UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

POSGRADO EN CIENCIA E INGENIERÍA DE LA COMPUTACIÓN

IMPLEMENTACIÓN DEL ALGORITMO DE BERLEKAMP PARA FACTORIZACIÓN DE POLINOMIOS SOBRE CAMPOS FINITOS

T E S I S

QUE PARA OBTENER EL GRADO DE:

MAESTRO EN CIENCIAS (COMPUTACIÓN)

P R E S E N T A:

CARLOS ALBERTO VÁZQUEZ FERNÁNDEZ

DIRECTOR DE LA TESIS: DR. GERARDO VEGA HERNÁNDEZ

MÉXICO, D.F. **FEBRERO DE 2009.**

CONTENIDO

AGRADECIMIENTOS

Agradezco a la Universidad Nacional Autónoma de México, de manera especial, a su Facultad de Ingeniería, por haberme permitido el estudiar la maestría en Ciencia e Ingeniería de la Computación.

Al IIMAS, que me brindó el espacio físico para poder desarrollar las actividades necesarias durante los últimos dos años.

El agradecimiento a la Coordinación del Posgrado, al doctor Boris Escalante Ramírez, y al doctor Fernando Arámbula Cosio, coordinadores durante este periodo, y a todo su equipo de trabajo, sin los cuales no hubiera sido posible el logro obtenido.

A mi tutor, el doctor Gerardo Vega Hernández, por su paciencia y sus consejos, a lo largo de este ciclo.

Finalmente, a mis sinodales, por el valioso tiempo que dedicaron a la revisión y corrección de mi propuesta de tesis: doctores Vladislav Khartchenko, Enrique Daltabuit Godas, Francisco García Ugalde, Pablo Barrera Sánchez.

RESUMEN

Al hacer una revisión de los paquetes de computación científica que tenemos a nuestra disposición, como son Maple, Mathematica y Matlab, nos hemos encontrado con varias carencias y restricciones en lo referente a factorización de polinomios, de manera particular, cuando esta se lleva a cabo sobre campos finitos no primos. Por ello, en este trabajo, hemos decidido aportar la implementación de un método para factorizar polinomios que cubra las limitaciones de los paquetes mencionados, y de esta forma contar con una herramienta un poco más completa sobre factorización, lo cual es importante para diversas aplicaciones. Tal método es el algoritmo de Berlekamp, un algoritmo para factorización de polinomios sobre campos finitos, de gran difusión en matemáticas aplicadas, particularmente en el campo del álgebra computacional. En este trabajo, realizamos una descripción del algoritmo explicando su funcionamiento tanto de manera teórica, como de manera experimental, esto último apoyándonos en la implementación realizada.

Como complemento al trabajo realizado, se hace una breve revisión de teoría algebraica básica, en la que se aborda a los campos finitos, y también se revisan conceptos básicos de polinomios, ambos estrechamente relacionados con el trabajo en cuestión.

Capítulo 1

INTRODUCCIÓN

Un problema interesante a resolver, desde el punto de vista computacional, es la factorización de polinomios. De manera particular, la factorización de polinomios sobre campos finitos. Por lo que el objetivo que nos hemos propuesto en este trabajo, es lograr una factorización completa dentro de dichos campos.

Un polinomio es una expresión algebraica con varias propiedades. En el capítulo 2 definiremos formalmente sus rasgos básicos, ahora nos basta con hablar de su utilidad e importancia para nuestro propósito: queremos tomar un polinomio sobre un campo finito, y lo queremos factorizar sobre dicha estructura, ya que los factores que resultan de este procedimiento son de gran utilidad para otras aplicaciones. Las aplicaciones pueden ser en Teoría de la Información, y áreas que se relacionan con ésta, como generación de códigos [1], teoría de campos finitos [1,2], criptografía [3], generación de secuencias seudoaleatorias de bits [4], entre otras. Tales aplicaciones caen tanto en áreas de las matemáticas como en el área de la ingeniería, por lo que contar con una herramienta un poco más completa sobre factorización de polinomios es importante.

1. ANTECEDENTES

Haciendo una revisión de las herramientas comerciales a nuestra disposición para modelado matemático, podemos decir que Mat-

lab versión 7 para Windows ni siquiera contempla factorización de polinomios sobre campos finitos, y en Mathematica versión 5 y Maple versión 9.5, ambos para Windows, nos encontramos con algunos comandos y funciones que realizan factorizaciones de polinomios sobre campos finitos, pero con ciertas carencias y restricciones:

Poca claridad en las extensiones de campos finitos primos: corremos Mathematica versión 5, bajo el sistema Windows; al hacer esto nos aparece la ventana principal del programa y una página en blanco conocida como cuaderno; en ella se escriben y ejecutan los comandos en secuencia; existe el comando *Factor*, para factorización de polinomios sobre campos finitos, al cual se le pasan como argumentos el polinomio a factorizar, y un número primo, para que se forme el campo finito primo sobre el cual se va a factorizar. La factorización que da como respuesta el programa es correcta. Pero no hay opciones claras para la factorización sobre extensiones, es decir, cuando se toman las potencias subsecuentes de algún número primo para formar campos finitos no primos.

Con Maple sucede algo similar: corremos Maple versión 9.5, bajo el sistema Windows; aparece una ventana con el cuaderno principal de trabajo; escribimos el comando *Berlekamp*, el cual también toma como sus argumentos un polinomio y un número primo para formar el campo. El programa factoriza correctamente, pero existe la misma limitante en cuanto a extensiones, que con el comando *Factor* de Mathematica.

Las implementaciones no son claras: Los aspectos de la

implementación en estas herramientas comerciales se encuentran como cajas negras, haciendo difícil el seguimiento de los pasos de la factorización; además, se hacen continuas llamadas a funciones internas, y uso de llamadas a funciones del sistema operativo, por lo que no es posible calcular el tiempo que lleva el proceso de factorización.

Se desconoce el manejo de memoria: No sabemos qué cantidad de memoria se asigna a cada paso de la factorización, y cuál es el límite de procesamiento antes de que los programas se queden colgados y ya no respondan. Con ello vemos que hay reestricción en la longitud de los campos finitos que se pueden utilizar.

2. LA SOLUCIÓN PROPUESTA

Para desarrollar una herramienta un poco más completa sobre factorización, se necesita un algoritmo que sea capaz de resolver el problema de factorización de forma eficiente y en un periodo de tiempo aceptable. Por esto, optamos por realizar la implementación del algoritmo clásico de Berlekamp para factorización de polinomios sobre campos finitos en lenguaje c, cubriendo todos los aspectos mencionados anteriormente. Es decir, se abarcan los campos finitos primos y sus extensiones, y se tiene toda la información necesaria para la corrección, depuración y ampliación del programa en el momento que sea necesario. Además, el manejo de memoria se hace de forma clara, evitando utilizar una cantidad exagerada que pudiera afectar la obtención de resultados. Dicho algoritmo es eficiente si el campo finito sobre el que se trabaja es relativamente pequeño, ya que el paso del algo-

ritmo, donde específicamente se buscan los factores irreducibles del polinomio a factorizar, requiere ser ejecutado q veces, donde q es el número de elementos del campo sobre el cual se realiza la factorización. Por lo que a mayor q, los tiempos de corrida se elevan de forma considerable. Tomando en cuenta este detalle, hemos elegido el trabajar con campos finitos primos pequeños y sus extensiones pequeñas, hasta un máximo de 81 elementos.

Con esto, se pretenden mejorar los resultados que pudieran obtenerse con alguno de los paquetes mencionados, ya que dichos paquetes también se restringen a campos pequeños, y sus resultados al factorizar sobre extensiones, son ambiguos.

3. ORGANIZACIÓN DE LA TESIS

El presente documento se encuentra organizado en un conjunto de 5 capítulos en total, siendo éste el primero de ellos. En el capítulo 2 se realiza una breve revisión de la teoría algebraica básica. Esta es pieza clave para el desarrollo de este trabajo, ya que este conocimiento es tanto un antecedente, como el lugar mismo de aplicación de resultados de la factorización. La parte teórica, que constituye el desarrollo de esta tesis, se encuentra explicada en el capítulo 3. Ahí, se entra de lleno en la descripción del algoritmo de Berlekamp, para factorización de polinomios sobre campos finitos. La implementación realizada, se explica en el capítulo 4, por medio de una descripción de las funciones principales que se utilizaron. En la última parte de este texto, se plantean la contribución y conclusión del trabajo desarrollado, y se incluye un apéndice con el listado de la implementación, y las referencias bibliográficas necesarias.

Capítulo 2

TEORIA ALGEBRAICA BÁSICA

En este capítulo revisamos la teoría algebraica básica. Es importante conocer estos fundamentos matemáticos, ya que son ampliamente utilizados en diversos campos del conocimiento, como los mencionados al inicio del capítulo anterior. Entonces, las estructuras algebraicas o sistemas algebraicos que se describirán, son conjuntos, en los cuales una o más operaciones están definidas. Solamente las definiciones y propiedades fundamentales de estos sistemas algebraicos son presentadas, ya que para nuestros propósitos con esto es suficiente. Dentro de las estructuras presentadas, abarcamos los campos, dentro de los que se incluyen como caso particular, a los campos finitos, además de revisar las definiciones correspondientes a grupos y anillos. También se describen los conceptos de las expresiones algebraicas llamadas polinomios.

1. GRUPOS

De los diversos sistemas algebraicos existentes, los grupos han sido por mucho, los más estudiados. Esto se debe a que la teoría de grupos es una de las partes más antiguas del álgebra abstracta, así como una de las más ricas en aplicaciones. Basta con mencionar su utilidad en aplicaciones en teoría de la información, por ejemplo, en generación de códigos [1], en ingeniería, para generación de secuencias seudoaleatorias de bits[4], entre otras. La definición formal de grupo es la siguiente:

Definición 2.1: Un conjunto $G \neq \emptyset$ es un *grupo* si en G está definida una operación $(\cdot)$, tal que las siguientes propiedades se cumplen:

1. $(\cdot)$ es *cerrada*; esto es, para cada $a, b \in G$,

$$a \cdot b \in G.$$

2. $(\cdot)$ es *asociativa*; esto es, para cada $a, b, c \in G$,

$$a \cdot (b \cdot c) = (a \cdot b) \cdot c.$$

3. Existe un *elemento identidad (o neutro)* $e \in G$, tal que para toda $a \in G$,

$$a \cdot e = e \cdot a = a.$$

4. Para cada $a \in G$, existe un elemento inverso $a^{-1} \in G$, tal que

$$a \cdot a^{-1} = a^{-1} \cdot a = e.$$

Si el grupo también satisface

5. Para toda $a, b \in G$,

$$a \cdot b = b \cdot a,$$

Entonces el grupo es llamado *abeliano* (o *conmutativo*).

2. ANILLOS

Presentamos otras estructuras algebraicas interesantes: los anillos, los cuales son estructuras que tienen más propiedades que los grupos, y de las cuales también encontramos diversas aplicaciones. La generación de códigos [1], es una de las áreas donde se aplican tales estructuras. Otra aplicación se encuentra en la aritmética matricial, donde las matrices de 3×3 definidas sobre los números reales, forman un anillo [6]. También dentro de los números reales, las funciones que toman números

reales y los mapean hacia números reales, forman un anillo [2, 6]. Veamos su definición formal:

Definición 2.2: Un *anillo* R, es un conjunto no vacío, con dos operaciones $(\cdot, +)$, que satisface las siguientes propiedades:

1. R es un grupo abeliano bajo $(+)$.

2. $(\cdot)$ es *cerrada*. Es decir, para cada $a, b \in R$,

$$a \cdot b \in R.$$

3. $(\cdot)$ es *asociativa*. Esto es, para toda $a, b, c \in R$, tenemos

$$(a \cdot b) \cdot c = a \cdot (b \cdot c).$$

4. Se cumple la ley distributiva; esto es, para toda $a, b, c \in R$, tenemos

$$a \cdot (b + c) = a \cdot b + a \cdot c,$$

y

$$(b + c) \cdot a = b \cdot a + c \cdot a.$$

Si el anillo también satisface la propiedad adicional

5. Para toda $a, b \in R$,

$$a \cdot b = b \cdot a,$$

entonces el anillo es llamado *conmutativo*.

Algo que hay que hacer notar, es que las operaciones $(+)$ y $(\cdot)$ en el anillo R, no son necesariamente las operaciones ordinarias con números. Como convención, usamos 0 (llamado el *elemento cero*) para denotar el elemento identidad del grupo abeliano en R, con respecto a la suma, y el inverso aditivo de a es denotado por $-a$; también, $a + (-b)$ se abrevia como $a - b$. Y $a \cdot b$ es escrito como ab. Como una consecuencia de la definición de anillo, obtenemos la propiedad general $a0{=}0a{=}0$, para todo $a \in R$. Esto implica que $(-a)b{=}a(-b){=}-ab$, para toda $a, b \in R$.

3. POLINOMIOS

Desde las epocas tempranas de las matemáticas, han aparecido problemas relacionados con los polinomios o ecuaciones polinomiales: se tiene conocimiento de que, cerca del año 1600 A.C., los babilonios poseían métodos generales para resolver polinomios cuadráticos, a pesar de no contar con una notación algebraica formal. Posteriormente, en la Grecia antigua, se desarrollaron métodos para resolver polinomios cúbicos, que involucraban los puntos de intersección de cónicas, cerca del año 100 D.C. Ya en el renacimiento, los matemáticos descubieron

la solución algebraica de los polinomios cúbicos. Años después, matemáticos de la talla de Tartaglia, Leibniz y Euler trabajaron en la búsqueda de métodos y demostraciones generales para tales expresiones. Desde entonces, y hasta nuestra actualidad, se han ampliado las aplicaciones o contextos en que aparecen los polinomios. En años recientes, podemos mencionar por ejemplo, los polinomios cuya solución satisface relaciones de recurrencia lineal, o aplicaciones en cosmología, particularmente, polinomios que modelan el comportamiento de hoyos negros [7]. Y en nuestro caso, consideramos los polinomios sobre campos finitos, con los cuales, podemos entre otras cosas, generar códigos cíclicos [1], construir campos finitos, o expresar a los elementos de tales campos [1,2]. Una vez que hemos mencionado brevemente la importancia que tienen los polinomios, la siguiente tarea es abordarlos formalmente:

Del álgebra elemental, recordamos a un polinomio como una expresión de la forma $a_0 + a_1x + \cdots + a_sx^s$. Los a_i's son llamados coeficientes, y generalmente son números reales o complejos; x es visto como una variable: esto es, sustituyendo un número arbitrario α por x, un número bien definido $a_0 + a_1\alpha + \cdots + a_s\alpha^s$ es obtenido. La aritmética de polinomios es gobernada por reglas conocidas. El concepto de polinomio y las operaciones asociadas pueden ser generalizados a un ambiente algebraico formal:

Sea R un anillo arbitrario. Un *polinomio* sobre R es una expresión de la forma

$$f(x) = \sum_{i=0}^{s} a_i x^i = a_0 + a_1x + \cdots + a_sx^s,$$

donde s es un entero positivo, los *coeficientes* a_i, para $0 \leq i \leq s$, son elementos de R, y x es un símbolo no perteneciente a R, llamado la *indeterminada* sobre R. Cuando sabemos qué indeterminada se utiliza, podemos usar f para designar al polinomio $f(x)$. Se adopta la convención de que un término $a_i x^i$ con $a_i=0$, no necesita ser escrito en la expresión que representa al polinomio. En particular, el polinomio $f(x)$ anterior, puede representarse en la forma equivalente

$$f(x) = a_0 + a_1 x + \cdots + a_s x^s + 0x^{s+1} + \cdots + 0x^{s+t},$$

donde t es cualquier entero positivo. Cuando comparamos dos polinomios $f(x)$ y $g(x)$ sobre R, es posible asumir que ambos contienen las mismas potencias de x. Los polinomios

$$f(x) = \sum_{i=0}^{s} a_i x^i$$

y

$$g(x) = \sum_{i=0}^{u} b_i x^i$$

sobre R son considerados iguales si y sólo si $s = u$ y $a_i = b_i$ para $0 \leq i \leq s$. Definimos la *suma* de $f(x)$ y $g(x)$ como

$$f(x) + g(x) = \sum_{i=0}^{s'} (a_i + b_i)x^i.$$

donde s' es el entero mayor entre s y u.

