i} + \left\lfloor \frac{t}{p_i} \right\rfloor \right) + \sum \left(\frac{s\ m_k}{p_i p_j} + \left\lfloor \frac{t}{p_i p_j} \right\rfloor \right) + \\
&\quad - \sum \left(\frac{s\ m_k}{p_i p_j p_k} + \left\lfloor \frac{t}{p_i p_j p_k} \right\rfloor \right) + \cdots \\
&= \left[s \cdot m_k - \sum \frac{s \cdot m_k}{p_i} + \sum \frac{s \cdot m_k}{p_i p_j} - \sum \frac{s \cdot m_k}{p_i p_j p_k} + \cdots \right] + \\
&\quad + \left[\lfloor t \rfloor - \sum \left\lfloor \frac{t}{p_i} \right\rfloor + \sum \left\lfloor \frac{t}{p_i p_j} \right\rfloor - \sum \left\lfloor \frac{t}{p_i p_j p_k} \right\rfloor + \cdots \right] \\
&= s \cdot \phi\left(m_k, a\right) + \phi\left(t, a\right)
\end{aligned}$$

Para t entre 0 y m_k.

4. Además, se tiene $\phi\left(t, k\right) = \varphi(m_k) - \phi\left(m_k - t - 1, k\right)$ para $t > \frac{m_k}{2}$donde $\varphi(m_k) = \phi\left(m_k, k\right) = \prod_{i=1}^{k} \left(p_i - 1\right)$

Por ejemplo $m_4 = 210$ y para $t \leq m_4/2 = 105$ se genera la tabla siguiente:

tabla crítica de $\phi\left(x, 4\right)$					
1 1	19 5	37 9	53 13	71 17	89 21
11 2	23 6	41 10	59 14	73 18	97 22
13 3	29 7	43 11	61 15	79 19	101 23
17 4	31 8	47 12	67 16	83 20	103 24

5. Finalmente para $t > m_4$, donde $t = s \cdot m_k - h$, para algún $h \leq \frac{m_k}{2}$ combinando las fórmulas se tiene que :

   $\phi(t,k) = \phi(s \cdot m_k - h, k) = s \cdot \phi(m_k, k) - \phi(h-1,k) = s \cdot \varphi(m_k) - \phi(h-1,k)$ para algún s y un $h \leq \frac{m_k}{2}$, es decir:

   $$\phi(s \cdot m_k - h, k) = s \cdot \varphi(m_k) - \phi(h-1,k)$$

   lo que también nos ayuda a reducir el cálculo. Por ejemplo,

   $$\phi(10000,4) = \phi(48 \cdot 210 - 80, 4) = 48 \cdot \varphi(m_4) - \phi(79,4) = 48 \cdot 48 - 19 = 2285$$
   $$\phi(909,4) = \phi(4 \cdot 210 + 69, 4) = 4 \cdot \varphi(m_4) + \phi(69,4) = 4 \cdot 48 + 16 = 208$$

La idea es combinar todas esta fórmulas y los valores de una tabla crítica para simplificar los cálculos que se deban realizar al usar la fórmula de Meissel o la fórmula de Lehmer.

Ejemplo. Si $x = 10^4$, obteniendo los valores de los límites de las sumatorias, se tiene que $b = \pi(\sqrt{x}) = \pi(100) = 25$, $c = \pi\left(\sqrt{[}3]x\right) = \pi(21) = 8$, y aplicando la fórmula de Meissel, se obtiene lo siguiente:

$$\begin{aligned}\pi(x) &= \lfloor x \rfloor - \sum_{i=1}^{c} \lfloor \frac{x}{p_i} \rfloor + \sum_{1 \leq i < j \leq c} \lfloor \frac{x}{p_i p_j} \rfloor - \cdots + \frac{(b+c-2)(b-c+1)}{2} - \sum_{i=c+1}^{b} \pi\left(\frac{x}{p_i}\right) \\ &= \phi(x,8) + \tfrac{(25+8-2)(25-8+1)}{2} - \sum_{i=9}^{25} \pi\left(\frac{x}{p_i}\right) = \phi(x,8) + \tfrac{31 \cdot 18}{2} - \sum_{i=9}^{25} \pi\left(\frac{x}{p_i}\right)\end{aligned}$$

