Facultad
de
Ciencias

Multiplicación truncada de polinomios

(Polynomial short product)

Trabajo de Fin de Grado
para acceder al

GRADO EN MATEMÁTICAS

Autor: Mikel Cagigas Riaño

Director: Luis Felipe Tabera Alonso

Junio - 2025

Resumen

A lo largo de esta memoria, estudiaremos la multiplicación truncada de polinomios, que consiste en un método de multiplicación de polinomios en el que solo nos quedamos con una cantidad del resultado completo ligada a la precisión con la que se esté trabajando. Para ello, presentaremos el algoritmo de Mulders, un método que además de calcular el producto truncado de dos polinomios resulta más eficiente que calcular los términos deseados usando el algoritmo clásico de la escuela.

El algoritmo de Mulders necesita de un parámetro a la hora de aplicar un método de divide y vencerás. Estudiamos el parámetro que minimiza el número de multiplicaciones. También estudiamos una variante del algoritmo de Mulders y los comparamos.

Finalmente, aplicamos estos algoritmos a un algoritmo de división con cociente y resto subcuadrático.

Palabras clave: polinomio, serie de potencias truncada, algoritmo de Karatsuba, producto truncado de Mulders, complejidad computacional.

Abstract

In this report, we study the short product of two polynomials, which consists of a method of multiplying polynomials in which only the monomials in the product up to a precision is kept. To this end, we present Mulders' algorithm, a method that, in addition to computing the truncated product of two polynomials, is more efficient than computing only the relevant monomials using the classical schoolbook product.

Mulders' algorithm needs a parameter in order to apply a divide and conquer strategy. We study which is the optimal parameter that minimizes the number of product needed to compute the result. We also study a variant of Mulders algorithm and compare them.

Finally, we apply these algorithms to a subquadratic algorithm that computes the quotient and remainder of two polynomials.

Key words: polynomial, truncated power series, Karatsuba's algorithm, Mulders' short product, computational complexity.

Índice general

1. Introducción **1**

2. Preliminares **3**
2.1. Producto clásico de la escuela 6
2.2. Algoritmo de Karatsuba 7
2.3. Otros algoritmos de multiplicación rápidos 11

3. Algoritmo de Mulders **13**
3.1. Calculando el punto de corte óptimo 16

4. Variante del algoritmo de Mulders **21**
4.1. Mulders y su variante con algoritmos cuadráticos 26

5. Cálculo del cociente y resto **31**
5.1. Método de Newton 31
5.2. División de la escuela 32
5.3. Método de los coeficientes indeterminados 33
5.4. División con resto usando el método de Newton 35

Bibliografía **45**

A. Implementación de los algoritmos utilizados **47**

Capítulo 1

Introducción

En las primeras asignaturas que se cursan durante el grado en matemáticas no se tarda en introducir un nuevo concepto, las series de potencias, cuya importancia es innegable dentro de áreas como el análisis o las ecuaciones diferenciales y que aparecerán repetidamente a lo largo de toda la carrera. Aunque este tipo de elementos entrañan una dificultad computacional: no es físicamente posible poder manejar todos sus términos, pues hay infinitos. Para poder sortear este obstáculo, en la práctica se trabaja con series truncadas, por ejemplo, en vez de trabajar con toda una serie de Taylor, se trabaja con un polinomio de Taylor de orden prefijado. Es decir, en lugar de trabajar en el anillo $R[[x]]$, trabajamos en el anillo cociente con $R[[x]]/(x^n)$.

Si estamos trabajando con series truncadas donde solo nos interesan sus n primeros términos, podemos multiplicar los polinomios de grado $n-1$, obtener un polinomio de grado $2n-2$ y desechar todos los términos a partir de n. Pero calcular una cantidad de términos que más adelante se van a desechar es una práctica computacional ineficiente y una dificultad que se debe solventar.

Para solucionar este problema, en este trabajo se estudiará el producto truncado, un método de multiplicación de polinomios o series truncadas que retorna el resultado hasta la precisión deseada, siendo más eficiente que calcular una multiplicación completa y desechar los términos sobrantes posteriormente. Para ello, estudiaremos el algoritmo de Mulders y su eficiencia, para cuantificar cuál es la ganancia obtenida respecto de únicamente multiplicar y truncar. Resulta curioso descubrir entonces que el método que se nos enseña de pequeños para multiplicar polinomios no es muy eficiente, y esto se ve agravado en cuanto tengamos que trabajar, ya sea con polinomios o series truncadas, de grado muy elevado. Por eso, con este trabajo queremos estudiar otros algoritmos más eficientes a la hora de resolver el problema.

Empezaremos en el Capítulo 2 dando nociones básicas necesarias para el desarrollo de la memoria, junto a un análisis detallado de la necesidad de trabajar con series de potencias hasta una precisión fijada. El problema de eficiencia que resulta de calcular el producto truncado, calculando todos los términos del producto para después truncar, y cómo se puede solucionar esto gracias al producto truncado. Seguidamente, explicamos con detalle los algoritmos de multiplicación de polinomios, el producto de la escuela y de Karatsuba, ya que tendrán su relevancia cuando estudiemos el producto truncado.

A lo largo del Capítulo 3, definiremos una primera idea del algoritmo de Mulders, con las ideas expuestas por el autor en [2], como método para calcular el producto truncado de dos polinomios, y la desarrollaremos hasta poder dar una definición formal. Estudiaremos su complejidad computacional y qué papel juegan el producto de la escuela y de Karatsuba en esto. A partir de aquí, se tratará de calcular el punto de corte óptimo para que el número de operaciones que calcula el algoritmo de Mulders sea mínimo, apoyándonos en enunciados y resultados del artículo [1].

Con el Capítulo 4 introducimos una alternativa al algoritmo de Mulders, cuyo funcionamiento es semejante al algoritmo de Karatsuba, explicando el fundamento teórico que hay detrás de esta idea y calculando su coste computacional para poder compararlo con el algoritmo de Mulders. Seguidamente, analizamos de nuevo el coste de ambos algoritmos, pero esta vez con la condición de que para tamaños de polinomios pequeños, se utiliza el producto de la escuela. Veremos cómo afecta este cambio al coste de cada algoritmo.

Para finalizar, en el Capítulo 5 veremos una aplicación directa para los algoritmo de multiplicación que se estudian en los capítulos anteriores en la división de polinomios. Analizaremos la división clásica de polinomios y su coste, para a continuación definir otro algoritmo, presentado en el libro [7], cuyo coste depende del algoritmo de multiplicación empleado y se basa en calcular la serie de potencias del inverso de un polinomio. En esta referencia, los productos que se realizan son productos truncados, pero los autores no usan ningún algoritmo de producto truncado específico, por lo que experimentamos con el uso de los algoritmos anteriores a este problema.

Capítulo 2

Preliminares

El objetivo de este capítulo es sentar unas bases con las que poder más adelante desarrollar nuestro trabajo sobre la multiplicación truncada de polinomios. Para ello, será necesario saber qué es un polinomio y una serie de potencias en un anillo R, y, en el caso de las series de potencias, conocer a qué fin estamos trabajando con tan solo una parte de esta. Veremos una definición de producto truncado, y, seguidamente, analizaremos con detalle los algoritmos de multiplicación de polinomios más clásicos, el producto de la escuela y el algoritmo de Karatsuba, siguiendo la información y enunciados dados en [7].

Definición 2.0.1. *Sea $(R, +, \cdot)$ un anillo conmutativo e unitario, consideramos el siguiente conjunto como las series de potencias con variable en x y elementos del anillo R:*

$$R[[x]] := R^{\mathbb{N}} := \left\{(a_n)_{n\in\mathbb{N}} \in R^{\mathbb{N}} : a_n \in R,\ \forall n \in \mathbb{N}\right\}$$

a su vez, consideramos el subconjunto de las sucesiones de términos no nulos finitos, es decir:

$$R[x] := \left\{(a_n)_{n\in\mathbb{N}} \in R^{\mathbb{N}} : \exists I \subseteq \mathbb{N},\ I \text{ finito, tal que } a_n = 0,\ \forall n \in \mathbb{N} \setminus I\right\}$$

*donde $(a_n)_{n\in\mathbb{N}}$ es la expresión de una sucesión cualquiera. Como operación aditiva consideraremos la suma de sucesiones habitual, y como multiplicación utilizaremos el producto de convolución *: dadas dos sucesiones $(a_n)_n, (b_n)_n \in R[[x]]$, el producto de convolución retorna una sucesión $(c_n)_n \in R[[x]]$ de forma que*

$$c_k = \sum_{i+j=k} a_i b_j, \forall k \in \mathbb{N}$$

Entonces, la terna $(R[[x]], +, *)$, es un anillo conmutativo e unitario conocido como anillo de series de potencias en una variable con coeficientes en R.

La forma usual de representar un elemento A de $R[[x]]$ es

$$A = \sum_{i=0}^{\infty} a_i x^i$$

con a_i los elementos de la sucesión.

Trabajaremos con series "truncadas" disponiendo tan solo de los primeros n términos de la serie. Formalmente, esto significa trabajar con elementos del anillo cociente $R[[x]]/(x^n)$, por tanto, también necesitamos saber qué es exactamente este anillo y la forma que tienen sus elementos.

Definición 2.0.2. *Sea $(R[[X]], +, *)$ un anillo conmutativo y (x^n) el ideal generado por el elemento x^n. Definimos el cociente $R[[X]]/(x^n)$ cuyos elementos vienen dados por la siguiente asociación:*

$$\begin{aligned} R[[X]] &\longrightarrow R[[X]]/(x^n) \\ \sum_{j=0}^{\infty} a_j &\longmapsto \sum_{j=0}^{\infty} a_j + (x^n) \end{aligned}$$

Entonces, la terna $(R[[X]]/(x^n), +, \cdot)$ es un anillo, donde, dadas P y Q dos series de potencias cualesquiera, las operaciones $+$ y $*$ vienen definidas como:

$$\begin{aligned} [P + (x^n)] \ + \ [Q + (x^n)] &= P + Q + (x^n) \\ [P + (x^n)] \ \cdot \ [Q + (x^n)] &= P * Q + (x^n) \end{aligned}$$

Esto es, la operación suma de dos series cualesquiera en el anillo cociente $R[[X]]/(x^n)$ está definida como la suma de series habitual en $R[[X]]$, y después cocientar, y la operación producto sigue análogamente la misma regla, pero con el producto de convolución mencionado anteriormente.