Para definir el *producto* de dos polinomios sobre R, sean

$$f(x) = \sum_{i=0}^{s} a_i x^i$$

y

$$g(x) = \sum_{j=0}^{u} b_j x^j$$

y establecemos

$$f(x)g(x) = \sum_{k=0}^{s+u} c_k x^k,$$

donde $c_k = \sum_{i+j=k} a_i b_j$, para $0 \leq i \leq s,\ y\ 0 \leq j \leq u$.

Con las operaciones anteriores, el conjunto de polinomios sobre R forma un anillo:

Definición 2.3: El anillo formado por los polinomios sobre R con las operaciones anteriores es llamado el *anillo de polinomios* sobre R, y denotado por $R[x]$.

El elemento cero de $R[x]$ es el polinomio cuyos coeficientes son todos 0. Este polinomio es llamado el *polinomio cero* y denotado por 0. Según el contexto, se entiende a 0 como el elemento cero de R, ó 0 como el polinomio cero de $R[x]$.

Otros conceptos básicos sobre polinomios son cubiertos por la siguiente definición:

Definición 2.4: Sea $f(x) = \sum_{i=0}^{s} a_i x^i$ un polinomio sobre R, que no es el polinomio cero, por lo que podemos suponer que $a_s \neq 0$. Entonces a_s es llamado el *coeficiente líder* de $f(x)$ y a_0 el *término constante*, mientras s es llamado el *grado* de $f(x)$. En símbolos, s=*grado*$(f(x))$=*grado*(f). Por convención, establecemos que grado(0)= -∞. Los polinomios de grado ≤ 0 son llamados *polinomios constantes*. Si R tiene como identidad al 1 y si el coeficiente líder de $f(x)$ es 1, entonces $f(x)$ es llamado un *polinomio mónico*.

4. CAMPOS

Estructuras más ricas que los grupos y los anillos, respecto a las propiedades que presentan, son los campos. Basta con mencionar campos tales como los números racionales, los números reales, y los números complejos, para darnos una idea de su importancia. A continuación, la definición formal:

Definición 2.5: Un campo F es un conjunto no vacío, en donde están definidas dos operaciones $(\cdot, +)$, en el cual se cumplen los siguientes axiomas:

1. El conjunto F es un grupo abeliano bajo $(+)$.

2. F es cerrado bajo $(\cdot)$, y el conjunto de todos los elementos diferentes a cero forman un grupo abeliano bajo la multiplicación.

3. Para cada $a, b, c \in F$ se cumple la ley distributiva

$$a \cdot (b + c) = a \cdot b + a \cdot c$$

Conceptos importantes relacionados con campos son los siguientes:

El *orden de un campo* es el número de elementos en el campo. Si el orden es infinito, llamamos al campo un *campo infinito*; si

el orden es finito, llamamos al campo un *campo finito*.

Sea F un campo. Si un subconjunto K de F es en si mismo un campo bajo las operaciones de F, entonces éste es llamado un *subcampo* de F. En este contexto, a F se le llama una *extensión* de K. Si $K \neq F$, decimos que K es un *subcampo propio* de F.

Adicionalmente, sobre un campo F, existe un anillo de polinomios, denotado por $F[x]$. Entonces, ahora que sabemos qué es un campo y un anillo de polinomios sobre él, podemos establecer hechos adicionales a los ya presentados, para polinomios, como son la división de polinomios, la irreducibilidad, y la factorización:

Teorema 2.6. Algoritmo de División: Sea $g \neq 0$ un polinomio en $F[x]$. Entonces para cualquier $f \in F[x]$, existen polinomios c, $r \in F[x]$, tal que

$$f = cg + r, \ \textit{donde } grado(r) < grado(g).$$

c y r son únicos. El polinomio c es llamado el *cociente*, mientras que el polinomio r es llamado el *residuo*. El residuo r de la división se denota como $f \ mod \ g$, y el cociente c como $f \ div \ g$.

Los detalles del teorema anterior se encuentran en [2, capítulo 1, páginas 20-21].

Mientras que, para saber si un polinomio es o no irreducible, nos auxiliamos del siguiente hecho:

Definición 2.7: Un polinomio $f(x) \in F[x]$, es *irreducible sobre* F (o *irreducible en* $F[x]$), si $f(x)$ tiene grado positivo y siempre que $f(x) = b(x)e(x)$, con $b(x), e(x) \in F[x]$, implica que $b(x)$ ó $e(x)$, es un polinomio constante.

Brevemente, un polinomio de grado positivo es irreducible sobre F si este permite solamente factorizaciones triviales. Un polinomio en $F[x]$, de grado positivo que no es irreducible sobre F es llamado *reducible sobre F*. La reducibilidad o la irreducibilidad de un polinomio dado, depende del campo bajo consideración. Puede ser irreducible sobre un campo o campos, y reducible sobre otro u otros.

Otro aspecto importante a mencionar, en lo referente a polinomios, es el concepto de *raíz de un polinomio*.

Tenemos que, un elemento $b \in F$, es llamado una *raíz* (o un *cero*) del polinomio $f \in F[x]$, si $f(b)=0$.

Otro hecho básico es que, al ser $b \in F$, una raíz del polinomio $f \in F[x]$, resulta que $x - b$ debe dividir a $f(x)$. Para ahondar en este concepto, puede consultarse [2, capítulo 1, página 27].

Ahora, ya que sabemos lo que es la raíz de un polinomio, podemos diferenciar entre una raíz simple y una raíz múltiple:

Si $b \in F$ es una raíz del polinomio $f \in F[x]$, y k es un entero positivo tal que $f(x)$ es divisible por $(x-b)^k$, pero no por $(x-b)^{k+1}$, entonces k es llamado la *multiplicidad* de b. Si k=1, entonces b es llamado una *raíz simple* (o un *cero simple*) de f, y si $k \geq 2$, entonces b es llamado una *raíz múltiple* (o un *cero múltiple*) de f.

Extendiendo un poco los conceptos de raíces y multiplicidad, mencionemos nuevamente a $f \in F[x]$, con $grado(f) = s \geq 0$. Si $b_1,\ldots,b_m \in F$ son raíces distintas de f, con multiplicidades $k_1, \ldots, k_m$, respectivamente, entonces $(x-b_1)^{k_1} \cdots (x-b_m)^{k_m}$ divide $f(x)$. En consecuencia, $k_1+\cdots+k_m \leq s$, y f puede tener a lo más s raíces distintas en F.

Más acerca de este concepto, puede consultarse en [2, capítulo 1, página 27].

Igual de importante, es saber a qué le llamamos la *derivada* de un polinomio: Si $f(x)=a_0+a_1x+a_2x^2+\cdots+a_sx^s \in F[x]$, entonces la *derivada* f' de f se define como $f' = f'(x) = a_1 + 2\ a_2x + \cdots + sa_sx^{s-1} \in F[x]$.

Como consecuencia final de lo que hemos hablado, a partir del concepto de raíz de un polinomio, hasta este punto, tenemos el siguiente resultado:

Teorema 2.8: El elemento $b \in F$ es una raíz múltiple de $f \in F[x]$, si y sólo si dicho elemento es una raíz tanto de f como de f'.

El último concepto sobre polinomios a presentar en esta parte, es el teorema de factorización. Es uno de los resultados esenciales de dicha expresión algebraica.

Teorema 2.9: Factorización única en $F[x]$. Cualquier polinomio $f \in F[x]$ de grado positivo puede ser escrito en la forma

$$f = af_1^{e_1} \cdots f_k^{e_k} \tag{1}$$

donde $a \in F$, $f_1, \ldots, f_k$ son polinomios mónicos irreducibles distintos en $F[x]$, y $e_1, \ldots, e_k$ son enteros positivos. Esta factorización es única, salvo el orden de los factores. A (1) se le conoce como la *factorización canónica* del polinomio $f \in F[x]$.

Se omite la demostración del teorema anterior, la cual se puede consultar en [2, capítulo 1, página 24].

5. CAMPOS FINITOS

Los campos finitos, juegan un rol importante en teoría general de campos. A su vez, son utilizados en teoría numérica, teoría de grupos, y en geometría proyectiva [7]. También tienen aplicaciones prácticas, especialmente en el área de codificación, en comunicaciones digitales [2]. Antes de definirlos, vamos a mencionar un concepto general, que es esencial para campos, conocido como la característica:

Sea 1 el elemento identidad de un campo. La *característica* de un campo es 0 si al sumar l veces la unidad, esto es, $1+1+\cdots+1$, el resultado nunca es igual a 0, para cada $l \geq 1$. Es el caso de los campos infinitos. En caso contrario, la característica del campo es el entero positivo más pequeño l, tal que $\sum_{i=1}^{l} 1 = 0$. Y ya que l no es 0, entonces l es un número primo. Por lo que, en el caso de campos finitos, su característica es un número primo, que en vez de l, ahora, para particularizarlo, es conveniente denotarlo como p. La demostración de porqué la característica de un campo es 0 o un número primo, puede consultarse en [9, capítulo 2, página 9].

Ahora bien, para asegurarnos de que existan los campos finitos, debemos establecer el siguiente teorema:

Teorema 2.10: Existencia única de campos finitos. Para cada número primo p, y cada entero positivo n, existe un campo finito con $q = p^n$ elementos.

La demostración del teorema anterior se encuentra en [2, capítulo 2, página 46].

Otro aspecto importante de estos campos, es establecer bajo qué criterio podemos hablar de la existencia de subcampos. Esto se cubre con el siguiente teorema:

Teorema 2.11: Criterio del subcampo. Sea F_q el campo finito con $q = p^n$ elementos. Entonces cada subcampo de F_q tiene orden p^m, donde m es un divisor positivo de n. Inversa-

mente, si m es un divisor positivo de n, entonces existe exactamente un subcampo de F_q, que contiene p^m elementos.

La demostración de este teorema puede consultarse en [2, capítulo 2, página 47].

Un lema importante e interesante, es el siguiente, el cual relaciona polinomios, con los conceptos de campo finito y subcampo de un campo finito:

Lema 2.12: Si F es un campo finito con q elementos, y K es un subcampo de F, entonces el polinomio $x^q - x$ en $K[x]$, se factoriza en $F[x]$ como

$$x^q - x = \prod_{a \in F} (x - a),$$

y es un hecho conocido, que todos los elementos de F, son todas las raíces del polinomio $x^q - x \in K[x]$.

La demostración de este lema puede consultarse en [2, capítulo 2, página 46].

El hablar de subcampos de un campo finito, nos permite saber a qué le llamamos campo finito primo y campo finito no primo:

Definición 2.13: Un campo finito que no contiene subcampos propios es llamado un *campo finito primo*.

En este caso, $n = 1$, por lo que $q = p$. Entonces, podemos denotar al campo finito primo como F_p, para diferenciarlo de un campo finito no primo, que denotaremos como F_q.

Un ejemplo clásico de este tipo de campo, son los enteros *modulo* un primo p, denotado por Z_p.

Por medio de p, podemos definir las siguientes operaciones en el campo finito primo F_p: para cada a,$b \in F_p$,

$$a + b := (a + b)\ mod\ p,$$

y

$$a \cdot b := (a \cdot b)\ mod\ p,$$

Donde $(a + b)\ mod\ p$ y $(a \cdot b)\ mod\ p$, indican que después de efectuar la operación correspondiente, el resultado de ella se divide por p, y se toma como resultado final el residuo entero de tal división.

Los p elementos de F_p, son los enteros $mod\ p$, es decir, { 0,1,2,...,p-1 }.

De la definición 2.13, vemos que toda *extensión* de un campo finito primo es un *campo finito no primo* F_q. En este caso, $n > 1$, y la extensión contiene $q = p^n$ elementos.

La existencia de tales campos, es formalmente establecida por el siguiente teorema:

Teorema 2.14: Supóngase que $f(x)$ es un polinomio irreducible de grado n, sobre F_p. Entonces, el conjunto de todos los polinomios en x, de grado $\leq n-1$, cuyos coeficientes pertenecen a F_p, con cálculos ejecutados $mod\ f(x)$, forman un campo de orden $q = p^n$.

los detalles del teorema anterior, se encuentran en [1, capítulo 4, página 94].

Del teorema anterior, vemos que los q elementos de F_q, que son los enteros $mod\ q$, es decir, { 0,1,2,. . .,q-1 }, pueden ser representados de diversas formas, por ejemplo, como polinomios. Gracias a dicha representación, se puede establecer la aritmética en estos campos. Si dos elementos a, $b \in F_q$, son representados en forma polinomial como $a(x)$, $b(x) \in F_p[x]$, respectivamente, entonces podemos definir las operaciones

$$a(x) + b(x) := (a(x) + b(x))\ mod\ f(x),$$

y

$$a(x) \cdot b(x) := (a(x) \cdot b(x)) \ mod \ f(x),$$

las cuales se llevan a cabo, mediante lo establecido en el teorema 2.6.

Finalmente, debemos reiterar que, las definiciones, explicaciones y teoremas presentados en este capítulo, son sólo una breve introducción a estructuras algebraicas, tal como se haría en un libro introductorio sobre algebra moderna, o abstracta.

Capítulo 3

Algoritmo de Berlekamp para factorización de polinomios sobre campos finitos

Vimos en el Capítulo 2, que cualquier polinomio no constante sobre un campo, puede ser expresado como el producto de polinomios irreducibles. En el caso particular de campos finitos, la disponibilidad de algoritmos veloces y eficientes para la factorización de polinomios es un asunto importante. En dicho caso, podemos mencionar, de entre las aplicaciones mencionadas en el capítulo 1, aplicaciones en Teoría de la información [1], y en relaciones de recurrencia lineal [4]. Y viendo mas allá del mundo de los campos finitos, existen varios problemas computacionales en Álgebra y Teoría de números, que dependen de una u otra forma de la factorización de polinomios sobre campos finitos.

En Matemáticas, de manera particular en el área del Álgebra Computacional, el algoritmo de Berlekamp es un método popular para factorización de polinomios sobre campos finitos; dicho algoritmo consiste pricipalmente de construir y reducir matrices, y del cálculo del máximo común divisor entre polinomios. Este algoritmo es eficiente o está mejor adaptado para aplicarse sobre campos finitos pequeños, como se verá más adelante en este capítulo.