Ahora, si nos centramos primero en calcular $\phi(x,8)$, tendrámos el siguiente desarrollo:

$$\begin{aligned}\phi(x,8) &= \phi(x,7) - \phi\left(\frac{x}{19},7\right) = \phi(x,7) - \phi(526,8) \\ \phi(x,7) &= \phi(x,6) - \phi\left(\frac{x}{17},6\right) = \phi(x,6) - \phi(588,6) \\ \phi(x,6) &= \phi(x,5) - \phi\left(\frac{x}{13},5\right) = \phi(x,5) - \phi(769,5) \\ \phi(x,5) &= \phi(x,4) - \phi\left(\frac{x}{11},4\right) = \phi(x,4) - \phi(909,4)\end{aligned}$$

así que $\phi(x,8) = \phi(x,4) - \phi(909,4) - \phi(769,5) - \phi(588,6) - \phi(526,8)$, donde los dos primeros términos se calcularon previamente por medio de la tabla crítica y las propiedades estudiadas, de este modo:

$\phi(x,4) - \phi(909,4) = \phi(10000,4) - \phi(909,4) = 2285 - 208 = 2077$

Mientras que los otros términos: $\phi(769,5)$, $\phi(588,6)$, $\phi(526,8)$, se calculan de manera similar, pero estos cálculos se omiten aquí.

5.2. El método de Mapes.

Las fórmulas de Meissel y Lehmer para calcular $\pi(x)$ están basadas en la fórmula de Legendre, de hecho estas son simplemente reordenamientos y agrupamientos para facilitar su cálculo. En 1963 Mapes halló un modo más eficiente de realizar esta suma, el método se explica a continuación.

Recordemos que $\phi\left(x,a\right)=\lfloor x\rfloor-\sum\left\lfloor\frac{x}{p_i}\right\rfloor+\sum\left\lfloor\frac{x}{p_ip_j}\right\rfloor-\sum\left\lfloor\frac{x}{p_ip_jp_k}\right\rfloor+\cdots$, donde $p_i<p_j<p_k\leq p_a$, y con p_a el a -ésimo número primo. Si hacemos un conteo de estos términos se puede notar que es equivalente al número de subconjuntos que se puede obtener de $\{1,2,\ldots,a\}$, que como sabemos es 2^a. Por otro lado resulta que cada uno de estos términos puede escribirse como un caso particular de la siguiente expresión:

$$T_k(x,a)=(-1)^{\beta_0+\beta_1+\cdots+\beta_{a-1}}\left\lfloor\frac{x}{p_1^{\beta_0}\;p_2^{\beta_1}\;\cdots p_a^{\beta_{a-1}}}\right\rfloor$$

Donde los β_i son los dígitos de la descomposición binaria del número k , es decir: $(k)_2=\beta_{a-1}\ldots\beta_1\beta_0$

Ahora bien, si asociamos al $n-$ ésimo primo p_n con la potencia 2^{n-1} y también, asociamos a cada producto de primos distintos con la suma de sus correspondientes potencias, así por ejemplo el producto $11\cdot 7\cdot 3$ se asocia con $2^4+2^3+2^1$. Es claro que $p_1^{\beta_0}\;p_2^{\beta_1}\;\cdots p_a^{\beta_{a-1}}$ recorre todos los productos de factores primos distintos, para factores primos menores o iguales a p_a y donde k varía por todas las diferentes sumas de potencias de 2 que sean menores que 2^a. Finalmente, la expresión $(-1)^{\beta_0+\beta_1+\cdots+\beta_{a-1}}$ se utiliza para corresponder con el signo de los diferentes términos de la expresión original.

Por ejemplo si $a=3$ la fórmula de Legendre es :

$$\lfloor x\rfloor-\left\lfloor\frac{x}{p_1}\right\rfloor-\left\lfloor\frac{x}{p_2}\right\rfloor-\left\lfloor\frac{x}{p_3}\right\rfloor+\left\lfloor\frac{x}{p_1p_2}\right\rfloor+\left\lfloor\frac{x}{p_1p_3}\right\rfloor+\left\lfloor\frac{x}{p_2p_3}\right\rfloor-\left\lfloor\frac{x}{p_1p_2p_3}\right\rfloor$$

y cada uno de los términos que la componen pueden ser escritos con esta nueva notación del siguiente modo:

Por definición se toma $T_0(x,3)=\lfloor x\rfloor$ y por lo anterior se tiene que

$T_1(x,3)=T_{(001)_2}(x,3)=-\left\lfloor\frac{x}{p_1}\right\rfloor$

$T_2(x,3)=T_{(010)_2}(x,3)=-\left\lfloor\frac{x}{p_2}\right\rfloor$

$T_4(x,3)=T_{(100)_2}(x,3)=-\left\lfloor\frac{x}{p_3}\right\rfloor$

$T_3(x,3)=T_{(011)_2}(x,3)=+\left\lfloor\frac{x}{p_1p_2}\right\rfloor$

$T_5(x,3)=T_{(101)_2}(x,3)=+\left\lfloor\frac{x}{p_1p_3}\right\rfloor$

$T_6(x,3)=T_{(110)_2}(x,3)=+\left\lfloor\frac{x}{p_2p_3}\right\rfloor$

y finalmente $T_7(x,3)=T_{(111)_2}(x,3)=-\left\lfloor\frac{x}{p_1p_2p_3}\right\rfloor$

Así en general la fórmula de Legendre pude ser escrita del siguiente modo:

$$\phi\left(x,a\right)=\sum_{k=0}^{2^a-1}T_k(x,a)$$

Sea M un entero $< 2^a$ y si 2^i es la mayor potencia de 2 que lo divide, se definimos $\gamma\,(M,x,a)$ como sigue:

$$\gamma\,(M,x,a) = \sum_{k=M}^{M+2^i-1} T_k(x,a)$$

entonces, tenemos la siguiente igualdad:

$$\phi\,(x,a) = T_0(x,a) + \gamma\left(2^0,x,a\right) + \gamma\left(2^1,x,a\right) + \cdots + \gamma\left(2^{a-1},x,a\right)$$

Además tenemos que $sign\{T_k(x,a)\} = (-1)^{\beta_i+\beta_{i+1}+\cdots+\beta_{a-1}}$ si $2^i|\ k$, dado que los últimos i dígitos de k son iguales a cero y en este caso se tiene que:

$$|T_k(x,a)| = \left\lfloor \frac{x}{p_{i+1}^{\beta_i}\ p_{i+2}^{\beta_{i+1}}\ \cdots p_a^{\beta_{a-1}}} \right\rfloor \quad \text{siempre que } 2^i|k$$

Aplicando la definición de T para las siguientes condiciones $2^i|\ k$ y $0 \le k' < 2^i$, donde $(k')_2 = \beta'_{i-1}\ldots\beta'_1\beta'_0$, tenemos que

$$T_{k'}\left\{T_k(x,a),i\right\} = (-1)^{\beta'_0+\beta'_1+\cdots+\beta'_{i-1}} \left\lfloor \frac{T_k(x,a)}{p_1^{\beta'_0}\ p_2^{\beta'_1}\ \cdots p_a^{\beta'_{i-1}}} \right\rfloor \quad \text{y como}$$

$$T_k(x,a) = (-1)^{\beta_i+\beta_{i+1}+\cdots+\beta_{a-1}} \left\lfloor \frac{x}{p_{i+1}^{\beta_i}\ p_{i+2}^{\beta_{i+1}}\ \cdots p_a^{\beta_{a-1}}} \right\rfloor$$

y si definimos $T_k(-x,a) = -T_k(x,a)$, entonces obtenemos lo siguiente:

$$\begin{aligned} T_{k'}\left\{T_k(x,a),i\right\} &= (-1)^{\beta'_0+\beta'_1+\cdots+\beta'_{i-1}}\,(-1)^{\beta_i+\beta_{i+1}+\cdots+\beta_{a-1}} \left\lfloor \frac{x}{p_1^{\beta'_0}\ p_2^{\beta'_1}\ \cdots p_i^{\beta'_{i-1}} p_{i+1}^{\beta_i}\ p_{i+2}^{\beta_{i+1}}\ \cdots p_a^{\beta_{a-1}}} \right\rfloor \\ &= (-1)^{\beta'_0+\beta'_1+\cdots+\beta'_{i-1}+\beta_i+\beta_{i+1}+\cdots+\beta_{a-1}} \left\lfloor \frac{x}{p_1^{\beta'_0}\ p_2^{\beta'_1}\ \cdots p_i^{\beta'_{i-1}} p_{i+1}^{\beta_i}\ p_{i+2}^{\beta_{i+1}}\ \cdots p_a^{\beta_{a-1}}} \right\rfloor \\ &= T_{k'+k}(x,a) \end{aligned}$$