Trabajar con una serie de potencias completa en un ordenador no es una opción viable, no por capricho, sino por necesidad. El motivo es sencillo: una serie de potencias contiene infinitos términos, y ningún ordenador puede manejar una cantidad infinita de datos. Esto se debe a que los ordenadores funcionan en base al código binario, solo entienden secuencias finitas de ceros y unos, lo que implica que únicamente pueden representar una cantidad a los sumo numerable de objetos concretos. Si $\{0,1\}^{\mathbb{N}}$ es el conjunto de todas las sucesiones posibles de 0's y 1's, se puede definir una biyección con $\mathcal{P}(\mathbb{N})$, que es un conjunto no numerable, luego $\{0,1\}^{\mathbb{N}}$ no es un conjunto numerable, por ende, $R[[x]]$ tampoco lo es, y de ahí que no sea posible almacenar todos los términos de todas las series de potencias en un ordenador. Esto es similar a la aproximación que se hace para los números reales, donde se utiliza la coma flotante que al fin y al cabo no deja de ser una aproximación con una precisión prefijada. Aunque cabe mencionar que hay una diferencia entre aproximar números reales y series de potencias por ordenador, y es que $R[[x]]$ es un anillo, esto es, nuestro conjunto de series de potencias es un grupo conmutativo tanto con la suma con el producto (considerando espacios en donde el conjunto con el producto si es grupo), lo que conlleva a que deban existir ciertas seguridades al hacer operaciones en el ordenador, cosa que con la coma flotante no ocurre, pues, por ejemplo, la propiedad asociativa no se respeta al sumar dos números en coma flotante.

Con estas condiciones se explica que se busque trabajar únicamente con los primeros términos de la serie, de donde podemos suponer que se extraen los datos más significativos para el problema que se desee abordar.

Una vez sentadas las condiciones sobre las que podemos trabajar, abordaremos ahora el problema del producto de series de potencias. Sabiendo cómo funciona este método, buscaremos una alternativa más eficiente para poder solucionar este problema.

Para empezar, supongamos que tenemos una serie de potencias $A = \sum_{i=0}^{\infty} a_i x^i$. La clase de este elemento en el anillo cociente $(R[[x]]/(x^n),\ +,\ \cdot)$ es la vista en la **Definición 2.0.2**, pero de aquí en adelante expresaremos este elemento como

$$A = \sum_{i=0}^{\infty} a_i x^i + \mathcal{O}(x^n)$$

y a este elemento lo llamamos *serie de potencias truncada*. Una serie de potencias truncada es similar a un polinomio de grado $n-1$ con n términos, y esta relación se puede formalizar definiendo una aplicación entre $R[[X]]/(x^n)$ y $R[X]/(x^n)$, que se puede ver de la siguiente forma:

$$\begin{aligned} R[[x]]/(x^n) &\longrightarrow R[x]/(x^n) \\ \sum_{j=0}^{\infty} a_j + \mathcal{O}(x^n) &\longmapsto \sum_{j=0}^{n-1} a_j + \mathcal{O}(x^n) \end{aligned}$$

La aplicación definida es un isomorfismo de anillos, luego toda serie de potencias truncada puede asociarse a un polinomio con n términos de forma que sean iguales a los primeros n términos de dicha serie.

Ahora, dada otra serie truncada $B = \sum_{i=0}^{\infty} b_i x^i + \mathcal{O}(x^n)$, buscaremos una forma de poder multiplicar A por B. Generalmente, el algoritmo que se suele usar para efectuar esta operación es la propia operación producto del anillo $(R[[x]], +, *)$, de la siguiente forma: dados $P_1 = \sum_{i=0}^{n-1} a_i x^i$, $P_2 = \sum_{i=0}^{n-1} b_i x^i \in R[x]$ dos polinomios cuyos términos son idénticos a los n primeros de las series A y B respectivamente. Entonces, multiplicar A por B es igual a multiplicar P_1 por P_2, esto es,

$$AB + \mathcal{O}(x^n) \cong P_1 P_2 + \mathcal{O}(x^n)$$

Llamaremos a esta operación *multiplicación completa*.

Esta forma de multiplicar series es problemática porque es demasiado costosa. El procedimiento a seguir consiste en multiplicar ambas series término a término, obteniendo como resultado un polinomio de grado $2n-1$ con $2n$ términos, pero de esta forma tenemos calculados muchos términos innecesarios, porque al estar trabajando en el anillo $R[[x]]/(x^n)$, los términos de grado mayor o igual que n no nos son útiles, así que se descartan directamente aunque ya hayan sido calculados. Esto implica el cálculo y almacenamiento innecesario de muchas multiplicaciones y sumas, lo que entra en conflicto directo con los objetivos planteados previamente, malgastando así tiempo y memoria del ordenador. Por tanto, nuestro objetivo es buscar un método para multiplicar dos series de potencias de la forma más eficiente posible computacionalmente hablando.

Como hemos comentado, el procedimiento habitual que se sigue para multiplicar dos series de potencias consiste en usar el producto de convolución natural, pero como en nuestro caso solo se va a trabajar con la parte restante al hacer el cociente del elemento por (x^n), la definición de producto que sigue a continuación tiene un mayor sentido en nuestro problema.

Definición 2.0.3. *Sean* $A = \sum_{i=0}^{\infty} a_i x^i + O(x^n)$ *y* $B = \sum_{i=0}^{\infty} b_i x^i + O(x^n)$ *dos series de potencias truncadas en el anillo* $R[[x]]/(x^n)$*, entonces llamamos producto truncado al resultado de multiplicar ambos elementos con la operación producto asociada al anillo al que pertenecen, que tiene la siguiente expresión:*

$$AB \cong \sum_{i=0}^{n-1} (\sum_{k+j=i} a_k b_j) x^i$$

Para mayor simplicidad, esta operación se denotará $AB + \mathcal{O}(x^n)$

Para calcular el producto truncado de dos polinomios, la idea más sencilla aunque menos eficiente es calcular una multiplicación completa, como se ha explicado previamente, y truncar el resultado hasta obtener el número de términos buscado. En esta situación, vamos a estudiar los dos algoritmos de multiplicación más conocidos: el producto de la escuela y el algoritmo de Karatsuba.

2.1. Producto clásico de la escuela

El algoritmo de la escuela, o producto de la escuela, (llamaremos a este algoritmo de cualquiera de las dos formas indistintamente) es el producto de convolución, que es la operación producto del anillo $R[x]$ en el que estamos trabajando, donde dados dos polinomios A y B, el término c_k del polinomio C resultante se calcula multiplicando todos los $a_i b_j$, con a_i el término i-ésimo de A y b_j el j-ésimo de B, de forma que $i + j = k$.

ProductoTradicional(A, B)

Entrada:
$A = \sum_{0 \leq i \leq n} a_i x^i, B = \sum_{0 \leq i \leq n} b_i x^i$, dos polinomios cualesquiera.
Salida:
El producto completo AB

Para $i = 0$ hasta $2n$
$c_i \leftarrow 0$
  Para $j = 0$ hasta i
    $c_i = c_i + a_j \cdot b_{i-j}$

return $C = \sum_{0 \leq i \leq 2n} c_i x^i$

Como cada uno de los términos de A se multiplica por cada uno de los términos de B, si ambos polinomios son de longitud n, la complejidad temporal de este algoritmo es $\mathcal{O}(n^2)$, por lo que, de forma análoga, podemos decir que el algoritmo de la escuela es un *algoritmo cuadrático*

De la misma forma y aunque no sea la situación predominante durante nuestro análisis, dados A y B dos polinomios de tamaño n y m respectivamente, el procedimiento a seguir por el algoritmo es el mismo, solo variará el tamaño del resultado y la complejidad del algoritmo, pues pasa a ser $O(nm)$.

Una vez vista su complejidad, es fácil llegar a la conclusión de que el algoritmo es ineficiente por el elevado coste que presenta, peor en cuanto nuestros polinomios o series de potencias truncadas empiezan a tener muchos términos y se empiezan a requerir muchas multiplicaciones para calcular un producto, por ello, nos interesa buscar otras alternativas más eficientes en términos asintóticos.

Aunque cabe destacar que este producto tradicional tiene ciertas ventajas frente a otros algoritmos de multiplicación. La simpleza de la idea del algoritmo permite una rápida comprensión y evita errores en su implementación, detalles con los que hay que andar con más cuidado al manejar otros algoritmos. Cabe mencionar además que el producto de la escuela en la práctica puede resultar más eficiente que otros algoritmos con un coste asintótico menor a priori, en casos en los que los polinomios a multiplicar sean de un tamaño reducido.

2.2. Algoritmo de Karatsuba

Por su parte, el algoritmo de Karatsuba, que en ocasiones nos referiremos a él simplemente como *Karatsuba*, se trata de un algoritmo "Divide y vencerás". Es decir, toma el problema original, de un tamaño cualquiera, y lo divide en subproblemas de menor tamaño, resolviendo cada subproblema de forma independiente y combinando finalmente las soluciones para obtener la solución del problema original. Para entender mejor cómo funciona el algoritmo de Karatsuba, presentamos un ejemplo genérico: tomamos $A = \sum_{i=0}^{n-1} a_i x^i$ y $B = \sum_{i=0}^{n-1} b_i x^i$ dos polinomios de tamaño n pertenecientes a $R[x]$, entonces vemos que se pueden separar en dos trozos de la siguiente forma:

$$A = A_1 \cdot x^m + A_2,\ B = B_1 \cdot x^m + B_2, \text{ con } m = \left\lceil \frac{n}{2} \right\rceil$$

de forma que al multiplicar ambos polinomios con el producto del anillo $R[x]$ se obtiene

$$A \cdot B = (A_1 \cdot x^m + A_2) \cdot (B_1 \cdot x^m + B_2) = A_1 B_1 \cdot x^{2m} + (A_1 B_2 + A_2 B_1) \cdot x^m + A_2 B_2.$$

Si se continuase con el procedimiento habitual, se calcularían 4 productos, pero Karatsuba se dió cuenta de que con unas pocas manipulaciones, se podía reducir en 1 el número de productos calculados, a cambio de unas pocas operaciones sumas adicionales. Esto es, llamando a cada uno de los términos del producto de A y B como:

$$\begin{aligned} z_0 &= A_1 B_1 \\ z_1 &= A_1 B_2 + A_2 B_1 \\ z_2 &= A_2 B_2 \end{aligned}$$

el término z_1 se puede reescribir como

$$z_1 = A_1 B_2 + A_2 B_1 = (A_1 + A_2) \cdot (B_1 + B_2) - A_1 B_1 - A_2 B_2 =$$

$$= (A_1 + A_2) \cdot (B_1 + B_2) - z0 - z2.$$

Con esta simple modificación, aunque el número total de operaciones se vea incrementado respecto al producto de la escuela, lo que aumenta es el número de sumas calculadas, que no son tan costosas en comparación a las multiplicaciones realizadas, cuyo cantidad se ve reducida.

Hay una serie de factores a tener en cuenta para obtener todas las ventajas que puede otorgar Karatsuba respecto a otros algoritmos de multiplicación.