Para cubrir la revisión total del algoritmo en cuestión, el orden

de este capítulo es el siguiente: primeramente, presentamos las herramientas matemáticas que utiliza el algoritmo de factorización de Berlekamp, para cumplir con su objetivo; posteriormente, explicamos el algoritmo, y como éste utiliza dichas herramientas a cada paso de su ejecución. Y al final de este capítulo, presentamos un ejemplo de factorización.

1. HERRAMIENTAS MATEMÁTICAS

El algoritmo de Berlekamp para factorización de polinomios sobre campos finitos, utiliza operaciones del álgebra lineal elemental, como la resta de matrices, o las operaciones sobre renglones y columnas de una matriz, para llevarla a su forma escalonada, y hallar la solución de ella. Estas operaciones son las más elementales que se llevan a cabo dentro del algoritmo. Aunadas a las operaciones mencionadas, existen 2 herramientas matemáticas un poco más sofisticadas, las cuales son utilizadas de manera principal en el desarrollo del algoritmo: primeramente, el algoritmo de Euclides, en su versión para polinomios, el cual es una variante del algoritmo clásico de Euclides que calcula el máximo común divisor de dos números enteros. Antes de abordar tal algoritmo, veamos unos conceptos previos necesarios:

Si $f_1,\ldots,f_s$ son polinomios en $F[x]$, no todos 0, entonces debe existir un único polinomio mónico $d \in F[x]$, de mayor grado, con las siguientes propiedades: (i) d divide a cada f_j, para $1 \leq j \leq s$; (ii) cualquier polinomio $e \in F[x]$ que divide a cada f_j, para $1 \leq j \leq s$, divide a d. A dicho polinomio d, lo podemos expresar en la forma

$$d = b_1 f_1 + \cdots + b_s f_s,$$

con

$$b_1, \ldots, b_s \in F[x].$$

El polinomio d es llamado el *máximo común divisor* de $f_1,\ldots,f_s$, escrito de la forma $d = mcd(f_1, \ldots, f_s)$. Si $mcd(f_1,\ldots,f_s)=1$, entonces los polinomios $f_1,\ldots,f_s$ son llamados *primos relativos*. Ellos son llamados *primos relativos en pares* si $mcd(f_i,f_j)=1$ para $1 \leq i < j \leq s$.

Ahora bien, si necesitamos calcular el máximo común divisor de dos polinomios $f, g \in F[x]$, lo podemos hacer mediante el *algoritmo de Euclides*:

Teorema 3.1. Algoritmo de Euclides para polinomios en $F[x]$: Dados polinomios f , $g \in F[x]$, con $grado(g) \leq grado(f)$; supongamos, sin pérdida de generalidad, que $g \neq 0$ y que g no divide a f. Entonces podemos utilizar repetidamente el algoritmo de la división, presentado en el teorema 2.6 del capítulo 2, de la siguiente forma:

$$\begin{aligned}
f &= c_1 g + r_1 && 0 \leq grado(r_1) < grado(g) \\
g &= c_2 r_1 + r_2 && 0 \leq grado(r_2) < grado(r_1) \\
r_1 &= c_3 r_2 + r_3 && 0 \leq grado(r_3) < grado(r_2) \\
&\vdots && \vdots \\
r_{v-2} &= c_v r_{v-1} + r_v && 0 \leq grado(r_v) < grado(r_{v-1}) \\
r_{v-1} &= c_{v+1} r_v. &&
\end{aligned}$$

Aquí $c_1, \ldots, c_{v+1}$ y $r_1, \ldots, r_v$ son polinomios en $F[x]$. Dado que el grado de g es finito, el procedimiento tiene que detenerse después de un número finito de pasos. Si el último residuo distinto a cero r_v, tiene *coeficiente líder* b, entonces $mcd(f, g)$=$b^{-1}r_v$. A fin de encontrar $mcd(f_1, \ldots, f_s)$ para $s > 2$ y polinomios f_i distintos a cero, primero se calcula $mcd(f_1, f_2)$, después $mcd(mcd(f_1,f_2),f_3)$, y así sucesivamente, por el algoritmo de Euclides.

Se omite la demostración del teorema anterior, la cual puede consultarse en [1, capítulo 12, páginas 362-363].

La segunda herramienta matemática es el Teorema chino del residuo para polinomios en $F[x]$, el cual es la variante del Teorema chino del residuo para enteros positivos:

Teorema 3.2. Teorema chino del residuo para polinomios en $F[x]$: Dado un campo F, polinomios distintos a cero f_1, f_2, $\ldots$, $f_k \in F[x]$ que son *primos relativos en pares*, y polinomios arbitrarios g_1, g_2, $\ldots$, $g_k \in F[x]$, entonces las congruencias simultáneas

$$h \quad \equiv \quad g_i \ mod \ f_i,$$

para $i = 1, 2, \ldots, k$, tienen una solución única $h(\in F[x])$ mod $f = f_1 \cdots f_k$.

Se omite la demostración de este teorema, la cual se encuentra en [5, capítulo 2, página 29].

Una vez presentadas las herramientas matemáticas que nos interesan en este capítulo, pasamos a la descripción detallada de las ideas que involucran la tarea de factorización, dentro del algoritmo de Berlekamp.

2. EL ALGORITMO DE FACTORIZACIÓN DE BERLEKAMP

Recordando, F denota un campo, y $F[x]$ es el anillo de polinomios sobre F. Si particularizamos lo anterior a un campo finito en general, es decir, campo finito primo o campo finito no primo, tenemos que F_q denota un campo finito de orden q, y $F_q[x]$ es el anillo de polinomios sobre F_q. Y sabemos que cualquier polinomio $f \in F_q[x]$ de grado positivo, tiene una factorización canónica en $F_q[x]$, por el teorema 2.9 del capítulo 2. Para algoritmos de factorización es suficiente considerar solamente polinomios mónicos. Entonces nuestro objetivo es expresar un polinomio mónico $f \in F_q[x]$ de grado positivo en la forma

$$f = f_1^{e_1} \cdots f_k^{e_k}, \tag{2}$$

donde $f_1, \ldots, f_k$ son polinomios monicos irreducibles distintos en $F_q[x]$ y $e_1, \ldots, e_k$ son enteros positivos.

Primero, vamos a simplificar la tarea de factorización, haciendo ver que el problema puede reducirse a factorizar un polinomio con *factores no repetidos,* lo que nos indica que los ex-

ponentes $e_1, \ldots, e_k$ en (2) deben ser iguales a 1 (o, de manera equivalente, que el polinomio no tiene raíces múltiples). Para este fin, vamos a calcular

$$d(x) = mcd(f(x), f\,'(x)),$$

el máximo común divisor de $f(x)$ y su derivada $f\,'(x)$, por medio del algoritmo de Euclides.

Realizamos el cálculo de $d(x)$, ya que esto nos va a permitir utilizar el teorema 2.8, como criterio de identificación, entre los diversos casos de factorización que pueden aparecer:

Caso 1: Si $d(x) = 1$, entonces sabemos que $f(x)$ no tiene factores repetidos, precisamente, por lo dicho en el teorema 2.8. Este caso se aborda a detalle en el ejemplo que se encuentra al final de este capítulo.

En caso contrario, cuando $d(x) \neq 1$, se presentan dos casos más:

Caso 2: Si $d(x) = f(x)$, debemos tener $f\,'(x) = 0$. Escribimos a $f(x)$ como $f(x) = g(x)^p$, para algún polinomio $g(x)$ en $F_q[x]$, y p es la característica de F_q. Si es necesario, el proceso de reduccción debe continuar, ahora aplicando el método a $g(x)$.

Para esclarecer este caso, veamos un ejemplo. Ya que el

aporte más importante de nuestro trabajo, es el operar en extensiones de campos finitos, entonces, para este ejemplo, utilizamos el polinomio $f(x)= x^3+3$, sobre el campo F_9, el cual es una extensión del campo finito primo F_3. Lo primero es, calcular la derivada $f\,'(x)$, y después el máximo común divisor entre $f(x)$ y $f\,'(x)$. Tenemos:

$$f\,'(x) = 0,$$

$$y$$

$$d(x) = mcd(f(x), f\,'(x)) = mcd(x^3+3, 0) = x^3+3.$$

Entonces, $d(x) = x^3+3 = f(x)$. Vemos que $f(x)$ puede ser escrito como $f(x)=(x+8)^3$. De aquí, identificamos a $g(x)= x+8$, y p=3. De ser necesario, se aplica recursivamente el proceso de reducción a $g(x)$, pero en este caso, $g(x)$ es irreducible, por lo que el proceso de reducción es simplemente escribir p veces $g(x)$:

$$f(x) = (x+8)(x+8)(x+8).$$

Caso 3: Si $d(x) \neq 1$ y $d(x) \neq f(x)$, entonces $d(x)$ es un factor no trivial de $f(x)$. Y $f(x)/d(x)$ no tiene factores repetidos. / denota división. La factorización de $f(x)$ se lleva a cabo

factorizando $d(x)$ y $f(x)/d(x)$ separadamente. En caso de que $d(x)$ aún tenga factores repetidos, deben realizarse aplicaciones adicionales del proceso de reducción.

Por ejemplo, sea el polinomio $f(x)=x^4 + x \in F_9[x]$. Al calcular $f\,'(x)$ y $d(x)$, obtenemos:

$$f\,'(x) = x^3 + 1,$$

$$y$$

$$d(x) = mcd(f(x), f\,'(x)) = mcd(x^4 + x, x^3 + 1) = x^3 + 1.$$

Notamos que $d(x) \neq 1$ y $d(x) \neq f(x)$, y efectivamente, $d(x)$ es factor no trivial de $f(x)$. Ahora, hay que factorizar $d(x)$ y $f(x)/d(x)$, por separado:

Primero, $f(x)/d(x)= x^4 + x/x^3 + 1=x$. Con esto concluye la factorización de $f(x)/d(x)$, ya que x es irreducible, y además se comprueba que $f(x)/d(x)$ no tiene factores repetidos. Ahora, para factorizar $d(x)$, vemos que puede ser escrito como $d(x)=x^3+ 1=(x + 1)^3$, donde identificamos el caso 2 anterior, es decir, $d(x)=g(x)^p=(x + 1)^3$. Entonces, $g(x)$ es irreducible, y aplicar recursivamente el proceso de reducción a $d(x)$, es simplemente escribirlo como p veces $g(x)$:

$$d(x) = (x+1)(x+1)(x+1).$$

Unimos las factorizaciones obtenidas por separado, y finalmente obtenemos

$$f(x) = x(x+1)^3.$$

De los casos anteriores, podemos ver que, al aplicar este proceso de reducción un número suficiente de veces, el problema original se reduce a factorizar un cierto número de polinomios con factores no repetidos. Y las factorizaciones canónicas de tales polinomios nos darán la factorización canónica del polinomio original. Por lo tanto, debemos centrar nuestra atención en polinomios con factores no repetidos. Para ello, presentamos el siguiente teorema, que además de ser un resultado clave, es una primera aproximación para obtener una factorización de f:

Teorema 3.3: Si $f \in F_q[x]$ es mónico, y $h \in F_q[x]$ es tal que $h^q \equiv h \ mod \ f$, entonces

$$f(x) = \prod_{c \in F_q} mcd(f(x), h(x) - c). \tag{3}$$

Vamos a analizar el resultado anterior. En el lado derecho de (3), los polinomios $h(x) - c$, con $c \in F_q$, son primos relativos

en pares. Esto implica, que al no compartir factores entre sí, podemos tomar a cada uno de ellos, junto con $f(x)$, para hallar cada factor de $f(x)$, mediante el cálculo del máximo común divisor entre ellos. Entonces, cada máximo común divisor del lado derecho de (3), divide a $f(x)$.

Ahora, veamos porqué $f(x)$, del lado izquierdo de (3), divide el lado derecho. Recordando el lema 2.12, si en la identidad de dicho lema, hacemos un cambio de variable, es decir, si $x = h(x)$, y $a = c$, podemos expresar dicha identidad como

$$h(x)^q - h(x) = \prod_{c \in F_q} (h(x) - c), \tag{4}$$

y el teorema 3.3 nos dice que $h^q \equiv h \ mod \ f$, y sabemos, de la definición de congruencia, que esto implica que f divide a $h^q - h$. De aquí vemos que $f(x)$ divide a (4). Por lo tanto, $f(x)$ divide el lado derecho de (3). Tenemos entonces que, los dos lados de (3), son polinomios mónicos que se dividen uno al otro, y por lo tanto son iguales.

Del teorema anterior, podemos ver, en general, que (3) no nos da la factorización completa de f, dado que $mcd(f(x), h(x) - c)$ puede ser reducible en $F_q[x]$. Para ver esto, observemos que si $h(x) \equiv c \ mod \ f(x)$ para algún $c \in F_q$, entonces el Teorema 3.3 nos da una factorización trivial de f y por lo tanto no es útil. Es decir, tal factorización trivial ocurre, cuando $mcd(f(x), h(x) - c\,') = f(x)$, para $c\,' = c$, o bien, $mcd(f(x), h(x) - c\,') = 1$, si $c\,' \neq c$. Sin embargo, si h es tal, que el Teorema 3.3 nos da una

factorización no trivial de f, decimos entonces que h es un *polinomio f-reducible.* Observe que cualquier h con $h^q \equiv h \ mod \ f$ y $0 < grado(h) < grado(f)$ es *f-reducible.* El hecho de que $0 < grado(h)$, es el que nos indica que h es reducible, y la factorización es no trivial. Entonces, si h no es *f-reducible*, puede ser que, si utilizamos directamente el teorema 3.3, obtengamos factorizaciones triviales. Por lo que, es conveniente utilizar la idea que nos da dicho teorema, pero debemos ampliarla, de tal manera, que incluya el obtener polinomios *f-reducibles.* Entonces, para hallar algoritmos de factorización basados en el Teorema 3.3, debemos encontrar métodos para construir polinomios *f-reducibles.* Para ello, entonces, nos basamos en la ecuación (3), pero debemos tomar en cuenta que dicha factorización depende del cálculo de q máximos comunes divisores, por lo que una aplicación directa de tal fórmula solo es viable para campos finitos pequeños F_q.

Entonces, para ampliar la idea del teorema 3.3, en aras de tener un método de construcción de polinomios *f-reducibles*, el algoritmo de Berlekamp, el cual será el método que vamos a utilizar, hace uso del Teorema chino del residuo para polinomios. Asumimos que f no tiene factores repetidos, de manera tal que

$$f = f_1 \cdots f_k,$$

es un producto de polinomios mónicos irreducibles distintos sobre F_q. Si la expresión $(c_1, \ldots, c_k)$, representa a cualquier con-

junto de k elementos de F_q, el Teorema chino del residuo implica que existe un único $h \in F_q[x]$ con $h(x) \equiv c_i \ mod \ f_i(x)$ para $1 \leq i \leq k$ y $grado(h) < grado(f)$.

Entonces, el polinomio $h(x)$ que buscamos, debe satisfacer la condición

$$h(x)^q \equiv c_i^q = c_i \equiv h(x) \ mod \ f_i(x),$$

para $1 \leq i \leq k$.