Lo anterior se puede resumir en el siguiente resultado:

$$T_{k'}\left\{T_k(x,a),i\right\} = T_{k'+k}(x,a) \text{ , donde , } 2^i|k \ y \ k' < 2^i$$

Volviendo al primer resultado combinado con este último se tiene que

$$\phi\left(T_M(x,a),i\right) = \sum_{k'=0}^{2^i-1} T_{k'}(T_M(x,a),i) = \sum_{k=M}^{M+2^i-1} T_k(x,a) = \gamma\,(M,x,a)$$

Ahora, recordando que $\phi\,(x,a) = T_0(x,a) + \gamma\left(2^0,x,a\right) + \gamma\left(2^1,x,a\right) + \cdots + \gamma\left(2^{a-1},x,a\right)$, entonces la fórmula de Legendre se puede rescribir del siguiente modo:

$$\phi\,(x,a) = T_0(x,a) + \phi\,(T_{2^0}(x,a),0) + \phi\,(T_{2^1}(x,a),1) + \cdots + \phi\,(T_{2^{a-1}}(x,a),a-1)$$

y reemplazando x por $T_M(x,a)$ y a la vez a por i se tiene que

$$\begin{aligned}\phi\,(T_M(x,a),i) &= T_0(T_M(x,a),i) + \phi\,(T_{2^0}(T_M(x,a),i),0) + \cdots + \phi\,(T_{2^{a-1}}(T_M(x,a),i),i-1)\\ &= T_M(x,a) + \phi\,(T_{M+1}(x,a),0) + \phi\,(T_{M+2^1}(x,a),1) + \cdots + \phi\,(T_{M+2^{a-1}}(x,a),i-1)\end{aligned}$$

todo esto siempre que $2^i|\ k$, recordando que $\phi\,(T_{M+1}(x,a),0) = T_0(T_{M+1}(x,a),0) = T_{M+1}(x,a)$ se tiene lo siguiente:

$$\phi\,(T_M(x,a),i) = T_M(x,a) + T_{M+1}(x,a) + \phi\,(T_{M+2^1}(x,a),1) + \cdots + \phi\,(T_{M+2^{a-1}}(x,a),i-1) \qquad (5.2.6)$$

Empezando con $\phi\,(x,a) = \phi\,(T_0(x,a),a)$ y aplicando esta última fórmula (**??**) recursivamente, se puede obtener el valor de $\phi\,(x,a)$. Este método parece ser más complicado, pero para valores grandes de x éste es mucho más rápido. De hecho, para valores grandes de x el algoritmo de Mapes requiere un tiempo computacional aproximadamente proporcional a $x^{0,7}$, mientras que la fórmula de Lehmer es algo más lenta.

Capítulo 6

MCD de dos Polinomios en $\mathbb{Z}[x_1, ..., x_k]$, $\mathbb{Z}_p[x]$ y $\mathbb{Q}[x]$.

Walter Mora Flores
`wmora2@yahoo.com.mx`
Escuela de Matemática – Centro de Recursos Virtuales (CRV)
Instituto Tecnológico de Costa Rica

6.1. 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.

6.2. 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 “·”, 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 “·”.
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:

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

donde p es primo.

- La aritmética módulo p es muy sencilla. Las operaciones con “+” y “·” 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 $a \equiv b \pmod{p/\text{mcd}(c, p)}$.

Más adelante vamos a volver a las congruencias.

6.2.1. 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 3 (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 4 **(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 $a(x)=a_nx^n+...+a_0$.

- 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 $a(x)=a_nx^n+...+a_0$.