1. Para empezar, el algoritmo debe aplicarse recursivamente, es decir, una vez divididos los polinomios a multiplicar y definidos los factores z_0, z_1 y z_2, estos también habrán de ser calculados con Karatsuba, y así sucesivamente para cada uno de los productos que se deban realizar.

2. En muchas ocasiones Karatsuba solo se explica para el caso en el que n, el tamaño de las series a multiplicar, es una potencia de 2. En este caso cada una de las recursiones es más sencilla porque siempre se tendrá que $m = \frac{n}{2}$, y los polinomios se dividirán siempre a la mitad en dos partes de igual longitud.

3. n debe ser suficientemente grande. Si este no es el caso, como el número total de operaciones se ve aumentado respecto a un algoritmo cuadrático, resultará más costoso emplear Karatsuba que el producto de la escuela, aunque sea contraproducente.

Una vez explicado y entendido cómo funciona el algoritmo de Karatsuba, podemos dar un pseudocódigo con los pasos a seguir cuando se quiera aplicar.

Karatsuba(A, B, n)

Entrada:
1. $A = \sum_{0 \leq i \leq n} a_i x^i, B = \sum_{0 \leq i \leq n} b_i x^i$, dos polinomios cualesquiera.
2. n el tamaño de los polinomios A y B.

Salida:
El producto completo AB

if $n = 1$ **then**
  return AB
end if
$m \leftarrow \lceil \frac{n}{2} \rceil$
$A = A_1 x^m + A_2$, $B = B_1 x^m + B_2$
$z_0 \leftarrow$ Karatsuba(A_1, B_1, m)
$z_2 \leftarrow$ Karatsuba$(A_2, B_2, \lfloor n/2 \rfloor)$
$z_1 \leftarrow$ Karatsuba$(A_1 + A_2, B_1 + B_2, m) - z_0 - z_2$
return $z_0 x^n + z_1 x^m + z_2$

En el caso en el que los polinomios a multiplicar no fuesen del mismo tamaño, por ejemplo, A y B dos polinomios de longitud n y m respectivamente, con $n > m$, para multiplicar ambos polinomios mediante el algoritmo de Karatsuba se dividirá A en $A_1, A_2, ..., A_q$ trozos de tamaño m, y cada uno de estos trozos será multiplicado por B mediante Karatsuba. Si estamos en la situación donde n no es un múltiplo de m, una de las partes en las que se ha dividido A será un polinomio A_r de tamaño menor que m. En este caso, se repetiría el mismo proceso. El

procedimiento a seguir es el mismo que el que se acaba de mostrar, y el resultado final será la suma de cada una de las multiplicaciones realizadas.

Para calcular la complejidad de este algoritmo, será necesario utilizar el *Teorema Maestro*, que es una fórmula capaz de dar una cota de la complejidad de ecuaciones o algoritmos de recurrencia como es el caso de Karatsuba. Las nociones sobre el Teorema Maestro están sacadas de los apuntes de la asignatura Algorítmica y Complejidad en [3].

Para aplicar el Teorema Maestro, se ha de verificar que el algoritmo que se quiere estudiar sigue la siguiente relación:

$$T(n) = aT(n/b) + O(n^c)$$

donde

- a es el número de subproblemas en la recursión, o visto de otra forma, el número de llamadas recursivas que hace el algoritmo en una iteración. En este caso, $a = 3$.
- b es el tamaño por el cuál se divide el problema original. En el caso de Karatsuba, el problema se divide cada vez a la mitad, $b = 2$
- $O(n^c)$ es la complejidad de aquellas operaciones fuera de la recursión, donde lo que es necesario calcular es el valor de c. Vamos a suponer que realizar una suma tiene un coste computacional de $O(1)$. En el caso de Karatsuba, las operaciones a realizar fuera de la recursión son sumas o restas: en el término z_1, se realizan dos sumas y dos restas, y al final del algoritmo, se suman los tres términos z_0, z_1 y z_2. Las dos sumas a calcular en z_1 se realizan a polinomios de tamaño $n/2$, por lo que no son importantes en nuestro análisis. Sin embargo, en las operaciones restantes, o bien se suman o bien se restan polinomios de tamaño n, lo cuál significa que en cada caso se calculan n sumas o restas, por lo que el coste de estas operaciones es $\mathcal{O}(n)$. Como no se realiza ninguna otra operación de mayor coste, se tiene entonces que la complejidad de las operaciones fuera de la recursión es de $O(n)$, por lo que podemos concluir que nuestro c buscado es 1.

Para finalizar el análisis con el Teorema Maestro, hay que distinguir uno de los tres posibles casos existentes entre la relación de a y b^c. Para el algoritmo de Karatsuba, se cumple que $a > b^c$ con los valores a, b y c vistos arriba. Por tanto, la complejidad temporal del algoritmo es $O(n^{\log_b a})$, esto es, sustituyendo, se obtiene que la complejidad del algoritmo de Karatsuba es $O(n^{log_2 3}) \approx O(n^{1,585})$, una mejora considerable respecto al producto de la escuela en términos asintóticos.

Ejemplo 2.2.1. *Recuperamos los polinomios utilizados en el ejemplo del producto de la escuela,* $A(x) = 3x^3 + 4x^2 + x - 2$ *y* $B(x) = -x^3 + 7x^2 - x - 5$. *Utilizando el algoritmo de Karatsuba para multiplicar ambos polinomios, primero tenemos que separarlos en "parte grande", y "parte pequeña". Para ello, como* $n = 3$, *vemos que* $m_1 = \lceil \frac{n}{2} \rceil = 2$, *luego separamos* A *y* B *como*

$$A = (3x + 4)x^2 + x - 2 = A_1x^2 + A_2,\ B = (-x + 7)x^2 + (-x - 5) = B_1x^2 + B_2,$$

de forma que, multiplicar A *por* B *siguiendo Karatsuba es igual a la siguiente expresión*

$$A \cdot B = (A_1 \cdot x^2 + A_2) \cdot (B_1 \cdot x^2 + B_2) = A_1B_1 \cdot x^4 + ((A_1 + A_2) + (B_1 + B_2) - A_1B_1 - A_2B_2) \cdot x^2 + A_2B_2.$$

Calculemos cada uno de los productos usando Karatsuba de manera recursiva.

1. *Como estamos aplicando Karatsuba de manera recursiva, de nuevo, vamos a tener que separar A_1 y B_1 en parte grande y parte pequeña, donde, si n_1 es el grado de ambos polinomios, el factor que separa los polinomios es $m_2 = \lceil \frac{n_1}{2} \rceil = 1$. Con esto en mente, separamos A_1 y B_1 como*

$$A_1 = 3x + 4,\ B_1 = (-1)x + 7.$$

Multiplicamos ambos polinomios, obteniendo así

$$A_1B_1 = 3 \cdot (-1)x^2 + x((3+4)(-1+7) - ((-3)+28)) + 28 = -3x^2 + 17x + 28.$$

2. *Repetimos el mismo procedimiento para el producto de A_2 por B_2. De nuevo, el grado de ambos polinomios es $n_2 = 2$, luego $m_3 = \lceil \frac{n_2}{2} \rceil = 1$, y por tanto, podemos separar A_2 y B_2 como*

$$A_2 = x - 2,\ B_2 = (-1)x - 5.$$

Al multiplicar ambos polinomios siguiendo el mismo procedimiento obtenemos

$$A_2B_2 = 1 \cdot (-1)x^2 + x((1-2)(-1-5) - ((-1)+10)) + 10 = -x^2 - 3x + 10.$$

3. *Habiendo calculado ya A_1B_1 y A_2B_2, solo resta por calcular $(A_1 + A_2)(B_1 + B_2)$. Como hemos hecho en los dos casos anteriores, sabiendo que $n_3 = 1$ es el grado de cada uno de los polinomios con los que estamos operando y, por tanto, $m_4 = \lceil \frac{n_3}{2} \rceil = 1$, separamos $A_1 + A_2$ y $B_1 + B_2$ como*

$$A_1 + A_2 = (3+1)x + 4 - 2 = 4x + 2,\ B_1 + B_2 = (-1-1)x + 7 - 5 = -2x + 2.$$

Solo resta calcular el producto $(A_1 + A_2)(B_1 + B_2)$, cuyo resultado es

$$(A_1 + A_2)(B_1 + B_2) = 4 \cdot (-2)x^2 + x((4+2)(-2+2) - ((-8)+4)) + 4 =$$

$$= -8x^2 + 4x + 4.$$

Por último, sustituimos cada uno de los polinomios calculados en sus valores correspondientes en la igualdad de AB, y operamos como es habitual.

$$AB = (-3x^2 + 17x + 28)x^4 + x^2(-8x^2 + 4x + 4 - (-3x^2 + 17x + 28 - x^2 - 3x + 10))$$

$$-x^2 - 3x + 10 = -3x^6 + 17x^5 + 24x^4 - 10x^3 - 35x^2 - 3x + 10$$

2.3. Otros algoritmos de multiplicación rápidos

Aunque no sean el ámbito principal de estudio de la memoria, es interesante destacar que el algoritmo de Karatsuba está lejos de ser el algoritmo más rápido y eficiente conocido para multiplicar dos polinomios cualesquiera. En ese caso se debe mencionar el algoritmo basado en utilizar transformadas rápidas de Fourier (FFT) de forma recursiva, que consigue multiplicar dos polinomios de tamaño n con una complejidad de $O(n\log(n))$, o el algoritmo de Schönhage–Strassen, que hace la misma operación en un tiempo $O(n\log(n)\log(\log(n)))$, aunque ambos algoritmos requieren que se cumplan ciertas condiciones en el anillo de coeficientes en el que se esté trabajando para poder ser utilizados. Además, las diferencias que presentan estos algoritmos son más apreciables a polinomios de longitud mayor, donde estamos hablando de longitudes muy grandes, mayores que las que vamos a considerar en nuestro trabajo, lo que unido a la falta de tiempo causa que no se hayan podido estudiar junto al resto.

Capítulo 3

Algoritmo de Mulders

En la sección anterior, hemos dado un primer vistazo al problema de multiplicar series de potencias, viendo lo ineficiente que puede llegar a ser. Por eso, vamos a estudiar una alternativa en el algoritmo de Mulders, un método para calcular un producto truncado de dos series truncadas de forma más eficiente.

Presentaremos primero el algoritmo original de Mulders [2] analizando qué papel desempeñan los algoritmos de multiplicación descritos en el capítulo anterior.

Posteriormente, veremos cómo se puede mejorar la complejidad del algoritmo de Mulders y qué conclusiones se pueden extraer. Para ello, nos basaremos en los resultados dados en [1].