En la expresión anterior, la igualdad central, $c_i^q = c_i$, es una propiedad importante que cumple todo elemento $c_i \in F_q$, y dicha proiedad se enuncia en detalle en un resultado ampliamente conocido, llamado el *Teorema de Fermat* [1, capítulo 4, página 96], y que en este caso, nos sirve para reforzar la congruencia anterior, la cual debe satisfacer el polinomio $h(x)$.

y por lo tanto, al satisfacer la condición anterior, h satisface la congruencia

$$h^q \equiv h \ mod \ f, \ con \ grado(h) < grado(f). \tag{5}$$

Por otro lado, si con la condición anterior, aseguramos que h es una solución de (5), entonces la identidad

$$h(x)^q - h(x) = \prod_{c \in F_q} (h(x) - c)$$

la cual se desprende del lema 2.12, implica que cada factor irreducible de f divide uno de los polinomios $h(x) - c$, por el análisis que hicimos del teorema 3.3, ya que en dicho teorema, vimos porqué f divide a la identidad anterior, y ahora, viendo a f como producto de sus factores f_i, cada uno de ellos seguirá cumpliendo tal condición. Entonces, todas las soluciones de (5) satisfacen $h(x) \equiv c_i \ mod \ f_i(x)$, con $1 \leq i \leq k$, para algún conjunto $(c_1, \ldots, c_k)$ de k elementos de F_q. Y puesto que tomamos k elementos, de un total de q, el máximo de combinaciones posibles es q^k. En consecuencia, existen exactamente q^k soluciones de (5).

Ahora, para encontrar esas soluciones, debemos reducir (5) a un sistema de ecuaciones lineales. Para ello, construimos una matriz, que aloje dicho sistema. Entonces, con $s = grado(f)$, construimos la matriz $B = (b_{ij})$ de $s \times s$, con $0 \leq i, j \leq s-1$, donde b_{ij} es un elemento en F_q, y la entrada (i, j) de B, es precisamente el elemento correspondiente b_{ij}. Para construir B, calculamos las potencias $x^{iq} \ mod \ f(x)$, las cuales son, de manera específica

$$x^{iq} \equiv \sum_{j=0}^{s-1} b_{ij} x^j \ mod \ f(x), \ con \ 0 \ \leq i \leq s-1. \qquad (6)$$

Para esclarecer como se debe formar el sistema de ecuaciones

lineales que forma a la matriz B, y que da solución a (5), veamos un caso específico. Por ejemplo, si tenemos el polinomio $f(x)=x^8+x^6+x^4+x^3+1 \in F_2[x]$, cuyo grado es $s=8$. Entonces, $q=2$. Para hallar las ecuaciones solución, que a su vez, formen la matriz B, tenemos:

para $i=0$: $x^{iq}=x^0$. Y calculamos $x^{iq}\ mod\ f(x)$, de acuerdo a lo descrito en el teorema 2.6. Entonces, $x^{iq}\ mod\ f(x)=1\ mod\ f(x)=1$. Este resultado, corresponde al lado derecho de (6), puesto que, si desarrollamos ese lado, tenemos, para $i=0$:

$$\sum_{j=0}^{s-1} b_{ij}x^j\ mod\ f(x) =$$

$$(b_{0,0}x^0+b_{0,1}x+b_{0,2}x^2+b_{0,3}x^3+b_{0,4}x^4+b_{0,5}x^5+b_{0,6}x^6+b_{0,7}x^7)\ mod\ f(x).$$

pero, $x^{iq}\ mod\ f(x)=1$, entonces, del desarrollo anterior, el único coeficiente $b_{ij}=1$, será $b_{0,0}$, los coeficientes b_{ij} restantes valen cero. Por lo que la congruencia en (6), para $i=0$, es

$$x^0 \equiv b_{0,0}x^0\ mod\ f(x) = 1\ mod\ f(x) = 1.$$

Ahora, para $i=1$, utilizamos el mismo procedimiento: $x^{iq}=x^2$. Y $x^{iq}\ mod\ f(x)=\ x^2\ mod\ f(x)=x^2$. De la sumatoria del lado

derecho de (6), el único coeficiente $b_{ij} \neq 0$, es $b_{1,2}$, coeficiente que corresponde al término x^j=x^2. Entonces, para i=1, (6) es

$$x^2 \equiv b_{1,2}x^2 \ mod \ f(x) = x^2 \ mod \ f(x) = x^2.$$

para i=2: x^{iq}=x^4. Y $x^{iq} \ mod \ f(x) = x^4 \ mod \ f(x)$= x^4. Con b_{ij}=1, para x^j=x^4, y la congruencia correspondiente es

$$x^4 \equiv b_{2,4}x^4 \ mod \ f(x) = x^4 \ mod \ f(x) = x^4.$$

para i=3: x^{iq}=x^6. Y $x^{iq} \ mod \ f(x) = x^6 \ mod \ f(x)$= x^6. Con b_{ij}=1, para x^j=x^6, y la congruencia correspondiente es

$$x^6 \equiv b_{3,6}x^6 \ mod \ f(x) = x^6 \ mod \ f(x) = x^6.$$

para i=4: x^{iq}=x^8. Y $x^{iq} \ mod \ f(x) = x^8 \ mod \ f(x)$= $1+x^3+x^4+x^6$. Ahora, b_{ij}=1, para los términos 1, x^3, x^4 y x^6, del lado derecho de (6), correspondientes a j=0, 3, 4, 6, respectivamente. entonces, la congruencia en (6), es

$$x^8 \equiv (b_{4,0}x^0+b_{4,3}x^3+b_{4,4}x^4+b_{4,6}x^6) \ mod \ f(x) = 1+x^3+x^4+x^6.$$

Completando el proceso, para los valores restantes de i, que son i= 5,6,7, respectivamente, tenemos

$$x^{10} \equiv 1 + x^2 + x^3 + x^4 + x^5,$$

$$x^{12} \equiv x^2 + x^4 + x^5 + x^6 + x^7,$$

$$x^{14} \equiv 1 + x + x^3 + x^4 + x^5.$$

Acomodando todas las congruencias obtenidas, tenemos el sistema:

$$\begin{array}{lcccccccc}
x^0 & \equiv & 1 & & & & & & \\
x^2 & \equiv & & & x^2 & & & & \\
x^4 & \equiv & & & & & x^4 & & \\
x^6 & \equiv & & & & & & & x^6 \\
x^8 & \equiv & 1 & & & +x^3 & +x^4 & & +x^6 \\
x^{10} & \equiv & 1 & & +x^2 & +x^3 & +x^4 & +x^5 & \\
x^{12} & \equiv & & & x^2 & & +x^4 & +x^5 & +x^6 \quad +x^7 \\
x^{14} & \equiv & 1 & +x & & +x^3 & +x^4 & +x^5 &
\end{array}$$

cuyos términos aparecen ordenados en columnas, de acuerdo a las potencias de x. Entonces, cada renglón de B, se forma con

todos los coeficientes b_{ij}, ya sean cero, o no, de la congruencia correspondiente:

$$B = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 \end{bmatrix}$$

Ahora bien, la matriz B debe ser tal que:

$$h(x) = a_0 + a_1 x + \cdots + a_{s-1} x^{s-1} \in F_q[x]$$

es una solución de (5) si y sólo si

$$(a_0, a_1, \ldots, a_{s-1})B = (a_0, a_1, \ldots, a_{s-1}). \tag{7}$$

Es decir, el vector de coeficientes de $h(x)$, satisface (7). Para que (7) sea válido, debe ser escrito en términos tanto de h como de B, de la siguiente manera: Escribimos a $h(x)$ como $h(x) = \sum_{j=0}^{s-1} a_j x^j$, y al producto $h(x)B$, como $h(x)B = \sum_{j=0}^{s-1} \sum_{i=0}^{s-1} a_i b_{ij} x^j$, tal que, ahora, incluyendo al término en x, podemos rescribir (7) como

$$h(x)B = h(x),$$

es decir,

$$\sum_{j=0}^{s-1}\sum_{i=0}^{s-1} a_i b_{ij} x^j = \sum_{j=0}^{s-1} a_j x^j,$$

y el escribir el sistema (7) de esta forma, no sólo es válido, sino que además, se sigue cumpliendo la congruencia en (5), ya que, si escribimos el polinomio $h(x)^q$, como $h(x)^q = \sum_{i=0}^{s-1} a_i x^{iq}$, entonces, podemos rescribir a (5) como:

$$\begin{aligned} h(x) &= \sum_{j=0}^{s-1} a_j x^j \\ &= \sum_{j=0}^{s-1}\sum_{i=0}^{s-1} a_i b_{ij} x^j \\ &\equiv \sum_{i=0}^{s-1} a_i x^{iq} = h(x)^q \ mod\ f(x). \end{aligned}$$

Ahora bien, el sistema (7) puede ser escrito de manera equivalente, como

$$(a_0, a_1, \ldots, a_{s-1})(B - I) = (0, 0, \ldots, 0), \tag{8}$$

Donde I es la matriz identidad de $s \times s$ sobre F_q. Por las consideraciones anteriores, el sistema (8) tiene q^k soluciones. Es decir, las mismas q^k soluciones de (5). Entonces, *la dimensión del espacio nulo de la matriz B-I es k*, la cual es el número de vectores que forman una base para el espacio nulo de la matriz $B - I$, y que a su vez, son el número de factores mónicos irreducibles distintos de f, y *el rango de B-I es s-k.*

Dado que el polinomio constante $h_1(x) = 1$ siempre es una solución de (5), el vector $(1, 0, \ldots, 0)$ es siempre una solución de (8), la cual puede ser verificada directamente. Esto es así, ya que el primer renglón de la matriz B siempre es (1,0,. . .,0). Veamos porqué: para calcular la matriz B, utilizamos x^{iq} mod $f(x)$, con $0 \leq i \leq s-1$. Entonces, para obtener el primer renglón de B, tenemos i=0; para este caso, x^{iq} mod $f(x)$=1 mod $f(x)$=1, que, expresado como vector es (1,0,. . .,0). Teniendo lo anterior en mente, y sustituyendo en (8), el vector correspondiente a $h_1(x)$=1, el cual es (1,0,. . .,0), tenemos en (8) que

$$(1, 0, \ldots, 0)(B - I) = (0, 0, \ldots, 0).$$

Entonces, deben existir polinomios $h_2(x), \ldots, h_k(x)$ de grado $\leq s - 1$, tal que los vectores correspondientes a $h_1(x)$,$h_2(x)$, . . . ,$h_k(x)$, forman una base para el espacio nulo de $B - I$. Los polinomios $h_2(x), \ldots, h_k(x)$ tienen grado positivo y entonces son

f-reducibles.

En este enfoque, un rol importante es jugado por la determinación del rango $\Re$ de la matriz $B - I$. Tenemos $\Re = s - k$, como se notó anteriormente, de tal forma que una vez que el rango $\Re$ es encontrado, entonces sabremos que *el número de factores mónicos irreducibles distintos de f está dado por $s - \Re$.* Tomando como base esta información, podemos decidir, cuándo detener el procedimiento de factorización.

Hasta aquí, hemos visto cómo se forma el sistema que necesitamos resolver, para obtener los polinomios *f-reducibles*, que, eventualmente, nos darán la factorización deseada. Ahora, necesitamos decidir cómo resolver dicho sistema. Para ello, la opción a utilizar, no es otra mas que emplear operaciones elementales sobre matrices. Entonces, el rango de $B - I$ puede ser determinado, utilizando operaciones en los renglones y las columnas, para reducir la matriz a su forma escalonada. Sin embargo, dado que nosotros queremos resolver el sistema (8), es recomendable utilizar solamente operaciones sobre las columnas, ya que ellas nos dejarán el espacio nulo invariante. Entonces, nos permitimos multiplicar cada columna de la matriz $B - I$, por un elemento diferente de cero, del campo finito F_q, y sumar cada múltiplo de una de esas columnas a una columna distinta. El rango $\Re$ es el número de columnas distintas de cero, en la forma escalonada en las columnas, de la matriz $B - I$.

Habiendo encontrado $\Re$, formamos $k = s - \Re$. Si $k = 1$, sabemos que f es irreducible sobre F_q, y el procedimiento termina. En este caso, las únicas soluciones de (5) serán los polinomios

constantes, y el espacio nulo de $B - I$ contiene únicamente los vectores de la forma $(c, 0, \ldots, 0)$ con $c \in F_q$. Si $k \geq 2$, tomamos de la base polinomial, el polinomio *f-reducible* $h_2(x)$, y calculamos $mcd(f(x), h_2(x) - c)$ para todo $c \in F_q$. El resultado puede ser una factorización no trivial de $f(x)$ proporcionada por (3). Si el uso de $h_2(x)$ no tiene éxito en separar $f(x)$ en k factores, calculamos $mcd(g(x), h_3(x) - c)$ para todo $c \in F_q$, y para todos los factores no triviales $g(x)$ hallados hasta aquí. Esto es así, porque, si $g(x)$ es factor no trivial de $f(x)$, entonces $g(x)|f(x)$, y como $f(x)|h(x)^q - h(x)$, esto implica que $g(x)|h(x)^q - h(x)$, es decir, $h(x)^q \equiv h(x)\ mod\ g(x)$, y, por lo tanto, $g(x) = \prod_{c \in F_q} mcd(g(x), h(x) - c)$. Por lo que, se garantiza que se cumpla (3), pero ahora, para cada factor no trivial $g(x)$, de $f(x)$. Este prodecimiento es continuado hasta que k factores de $f(x)$ son obtenidos.

Un punto importante a destacar, es que, el proceso descrito anteriormente, nos debe dar todos los factores buscados, pues cada dos factores mónicos irreducibles distintos de $f(x)$, podrán ser separados por algún polinomio h_j, con $1 \leq j \leq k$. Aclaremos este punto. Vamos a considerar, como hemos dicho aquí, dos factores mónicos irreducibles distintos de $f(x)$, llamados $f_1(x)$ y $f_2(x)$. Siguiendo la idea de (5), deben existir elementos c_{j1}, $c_{j2} \in F_q$, tal que $h_j(x) \equiv c_{j1}\ mod\ f_1(x)$, y $h_j(x) \equiv c_{j2}\ mod\ f_2(x)$, con $1 \leq j \leq k$. Es decir, con un polinomio $h_j(x)$, y con dos elementos distintos en F_q, explícitamente, $c_{j1} \neq c_{j2}$, obtendremos dos factores distintos de $f(x)$. En otras palabras, con $h_j(x)$ y c_{j1}, obtenemos $f_1(x)$, y con $h_j(x)$ y c_{j2}, obtenemos $f_2(x)$. Por lo que jamás tendremos $c_{j1} = c_{j2} = c\,'$, tal que $f_1, f_2 | h_j - c\,'$, o bien,

$h_j \equiv c\ 'mod(f_1, f_2)$, para toda j. Aunado a lo anterior, sabemos que, siempre tendremos $h_1(x)$=1, como solución trivial, por lo que debemos tener como mínimo, otro polinomio $h_j(x)$, es decir, $j \geq 2$.

El siguiente ejemplo nos ayudará a entender los conceptos anteriores: si factorizamos el polinomio $f(x)=x^8+x^6+x^4+x^3+1$ sobre F_2, tenemos:

$$f(x) = f_1(x)f_2(x) = (x^6 + x^5 + x^4 + x + 1)(x^2 + x + 1),$$

y como polinomios $h(x)$, a $h_1(x)$=1, y a $h_2(x) = x + x^2 + x^5 + x^6 + x^7$. Nos enfocamos en $h_2(x)$, ya que $h_1(x)$ es la solución trivial.