- 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 5

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 10

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 11

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 $\text{grado}_i(a(x_1,...,x_v))$

6.3. 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,

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

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

Teorema 12

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 MCD (a, b) = 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) $r_1(x) = x^3 - 8$

$$\text{Dividimos } r_0(x) \text{ por } r_1(x), \quad \begin{array}{l|l} x^5 - 32 & x^3 - 8 \\ \cline{2-2} \cdots & x^2 \\ \hline \text{residuo: } 8x^2 - 32 & \end{array}$$

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

$$\text{Ahora, dividimos } r_1(x) \text{ por } r_2(x), \quad \begin{array}{l|l} x^3 - 8 & 8x^2 - 32 \\ \cline{2-2} \cdots & x/8 \\ \hline \text{residuo: } 4x - 8 & \end{array}$$

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

$$\text{Ahora, dividimos } r_2(x) \text{ por } r_3(x), \quad \begin{array}{l|l} 8x^2 - 32 & 4x - 8 \\ \cline{2-2} \cdots & 2x + 4 \\ \hline \text{residuo: } 0 & \end{array}$$

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 grado$\,(a(x)) = m$ y 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	$12593338795500743100931141992187500$

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

• 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: $\beta_i = \operatorname{cont}(\operatorname{prem}(r_i(x), r_{i-1}(x)))$, 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 $\delta_i = \operatorname{grado}(r_{i-1}(x)) - \operatorname{grado}(r_i(x))$.

Necesitamos algunas cosas antes de establecer el algoritmo.

Definición 6

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 $\operatorname{cont}(a(x))$ y se define como el MCD de los coeficientes de $a(x)$

• Con estas definiciones podemos podemos ver que

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

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

Ejemplo.

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

- En $\mathbb{Z}[x]$, $\operatorname{cont}(a) = 6$ y $\operatorname{pp}(a) = 8x^3 - 14x^2 + 7x - 6$

- 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 3 (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 $m =$ grado$(a(x))$ y $n =$ grado$\,(b(x))$.

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

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

• "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 13

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 $\text{grado}\,(a(x)) \geq \text{grado}\,(b(x))$, sean $q(x)$ y $r(x)$ el seudo-cociente y el seudo-residuo, entonces

$$\text{MCD}\,(a(x), b(x)) \;=\; \text{MCD}\,(b(x), \text{pp}(r(x))) \tag{6.3.1}$$

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 **??** 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 $\text{pp}(r(x))$, por esto, a este algoritmo se le llama el algoritmo primitivo de Euclides.

Algoritmo 6.3.1: Algoritmo Primitivo de Euclides.

```
Data: Polinomios $a(x), b(x) \in D[x]$, $D$ DFU.
Result: $c(x) = \text{MCD}\,(a(x), b(x))$
$c(x) = \text{pp}(a(x))$ ;
$d(x) = \text{pp}(b(x))$ ;
while $d(x) \neq 0$ do
    $r(x) = \text{prem}(c(x), d(x))$ ;
    $c(x) = d(x)$ ;
    $d(x) = \text{pp}(r(x))$ ;
$\lambda = \text{MCD}(\text{cont}(a(x)), \text{cont}(b(x)))$ ;
$g(x) = \lambda\, c(x)$ ;
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 6.1: 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
$$\begin{array}{l|l} yx^2+y^2x & yx^2+(y^2+1)x+y \\ \cline{2-2} -yx^2-(y^2+1)x-y & 1 \\ \cline{1-1} \text{residuo: } -x-y & \end{array}$$

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
$$\begin{array}{l|l} yx^2+(y^2+1)x+y & x+y \\ \cline{2-2} -yx^2-y^2x & yx+1 \\ \cline{1-1} x+y & \\ -x-y & \\ \cline{1-1} \text{residuo: } 0 & \end{array}$$

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 $\varrho = \text{Máx}_{1\leq i\leq k}\,\{\text{grado}\, r_{i-1}(x) - \text{grado}\, r_i(x)\}$ con k el subíndice del último residuo.

6.4. 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

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

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 **??** se puede observar la relación entre los coeficientes principales de los residuos $r_i(x)$ y r_{i-1}

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

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

Definición 7 **(Sucesión de Residuos Polinomiales)**

Sean $a(x), a(x) \in R[x]$ con grado $a(x) \geq$ 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",

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

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

  Si ponemos $q_i(x) = \mathrm{pquo}(r_{i-1}(x), r_i(x))$, 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 $1-\delta_{i-1} \le 0$, 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 [11]). 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 6.4.1: Algoritmo PRS Subresultante.