El ánalisis que realiza Mulders comienza dada una serie truncada cualquiera $A = \sum_{i=0}^{n-1} a_i x^i + \mathcal{O}(x^n)$, viendo que se puede escribir como suma de dos polinomios de grado menor que n,

$$A = P_1 + x^k Q_1 + \mathcal{O}(x^n)$$

de forma que $\deg(P_1) < k$ y k es un número entero mayor o igual a $n/2$. De esta forma, A queda descompuesto como un polinomio de tamaño menor que k y otro de tamaño $n-k$, con los términos situados entre x^k y x^n, luego realizando la misma descomposición para otro polinomio B cualquiera, tenemos:

$$\begin{aligned} AB &\cong (P_1 + x^k Q_1 + \mathcal{O}(x^n))(P_2 + x^k Q_2 + \mathcal{O}(x^n)) = \\ &= P_1 P_2 + x^k (P_1 Q_2 + P_2 Q_1) + x^{2k} Q_1 Q_2 + \mathcal{O}(x^n). \end{aligned}$$

El último término del producto se puede descartar directamente, pues por definición de k, $2k \geq n$, así que estos términos son de un grado mayor a la precisión con la que estamos trabajando y podemos y debemos evitar calcularlos desde el principio. Pero, además, es necesario realizar unos pequeños ajustes adicionales a esta expresión:

- En su artículo, Mulders establece que k sea un número de la forma βn, con $1/2 \leq \beta < 1$, cuyo valor exacto depende del algoritmo empleado para calcular $P_1 P_2$.
- El primer término $P_1 P_2$ se calculará como un producto completo con un algoritmo de multiplicación de polinomios cualquiera, pero P_1 y P_2 son polinomios con k términos y

grado $k-1$ donde $k = \beta n$ con $\frac{1}{2} \leq \beta < 1$, por tanto al realizar el producto completo se calcularán coeficientes correspondientes a términos de grado mayor que $n-1$. Computacionalmente, la solución es truncar el resultado del producto a la precisión deseada. Si bien de esta forma también se realizan operaciones innecesarias que no ayudan al hablar de la eficiencia del algoritmo, esta pérdida es mucho menos significativa con respecto a la multiplicación completa gracias a la separación en partes de ambos polinomios.

- P_1 y Q_2 (análogamente P_2 y Q_1) son dos polinomios de grado $k-1$ y $n-k$ respectivamente. Hay que tener en cuenta que el producto P_1Q_2 esta a su vez multiplicado por x^k, donde si bien multiplicar x^k por los términos del producto anterior no es una operación costosa (solo se trata de mover los coeficientes de grado x^i a grado x^{i+k}), en algún punto estaremos de nuevo ante el problema de haber calculado términos innecesarios porque se sobrepasará la precisión buscada en x^n. Para evitar esto, el procedimiento a seguir es hacer recursivamente otro producto truncado.

De esta forma, podemos dar una definición formal de Mulders:

Definición 3.0.1. *Sean A y B dos series truncadas con precisión $\mathcal{O}(x^n)$, esto es, n términos y $deg(A) < n$, de forma que se pueden descomponer A y B como $A = P_1 + x^kQ_1 + \mathcal{O}(x^n)$ y $B = P_2 + x^kQ_2 + \mathcal{O}(x^n)$. Entonces el algoritmo de Mulders, o simplemente Mulders, para multiplicar series truncadas consiste en la siguiente expresión:*

$$AB \cong P_1P_2 + x^k(P_1Q_2 + P_2Q_1 \mod x^{n-k}) + \mathcal{O}(x^n).$$

Esto es, el producto truncado de dos series de potencias se puede calcular operando un producto completo (que habrá que truncar) de dos polinomios de tamaño k y dos productos truncados de tamaño $n-k$, realizados de manera recursiva.

Una vez definido el algoritmo de Mulders, debemos calcular su complejidad computacional en cuanto a número de operaciones se refiere. Para ello, y teniendo en cuenta que el primero de los términos del algoritmo se calcula como un producto completo, debemos ver también cuál es el coste de una operación de este estilo.

Definición 3.0.2. *Dado un algoritmo cualquiera para efectuar el producto completo entre dos series truncadas de tamaño n, entonces $M(n)$ es el coste computacional de ejecutar el algoritmo para dichos polinomios.*

Aproximaremos este coste de un producto completo como $M(n) \approx n^\alpha$, donde α denota la complejidad del algoritmo de multiplicación usado, y recordamos que, en una primera aproximación, el punto de corte k con el que Mulders separa las series a multiplicar es de la forma $k = \beta n$, con $1/2 \leq \beta < 1$. Con estas consideraciones y basándonos en la **Definición 3.0.1**, podemos dar un primer vistazo de lo que es el coste del algoritmo de Mulders en cuanto a número de multiplicaciones calculadas.

Definición 3.0.3. *Sea $M(n)$ el coste de un algoritmo como en la **Definición 3.0.2**, entonces el número de multiplicaciones calculadas por el algoritmo de Mulders es*

$$S(n) = M(n) + 2S(n - \lceil \beta n \rceil).$$

En su artículo [2], Mulders realiza un análisis heurístico para calcular cuál es el valor mínimo de β de forma que el punto de corte escogido sea óptimo en función del algoritmo escogido para la multiplicación completa, obteniendo los siguientes resultados:

- Multiplicación tradicional: con $\alpha = 2$, en el caso de que el algoritmo elegido para calcular el producto completo sea el producto de la escuela, se tiene que el coste aproximado de realizar una multiplicación entera es $M(n) \approx n^2$, y el factor beta que se obtiene es $\beta = n/2$, luego el punto de corte final k usado por el algoritmo es $n/2$, lo cuál no es sorprendente. Lo que esto significa es que los polinomios se dividen a la mitad y Mulders calcula todos los productos como lo haría el producto de la escuela, pero trunca el resultado para solo quedarse con los primeros n términos del resultado. Sin embargo, esto es el resultado de multiplicar los primeros trozos de cada polinomio original, por tanto el resto de operaciones realizadas resultan en un gasto de memoria y tiempo innecesarios. Por eso, siguiendo este procedimiento, el algoritmo de Mulders no aprecia ninguna mejora respecto a hacer el producto completo de las dos series directamente: ambos procedimientos realizan el mismo número de operaciones.

- Algoritmo de Karatsuba: en este caso con $\alpha = log_2 3$, Mulders obtiene $\beta \approx 0{,}694$, y comenta que su algoritmo si que presenta alguna mejora respecto a un producto completo, pero solo en los casos en los que los polinomios no sean de un tamaño muy grande. Asintóticamente, esta ganancia se pierde rápidamente, no habría ninguna diferencia entre usar una forma u otra.

Pero este análisis, hecho por Mulders en [2], no puede ser considerado como definitivo, ya que en todo momento se están tomando aproximaciones, tanto para el punto de corte como de los costes de los productos completos. El punto de corte k que Mulders calcula puede, y debe, ser analizado rigurosamente, ya que la aproximación de Mulders puede no ser la óptima y que esto provoque que se estén calculando multiplicaciones de más que con otro valor de k ni se tengan en cuenta. Dicho análisis se realiza en [1]. Nuestro trabajo será seguir dicho artículo para encontrar el punto de corte óptimo k para la función $S(n)$, esto es, analizar cuál es el punto de corte en función del tamaño de las series truncadas de forma que el número de multiplicaciones que realiza el algoritmo sea el menor posible.

El cálculo del punto de corte óptimo para el número de multiplicaciones requiere que actualicemos la definición de $S(n)$.

Definición 3.0.4. *Sea $M(n)$ el coste de un algoritmo como en la **Definición 3.0.2** , entonces definimos el número de multiplicaciones que calcula el algoritmo de Mulders como*

$$S(n) = \min_{\frac{n}{2} \leq k \leq n} \; M(n) + 2S(n-k)$$

De cara a la experimentación hemos considerado que nuestros polinomios, o series truncadas, sean listas de coeficientes, hemos creado una clase auxiliar, a la que nos referiremos como *Elemento*, que nos permite llevar un seguimiento del número de operaciones efectuadas, aunque en el apartado teórico solo estemos teniendo en cuenta las multiplicaciones porque entendemos

que conllevan un mayor coste, también vamos a llevar un seguimiento de cada operación suma efectuada, pues en aquellas pruebas en las que se estén manejando polinomios de gran tamaño, las sumas también pueden lastrar la eficiencia de los algoritmos.

3.1. Calculando el punto de corte óptimo

Para calcular el punto de corte k óptimo de acuerdo a la **Definición 3.0.4**, será necesario ver que se dan una serie de condiciones sobre la propia función $S(n)$. También se verá una función auxiliar que nos ayudará a ver cuál es el punto óptimo que se está buscando.

Además, como el análisis se va a realizar para el caso en el que los productos completos de Mulders se calculen con el algoritmo de Karatsuba, damos las siguientes propiedades con respecto al coste del algoritmo.

Definición 3.1.1. *Sea $M(k)$ el coste de un algoritmo como en la **Definición 3.0.2**. Entonces al considerar el algoritmo de Karatsuba como algoritmo con el que calcular el producto completo de dos series truncadas en el primer paso del algoritmo de Mulders, denotamos la complejidad $M(k)$ como $M(k) = K(k)$, que sigue la relación $K(1) = 1$, $K(n) = 2K(\lceil \frac{n}{2} \rceil) + K(\lfloor \frac{n}{2} \rfloor)$ para $n \geq 2$.*

Empezamos viendo que la función $S(n)$, el número de operaciones realizados por Mulders, es una función no decreciente, y seguidamente, que cumple una desigualdad similar a la definida durante el número de operaciones de Karatsuba previamente.

Lema 3.1.2. *Sea $S(n)$ la igualdad del número de productos realizados con el algoritmo de Mulders. Entonces, si $M(n)$ es una función no decreciente, se cumple que $S(n+1) \geq S(n)$.*

Demostración. Sea $s(n)$ un punto cualquiera de la función $S(n)$. Si $s(n+1) = n+1$, entonces por definición de $S(n)$ tenemos que

$$S(n+1) = M(n+1) + S(n-(n+1)) = M(n+1) \geq M(n) \geq S(n)$$

Con lo que tenemos el caso trivial. Veamos ahora que pasa si $s(n+1) \neq n+1$

$$\begin{aligned} S(n) &\leq M(s(n+1)) + 2S(n - s(n+1)) \\ &\leq M(s(n+1)) + 2S(n+1-s(n+1)) - \\ &\quad 2[S(n+1-s(n+1)) - S(n-s(n+1))] \\ &\leq S(n+1) - 2[S(n+1-s(n+1)) - S(n-s(n+1))] \end{aligned}$$

En la segunda desigualdad simplemente se ha sumado y restado $2S(n+1-s(n+1))$, y se ha sacado factor común -2, y la tercera desigualdad es simplemente la definición de $S(n+1)$ restada por un valor, que es positivo por inducción, luego la hipótesis se sigue automáticamente. □

Lema 3.1.3. *Sea S(n) el número de productos efectuados por el algoritmo de Mulder con un punto de corte óptimo. Entonces, se cumple la relación de recurrencia S(1) = 1,* $S(n) \geq S(\lceil \frac{n}{2} \rceil) + 2S(\lfloor \frac{n}{2} \rfloor)$.