Además, c_{j1}=0, y c_{j2}=1, con $c_{j1}, c_{j2} \in F_2$, y $c_{j1} \neq c_{j2}$.

De los hechos anteriores, tenemos:

a)

$$f_1 | h_2 - c_{j1} \Rightarrow x^6 + x^5 + x^4 + x + 1 | x^7 + x^6 + x^5 + x^2 + x,$$

y f_1 no divide a $h_2 - c_{j2}$.

b)

$$f_2|h_2 - c_{j2} \Rightarrow x^2 + x + 1|x^7 + x^6 + x^5 + x^2 + x + 1,$$

y f_2 no divide a $h_2 - c_{j1}$.

Vemos entonces que, con un polinomio $h_j(x)$, en este caso, $h_2(x)$, y dos elementos distintos en F_2, que son 0 y 1, obtenemos dos factores mónicos irreducibles distintos de $f(x)$, que para este ejemplo, son los únicos factores de $f(x)$.

Todos los aspectos descritos hasta aquí, los cuales forman parte de un algoritmo de factorización, basado en determinar polinomios *f-reducibles*, para resolver el sistema (8), es el algoritmo de Berlekamp.

Recapitulando, el desarrollo del algoritmo de factorización propuesto, consiste en:

teniendo como entrada del algoritmo, a $f(x) \in F_q[x]$, un polinomio mónico, de grado s, utilizar el criterio del teorema 2.8, mediante $d(x) = mcd(f(x), f\,'(x))$, con el propósito de distinguir entre los diversos casos de factorización; posteriormente, efectuamos los siguientes pasos del algoritmo de Berlekamp:

1. Para cada i, con $0 \leq i \leq s-1$, calcúlese

$$x^{iq} \equiv \sum_{j=0}^{s-1} b_{ij} x^j \ mod\ f(x),$$

$$para\ 0 \leq i \leq s-1.$$

Notemos que cada b_{ij} es un elemento de F_q.

2. Fórmese la matriz B de $s \times s$, cuya entrada (i, j) es b_{ij}.

3. Hallar una base $h_1(x), h_2(x), \ldots, h_k(x)$ para el espacio nulo de la matriz $B - I$, mediante la determinación de $\Re$. Entonces, el número de factores irreducibles de $f(x)$ será $s - \Re = k$.

4. Si k=1, sólo tenemos $h_k(x)$=$h_1(x)$=1, por lo que $f(x)$ es irreducible sobre F_q, y termina el proceso. Si $k \geq$2, calculamos $mcd(f(x), h_2(x)-c)$, para todo $c \in F_q$. Si no se puede separar $f(x)$ en k factores, se calcula $mcd(g(x), h_3(x) - c)$, para todo $c \in F_q$, y para todos los factores no triviales $g(x)$ encontrados hasta aquí. Esto se realiza aplicando el criterio del teorema 3.3, por medio de (3). El proceso continúa recursivamente hasta obtener los k factores irreducibles de $f(x)$.

a la salida del algoritmo, obtenemos la factorización de $f(x)$, en polinomios mónicos irreducibles.

Para finalizar el capítulo, presentamos un ejemplo paso a paso de la aplicación del algoritmo de factorización de Berlekamp. Vamos a cubrir el caso pendiente de factorización, que es el caso 1 de la página 31, cuando $d(x)$=1. Nuestro aporte más importante, es el trabajar en extensiones de campos finitos, por tanto trabajamos sobre una de tales extensiones, que es F_9, una extensión del campo finito primo F_3.

3. EJEMPLO DE FACTORIZACIÓN

Ejemplo 3.4: Factorizar el polinomio $f(x) = x^{12} + 3x^8 + 6x^7 + 8x^6 + 4x^3 + 2x^2 + 2x + 5$, sobre el campo F_9.

Lo primero, es calcular $d(x)$. Puesto que

$d(x)$=$mcd(f(x),\ f\ '(x))$=1, es decir, $d(x)$=$mcd(x^{12} + 3x^8 + 6x^7 + 8x^6 + 4x^3+2x^2+2x+5, 6x^7+6x^6+x+2)$ =1, $f(x)$ no tiene factores repetidos. Una vez asegurado lo anterior, procedemos con los siguientes pasos del algoritmo. Tenemos:

1. Cálculo del sistema de congruencias:

Calculamos $x^{iq}\ mod\ f(x)$, para q=9 y $0 \leq i \leq 11$. Obtenemos las siguientes congruencias mod $f(x)$:

x^0	$\equiv$	1											
x^9	$\equiv$										x^9		
x^{18}	$\equiv$	3+		$2x^2$+	$7x^3$+	$8x^4$+	x^5+			$6x^8$+	$3x^9$+	$7x^{10}$	
x^{27}	$\equiv$	7+	$4x$+	$7x^2$+	$5x^3$+	x^4+		$4x^6$+	$7x^7$+	$5x^8$+	x^9+	x^{10}	
x^{36}	$\equiv$	1+	$8x$+	$6x^2$+	$5x^3$+	$7x^4$+	$6x^5$+		$4x^7$+	$2x^8$+	$7x^9$+	$6x^{10}$+	$6x^{11}$
x^{45}	$\equiv$	6+	$5x$+	$5x^2$+	$3x^3$+		$7x^5$+	$7x^6$+	$2x^7$+	$6x^8$+	$3x^9$+	$7x^{10}$	
x^{54}	$\equiv$		$6x$+	$5x^2$+	$6x^3$+	$5x^4$+	$7x^5$+	$8x^6$+	x^7+	$8x^8$+	$2x^9$+	$5x^{10}$+	$4x^{11}$
x^{63}	$\equiv$	3+	$8x$+			x^4+	$8x^5$+	$6x^6$+	$8x^7$+	$7x^8$+	x^9+		$3x^{11}$
x^{72}	$\equiv$	4+	$3x$+	x^2+	$3x^3$+	$7x^4$+	$3x^5$+		$4x^7$+	x^8	$3x^9$+	$6x^{10}$+	$7x^{11}$
x^{81}	$\equiv$	6+	$3x$+		$3x^3$+		$4x^5$+	$2x^6$+	$4x^7$+	$4x^8$+	$3x^9$+	$3x^{10}$+	$3x^{11}$
x^{90}	$\equiv$		$2x$+	$8x^2$	$5x^3$+	$5x^4$+	$5x^5$+	x^6+	$7x^7$+	$5x^8$+	$6x^9$		
x^{99}	$\equiv$		$8x$+	$3x^2$+	x^3+	$2x^4$+	$6x^5$+		$6x^7$+	$7x^8$+	$7x^9$+	$8x^{10}$+	$7x^{11}$

2. Formar la matriz B: De las congruencias anteriores, formamos la matriz B de $s \times s = 12 \times 12$. Cada congruencia es representada por medio de los coeficientes de cada uno de sus términos, de izquierda a derecha, y de manera ascendente, respecto a las potencias de x. Cada congruencia es colocada en un renglón distinto de la matriz B:

$$B = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 3 & 0 & 2 & 7 & 8 & 1 & 0 & 0 & 6 & 3 & 7 & 0 \\ 7 & 4 & 7 & 5 & 1 & 0 & 4 & 7 & 5 & 1 & 1 & 0 \\ 1 & 8 & 6 & 5 & 7 & 6 & 0 & 4 & 2 & 7 & 6 & 6 \\ 6 & 5 & 5 & 3 & 0 & 7 & 7 & 2 & 6 & 3 & 7 & 0 \\ 0 & 6 & 5 & 6 & 5 & 7 & 8 & 1 & 8 & 2 & 5 & 4 \\ 3 & 8 & 0 & 0 & 1 & 8 & 6 & 8 & 7 & 1 & 0 & 3 \\ 4 & 3 & 1 & 3 & 7 & 3 & 0 & 4 & 1 & 3 & 6 & 7 \\ 6 & 3 & 0 & 3 & 0 & 4 & 2 & 4 & 4 & 3 & 3 & 3 \\ 0 & 2 & 8 & 5 & 5 & 5 & 1 & 7 & 5 & 6 & 0 & 0 \\ 0 & 8 & 3 & 1 & 2 & 6 & 0 & 6 & 7 & 7 & 8 & 7 \end{bmatrix}$$

3. Determinar una base para el espacio nulo de $B - I$: Formamos la matriz $B - I$, la cual es:

$$B - I = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 3 & 0 & 1 & 7 & 8 & 1 & 0 & 0 & 6 & 3 & 7 & 0 \\ 7 & 4 & 7 & 4 & 1 & 0 & 4 & 7 & 5 & 1 & 1 & 0 \\ 1 & 8 & 6 & 5 & 6 & 6 & 0 & 4 & 2 & 7 & 6 & 6 \\ 6 & 5 & 5 & 3 & 0 & 6 & 7 & 2 & 6 & 3 & 7 & 0 \\ 0 & 6 & 5 & 6 & 5 & 7 & 7 & 1 & 8 & 2 & 5 & 4 \\ 3 & 8 & 0 & 0 & 1 & 8 & 6 & 7 & 7 & 1 & 0 & 3 \\ 4 & 3 & 1 & 3 & 7 & 3 & 0 & 4 & 0 & 3 & 6 & 7 \\ 6 & 3 & 0 & 3 & 0 & 4 & 2 & 4 & 4 & 5 & 3 & 3 \\ 0 & 2 & 8 & 5 & 5 & 5 & 1 & 7 & 5 & 6 & 2 & 0 \\ 0 & 8 & 3 & 1 & 2 & 6 & 0 & 6 & 7 & 7 & 8 & 6 \end{bmatrix}$$

Al llevar a la matriz $B - I$ a su forma escalonada, para hallar una base,

obtenemos:

$$\begin{bmatrix}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 3 & 3 & 6 & 1 & 8 & 7 & 7 & 8 & 5 & 3 & 0
\end{bmatrix}$$

Observamos que la matriz escalonada tiene 10 columnas distintas de cero. Entonces, su rango es $\Re = 10$, y por lo tanto, $s - \Re = k = 2$. Entonces, debemos tener dos vectores, que son

$$(1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)$$

y

$$(0, 6, 6, 3, 2, 4, 5, 5, 4, 7, 6, 1),$$

los cuales forman una base del espacio nulo de la matriz $B - I$. Si recorremos los elementos de cada vector, de izquierda a derecha, y a cada elemento distinto de cero le asignamos una potencia de x, de manera ascendente, a dichos vectores les corresponden los polinomios:

$$h_1(x) = 1,$$
$$h_2(x) = 6x + 6x^2 + 3x^3 + 2x^4 + 4x^5 + 5x^6 + 5x^7 + 4x^8 + 7x^9 + 6x^{10} + x^{11}.$$

4. Para i=1, tenemos $h_1(x)$=1, que es un factor trivial, por lo cual lo descartamos. Para i=2, es decir, para $grado(h_i(x)) \geq 1$, calculamos el $mcd(f(x), h_2(x) - c)$ para cada $c \in F_q$. Esto se realiza mediante el algoritmo de Euclides. El programa prueba todos los elementos de F_9, encontrando dos valores correctos para c:

para c=3
$mcd(f(x), h_2(x) - 3)$=$mcd(f(x), h_2(x) + 6)$

$$= mcd(x^{12} + 3x^8 + 6x^7 + 8x^6 + 4x^3 + 2x^2 + 2x + 5\ ,\ x^{11} + 6x^{10} + 7x^9 + 4x^8 + 5x^7 + 5x^6 + 4x^5 + 2x^4 + 3x^3 + 6x^2 + 6x + 6)$$

$$= x^{11} + 6x^{10} + 7x^9 + 4x^8 + 5x^7 + 5x^6 + 4x^5 + 2x^4 + 3x^3 + 6x^2 + 6x + 6,$$

para c=8
$mcd(f(x), h_2(x) - 8)$=$mcd(f(x), h_2(x) + 4)$

$$= mcd(x^{12} + 3x^8 + 6x^7 + 8x^6 + 4x^3 + 2x^2 + 2x + 5\ ,\ x^{11} + 6x^{10} + 7x^9 + 4x^8 + 5x^7 + 5x^6 + 4x^5 + 2x^4 + 3x^3 + 6x^2 + 6x + 4)$$

$$= x + 3.$$

La factorización obtenida es:

$$f(x) = (x^{11}+6x^{10}+7x^9+4x^8+5x^7+5x^6+4x^5+2x^4+3x^3+6x^2+6x+6)(x+3).$$

□

Con el ejemplo anterior, hemos terminado la discusión de la parte teórica del algoritmo de Berlekamp. La explicación de manera práctica, se verá a continuación, en el capítulo 4. Y algunos detalles más del algoritmo, se abordarán en las conclusiones de este trabajo.

Capítulo 4

La implementación del algoritmo de Berlekamp para factorización de polinomios sobre campos finitos

En este capítulo, presentamos los aspectos principales de la implementación llevada a cabo. Dicha implementación fue programada en lenguaje C estandar. Gracias a esto, se puede trabajar tanto en compiladores de C para Linux, como para Windows. Esto permite enfocarse directamente en el aspecto de programación, sin tener que lidiar con conflictos de compatibilidad de software. se utilizó el compilador gcc, para C sobre Linux, y el compilador Dev C++, para C sobre Windows.

Entonces, hablando de la implementación, presentamos aquí las funciones, rutinas y estructuras principales del programa realizado, mediante una breve descripción de sus argumentos, su funcionamiento y su aportación al programa total. La mayor parte de las estructuras de datos, se construyeron con arreglos de carácteres no signados. Esto se hizo con el propósito de ocupar el menor espacio de memoria posible, ya que el tipo *carácter*, es de los que menos bits ocupa para representar datos.

En algunas funciones, se incluye el seudocódigo, con el fin de apoyar la explicación de la función presentada. En otros casos, al ser las funciones muy simples, no se incluye seudocódigo. Y otras funciones son largas, por lo que tampoco se incluye seudocódigo, ya que éste, pudiera prestarse a confusión. En tal caso, se sugiere consultar directamente el apéndice con el listado del programa.

Funciones principales de la implementación

1. FUNCIONES PARA LA ARITMÉTICA EN EL CAMPO FINITO:

Previo a implementar el algoritmo de Berlekamp, es necesario identificar la estructura algebraica sobre la cual vamos a factorizar. Es decir, identificar el campo finito, y en base a él, crear las tablas, que contengan las estructuras para realizar la aritmética en el campo. Por lo que, se necesitan dos funciones, una que haga la suma de elementos del campo, para construir la tabla de suma, y la otra, que haga la multiplicación de elementos del campo, para construir la tabla de multiplicar. Además, se debe crear una función para manejo de los inversos aditivos, y que también calcule los inversos multiplicativos, de los elementos del campo. Entonces, para crear el campo finito, realizar la aritmética en él, y realizar el manejo adecuado de los elementos del campo, llevamos a cabo la implementación de las funciones de esta sección.