```
   Data: Polinomios $a(x), b(x) \in D[x]$, $\text{grado}\, a(x) \geq \text{grado}\, b(x)$, $D$ DFU.
   Result: $c(x) = \text{MCD}\,(a(x), b(x))$
1  $r_0 = a(x)$ ;
2  $r_1 = b(x)$ ;
3  $deg_0 = \text{grado}(r_0)$, $deg_1 = \text{grado}(r_1)$, $cp_0 =$ coeficiente principal de $r_0$ ;
4  $cp_1 =$ coeficiente principal de $r_1$, $\delta_1 = deg_0 - deg_1$, $\delta_0 = \delta_1$ ;
5  $\alpha_1 = cp_1^{\delta_1+1}$, $\beta_1 = (-1)^{\delta_1+1}$, $\psi_1 = -1$, $\psi_0 = -1$ ;
6  while $r_1 \neq 0$ do
7      $c = \alpha_1\, r_0$, $q = \text{quo}(c, r_1)$, $r_0 = r_1 w\ r_1 = \text{quo}(c - q \cdot r_1, \beta_1)$ ;
8      $deg_0 = \text{grado}(r_0)$, $deg_1 = \text{grado}(r_1)$ ;
9      $cp_0 =$ coeficiente principal de $r_0$ ;
10     $cp_1 =$ coeficiente principal de $r_1$ ;
11     $\delta_0 = \delta_1$, $\delta_1 = deg_0 - deg_1$ ;
12     $\alpha_1 = cp_1^{\delta_1+1}$, $\psi_0 = \psi_1$ ;
13     if $\delta_0 > 0$ then
14         $\psi_1 = \text{quo}(-cp_0^{\delta_0}, \psi_0^{\delta_0-1})$ ;
15     else
16         $\psi_1 = -cp_0^{\delta_0} \cdot \psi_0$
17     $\beta_1 = -cp_0 \cdot \psi_1^{\delta_1}$ ;
18 Normalizar salida;
19 $r_0 =$ MCD (cont($a(x)$), cont($b(x)$)) $\cdot$ pp($r_0$);
20 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 $r_0 =$ MCD $(6, 2) \cdot (2x-3) == 4x-6$.

6.5. 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 MCD $(a(\xi), b(\xi)) = \gamma$. Veamos que podría pasar

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

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

Sea $g(x) =$ MCD $(a(x), b(x))$.

- si $\xi = 5$ entonces $g(5) = 48 \neq 24 =$ MCD $(a(5), b(5)) = \gamma$,
- si $\xi = 10$ entonces $g(10) = 224 =$ 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 reconstruir $g(x) =$ MCD $(a(x), b(x))$ usando una representación ξ-ádica de $\gamma =$ MCD $(a(\xi), b(\xi))$. Claro, tenemos el problema de escoger un ξ lo más pequeño que se pueda de tal manera que nos produzca polinomios con el grado adecuado. En polinomios de varias variables se debe hacer esto para cada variable, así que cada ξ debe mantenerse pequeño.

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

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

$$\gamma = u_0 + u_1\xi + ... + u_d\xi^d \tag{6.5.2}$$

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

```
Data: $m, p \in \mathbb{Z}$
Result: $u = m \,(\text{mód}\, p)$ en la representación simétrica de $\mathbb{Z}_p$, es decir $-p/2 < u \leq p/2$
$u = \text{rem}(m, p)$ ;
if $u > p/2$ then
    $u = u - p$
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 **??**, $\gamma \equiv u_0 \pmod{\xi}$ y entonces,

$$u_0 = \phi_\xi(\gamma) \tag{6.5.3}$$

- $\gamma - u_0$ 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

$$u_i = \phi_\xi\left(\frac{\gamma\, -\, (u_0 + u_1\xi + ... + u_{i-1}\xi^{i-1})}{\xi^i}\right),\ i = 1, ..., d \tag{6.5.4}$$

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 $u_{\max} = \max_e |u_e|$.

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

- 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 6.5.2: Representación ξ-ádica de γ.

```
Data: $\gamma, \xi \in \mathbb{Z}$
Result: $u_0, u_1, ..., u_d$ tal que $\gamma = u_0 + u_1\xi + ... + u_d\xi^d$ con $\xi^{d+1} > 2|\gamma|$ y $-\xi/2 < u_i \leq \xi/2$.
$e = \gamma$;
$i = 0$;
while $e \neq 0$ do
    $u_i = \phi_\xi(e)$;
    $e = (e - u_i)/\xi$;
    $i = i + 1$;
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.