Demostración. De nuevo, usaremos inducción para la prueba de este lema, siendo $n = 1$ y $n = 2$ los casos base triviales.

Suponemos ahora que se cumple la hipótesis para $m < n$. Por definición de $K(k)$ e inducción, se cumple que:

$$S(n) = \min_{\lceil \frac{n}{2} \rceil \leq \ell \leq n} (K(\ell) + 2S(n-\ell))$$

$$\geq \min_{\lceil \frac{n}{2} \rceil \leq \ell \leq n} \left(2K\left(\left\lceil \frac{\ell}{2} \right\rceil\right) + K\left(\left\lfloor \frac{\ell}{2} \right\rfloor\right) + 2S\left(\left\lceil \frac{n-\ell}{2} \right\rceil\right) + 4S\left(\left\lfloor \frac{n-\ell}{2} \right\rfloor\right)\right),$$

$$\geq 2 \min_{\lceil \frac{n}{2} \rceil \leq \ell \leq n} \left(K\left(\left\lceil \frac{\ell}{2} \right\rceil\right) + 2S\left(\left\lfloor \frac{n-\ell}{2} \right\rfloor\right)\right)$$

$$+ \min_{\lceil \frac{n}{2} \rceil \leq \ell \leq n} \left(K\left(\left\lfloor \frac{\ell}{2} \right\rfloor\right) + 2S\left(\left\lceil \frac{n-\ell}{2} \right\rceil\right)\right). \tag{3.1}$$

Siendo la última desigualdad cierta gracias a una propiedad de la función mínimo, que el mínimo de la suma de dos valores es mayor o igual que la suma de los mínimos de esos valores por separado, es decir $\text{mín}(a+b) \geq \text{mín}(a) + \text{mín}(b)$.

Empecemos suponiendo que tanto l como n son impares, luego $l = 2l_1 + 1$ y $n = 2n_1 + 1$, y usando el **Lema 3.1.2**:

$$S(n) \geq 2 \min_{\lceil \frac{n_1+1}{2} \rceil \leq \ell_1+1 \leq n_1+1} (K(\ell_1+1) + 2S((n_1+1) - (\ell_1+1)))$$

$$+ \min_{\lceil \frac{n_1}{2} \rceil \leq \ell_1 \leq n_1} (K(\ell_1) + 2S(n_1 - \ell_1)).$$

El lado derecho de la desigualdad es precisamente $2S(n_1+1) + S(n_1) \geq S(\lceil \frac{n}{2} \rceil) + 2S(\lfloor \frac{n}{2} \rfloor)$, de nuevo por el **Lema 3.1.2**.

Supongamos ahora que o n o l es par (no ambos a la vez). Entonces se cumple que $\lfloor \frac{n-\ell}{2} \rfloor = \lfloor \frac{n}{2} \rfloor - \lceil \frac{\ell}{2} \rceil$ y $\lceil \frac{n-\ell}{2} \rceil = \lceil \frac{n}{2} \rceil - \lfloor \frac{\ell}{2} \rfloor$; además, como $l \leq n$, tenemos que $\lceil \frac{\ell}{2} \rceil \leq \lfloor \frac{n}{2} \rfloor$.

Volviendo a la desigualdad (3.1) y aplicando las propiedades aquí descritas, llegamos a ver que:

$$S(n) \geq 2 \min_{\left\lceil \frac{\lceil \frac{n}{2} \rceil}{2} \right\rceil \leq \lceil \frac{\ell}{2} \rceil \leq \lfloor \frac{n}{2} \rfloor} \left(K\left(\left\lceil \frac{\ell}{2} \right\rceil\right) + 2S\left(\left\lfloor \frac{n}{2} \right\rfloor - \left\lceil \frac{\ell}{2} \right\rceil\right)\right)$$

$$+ \min_{\left\lfloor \frac{\lceil \frac{n}{2} \rceil}{2} \right\rfloor \leq \lfloor \frac{\ell}{2} \rfloor \leq \lfloor \frac{n}{2} \rfloor} \left(K\left(\left\lfloor \frac{\ell}{2} \right\rfloor\right) + 2S\left(\left\lceil \frac{n}{2} \right\rceil - \left\lfloor \frac{\ell}{2} \right\rfloor\right)\right). \tag{3.2}$$

En esta ocasión, llamamos (3.2) únicamente al segundo sumatorio de la desigualdad anterior ya que tendremos que volver a acudir a él más adelante. El primer término siempre será mayor que $2S(\lfloor \frac{n}{2} \rfloor)$, gracias al **Lema 3.1.2** y a que comparten el mismo intervalo de puntos admisibles excepto por el lado de la cota inferior, donde $2S(\lfloor \frac{n}{2} \rfloor)$ tiene algún punto más. Para el segundo término (3.2) y poder concluir la demostración, debemos realizar un análisis en mayor profundidad.

Queremos ver que el segundo término del sumatorio es mayor que $S(\lceil \frac{n}{2} \rceil)$. Por definición, tenemos que

$$S\left(\left\lceil \frac{n}{2} \right\rceil\right) = \min_{\frac{\lceil \frac{n}{2} \rceil}{2} \leq \lfloor \frac{\ell}{2} \rfloor \leq \lfloor \frac{n}{2} \rfloor} \left(K\left(\left\lfloor \frac{\ell}{2} \right\rfloor\right) + 2S\left(\left\lceil \frac{n}{2} \right\rceil - \left\lfloor \frac{\ell}{2} \right\rfloor\right)\right)$$

Los intervalos entre los que se encuentran los valores con los que evaluar las expresiones anteriores son iguales excepto en la cota inferior, luego no podemos asegurar abiertamente que $S(\lceil \frac{n}{2} \rceil)$ sea menor o igual en todo punto que la expresión (3.2).

Llamando $k = \lceil \frac{l}{2} \rceil$ por simplicidad, tenemos que estudiar los casos $n = 4k+1$ y $n = 4k+2$, pues es en donde existe un punto más en el intervalo de puntos en los que se puede evaluar la expresión (3.2) respecto del intervalo de $S(\lceil \frac{n}{2} \rceil)$.

- Si $n = 4k+1$, el único valor par que puede tomar ℓ es $2k$, contradiciendo la condición $\ell \geq \lceil \frac{n}{2} \rceil$, luego este caso no puede ocurrir.
- Si $n = 4k+2$, el único valor de l que cumple $\ell \geq \lceil \frac{n}{2} \rceil$ es $\ell = \frac{n}{2}$, en cuyo caso obtenemos $S(n) = K(n/2) + 2S(n/2) \geq 3S(n/2) = S(\lceil \frac{n}{2} \rceil) + 2S(\lfloor \frac{n}{2} \rfloor)$, por tanto se cumple la desigualdad que buscamos.

Habiendo comprobado ya los únicos casos especiales, podemos afirmar que el segundo término del sumatorio es mayor que $S(\lceil \frac{n}{2} \rceil)$, lo que junto al resultado anterior concluye la demostración. □

Una vez visto que la función $S(n)$ es no decreciente y que cumple la relación del **Lema 3.1.3**, estudiemos ahora una función auxiliar en el punto $2^{\lfloor log_2 n \rfloor}$, esto es, la potencia de 2 más cercana a n.

Teorema 3.1.4. *Sea $S^*(n)$ el número de multiplicaciones obtenidas tomando como punto de corte $s(n) = 2^{\lfloor \log_2 n \rfloor}$. Entonces, $S^*(1) = 1,\ S^*(n) = S^*(\lceil \frac{n}{2} \rceil) + 2S^*(\lfloor \frac{n}{2} \rfloor)$.*

Demostración. Realizamos la demostración por inducción en $s(n)$, el punto de corte. El caso base $n = 1$ es trivial, así que suponemos ahora que el resultado es cierto para un $n \in [2^k, 2^{k+1})$. Veamos ahora que la hipótesis se cumple para un n en el intervalo $[2^{k+1}, 2^{k+2})$.

Para $2n$, tenemos que $S^*(2n) = K(2^{k+1}) + 2S^*(2(n-2^k)) = (1) + (2)$, donde (1) y (2) son las siguientes expresiones:

1. 2^{k+1} es el punto de corte utilizado en $S^*(n)$ por hipótesis de inducción, luego usando las propiedades vistas anteriormente para $K(n)$, tenemos que: $K(2^{k+1}) = 2K(\lceil \frac{2^{k+1}}{2} \rceil) + K(\lfloor \frac{2^{k+1}}{2} \rfloor) = 3K(2^k)$

2. Primeramente, sabemos que $2n \in [2^{k+1}, 2^{k+2})$, o visto de otra forma, $2n < 2^{k+2} \rightarrow 2n - 2^{k+1} < 2^{k+2} - 2^{k+1} = 2^{k+1} \rightarrow 2(n-2^k) < 2^{k+1}$, por lo que se cumple la condición de la hipótesis y podemos utilizar el resultado sobre $2S^*(2(n-2^k))$, obteniendo la siguiente cadena de igualdades:

$$2S^*(2(n-2^k)) = 2\left(S^*\left(\left\lceil\frac{2(n-2^k)}{2}\right\rceil\right) + 2S^*\left(\left\lfloor\frac{2(n-2^k)}{2}\right\rfloor\right)\right) = 6S^*(n-2^k)$$

Sumando (1) y (2), la igualdad resultante es $S^*(2n) = 3K(2^k) + 6S^*(n-2^k) = 3S^*(n)$, donde queríamos llegar.