Previo a la presentación de las funciones, presentamos una estructura medular, que contiene los polinomios que empleamos, para construir los campos finitos que soporta nuestra implementación:

Macro CM: Es una macro, cuyos elementos son, los polinomios empleados, para construir los campos finitos primos posibles a utilizar, y sus extensiones. CM indica el valor máximo de

M, donde M es el grado del polinomio utilizado, para crear las tablas aritméticas del campo. En este punto, cabe hacer una explicación acerca de tales polinomios. Habíamos mencionado en el teorema 2.19, la existencia de un polinomio irreducible, el cual se emplea para hacer la aritmética, dentro de un campo finito. Entonces, tal polinomio nos puede ayudar a construir las tablas de nuestra implementación, para la aritmética en dicho campo. De esta forma, si tenemos un polinomio irreducible de grado M, sobre F_p, dicho polinomio nos servirá para generar las tablas del campo $F_{q=p^M}$ correspondiente. La parte más importante de nuestra implementación, es el poder factorizar sobre extensiones de campos finitos. Entonces, si queremos factorizar sobre una extensión, F_9 por ejemplo, tenemos, q=9, p^M=3^2, M=2, por lo que utilizamos el polinomio irreducible $2+x+x^2$, sobre F_3, para generar las tablas. Si queremos factorizar sobre F_{81}, el cual es la extensión máxima de nuestra implementación, se tiene q=81, p^M=3^4, M=4, por lo que utilizamos el polinomio irreducible $2+x+x^4$, sobre F_3, para crear las tablas aritméticas para la factorización. Utilizando el polinomio adecuado, el procedimiento es similar para el resto de extensiones de campos finitos de la implementación. Para construir las tablas de campos finitos primos, cuyo campo máximo implementado es F_{79}, utilizamos el polinomio irreducible $1+x$. Ahora bien, dentro de la estructura que contiene a los polinomios, estos se escriben como arreglos de enteros que se leen en forma ascendente, con respecto a las potencias de x. Por ejemplo, el polinomio $1 + x + x^2$, con el que se construye el campo finito F_4, es representado dentro de la macro, como $PI\_04[6] = \{1, 1, 1, 0, 0, 0\}$, que indica un arreglo de 6 enteros, llamado $PI\_04[6]$.

función sumaQ(x,y,z): Realiza la suma de elementos del campo finito en el que se trabaja, por medio de iteraciones, sin incluir la reducción modular. Esta operación se hace con el propósito de llenar la tabla para sumar, dentro del campo finito. Sus argumentos x y y, son dos arreglos de enteros, que representan a los elementos a ser sumados. El resultado es almacenado en el argumento z, el cual también es un arreglo vectorial de enteros. Se utiliza la variable M, para representar al grado del polinomio con que se construyó el campo.

Seudocódigo:

sumaQ(x, y, z)

1. definir arreglos de enteros $x[\ ]$, $y[\ ]$, $z[\ ]$;
2. declarar entero i;
3. **para** $i = 0$ **hasta** $i < M$ **hacer**
4. $z[i] = (x[i] + y[i])$;
5. **fin para**

función multQ(x,y,z): Realiza la multiplicación de elementos del campo finito en el que se trabaja, por medio de iteraciones, incluyendo la reducción modular, vía el polinomio con que se construye el campo. Dichas operaciones se hacen con el propósito de llenar la tabla para multiplicar, dentro del campo finito. Sus argumentos x y y, son dos arreglos de enteros, que representan a los elementos a ser multiplicados. El resultado es almacenado en el argumento z, el cual también es un arreglo vectorial de enteros. Se utiliza la variable M, para representar

al grado del polinomio con que se construyó el campo.

Seudocódigo:

multQ(x, y, z)

1. definir arreglos de enteros x, y, z.
2. definir enteros i, j.
3. **para** $i = 0$ **hasta** $i < M$ **hacer**
4. **para** $j = 0$ **hasta** $j < M$ **hacer**
5. calcula $z[i + j]$=$y[j] * x[i]$.
6. calcular z *mod* (polinomio primitivo).
7. **fin para** j
8. **fin para** i

función valor(z): Es la función que lleva a cabo la reducción modular del argumento z, el cual es el arreglo resultado de las funciones **sumaQ**, y **multQ**, respectivamente. Devuelve el resultado en la variable v. Tanto z como v, son arreglos vectoriales de enteros. A su vez, se utiliza la variable M, que representa al grado del polinomio con que se construyó el campo finito donde se está trabajando. Para llevar a cabo la redución, se requiere el valor de p, el cual, es la característica del campo finito en donde se factoriza.

Seudocódigo:

valor(z)

1. define arreglo de enteros z.
2. define enteros i, $v = 0$, w=1.
3. **para** i=0 **hasta** $i < M$, y $w = w * p$, **hacer**
4. $v = v + ((z[i]\ mod\ p) * w)$.
5. **fin para** i
6. regresa v.

función calTablas(): Esta función se encarga de calcular las dos estructuras aritméticas del campo finito en el que se va a factorizar. Las llamamos **TSuma** y **TMult**, que son la tabla de sumar, y la tabla de multiplicar, respectivamente. Ambas estructuras son representadas mediante arreglos bidimensionales, en los cuales, cada elemento es almacenado en un carácter no signado. Dichas estructuras, se almacenan en memoria, de tal forma que, cuando se efectúan los cálculos, los elementos se buscan dentro de estos arreglos, respecto al renglón y columna que ocupan. Por ejemplo, si calculamos las tablas correspondientes al campo finito F_9, que es una extensión del campo finito primo F_3, tenemos

tabla de sumar

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
[0]	0	1	2	3	4	5	6	7	8
[1]	1	2	0	4	5	3	7	8	6
[2]	2	0	1	5	3	4	8	6	7
[3]	3	4	5	6	7	8	0	1	2
[4]	4	5	3	7	8	6	1	2	0
[5]	5	3	4	8	6	7	2	0	1
[6]	6	7	8	0	1	2	3	4	5
[7]	7	8	6	1	2	0	4	5	3
[8]	8	6	7	2	0	1	5	3	4

si queremos sumar, $5 + 4$, en F_9, buscamos en la tabla anterior, el elemento que aparece en el cruce del renglón [5], y la columna [4], entonces, $5+4 = 6$, o bien, a la inversa, el cruce del renglón [4], y la columna [5]. Y así para cualquier caso requerido. En cuanto a la multiplicación, es de igual forma

tabla de multiplicar

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
[0]	0	0	0	0	0	0	0	0	0
[1]	0	1	2	3	4	5	6	7	8
[2]	0	2	1	6	8	7	3	5	4
[3]	0	3	6	7	1	4	5	8	2
[4]	0	4	8	1	5	6	2	3	7
[5]	0	5	7	4	6	2	8	1	3
[6]	0	6	3	5	2	8	7	4	1
[7]	0	7	5	8	3	1	4	2	6
[8]	0	8	4	2	7	3	1	6	5

si queremos multiplicar, 7 por 4, localizamos el cruce del renglón [7], y la columna [4], y tenemos 7*4 = 3, o bien, el cruce del renglón [4], y la columna [7].

las dimensiones máximas de los arreglos son de 81 X 81, con lo cual aseguramos cubrir todos los casos para las extensiones de campos finitos primos, hasta el campo F_{81}. Para calcular las estructuras descritas anteriormente, la función **calTablas**, se auxilia de las funciones **sumaQ**, **multQ**, y **valor**. Una breve descripción de la función es la siguiente.

Seudocódigo:

calTablas()

1. declarar **Q**, el orden del campo.
2. declarar **x**, **y**, y **z**, arreglos de enteros que representan los elementos del campo.
3. declarar una estructura **switch** que toma como argumento a **Q**.
4. leer **Q**.
5. **si Q** es válido **entonces**
6. extraer de **CM** el polinomio primitivo.
7. identificar a **p**, para la reducción modular.
8. **de lo contrario**
9. desplegar mensaje de campo no válido.
10. salir del programa.

```
11.       fin si
12.     para i=0 hasta i < M + 1 hacer
13.       para j=0 hasta j < Q hacer
14.         calcula TSum, mediante funciones sumaQ y valor.
15.         calcula TMult, mediante funciones multQ y valor.
16.       fin para j
17.     fin para i
18.     para i=0 hasta Q hacer
19.      para j=0 hasta Q hacer
20.         imprime tabla de suma TSum.
21.         imprime tabla de multiplicación TMult.
22.       fin para j
23.     fin para i
```

función calinv: Esta función calcula el contenido de dos arreglos llamados, **invM**, e **invA**. Ambos son arreglos de una dimensión, o vectores, de enteros, que son localizados mediante un índice, que los recorre. El primero de ellos, **invM**, contendrá a los inversos multiplicativos de los elementos del campo finito, tal que, cuando uno de ellos se requiere, sea localizado en esta estructura. El segundo de ellos, **invA**, contendrá a los inversos aditivos, de tal forma que, cuando se presente como dato, un elemento con signo negativo, éste pueda ser expresado con su equivalente positivo, dentro del campo. Esto con el propósito de poder hacer la aritmética, ya que la tabla de suma, **TSum**, contiene a todos los elementos del campo con signo positivo. También, para que se pueda expresar a la salida del programa, a todos los factores, mediante coeficientes con signo positivo. Ambos vectores permanecen en memoria, disponibles para cuando

sean requeridos. La función **calinv**, tiene presente que Q, es el orden del campo finito de trabajo.

Seudocódigo:

calinv()

```
1.    declarar indices enteros i, j.
2.      para i=0 hasta i < Q hacer
3.        para j=0 hasta j < Q hacer
4.          si TMult[i][j] = 1, entonces
5.            invM[i] = j.
6.          fin si
7.          si TSuma[i][j] = 0, entonces
8.            invA[i] = j.
9.          fin si
10.       fin para j
11.     fin para i
```

2. FUNCIONES PARA ARITMÉTICA MATRICIAL:

Las funciones de esta sección, son las que realizan los cálculos del algoritmo de Berlekamp, concernientes a los pasos que involucran a la matriz B. Es decir, para poder expresar el sistema de congruencias del paso 1 del algoritmo (página 48), mediante un matriz, a la que, en el paso 2 (página 49), se le resta la matriz identidad I. Una vez que se hace esto, el siguiente paso del algoritmo, que es cubierto por dichas funciones, es diagonalizar la matriz $B - I$, es decir, encontrar una base de vec-

tores $h(x)$, que resuelvan tal matriz. El resultado de esta etapa de la implementación, es la base de vectores $h(x)$ mencionada, la cual se almacena temporalmente, para ser procesada por la siguiente etapa, la cual es la etapa de funciones para manejo de polinomios, que finalmente arrojarán el resultado completo del algoritmo de factorización programado, el cual será $f(x)$, expresado en sus factores mónicos irreducibles.

La primera función a presentar en esta sección, es la función **grado**. Esto se hace, ya que, a pesar de que dicha función, es función para manejo de polinomios (siguiente sección), es empleada por las funciones de esta sección.

función grado(x): Esta función encuentra el grado del polinomio que se le pasa como argumento. Esto lo hace recorriendo el arreglo x, el cual es un vector de enteros, que representa a los coeficientes del polinomio en turno. Hace el recorrido del vector, hasta encontrar el elemento $\neq 0$, de mayor índice, que nos indique que ese, es el coeficiente líder.

función CalMatB(B,PL): Función que calcula las matrices B y $B - I$. B se forma mediante un arreglo bidemensional de carácteres no signados, la cual, se inicializa en ceros, y conforme avanza la ejecución, se va llenando con las congruencias respectivas. La función **CalMatB**, tiene como argumentos a B, y al polinomio a factorizar, denotado como PL. Lo primero que hace la función, es calcular el grado de PL, mediante la función **grado**. Con esta accción, se sabrá la dimensión de B. Una vez que se sabe lo anterior, se calcula el sistema de congruen-

cias con las que llenará la matriz B. Para ello hace uso de la función **Divide** (Esta función se encuentra en **3. Funciones para manejo de polinomios**). Con dicha función, se calcula x^{iq} *mod* PL, para $0 \leq i \leq$ *grado de* PL - 1, y con q, el orden del campo respectivo. Entonces, se van tomando los residuos polinomiales de cada división, y se guardan en B. Después, se suma a B, la matriz identidad I correspondiente, mediante la tabla de suma **TSuma**, y el vector de inversos aditivos, **invA**. Con esto, se crea la matriz $B - I$. Finalmente, se imprimen las matrices B y $B - I$.

Seudocódigo:

CalMatB(B,PL)

1. declara B y PL.
2. define indices para recorrido de elementos.
3. calcular el grado de PL, mediante la función **grado**.
4. define la longitud de B, según el grado de PL.
5. calcular las congruencias mediante la función **Divide**.
6. almacenar las congruencias en B.
7. calcular $B - I$.
8. imprimir matrices.

función DiagonalizaMatB(B,PL): Función que se encarga de llevar la matriz $B - I$, obtenida con la función **CalMatB**, a su forma escalonada, o diagonalizada. Recibe la matriz $B - I$, en su argumento B, y en el argumento PL, recibe el polinomio a factorizar, al cual le calcula el grado, mediante la función **grado**.

Y para diagonalizar, emplea las tablas de suma **TSuma**, y de multiplicación **TMult**, del campo en el que se trabaja. A su vez, utiliza las estructuras **invM**, e **invA**, para operar sobre los elementos de $B-I$, de tal manera que, en la matriz, se produzca la diagonal de unos, que indique la forma escalonada que se busca. Una vez que se tiene la matriz en su forma escalonada, se procede a encontrar una base de polinomios $h(x)$, para el espacio de soluciones. Para ello, se emplea un arreglo llamado S, el cual, está formado con carácteres no signados, que representan los coeficientes del polinomio $h(x)$ en turno, el cual satisface a $B-I$. De tal forma que, en cada iteración, S contiene un vector solución $h(x)$ de la matriz. Finalmente, se imprimen la matriz $B-I$, y S, que contiene a los vectores que la satisfacen. Estos vectores se imprimen en términos de la variable x.

3. FUNCIONES PARA MANEJO DE POLINOMIOS:

Una vez que se obtiene la base de vectores $h(x)$ para el espacio nulo de $B-I$, ahora, en esta etapa, con esa base, se realiza el cálculo final de los factores del polinomio que se quiere factorizar. Además de otras funciones, como son la suma de polinomios, la multiplicación de polinomios, la división sintética, y las funciones restantes para polinomios.

función Divide(QP,R,A,B): Realiza la división sintética entre polinomios, representados por A, el dividendo, y B, el divisor. En QP se guarda el cociente de la división, mientras que R guarda el residuo de tal división. Los 4 vectores son arreglos de carácteres sin signo. La función *Divide*, hace uso de la función **grado**, para calcular el grado del dividendo y el divisor, y se

auxilia de las tablas de sumar y multiplicar, **TSuma** y **TMult**, respectivamente, para realizar la aritmética necesaria, que involucra a la división. También se emplean los arreglos **invM**, e **invA**, para las operaciones y conversiones necesarias.