6.5.2. Reconstrucción del Máximo Común Divisor

Aunque en esta primera parte solo vamos a implementar el algoritmo en una indeterminada, vale la pena enunciar los teoremas de manera general.

Teorema 14

Sean P y Q dos polinomios dependiendo de las variables $x_1, x_2, ..., x_k$ con coeficientes enteros. Sea $\xi \in \mathbb{Z}$ tal que $\xi \geq 2 \cdot \text{Mín}\{\|P\|_\infty, ||Q||_\infty\} + 2$ donde $||P||_\infty$ denota el coeficiente numérico más grande de P (en valor absoluto). Si G es la *parte primitiva* de la reconstrucción que se obtiene a partir de la representación $\xi-$ádica de $\gamma = \text{ MCD}\,(P(x_1, ...x_{k-1}, \xi), Q(x_1, ...x_{k-1}, \xi))$ y si G divide a P y Q entonces $G = \text{ MCD}\,(P, Q)$.

Prueba. Ver [**?**].

- En realidad, para ξ suficientemente grande (posiblemente demasiado grande), $G|P$ y $G|Q$ siempre (tomando G como se indica en el teorema).

  El problema es que, para que el algoritmo que se deriva sea útil, ξ debe ser razonablemente pequeño. Así que se admite cierta posibilidad de falla y se mantiene solo la condición esencial de divisibilidad.

- Con la escogencia de ξ, según el teorema, se tiene un punto de partida para ir probando polinomios. El algoritmo heurístico es un algoritmo tipo "Las Vegas": siempre entrega la respuesta correcta, pero no garantiza el tiempo total de ejecución, aunque con alta probabilidad, éste será bajo.

- El teorema nos dice que con esta escogencia de ξ, reconocer un resultado correcto o incorrecto (a partir de la reconstrucción) es solo una cuestión de dividir.

- En el teorema se toma la parte primitiva de la reconstrucción 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 P y Q fallaría siempre. Aún removiendo el contenido, el teorema garantiza que el test de divisibilidad decide si G es correcto o no.

Ejemplo.

- Sea $a(x)=x^3-3x^2+2x=(x-2)(x-1)x$ y $b(x)=x^3+6x^2+11x+6=(x+1)(x+2)(x+3)$. $6\,|\,a(\xi)$, y $6\,|\,b(\xi)$ para todo $\xi\in\mathbb{Z}$. Observe sin embargo que estos polinomios son *primos relativos.*

- Como se señaló antes, 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, se obtuvo tiempos de $\boldsymbol{O}\left(n^4\right)$ para este algoritmo ([**?**], [11]).

- Hay que recordar que este algoritmo es muy eficiente en problemas pequeños (no más de cuatro variables) y se usa como complemento de otros algoritmos.

En el algoritmo que sigue, usamos el homomorfismo de evaluación $\phi_{(x_1-\xi)}:\mathbb{Z}[x_1,...x_k]\longrightarrow\mathbb{Z}[x_2,...,x_k]$ definido por

$$\phi_{(x_1-\xi)}(a(x_1,...,x_n))=a(\xi,...,x_n)$$

Por ejemplo, si $a(x,y)=(y^2+1)x^2+(y+y^3)\,x$, entonces $\phi_{(x-\xi)}=(y^2+1)\xi^2+(y+y^3)\,\xi$.

También, si $a(x_1,x_2,...,x_n)=\sum a_{i_1,i_2,...,i_n}x_1^{i_1}x_2^{i_2}\cdots x_n^{i_n}$, entonces

$$||a(x_1,x_2,...,x_n)||_\infty=\text{Máx}_{i_1,i_2,...,i_n}\{|a_{i_1,i_2,...,i_n}|\}$$