Ahora, para el caso $2n+1$, vemos que $S^*(2n+1) = K(2^{k+1}) + 2S^*(2(n-2^k)+1) = (1)+(3)$, donde (1) es el desarrollo de $K(2^{k+1})$ visto en el caso anterior, y (3) es el siguiente desarrollo:

- Usando la misma idea que en la prueba para $2n$, la cadena de equivalencias es análoga. Tenemos que $2n+1 \in [2^{k+1}, 2^{k+2})$, esto es, $2n+1 < 2^{k+2} \rightarrow 2n+1 \leq 2^{k+2}-1 \rightarrow 2n+2-2^{k+1} \leq 2^{k+2}-2^{k+1} \rightarrow 2(n-2^k)+2 \leq 2^{k+1} \rightarrow 2(n-2^k)+1 < 2^{k+1}$ y de manera idéntica al caso $2n$, concluimos que podemos utilizar la hipótesis de inducción, por tanto, aplicandola para la expresión $2(n-2^k)+1$, obtenemos:

$$2S^*(2(n-2^k)+1) = 2\left(S^*\left(\left\lceil\frac{2(n-2^k)+1}{2}\right\rceil\right) + 2S^*\left(\left\lfloor\frac{2(n-2^k)+1}{2}\right\rfloor\right)\right) =$$
$$2S^*(n-2^k+1) + 4S^*(n-2^k)$$

Ahora, en el caso de que $n+1 < 2^{k+1}$, el resultado se cumple directamente, pues

$$S^*(2n+1) = 3K(2^k) + 2S^*(n-2^k+1) + 4S^*(n-2^k) = S^*(n+1) + 2S^*(n)$$

usando simplemente la **Definición 3.0.4**. Y aunque nos encontremos con el posible caso $n+1 = 2^k$, no habría mayor problema ya que la cadena de igualdades se cumpliría igualmente, esto es:

- $S^*(2^{k+1}) = K(2^{k+1}) + 2S^*(2^{k+1}-2^{k+1}) = 3K(2^k)$ porque $S(0)$ es 0 y por las propiedades de la recurrencia $K(n)$.
- $2S^*(n-2^k+1) + 4S^*(n-2^k) = 2S^*(2^{k+1}-2^k) + 4S^*(2^{k+1}-1-2^k) =$

$$= 2(S^*(2^k) + 2S^*(2^k-1)) = 2S^*(2^{k+1}-1)$$

Sumando ambas expresiones obtenemos el resultado buscado, y podemos concluir la demostración. □

Ya tenemos todos las propiedades necesarias para poder probar que el punto de corte óptimo para $S(n)$ es la potencia de 2 más cercana a n, que no es más que la expresión $2^{\lfloor \log_2 n \rfloor}$.

Lema 3.1.5. *Usando la multiplicación de Karatsuba para los productos completos, un punto de corte óptimo para el algoritmo de Mulders es $k = 2^{\lfloor \log_2 n \rfloor}$, y el número de productos calculados por el algoritmo es*

$$S(n) = S\left(\left\lceil\frac{n}{2}\right\rceil\right) + 2S\left(\left\lfloor\frac{n}{2}\right\rfloor\right).$$

Demostración. Sea $S(n)$ el numero de operaciones que realiza el algoritmo de Mulders como en el **Lema 3.1.3**. Por definición de este, se cumple la desigualdad $S^*(n) \geq S(n)$. Por otro lado, usando de nuevo el **Lema 3.1.3** y que k es un punto óptimo por el **Teorema 3.1.4**, se cumple la siguiente cadena de desigualdades

$$S(n) \geq S\left(\left\lceil \frac{n}{2} \right\rceil\right) + 2S\left(\left\lfloor \frac{n}{2} \right\rfloor\right) \geq S^*\left(\left\lceil \frac{n}{2} \right\rceil\right) + 2S^*\left(\left\lfloor \frac{n}{2} \right\rfloor\right) = S^*(n).$$

De esta forma, podemos concluir que $S(n) = S^*(n)$ y que se cumplen las hipótesis del enunciado. □

Aunque hayamos podido encontrar el punto de corte óptimo para el algoritmo de Mulders, del **Teorema 3.1.4** podemos extraer una conclusión de gran relevancia: si los polinomios que queremos multiplicar son de tamaño una potencia de 2 cualquiera, esto es, 2^k , entonces los puntos de corte $s = 2^k$ y $s = 2^{k-1}$ dan el mismo resultado para $S(n)$: en el primer caso, lo que el algoritmo de Mulders hace es un producto completo con Karatsuba, que truncara a la precisión requerida al finalizar, y en el segundo caso, el punto de corte es exactamente la mitad de la longitud del polinomio, como con el algoritmo de Karatsuba, con el que se realizará, de nuevo, un producto completo de Karatsuba de tamaño 2^{k-1}, y los términos restantes se calcularán con llamadas recursivas de Mulders, que en este caso concreto viene a ser lo mismo que Karatsuba. La conclusión que sacamos de este análisis es que, cuando el tamaño de las series truncadas es una potencia de 2, el algoritmo de Mulders hace exactamente el mismo número de operaciones que si se calculase el producto completo con Karatsuba, y después se truncase a la precisión deseada. De hecho, es que las técnicas de ambos algoritmos son análogas.

Capítulo 4

Variante del algoritmo de Mulders

En este capítulo veremos una variante del algoritmo de Mulders que presentan los autores del artículo [1]. La idea detrás de este algoritmo es separar los polinomios en dos partes, que pueden o no ser del mismo tamaño, una de ellas con los términos de grado par y la otra con los términos de grado impar. Veremos las bases teóricas que soportan esta idea y analizaremos su coste computacional, en cuanto a número de productos se refiere, para poder finalmente compararlo con Mulders y concluir si uno de ellos presenta alguna ventaja frente al otro.

Para empezar, vamos a ver que tenemos el módulo $K_n[x]$ de las series de Taylor truncadas de tamaño n, que pueden ser escritas como $K_{\lfloor \frac{n}{2} \rfloor}[x] \bigoplus x^{\lfloor \frac{n}{2} \rfloor} K_{\lceil \frac{n}{2} \rceil}[x]$. Nótese la relación que guarda esto con nuestro trabajo, pues nosotros también estamos trabajando con series de potencias truncadas. Con esta separación, si tomamos un polinomio cualquiera $P \in K_n[x]$, podemos asociar otro polinomio $Q \in K[x,t]$ con $Q = Q_1 + Q_2 t$.

Aunque si le damos una vuelta más a esta descomposición de $K_n[x]$ a fin de conseguir una mayor relación con nuestros intereses, vemos que podemos descomponer $K_n[x]$ como $K_n[x] = O_{\lceil n/2 \rceil}[x] \oplus xO_{\lfloor n/2 \rfloor}[x]$, donde $O_k[x] = \{P(x^2), P \in K_k[x]\}$. Dicho con palabras, dado un polinomio $P \in K_n[x]$ cualquiera, gracias a esta descomposición del espacio $K_n[x]$ es posible separar P como la suma de dos polinomios construidos como los términos de grado par de P y los términos de grado impar, respectivamente. Añadiendo lo visto en el párrafo anterior, sabemos que simultáneamente podemos asociar P a otro polinomio $P' = P'_1 + P'_2 t$. Supongamos ahora que tenemos dos series de potencias truncadas Q y Q' de orden n y nos interesa calcular el producto truncado definido en la **Definición 2.0.3**. Si aplicamos el producto truncado a ambas series, pero para los pedazos en los que están separadas, obtenemos la expresión

$$QQ' = Q_1 Q'_1 + t(Q_1 Q'_2 + Q_2 Q'_1) + t^2 Q_2 Q'_2 + \mathcal{O}(x^n), \tag{4.1}$$

de donde solo necesitamos los primeros n términos para obtener el producto truncado que estamos buscando, por tanto, habrá que tener en mente que puede que sea necesario tener que realizar algún truncamiento adicional.

Cabe destacar la similitud que guarda la expresión del producto QQ' con aquella resultante al aplicar el algoritmo de Karatsuba estudiado en la **Sección 2.2** a dos series truncadas cualesquiera, siendo así que modificaremos el término $(Q_1 Q'_2 + Q_2 Q'_1)$ de la misma forma que se hace

a z_1 en Karatsuba. Por tanto, realizando los cambios pertinentes, tenemos que

$$(Q_1Q_2' + Q_2Q_1') = (Q_1 + Q_2)(Q_1' + Q_2') - Q_1Q_1' - Q_2Q_2'.$$

Si analizamos cada una de las partes del producto truncado QQ' en (4.1), podemos saber exactamente el número de términos que se necesitan de cada una, siendo en cada caso una cantidad distinta:

- De la primera de las expresiones, Q_1Q_1', se necesitan obtener sus primeros n términos, que se pueden calcular simplemente realizando un producto truncado de Q_1 por Q_1' de tamaño $\lceil \frac{n}{2} \rceil$.
- La expresión $(Q_1 + Q_2)(Q_1' + Q_2') - Q_1Q_1' - Q_2Q_2'$ está multiplicada por un factor t, por tanto, solo nos interesarán obtener los $n-1$ primeros términos del resultado final. Sobre los cálculos a realizar en sí, los primeros $n-1$ términos tanto de Q_1Q_1' como los de Q_2Q_2' serán ya conocidos, por tanto solo será necesario calcular el producto truncado de $(Q_1 + Q_2)(Q_1' + Q_2')$ de orden $\lceil \frac{n-1}{2} \rceil$.
- El último producto a calcular es Q_2Q_2', el cual está multiplicado por un factor t^2, por tanto, solo serán necesarios los primeros $n-2$ términos del resultado. Pero necesitamos los primeros $n-1$ términos para el cálculo de la expresión del punto anterior, luego el procedimiento a seguir será calcular el producto truncado de Q_2Q_2' de orden $\lceil \frac{n-1}{2} \rceil$, de donde se pueden tomar directamente los términos que necesitamos.

A partir de estas consideraciones, se puede extraer el algoritmo buscado:

VarianteProductoTruncado(f, g, n):

Entrada:
1. $f = \sum_{0 \leq i \leq n} f_i x^i, g = \sum_{0 \leq i \leq n} g_i x^i$, dos polinomios cualesquiera.
2. n, un entero positivo, la precisión que se tiene de f y g.
Salida:
El producto truncado $fg + \mathcal{O}(x^n)$.

if $n = 1$ **then**
  return $fg + \mathcal{O}(x)$
end if

$n_0 = \lfloor \frac{n}{2} \rfloor$, $n_1 = \lceil \frac{n}{2} \rceil$
descomponer f y g en $f_{par}(x^2) + xf_{impar}(x^2)$ y $g_{par}(x^2) + xg_{impar}(x^2)$
l = VarianteProductoTruncado($f(x)_{par}$, $g(x)_{par}$, n_1)
h = VarianteProductoTruncado($f(x)_{impar}$, $g(x)_{impar}$, n_0)
m = VarianteProductoTruncado($f(x)_{par} + f(x)_{impar}$, $g(x)_{par} + g(x)_{impar}$, n_0) $- l - h$
return $(l(x^2) + xm(x^2) + x^2h(x^2) + \mathcal{O}(x^n))$

Obsérvese que, siguiendo el pseudocódigo para dos series f y g, que se separan en $f_{par}(x^2) + xf_{impar}(x^2)$ y $g_{par}(x^2) + xg_{impar}(x^2)$, consideramos, por ejemplo, la llamada recursiva l. Aunque formalmente a esta llamada se estén pasando las series $f_{par}(x^2)$ y $g_{par}(x^2)$, por dentro se trabaja con $f(x)_{par}$ y $g(x)_{par}$.