Seudocódigo:

Divide(QP,R,A,B)

1. declarar QP, R, A, B, arreglos de carácteres no signados.
2. declaran enteros i, gb, gr, dif, fac.
3. **para** i=0 **hasta** $i < longitud\ de\ A$ **hacer**
4. asignar A a R.
5. asignar 0 a QP.
6. **fin para** i.
7. calcular grado de B y asignarlo a gb.
8. calcular grado de R y asignarlo a gr.
9. **mientras** $gr \geq gb$ **hacer**
10. $dif = gr$ - gb.
11. $fac = TMult[invM[B[gb]]][R[gr]]$.
12. $QP[dif] = fac$.
13. $fac = invA[fac]$.
14. **para** i=gb **hasta** $i \geq 0$ **hacer**
15. $R[i+dif] = TSuma[R[i+dif]][TMult[fac][B[i]]]$.
16. $gr = grado(R)$.
17. **fin para** i
18. **fin mientras**

función Cal_GCD(D,A,B): Calcula el máximo común divi-

sor entre polinomios. A y B representan los polinomios, por medio de vectores que contienen a sus coeficientes, representados por carácteres no signados. D es un arreglo de carácteres no signados, que se utiliza para comparación. Además, D sirve para almacenar el máximo común divisor. Al llevar a cabo el cálculo, si el máximo común divisor es 1, la función regresa 1, indicando que los polinomios no tienen factors repetidos. Y tendremos entonces, la factorización del caso 1, mencionado en el capítulo 3. De lo contrario, la función regresa 0, lo que indica que el resultado es no trivial, y se tendrá la factorización del caso 3, mencionado en el capítulo 3. Si la función regresa 2, el máximo común divisor es el polinomio A, lo que nos indica que se tendrá la factorización del caso 2, del capítulo 3. El polinomio resultado del cálculo del máximo común divisor, se guarda en el arreglo auxiliar R, que es un vetor de carácteres no signados. Esta función, **Cal_GCD**, también utiliza la función **Divide**, de manera repetida, así como las estructuras **invM** (inversos multiplicativos), y **TMult**, (tabla de multiplicar), para llevar a cabo sus cálculos.

Seudocódigo:

Cal_GCD(D,A,B)

1. declarar variables y arreglos a utilizar.
2. leer polinomios A y B.
3. utilizar repetidamente división sintética.
4. leer resultado de la división, el cual es el mcd.
5. **si** mcd=1 **entonces**
6. el resultado es trivial.

7. **de lo contrario**
8. es no trivial.
9. **fin si**

función EncuentraFactores(B,PL): Función que auxilia a la función **Factoriza**, en la factorización del polinomio PL. Por medio de la matriz B, y el empleo de las estructuras **TSuma** (tabla de suma) e **invA** (inversos aditivos), crea un arreglo llamado **GCD**, en el que se van almacenando los factores que se han encontrado, y que al final contendrá a todos los factores del polinomio, que se recibió como argumento de entrada. También se auxilia de la función **Cal_GCD**. Al final se imprimen los factores en términos de x.

función Factoriza(PL): Esta función se encarga de factorizar el polinomio PL. En el proceso, revisa si el polinomio tiene factores repetidos. Para ello, deriva el polinomio PL, y utiliza a la función **Cal_GCD**, para calcular el máximo común divisor entre PL y su derivada PP. Si el máximo común divisor es 1, PL no tiene factores repetidos, y el polinomio será enviado a etapas posteriores, donde otras funciones continúan el proceso. En caso contrario, la función se llama a sí misma recursivamente, para identificar los factores repetidos. En esta acción, se emplea a la función **Divide**. Para realizar la factorización, además de lo descrito anteriormente, emplea a las funciones **CalMatB**, **EncuentraFactores** y **DiagonalizaMatB**.

Seudocódigo:

Factoriza(PL)

1. leer PL.
2. calcular $PL\,'$.
3. calcular $mcd(PL, PL\,')$.
4. **si** mcd=1 **entonces**
5. llamar a la función **CalMatB**.
6. llamar a la función **DiagonalizaMatB**.
7. llamar a la función **EncuentraFactores**.
8. **de lo contrario**
9. llamarse recursivamente.
10. identificar el caso de factorización.
11. repetir 5., 6., y 7.
12. **fin si**.

4. FUNCIÓN COMPLEMENTARIA:

Mult(F,A,B): Función de comprobación. Recibe en A y en B, los factores obtenidos del proceso, los multiplica y los coloca en F. Esto se realiza a manera de comprobar que el resultado de la factorización es correcto. Si en F aparece el polinomio original a factorizar, todo salió bien. Esta función emplea a la función **grado** y a las tablas de suma y multiplicación dentro del campo, **TSuma** y **TMult**, respectivamente.

5. RUTINA PRINCIPAL:

int main: Rutina principal, que recibe los parámetros con los que se corre el programa. Lee Q, que representa el orden del campo finito, y con ello, calcula las tablas para la aritmética dentro de dicho campo. Lee PL, el polinomio a factorizar. Y llama a la función **grado**, y verifica si el polinomio es mónico. Si no lo es, utiliza la estructura **invM** (inversos multiplicativos), y la tabla de multiplicar **TMult**, para hacerlo mónico. Esto lo hace, multiplicando los coeficientes del polinomio, por el inverso multiplicativo de su coeficiente líder. A su vez, llama a la función que deriva el polinomio PL, a factorizar. Y llama a la función **Cal_GCD**, que calcula el máximo común divisor entre PL y su derivada. Esto lo hace con el propósito de verificar si PL tiene factores repetidos. Una vez que verifica lo anterior, llama a la función **Factoriza**, que se encarga del resto del proceso. Al final, despliega los factores obtenidos.

Cómo se corre. Una vez compilado el programa:

en la línea de comandos de Linux, se escribe:

./nombre de archivo. -q. campo. coeficientes del pol. en orden ascendente.

ejemplo: ./BerlekampQ -q 25 2 5 0 0 16 8 0 -0 -1 -8 1 0 1

en Windows, en Dev C++, en el menú **Execute**, en la opción **Parameters**, se escribe:

-q. campo. vector de coeficientes del polinomio en orden ascendente.

ejemplo: -q 25 2 5 0 0 16 8 0 -0 -1 -8 1 0 1

6. EJEMPLOS DE CORRIDA.

Ejemplo 1: Factorizamos el polinomio $x+x^{16}$, sobre F_2, en Dev C++. Según el procedimiento descrito, escribimos, -q 2 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1. Ejecutamos el programa y obtenemos:

$$f\,'(x) = 1,$$

$$MCD = 1,$$

$$Numero\ Total\ de\ Factores = 6,$$

$$x,\ x^4+x^3+x^2+x+1,\ x^2+x+1,\ x^4+x^3+1,\ x^4+x+1,\ x+1.$$

Lo interesante de este ejemplo, es que, de los factores que resultan, los últimos 4, son polinomios irreducibles que pueden utilizarse dentro de la misma implementación, para generar tablas

de suma y multiplicación, de algunos campos finitos.

Ejemplo 2: Factorizamos el polinomio $1+x+x^2+x^3+x^4+x^6+x^7+x^9+x^{11}+x^{12}+x^{100}+x^{101}+x^{104}$, sobre F_2, en Dev C++. Según el procedimiento descrito, escribimos, -q 2 1 1 1 1 1 0 1 1 0 1 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 1.

Ejecutamos el programa y obtenemos:

$$f\,'(x) = 1,$$

$$MCD = 1,$$

$$Numero\ Total\ de\ Factores = 2,$$

$$x^{100}+x^8+x^7+x^2+1,\ x^4+x+1.$$

Lo interesante de este ejemplo, es darnos cuenta, de que la implementación factoriza polinomios de grados grandes, y nos da el resultado correcto.

Conclusiones

Al culminar el presente trabajo, logramos cubrir los aspectos que involucran el poner en práctica un algoritmo de factorización de polinomios. Es decir, tomamos un algoritmo de factorización conocido, lo analizamos en detalle, y lo pusimos a prueba. Gracias a la ayuda de la computadora, pudimos obtener resultados, que verificaron, que la implementación realizada fue correcta. Esto fue posible al comparar, en algunos casos, nuestros resultados con los resultados arrojados por programas externos al nuestro. Y en otros casos, comparando nuestros resultados, con los resultados de algunas de las referencias bibliográficas. La implementación resultó exitosa, para el límite que nos fijamos, de factorizar en un campo de 81 elementos como máximo.

Pensamos que, al realizar exitosamente la implementación del algoritmo de Berlekamp, para factorización de polinomios sobre campos finitos, hemos hecho una breve contribución a las diversas áreas del conocimiento que involucran el uso de polinomios sobre campos finitos.

Queda como un reto a futuro, el complementar la implementación realizada. Esto podría ser llevado a cabo, incrementando el número de campos en los que se puede factorizar, mediante la adición de algunas funciones que procesen números grandes. O bien, que se pudieran agregar algunos aspectos a la imple-

mentación, tales como determinar cuáles de los factores que resultan, pueden ser utilizados en la generación de códigos cíclicos [1]. Lo anterior, lo podemos realizar actualmente inspeccionando directamente los resultados del programa, y gracias a conocimientos previos, pero sólo para algunos de los factores arrojados por el programa. Pero sería interesante contar con un programa que lo hiciera de manera directa, y para todos los factores que se obtienen con la implementación actual.

Apéndice

Presentamos el listado completo de la implementación realizada. Su uso es permitido libremente, con el reconocimiento de su origen. Para ello contactar al Dr. Gerardo Vega Hernández, <gerardov@servidor.unam.mx>.

El archivo .h:

```
/* BerlekampQ.h - Header para BerlekampQ.c

    Universidad Nacional Autonoma de Mexico,
Septiembre del 2008. */

/* Declaracion de constantes */
#define N_max 4096

/* Definicion de tipos */
typedef unsigned char B8;
```

La implementación C:

```
#include <stdio.h>
#include <stdlib.h>
#include "BerlekampQ.h"
```

//**declaración inicial de variables y estructuras**

```
char cadena[81];
char *bits();
int f;
int P, Q=2;
int ns;
```

// **arreglos para cálculo de inversos**

B8 invM[81], invA[81];

// **tablas para aritmética en el campo finito**

B8 TSuma[81][81], TMult[81][81];

char *sig[2] = {“ ”, “+ ”};

int pol[N_max];
B8 B[N_max][N_max];
B8 S[N_max][N_max];
B8 PP[N_max];
B8 POL_Cero[N_max], POL_Uno[N_max];
int ntf=0; /* Numero Total de Factores, sin importar repeticiones */

// **macro que contiene los polinomios primitivos para construir los**
// **campos finitos soportados por la implementación**

#define CM 32
/* CM=máximo valor de M (M es el grado del pol. prim usado)*/

int PI_Pr[6]={1,1,0,0,0,0}, PI_04[6]={1,1,1,0,0,0}, PI_08[6] ={1,1,0,1,0,0},
PI_16[6]={1,1,0,0,1,0}, PI_32[6]={1,0,1,0,0,1}, PI_64[7] ={1,1,0,0,0,0,1},
PI_09[6]={2,1,1,0,0,0}, PI_27[6]={1,2,0,1,0,0}, PI_81[6] ={2,1,0,0,1,0},
PI_25[6]={2,1,1,0,0,0}, PI_49[6]={3,1,1,0,0,0}, PI_Pr2[6]={2,1,0,0,0,0},
PI_41[6]={6,1,0,0,0,0}, PI_43[6]={9,1,0,0,0,0}, PI_59[6] ={3,1,0,0,0,0},
PI_67[6]={4,1,0,0,0,0}, PI_73[6]={5,1,0,0,0,0};

int *PI, MPI[6];
int M, gp;

//**FUNCIONES PARA ARITMÉTICA EN EL CAMPO FINITO:**

// **Función para sumar:**

/* Suma z(x) = x(x)+y(x) sin reducir modulo P */

```
/* la reduccion va acargo de la funcion valor */

sumaQ(x,y,z)
int x[  ],y[  ],z[  ];
      {
       int i;
       for(i=0;i<M;i++)     z[i] = (x[i]+y[i]);
      }
```

// **Función para multiplicar:**

```
/* Multiplica z(x) = x(x)*y(x) sin reducir modulo P */
/* la reduccion va acargo de la funcion valor */

multQ(x,y,z)

int x[  ],y[  ],z[  ];
      {
       int sum[2*CM];
       int i, j, gs;


        memset(sum,0,2*M*sizeof(int));

    /* multiplica sum(x) = x(x)*y(x) */


          for(i=0;i<M;i++)
           for(j=0;j<M;j++)     sum[i+j] += y[j]*x[i];

    /* residuo z(x) = sum(x) Mod PI(x) */


          for(gs=gradoQ(sum);gs>=gp;gs- -)

    /* PI es monico */
```

```
            for(j=0;j<M;j++)     sum[gs-M+j] += MPI[j]*sum[gs];
        memcpy(z,sum,M*sizeof(int));
}
```

// **Función auxiliar de multQ(x,y,z):**

```
gradoQ(x)
int x[  ];
      {
       int i;


       for(i=2*M-1;i>0;i- -)
       if(x[i]) break;
       return(i);
      }
```

// **Reducción modular:**

```
valor(z)
int z[  ];
      {
       int i, v=0, w;
       for(i=0,w=1;i<M;i++,w*=P)     v += ((z[i]%P)*w);
       return(v);
      }
```

// **Función que calcula las tablas**
// **para la aritmética en el campo:**

```
calTablas()
{
int i, j, k, l, NE=Q;
int x[CM], y[CM], z[CM];

switch(Q)
```

{

case 2:
P=2; PI=PI_Pr; M=1;
break;
case 3:
P=3; PI=PI_Pr; M=1;
break;
case 4:
P=2; PI=PI_04; M=2;
break;
case 5:
P=5; PI=PI_Pr; M=1;
break;
case 7:
P=7; PI=PI_Pr; M=1;
break;
case 8:
P=2; PI=PI_08; M=3;
break;
case 9:
P=3; PI=PI_09; M=2;
break;
case 11:
P=11; PI=PI_Pr; M=1;
break;
case 13:
P=13; PI=PI_Pr; M=1;
break;
case 16:
P=2; PI=PI_16; M=4;
break;
case 17:
P=17; PI=PI_Pr; M=1;
break;
case 19:

P=19; PI=PI_Pr; M=1;
break;
case 23:
P=23; PI=PI_Pr; M=1;
break;
case 25:
P=5; PI=PI_25; M=2;
break;
case 27:
P=3; PI=PI_27; M=3;
break;
case 29:
P=29; PI=PI_Pr; M=1;
break;
case 31:
P=31; PI=PI_Pr; M=1;
break;
case 32:
P=2; PI=PI_32; M=5;
break;
case 37:
P=37; PI=PI_Pr2; M=1;
break;
case 41:
P=41; PI=PI_41; M=1;
break;
case 43:
P=43; PI=PI_43; M=1;
break;
case 47:
P=47; PI=PI_Pr2; M=1;
break;
case 49:
P=7; PI=PI_49; M=2;
break;
case 53:
P=53; PI=PI_Pr2; M=1;

```
                    break;
                    case 59:
                P=59; PI=PI_59; M=1;
                    break;
                    case 61:
                P=61; PI=PI_Pr2; M=1;
                    break;
                    case 64:
                P=2; PI=PI_64; M=6;
                    break;
                    case 67:
                P=67; PI=PI_67; M=1;
                    break;
                    case 71:
                P=71; PI=PI_Pr2; M=1;
                    break;
                    case 73:
                P=73; PI=PI_73; M=1;
                    break;
                    case 79:
                P=79; PI=PI_Pr2; M=1;
                    break;
                    case 81:
                P=3; PI=PI_81; M=4;
                    break;
                    default:
            printf("Error campo invalido o no soportado\n");

system("PAUSE ");
exit(-1);
break;
}

gp = M;


for(i=0;i<M+1;i++)     MPI[i] = (P - PI[i])%P;
```

```
    for(i=0;i<NE;i++)
      { /* Tabla para sumar */


          for(k=0,l=i;k<M;k++,l/=P)     x[k] = l % P;
          for(j=0;j<NE;j++)
            {
            for(k=0,l=j;k<M;k++,l/=P)     y[k] = l % P;
            /* Suma x+y y el resultado en z */
            sumaQ(x,y,z);
            TSuma[i][j] = (B8) valor(z);
            }
      }

    for(i=0;i<NE;i++)
      { /* Tabla para multiplicar */


          for(k=0,l=i;k<M;k++,l/=P)     x[k] = l % P;
          for(j=0;j<NE;j++)
            {
            for(k=0,l=j;k<M;k++,l/=P)     y[k] = l % P;
            /* Multiplica x*y y el resultado en z */
            multQ(x,y,z);
            TMult[i][j] = (B8) valor(z);
            }
      }

// Imprime tablas

    for(i=0;i<Q;i++)
      {
       for(j=0;j<Q;j++)     printf("%2d ",TSuma[i][j]);
       printf("\n");
      }
```

```
    printf("\n");

    for(i=0;i<Q;i++)
      {
       for(j=0;j<Q;j++)     printf("%2d ",TMult[i][j]);
       printf("\n");
      }
}
```

// **Función para inversos:**

```
/* Calcula inversos de los elementos en el campo*/

calinv()
      {
       int i, j;

       for(i=0;i<Q;i++)
        for(j=0;j<Q;j++)
          {
           if(TMult[i][j]==1) invM[i] = j;
           if(TSuma[i][j]==0) invA[i] = j;
          }
}
```

//**FUNCIONES PARA ARITMÉTICA MATRICIAL:**
// **Hallar el grado del polinomio:**

```
grado(x)
B8 x[  ];
      {
       int i;

       for(i=N_max-1;i>0;i- -)
       if(x[i]) break;
       if((i==0)&&(!x[i])) return(-1);
```

```
      return(i);
     }
```

// **Cálculo de matrices:**

```
/* Calcula Matriz B y despues B-I */

CalMatB(B,PL)
B8 B[N_max][N_max], PL[ ];
     {
      int i, j, n;
      B8 X[N_max], F[N_max], QP[N_max], R[N_max];
      memset(B,0,N_max*N_max);
      n=grado(PL);
      memcpy(X,POL_Uno,N_max);
      memset(F,0,N_max);
      F[Q] = 1;

      for(i=0;i<n;i++)
       {
        Divide(QP,R,X,PL);
        memset(X,0,N_max);
        for(j=Q;j<N_max;j++)
        X[j] = R[j-Q];
        R[i] = TSuma[R[i]][invA[1]];     /* B-I */
        for(j=0;j<n;j++)    B[j][i] = R[j];     /* trasponiendo */
       }

//imprimir matrices:

//B-I

     for(i=0;i<n;i++)
      {
       for(j=0;j<n;j++) printf("%3d ",B[j][i]);
       printf("\ n");
      }
```

```
      printf("\ n");

//B-I transpuesta

      for(i=0;i<n;i++)
       {
        for(j=0;j<n;j++) printf("%3d ",B[i][j]);
        printf("\ n");
       }

      }
```

// **Función que diagonaliza la matriz B:**

```
DiagonalizaMatB(B,PL)
B8 B[N_max][N_max], PL[N_max];
      {
       int i, j, k, n, enceros, fac;
       B8 T[N_max];
       n=grado(PL);

       for(i=1;i<n;i++)
        {
         enceros = 0;
         if(!B[i][i])
          {
           for(j=0;j<i;j++)
           if((!B[j][j])&B[j][i]) break;
           if(j==i)
           for(j=i+1;j<n;j++)
           if(B[j][i]) break;
           if(j<n)
            {
             memcpy(T,&B[j][0],N_max);
             memcpy(&B[j][0],&B[i][0],N_max);
             memcpy(&B[i][0],T,N_max);
            }
```

```
            else enceros = 1;
           }
          if(enceros) continue;
          fac = invM[B[i][i]];
          for(k=0;k<n;k++)          B[i][k] = TMult[fac][B[i][k]];
          for(j=0;j<n;j++)
           {
            if(i==j) continue;
            if(B[j][i])
            for(k=0,fac=invA[B[j][i]];k<n;k++)
            B[j][k] = TSuma[B[j][k]][TMult[fac][B[i][k]]];
           }
         }

//Impresión de las matrices

printf("\ n");
for(i=0;i<n;i++)
      {

/* B-I diagonalizada, donde se ve el rango en renglones distintos de cero */


           for(j=0;j<n;j++) printf("%3d ",B[i][j]);
           printf("\ n");
          }

        printf("\ n");

for(i=0;i<n;i++)
      {

/* B-I diagonalizada transpuesta, donde se ve el rango en las columnas distintas de cero */

        for(j=0;j<n;j++) printf("%3d ",B[j][i]);
        printf("\ n");
```

```
      }

/* búsqueda de una base para */
/* el espacio de soluciones */

memset(S,0,N_max*N_max);
ns = 0;
for(i=1;i<n;i++)
      {
       if(!B[i][i])
         {
          S[ns][i]=1;
          for(j=0;j<n;j++)
          if(B[j][i])     S[ns][j]=invA[B[j][i]];
          ns++;
         }
      }

//Imprimir la base

printf("\ n");
for(i=0;i<ns;i++)
      {
       for(j=0;j<n;j++) printf("%3d ",S[i][j]);
       imprime(&S[i][0]); printf("\ n");
      }

}
```

//FUNCIONES PARA MANEJO DE POLINOMIOS:
// Función para la división sintética:

```
Divide(QP,R,A,B)

/* Division sintetica de A entre B */
```

```
B8 QP[  ], R[  ], A[  ], B[  ];
      {
       int i, j, gb, gr, dif, fac;

       for(i=0;i<N_max;i++)     { R[i] = A[i]; QP[i] = 0; }
       gb = grado(B);
       gr = grado(R);
       while(gr>=gb)
        {
         dif = gr - gb;
         fac = TMult[invM[B[gb]]][R[gr]];
         QP[dif] = fac;

         /* fac = - fac */

         fac = invA[fac];
         for(i=gb;i>=0;i- -)
         R[i+dif] = TSuma[R[i+dif]][TMult[fac][B[i]]];
         gr = grado(R);

        }

      }
```

// **Función para el máximo común divisor:**

```
/* Calcula el maximo comun divisor entre A y B el resultado en R */
/* Si mcd(A,B) = 1, la funcion regresa un 1, de otro modo 0 */

Cal_GCD(D,A,B)
B8 D[N_max], A[N_max], B[N_max];
      {
       int i, gd, fac;
       B8 Q[N_max], R[N_max], T[N_max];

       if(!memcmp(A,POL_Cero,N_max) || !memcmp(B,POL_Cero,N_max))
        {
```

```
        memcpy(R,A,N_max);
        return(2);
       }
       memcpy(T,A,N_max);
       memcpy(D,B,N_max);

       while(1)
        {

         /* Division sintetica de A entre B */

         Divide(Q,R,T,D);
         if(!memcmp(R,POL_Cero,N_max))     break;
         memcpy(T,D,N_max);
         memcpy(D,R,N_max);
        }

        gd = grado(D);
        if(D[gd]!=1)
        for(i=0,fac=invM[D[gd]];i<=gd;i++)
        D[i] = TMult[fac][D[i]];

        if(!memcmp(D,POL_Uno,N_max))     return(1);     /*mcd(A,B)=1*/

        if(!memcmp(A,R,N_max))     return(2);     /* A = mcd(A,B) */

        return(0);     /* mcd(A,B) es NO trivial */
    }
```

// **Función auxiliar de la función Factoriza:**

```
EncuentraFactores(B,PL)
B8 B[N_max][N_max], PL[N_max];
    {

     int i, j, k, l, esirr, grd, nf = 0, ban;
     B8 GCD[N_max], POL[N_max];
```

```
B8 R1[N_max], R2[N_max];

memcpy(&B[nf++][0],PL,N_max);

for(i=0;i<ns;i++)
 {

  for(j=0,k=0;j<nf;j++)
   {

    memcpy(POL,&B[j][0],N_max);
    for(l=0,ban=1;l<Q;l++)
     {

      S[i][0] = TSuma[S[i][0]][invA[l]];
      esirr = Cal_GCD(GCD,POL,&S[i][0]);
      S[i][0] = TSuma[S[i][0]][l];
      if(!esirr)
       {

        if(ban)
         {

          memcpy(&B[j][0],GCD,N_max);
          ban=0;
         }

        else
         {

          memcpy(&B[nf+k][0],GCD,N_max);
          k++;
         }
       }
   }
 }
```

```
            nf += k;
            if(nf==(ns+1)) break;
          }

/* se imprimen factores: */

for(i=0;i<nf;i++)
       {
        imprime(&B[i][0]); printf("");
       }

printf("\ n");

       if(nf>1)
        {

         Mult(R1,&B[0][0],&B[1][0]);
         for(i=2;i<nf;i++)
          {

           Mult(R2,R1,&B[i][0]);
           memcpy(R1,R2,N_max);
          }

          if(memcmp(PL,R1,N_max)) printf("Algo salio mal \ n");
        }

        return(nf);
       }
```

// **Función que verifica los factores que se van hallando:**

```
raizPesima(x)

B8 x;
    {
```

```
    int i, j;
    B8 y;

    for(i=1;i<Q;i++)
       {
        for(j=0,y=1;j<P;j++)     y = TMult[y][i];
        if(x==y) return(i);
       }

    printf("Super Error");
    exit(0);


     }
```

// **Función que factoriza el polinomio:**

```
Factoriza(PL)

    B8 PL[N_max];
{
int i, ordf, grd;
B8 Q[N_max], R[N_max], GCD[N_max];

/* Derivemos PL. el resultado en PP */

grd=grado(PL);
memset(PP,0,N_max);

for(i=1;i<=grd;i++)
PP[i-1] = TMult[i%P][PL[i]];

    if(Cal_GCD(GCD,PL,PP)!=1)
      {
       if(!memcmp(PP,POL_Cero,N_max))
        {
         memset(GCD,0,N_max);
```

```
        grd=grado(PL);
        for(i=0;i<=grd;i++)
        if(PL[i]) GCD[i/P] = (B8)    raizPesima(PL[i]);
        for(i=0;i<P;i++)
        Factoriza(GCD);
       }
      else
       {
        Divide(Q,R,PL,GCD);
        Factoriza(GCD);
        Factoriza(Q);
        }
      return;
      }

CalMatB(B,PL);
DiagonalizaMatB(B,PL);
ntf += EncuentraFactores(B,PL);
}
```

//FUNCIONES COMPLEMENTARIAS:
// Función auxiliar para imprimir:

```
imprime(x)
B8 x[  ];
     {

      char linea[512];
      int i, a, s=0;
      i = grado(x);
      printf("\ n");
      printf("(");

      if(i>1)
       {
        s = 1;
```

```
        a = x[i];
        if(a!=1) printf("% d*x^%d",a,i- -);
        else printf("x^%d",i- -);
       }

      for(;i>1;i- -)
       {
        if(!x[i])     continue;
        a = x[i];
        if(a!=1) printf("+%d*x^%d",a,i);
        else printf("+x^%d",i);
       }

      if(x[1])
       {
        a = x[1];
        if(a!=1) printf("%s%d*x",sig[s],a);
        else printf("%sx",sig[s]);
        s = 1;
       }

      if(x[0])
       {
        a = x[0];
        printf("%s%d",sig[s],a);
       }
       printf(")");
    }
```

// **Función de comprobación:**

```
/* Multiplica A y B el resultado lo coloca en R */
/*comprobacion de que todo esta bien*/

Mult(R,A,B)
B8 R[N_max], A[N_max], B[N_max];
    {
```

```
    int i, j, ga, gb;
    memset(R,0,N_max);
    ga=grado(A);
    gb=grado(B);
    for(i=0;i<=ga;i++)
     if(A[i])
      for(j=0;j<=gb;j++)
      R[i+j] = TSuma[R[i+j]][TMult[A[i]][B[j]]];
      printf("= "); imprime(R); printf("\ n");
   }
```

//RUTINA PRINCIPAL:
// El programa principal:

```
int main(int argc, char *argv[ ])
   {
    int i, ordf, grd, fac, off=0;
    B8 PL[N_max];
    B8 GCD[N_max];
    memset(POL_Cero,0,N_max);
    memset(POL_Uno,0,N_max);    POL_Uno[0]=1;
    memset(pol,0,N_max*sizeof(int));
    memset(PL,0,N_max);
    memset(PP,0,N_max);

    if(!strcmp(argv[1],"-q"))
     {
      Q = atoi(argv[2]);    /* Orden del campo */
      off = 2;
     }

   for(i=1;i<argc-off;i++)
    {
     pol[i-1] = (atoi(argv[i+off]));    /*Los coef. de grado menor primero*/
    }
```

```
calTablas();
calinv();
for(i=0;i<N_max;i++)
 {
  if(pol[i]<0) PL[i] = invA[(-pol[i])%Q];
  else PL[i] = pol[i]%Q;
 }

grd=grado(PL);

if(PL[grd]!=1)
 {
  printf("No es monico el polinomio; mult. x %d \ n",fac=invM[PL[grd]]);
   system("PAUSE ");
   for(i=0;i<=grd;i++)
   PL[i] = TMult[fac][PL[i]];
   imprime(PL);
   printf("\ n");
 }

 for(i=1;i<=grd;i++)
 PP[i-1] = TMult[i%P][PL[i]];
 imprime(PL);

 if(Cal_GCD(GCD,PL,PP)!=1)
  {
   printf("(Tiene raices multiples )");
   exit(0);
  }

  printf("= \ n");
  Factoriza(PL);
  printf("Numero Total de Factores = %d \ n",ntf);
  system("PAUSE ");
  return 0;
}
```

Bibliografía

1. MacWilliams, F.J., Sloane, N.J.A.: The theory of error-correcting codes. North-Holland Publishing Company, New York (1977).

2. Lidl, R., Niederreiter, H.: Introduction to finite fields and their applications. Cambridge Univ. Press, Cambridge (1994).

3. Menezes, A., van Oorschot, P., Vanstone, S.: Handbook of applied cryptography. CRC Press (1996).

4. Nelson, V.P., Troy Nagle, H., Carroll, B.D., Irwin, J.D.: Análisis y diseño de circuitos lógicos digitales. Prentice Hall Hispanoamericana, México (1996).

5. Berlekamp, E.R.: Algebraic coding theory. McGraw-Hill Book Company, New York (1968).

6. Dence, J.B., Dence, T.P.: Elements of the theory of numbers. Academic Press, San Diego, USA (1999).

7. Stewart, I.: Galois theory. Chapman & Hall/CRC Press, Florida, U.S. (2004).

8. Koshy, T.: Elementary number theory with applications. Harcourt/Academic Press, California, USA (2002).

9. Weintraub, S.H.: Galois theory. Springer Universitext, New York, USA (2006).