Algoritmo 6.5.3: MCDHEU(A,B).

```
   Data: $A, B \in \mathbb{Z}[x_1, ..., x_k]$
   Result: $G =$ MCD $(A,B)$ si el resultado de la búsqueda heurística da resultado, sino retorna $-1$
1  $vars = Indeterminadas(A) \bigcup Indeterminadas(B)$ ;
2  if $Card(vars) = 0$ then
3      return $\gamma =$ MCD $(A,B) \in \mathbb{Z}$
4  else
5      $x = vars[1]$
6  $\xi = 2 \cdot \text{Mín}\{||A||_\infty, ||B||_\infty\} + 2$ ;
7  $i = 0$ ;
8  while $i < 7$ do
9      if $length(\xi) \cdot máx\{grado_x A, grado_x B, \} > 5000$ then
10         return $-1$ (MCD no puede ser negativo, $-1$ se usa como indicador de fallo)
11     $\gamma =$ MCDHEU$(\phi_{(x-\xi)}(A), \phi_{(x-\xi)}(B))$ (llamada recursiva);
12     if $\gamma \neq -1$ then
13         Generar $G$ a partir de la expansión $\xi-$ádica de $\gamma$
14     if $G|A$ y $G|B$ then
15         return $G$
16     Crear un nuevo punto de evaluación;
17     $\xi =$ quo$(\xi \times 73794,\ 27011)$
18 return $-1$ ;
```

- En al línea 9 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 14, 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 17 lo que procura es tener algún grado de "aleatoriedad" en la escoegencia 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 $\mathbb{Z}[\boldsymbol{x}, \boldsymbol{y}]$. 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

$$\begin{aligned} 40 &= 4 + 1\cdot 36^1 \\ G_2 &= 4 + y \end{aligned}$$

- 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

$$\begin{aligned} 4+y &= y\cdot 4^0 + 1\cdot 4^1 \\ G_1 &= y + 1\cdot x \end{aligned}$$

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

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

6.6. 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,

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

  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 6.6.1: Algoritmo Extendido de Euclides.

```
Data: $a, b \in D$ dominio Euclidiano
Result: MCD $(a,b) = g$ y $s, t \in D$ tal que $g = sa + tb$.
$c = n(a),\ d = n(b)$ ;
$c_1 = 1,\ d_1 = 0$ ;
$c_2 = 0,\ d_2 = 1$ ;
while $d! = 0$ do
    $q = \text{quo}(c,d),\ r = c - qd$ ;
    $r_1 = c_1 - qd_1,\ r_2 = c_2 - qd_2$ ;
    $c = d,\ c_1 = d_1,\ c_2 = d_2$ ;
    $d = r,\ d_1 = r_1,\ d_2 = r_2$ ;
MCD $= n(c)$ ;
$s = c_1/(u(a) * u(c)), t = c_2/(u(b) * u(c))$ ;
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) $t = \dfrac{17x}{535} + \dfrac{71}{2140}$

6.7. 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.

6.7.1. 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.

6.7.2. 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);
    }
}
```

6.7.3. 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....

}//
```

6.7.4. 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;
}//

//MCD Heurístico en Z[x]
public Bpolinomio MCDHeu(Bpolinomio A, Bpolinomio B)
{
  //Agregar caso especial deg B > deg A.

  //vars={} pues  A,B en Z[x],
  //llamamos MCDHeu(phi_{x-xi}(A),phi_{x-xi}(B))

 BigInteger lda   = (A.cont()).gcd(B.cont());
 Bpolinomio G       = Bpolinomio.ZERO;
 Bpolinomio gamma   = new Bpolinomio();
 BigInteger BI2     = new BigInteger("2");
 Qpolinomio QpolObj = Qpolinomio.ZERO; //solo para llamar métodos.
 Qpolinomio QG,QA,QB,r1,r2; //requeiere división en Q[x]

 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) ex Z[x]

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

 for(int i= 1; i<=6; i++)
 {
    if(lengthxi*Math.max(A.deg, B.deg)>5000)
       return failflag; //sale

     gamma = MCDHeu(new Bpolinomio(A.evaluate(xi),0),  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);
      }else{ //gamma es polinomio
             //G.Reconstruccion_xi_adica(gamma, xi, var);
           }
     }
   //Viene división en Q[x]
    QG = QpolObj.leer(G.toString()); //lo lee como Qpolinomio
    //Test de divisibilidad
    r1 = QA.minus((QA.divides(QG)).times(QG));
    r2 = QB.minus((QB.divides(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;
    }
    //sino salió, crear un  nuevo punto de evaluación
    xi = (xi.multiply(new BigInteger("73794"))).divide(new BigInteger("27011"));
 }
 return failflag; //-1 si no encuentra algo.
}
```

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.