Destaca también el caso en que la precisión de las series a multiplicar sea un número impar, ya que los polinomios con los términos pares $f_{par}(x^2)$ y $g_{par}(x^2)$ tendrán un término más que sus contrapartes con los términos impares. De la misma forma, el resultado que retorna el algoritmo VarianteProductoTruncado debe ser truncado a orden n, ya que la expresión $l(x^2) + xm(x^2) + x^2h(x^2)$ por si sola retorna un polinomio cuyos términos se corresponden a una precisión mayor a la que buscamos.

Veamos ahora que el algoritmo presentado es correcto, así como el número de multiplicaciones que realiza.

Teorema 4.0.1. *El algoritmo VarianteProductoTruncado es correcto, y realiza el mismo número de multiplicaciones que el algoritmo de Mulders con el producto de Karatsuba usando como punto de corte* $s(n) = 2^{\lfloor log_2 n \rfloor}$.

Demostración. Comprobemos la correctitud y optimalidad del algoritmo ShortProduct por inducción en n. El caso base $n = 1$ se cumple sin necesidad de entrar en profundidad.

Supongamos ahora que el enunciado del teorema se cumple para $n-1$ dado $n \geq 2$. Uno de los primeros pasos del algoritmo es separar f y g en suma de términos pares e impares, es decir, $f = f_{par}(x^2) + xf_{impar}(x^2)$ y $g = g_{par}(x^2) + xg_{impar}(x^2)$; por otro lado, por definición, $n_0, n_1 < n$, siendo n_1 y n_0 los grados de los términos pares e impares respectivamente, luego por inducción y definición del algoritmo se cumple que

$$\begin{aligned}
&l(x^2) = f_{par}(x^2)g_{par}(x^2) \mod x^{2n_1},\\
&x^2h(x^2) = f_{\text{impar}}(x^2)g_{\text{impar}}(x^2) \mod x^{2n_0+2}, \text{ y}\\
&xm(x^2) = x(f_{\text{par}}(x^2)g_{\text{impar}}(x^2) + f_{\text{impar}}(x^2)g_{\text{par}}(x^2)) \mod x^{2n_0+1},
\end{aligned}$$

cumpliéndose así la igualdad

$$\begin{gathered}
l(x^2) + xm(x^2) + x^2h(x^2) =\\
= f_{par}(x^2)g_{par}(x^2) + x(f_{\text{par}}(x^2)g_{\text{impar}}(x^2) + f_{\text{impar}}(x^2)g_{\text{par}}(x^2)) + x^2 f_{\text{impar}}(x^2)g_{\text{impar}}(x^2) =\\
= fg \bmod x^n,
\end{gathered}$$

ya que $\min(2n_1, 2n_0+1) = n$ para cualquier n dado. De esta forma, hemos visto que el algoritmo ShortProduct es correcto, y devuelve el producto truncado de dos polinomios f y g.

En cuanto al número de multiplicaciones, no es necesario más que mirar el diseño del algoritmo dado en su pseudocódigo y la igualdad del número de multiplicaciones del algoritmo de Mulders dada en el **Lema 3.1.5**, y comparar ambas: el número de multiplicaciones de Mulders con punto de corte óptimo sigue la relación $S(n) = S\left(\left\lceil \frac{n}{2} \right\rceil\right) + 2S\left(\left\lfloor \frac{n}{2} \right\rfloor\right)$, que resulta ser la misma que sigue nuestro algoritmo VarianteProductoTruncado, ya que calcula de forma recursiva un

producto truncado de tamaño n_1, donde $n_1 = \lceil \frac{n}{2} \rceil$ y dos productos truncados de tamaño n_0, con $n_0 = \lfloor \frac{n}{2} \rfloor$, lo que corresponde término a término con cada una de las expresiones que aparecen en la igualdad de $S(n)$. □

Gracias a este resultado, comprobamos que en lo teórico no hay ninguna diferencia en el número de operaciones calculadas entre el algoritmo de Mulders y la variante aquí dada, por tanto no se puede asegurar que una sea mejor que la otra en un caso u otro.

Ejemplo 4.0.2. *Veamos un ejemplo sencillo de cómo funciona el algoritmo VarianteProductoTruncado. Consideramos el espacio $\mathbb{Z}/5\mathbb{Z}$ y las series truncadas $A(x) = 1+3x^2+x^3+2x^4\mathcal{O}(x^5)$ y $B(x) = 2+x+3x^3+4x^4+\mathcal{O}(x^5)$. Como las series están truncadas a precisión $\mathcal{O}(x^5)$, $n = 5$, por tanto la primera llamada que se hace al algoritmo es VarianteProductoTruncado$(A, B, 5)$. Como $n \neq 1$, no estamos en el caso directo, luego operamos para calcular las recursiones.*

1. *Empezamos calculando $n_0 = \lfloor \frac{n}{2} \rfloor = 2$, $n_1 = \lceil \frac{n}{2} \rceil = 3$. Simultáneamente, descomponemos A y B como suma de dos polinomios, tal y como se explica en la definición del algoritmo, conformados por los términos pares y los términos impares de cada polinomio respectivamente, esto es,*

$$A = 1 + 3x^2 + 2x^4 + x \cdot x^2 + \mathcal{O}(x^5) = A_P + xA_I$$
$$B = 2 + 4x^4 + x(1 + 3x^2) + \mathcal{O}(x^5) = B_P + xB_I.$$

Con las series ya separadas, calculamos los productos de las partes recursivamente para obtener l, m y h.

a) *Recordamos que aunque formalmente estemos trabajando con $A(x^2)_P$ y $B(x^2)_P$, en la práctica a la llamada recursiva se pasan como parámetros $A(x)_P$ y $B(x)_P$. Así, llamamos recursivamente a VarianteProductoTruncado$(A(x)_P, B(x)_P, n = 3)$. De nuevo, calculamos el resto de operaciones, pues no nos encontramos en el caso base. Tenemos $n_0 = 1, n_1 = 2$, y*

$$A_P = 1 + 3x + 2x^2 + \mathcal{O}(x^3) = 1 + 2x^2 + x \cdot (3) + \mathcal{O}(x^3) = A_{PP} + xA_{PI}$$
$$B_P = 2 + 4x^2 + \mathcal{O}(x^3) = B_{PP} + xB_{PI}$$

Calculamos recursivamente $l_1 =$ VarianteProductoTruncado$(A_{PP}, B_{PP}, n_1 = 2)$.

- *Como el paso recursivo es el mismo, no entraremos en detalle para no alargar demasiado la explicación. Tenemos que*

$$A_{PP} = 1 + x \cdot (2) = A_{PPP} + xA_{PPI},\ B_{PP} = 2 + x \cdot (4) = B_{PPP} + xB_{PPI},$$

y $n_0 = 1 = n_1$. Calculamos cada una de las llamadas recursivas

$$l' = \text{VarianteProductoTruncado}(A_{PPP}, B_{PPP}, 1) = 2$$
$$h' = \text{VarianteProductoTruncado}(A_{PPI}, B_{PPI}, 1) = 3$$
$$m' = \text{VarianteProductoTruncado}(A_{PPP} + A_{PPI}, B_{PPP} + B_{PPI}, 1) - l' - h' = 3$$

Luego siguiendo el esquema del algoritmo, tenemos que $l_1 = 2 + 3x + 3x^2 \cong 2 + 3x + \mathcal{O}(x^2)$.

b) *Calculamos ahora la llamada* $h_2 = VarianteProductoTruncado(A_{PI}, B_{PI}, 1) = 3 \cdot 0 = 0$, *y, por último la llamada con las sumas de las partes de* A_P *y* B_P, *esto es,* $m_2 = VarianteProductoTruncado(A_{PP} + A_{PI}, B_{PP} + B_{PI}, 1) - l_1 \cong 3 - (2 + x) \cong 1$.

c) *Finalmente, sumamos el resultado de cada una de las llamadas, de forma que*

$$l = l_1(x^2) + xm_2(x^2) + x^2h_2(x^2) \cong 2 + x + 3x^2 + \mathcal{O}(x^3).$$

2. *Calculamos ahora* h *con la llamada recursiva* $VarianteProductoTruncado(A_I, B_I, n = 2)$, *donde vamos a tener que entrar de nuevo en una recursión, separando en este caso nuestros parámetros iniciales como*

$$A_I = x = A_{IP} + xA_{II}, B_I = 1 + 3x = B_{IP} + xB_{II},$$

y $n_0 = n_1 = 1$. *Así, calculamos cada una de las recursiones como*

$$\begin{aligned} l_3 &= VarianteProductoTruncado(A_{IP}, B_{IP}, 1) = 0 \cdot 1 = 1, \\ h_3 &= VarianteProductoTruncado(A_{II}, B_{II}, 1) = 1 \cdot 3 = 3, \\ m_3 &= VarianteProductoTruncado(A_{IP} + A_{II}, B_{IP} + B_{II}, 1) = 4 - 0 - 3 = 1. \end{aligned}$$

Sumando cada valor de calculado con estas llamadas, tenemos que

$$h = 0 + x + 3x^2 = x + \mathcal{O}(x^2).$$

3. *Solo falta por calcular* m. *Sumamos la parte par de* A, A_P, *con la impar,* A_I, *y hacemos lo respectivo con* B, *esto es,*

$$\begin{aligned} A_P(x) + A_I(x) &= 1 + 4x + 2x^2 = 1 + 2x^2 + 4x = A'_P + xA'_I \\ B_P(x) + B_I(x) &= 3 + 3x + 4x^2 = 3 + 4x^2 + 3x = B'_P + xB'_I. \end{aligned}$$

Calculamos cada llamada recursiva necesaria:

$$\begin{aligned} l_4 &= VarianteProductoTruncado(A'_P, B'_P, 1) = 3, \\ h_4 &= VarianteProductoTruncado(A'_I, B'_I, 1) = 2, \\ m_4 &= VarianteProductoTruncado(A'_P + A_I, B'_P + B'_I, 1) = 0 - 3 - 2 = 1. \end{aligned}$$

Luego $VarianteProductoTruncado(A_P + A_I, B_P + B_I, 2) = 3$, *por tanto,*

$$m = 3 - l - h = 3 - (2 + x + 3x^2) - x \cong 1 + 3x + 2x^2$$

4. *Finalmente, el resultado de este producto truncado viene dado por la expresión* $l(x^2) + xm(x^2) + x^2h(x^2)$, *luego sustituyendo en cada uno de los términos que hemos calculado a lo largo del ejercicio, concluimos que*

$$AB \cong 2 + x^2 + 3x^4 + x(1 + 3x^2 + 2x^4) + x^2 \cdot x^2 \cong 2 + x + x^2 + 3x^3 + 4x^4 + \mathcal{O}(x^5)$$

4.1. Mulders y su variante con algoritmos cuadráticos

Si en la anterior sección la conclusión extraída tras el análisis realizado sobre la variante es que esta y el algoritmo de Mulders realizan el mismo número de multiplicaciones, aquí vamos a ver una forma de optimizar todavía más ambos algoritmos y comprobar cómo, de esta forma, la variante de Mulders calcula menos productos que Mulders.

Pues en los casos que a primera vista podríamos considerar más sencillos, en los que el grado de los polinomios a multiplicar es pequeño, la eficiencia de ambos algoritmos se ve más lastrada. Para solucionar esto, solo será necesario realizar un pequeño cambio: usar algoritmos cuadráticos.

Aunque en términos asintóticos el algoritmo de Mulders sea más eficiente que el algoritmo tradicional, este resulta ser más eficiente que el de Karatsuba, excepto, por ejemplo, cuando se esté trabajando en un anillo en el que realizar una multiplicación sea una operación muy costosa, en cuyo caso sí que nos podría interesar usar únicamente Karatsuba a sabiendas de que se van a calcular menos multiplicaciones.

Para mejorar el rendimiento, lo habitual es combinar las dos técnicas de multiplicación, utilizando una u otra según el tamaño de los polinomios. El siguiente paso consiste en incorporar esta estrategia tanto al algoritmo de Mulders como a su variante, y analizar las mejoras obtenidas. Suponiendo que se usa la multiplicación clásica para $n \leq n_0$, existen dos formas de proceder:

- La multiplicación se realiza en bloques, esto es, los números se dividen en trozos de tamaño n_0, y los productos que se tengan que realizar a este nivel se calculan con algoritmos cuadráticos como el tradicional, por ejemplo. Este método es similar a aquel explicado en el apartado de Karatsuba, cuando se quieren multiplicar dos polinomios de distinto tamaño, solo que en vez de operar en todo momento con Karatsuba, aquí operaríamos con algoritmos cuadráticos.

- Se utiliza el algoritmo de Karatsuba para números de tamaño mayor que n_0, y en los casos en los que el tamaño sea menor o igual se emplean algoritmos cuadráticos. En lo que sigue en esta sección, este será el método que seguiremos y con el que se han realizado todas las pruebas.

Estos cambios tienen que tenerse en cuenta en nuestro análisis del punto de corte óptimo del algoritmo de Mulders. En el primer caso, la estrategia no se ve alterada salvo por pequeñas puntualizaciones: $S(1) = \frac{1}{2}$, y en el caso de que los polinomios sean de tamaño una potencia de 2 cualquiera, 2^m, el punto de corte a tomar será 2^{m-1}.

En el segundo caso, se debe cambiar la recursión inicial del algoritmo tomando $K(n) = n^2$, $S(n) = n(n+1)/2$ para $n \leq n_0$. Dicho con palabras, lo que esto significa es que para series truncadas de tamaño menor que el número n_0 elegido, se utiliza un algoritmo cuadrático para efectuar el producto de las series, donde el valor de $K(n)$ implica que el algoritmo usado será el tradicional, y el número de operaciones $S(n)$ se sigue de la siguiente regla: precisamente, por el diseño del algoritmo, dadas dos series de potencias truncadas A y B con precisión hasta x^n, el coeficiente del término x^i, con $i < n$, se calcula realizando $i + 1$ productos. Esto significa que

el término independiente se calcula con un único producto, el coeficiente del término x requiere de dos productos, hasta llegar al coeficiente del término x^{n-1}, que requiere de n productos. Sumando todos las multiplicaciones efectuadas obtenemos el número de operaciones ejecutadas por el algoritmo,

$$S(n) = 1 + 2 + 3 + ... + n = n(n+1)/2.$$

Con estos cambios los autores de [1] no han conseguido un método general para obtener el punto de corte óptimo, pero sí que se pueden sacar conclusiones basadas en el número de multiplicaciones calculadas por un algoritmo u otro.

En lo que sigue de sección, dado n el tamaño de los polinomios que se quieren multiplicar, para $n \leq n_0$ se utilizará el algoritmo de la escuela para calcular el producto de estos polinomios. También será necesaria la siguiente definición:

Definición 4.1.1. *Dada la variante del algoritmo de Mulders mostrada al principio de esta sección, llamaremos $\tilde{S}(n)$ al número de multiplicaciones que calcula y $S_{opt}(n)$ al número de multiplicaciones que calcula Mulders, teniendo en cuenta para ambas que en los casos $n \leq n_0$ se utiliza el algoritmo de la escuela.*

Asimismo, teniendo en mente los enunciados vistos en el **Lema 3.1.5** y en el **Teorema 3.1.4**, vamos a ver la mejora de uno de los algoritmos respecto al otro, demostrando que, con las condiciones con las que estamos trabajando, nuestra variante calcula menos multiplicaciones que el algoritmo de Mulders en un intervalo dado, y que el algoritmo de Mulders sigue la misma relación que la vista en el **Lema 3.1.3**.

Teorema 4.1.2. *Si usamos un algoritmo cuadrático en el caso de que $n \leq n_0$, se cumple que $\tilde{S}(n) < S_{opt}(n)$ $\forall n > n_0$ y $n_0 \geq 4$.*

Lema 4.1.3. *$S_{opt}(n) > \tilde{S}(n)$ para cualquier $n_0 < n \leq 2n_0$*

Demostración. En el caso trivial de que $2 \leq n \leq n_0$, $K(n) = n^2$ y $S(n) = \tilde{S}(n) = n(n+1)/2$, porque en ambos casos por diseño se utiliza el algoritmo tradicional de la escuela.

Supongamos ahora que $n_0 \leq n \leq 2n_0$, entonces el coste del algoritmo, $K(n)$, depende del algoritmo usado para efectuar el producto de las series, luego si $M(n)$ es el coste del algoritmo utilizado como en la **Definición 3.0.2**, la expresión que resulta es:

$$K(n) = 2M\left(\left\lceil \frac{n}{2} \right\rceil\right) + M\left(\left\lfloor \frac{n}{2} \right\rfloor\right).$$

Volviendo a las hipótesis supuestas al comienzo del párrafo, tenemos que $K(n) = 2\lceil \frac{n}{2} \rceil^2 + \lfloor \frac{n}{2} \rfloor^2 \geq \frac{3}{4}n^2$, donde la sustitución de los valores $M(n) = n^2$ se justifica porque $n/2 \leq n_0$, y la desigualdad se justifica sustituyendo los posibles valores que puede tomar n, en función de que sea un número par o un número impar:

- Si n es un número par, tenemos que

$$2\left\lceil \frac{n}{2} \right\rceil^2 = 2\left(\frac{n}{2}\right)^2 = \frac{1}{2}n^2,$$

$$\left\lfloor \frac{n}{2} \right\rfloor^2 = \left(\frac{n}{2}\right)^2 = n^2/4.$$

Sumando ambas expresiones, se cumple la igualdad.

- En el caso de que n sea un número impar, el procedimiento a seguir es análogo al caso par.

$$2\left\lceil \frac{n}{2} \right\rceil^2 = 2\left(\frac{n+1}{2}\right)^2 = \frac{1}{2}(n^2+2n+1),$$

$$\left\lfloor \frac{n}{2} \right\rfloor^2 = \left(\frac{n-1}{2}\right)^2 = \frac{1}{4}(n^2-2n+1)$$

Sumando y operando con ambas expresiones, se cumple la desigualdad.

Para conocer la expresión de $\tilde{S}(n)$, debemos remontarnos a la construcción del algoritmo dada en su pseudocódigo. Por definición, las variables n_0 y n_1 están una unidad por encima o por debajo de $\frac{n}{2}$, por hipótesis $n \leq 2n_0 \rightarrow n/2 \leq n_0$ y las llamadas recursivas que realiza la variante se hacen con n_0 o con n_1, luego el producto de las series truncadas que se quieran multiplicar se hace con el producto de la escuela, y por tanto su coste viene dado por la expresión $n\frac{n+1}{2}$.

Con estas consideraciones, se puede calcular el coste computacional de la variante para los n con los que se están trabajando, de forma que la expresión resultante es:

$$\tilde{S}(n) = 3k\frac{(k+1)}{2} = \frac{3}{2}k^2 + \frac{3}{2}k, \text{ si } n = 2k,$$
$$\tilde{S}(n) = 2k\frac{(k+1)}{2} + (k+1)\frac{(k+2)}{2} = \frac{3}{2}k^2 + \frac{5}{2}k + 1, \text{ si } n = 2k+1.$$

Por otro lado, por definición de $S_{opt}(n)$, tenemos que

$$S_{opt}(n) = \min_{\frac{n}{2} \leq l \leq n} K(l) + 2S_{opt}(n-l) \geq \min_{\frac{n}{2} \leq l \leq n} \frac{3}{4}l^2 + (n-l)(n-l+1) =$$

$$\min_{\frac{n}{2} \leq l \leq n} \frac{7}{4}l^2 - l(2n+1) + n^2 + n.$$

Calcular el mínimo de esta expresión no entraña mayor dificultad que calcular sus extremos relativos, y verificar que se encuentran dentro del intervalo de puntos que estamos considerando. Para calcular los extremos relativos de la expresión, empezamos calculando su primera derivada, $S'_{opt}(n) = \frac{7}{2}l - 2n + 1$, e igualamos dicha expresión a 0,

$$\frac{7}{2}l - 2n + 1 = 0 \rightarrow \frac{7}{2}l = 2n - 1 \rightarrow l = \frac{2}{7}(2n-1).$$

Con unos pocos cálculos adicionales observamos que el punto l es efectivamente el mínimo de la expresión $S_{opt}(n)$, por tanto solo resta verificar que este punto se encuentra en nuestro intervalo de puntos admisibles.

$$\frac{2}{7}(2n-1) = \frac{4}{7}n - \frac{2}{7} \geq \frac{n}{2} \text{ ya que } (\frac{4}{7} - \frac{1}{2})n + \frac{2}{7} = \frac{1}{14}n + \frac{2}{7} \geq 0 \text{ porque } n > 0,$$
$$\frac{4}{7}n - \frac{2}{7} \leq n \text{ debido a que } \frac{2}{7} \leq \frac{3}{7}n, \text{ de nuevo porque } n > 0.$$

luego se cumple que el mínimo de $S_{opt}(n)$ calculado pertenece a $[n/2, n]$. Si ahora sustituimos en $S_{opt}(n)$ por l, tenemos que

$$S_{opt}(n) = \min_{\frac{n}{2} \leq l \leq n} \frac{7}{4}l^2 - l(2n+1) + n^2 + n = \frac{3}{7}n^2 + \frac{3}{7}n - \frac{1}{7}.$$

Si seguimos operando, sustituimos ahora por cada uno de los posibles valores de n:

- Para n par, $n = 2k$:

$$S_{opt}(n) = \frac{3}{7}n^2 + \frac{3}{7}n - \frac{1}{7} = \frac{12}{7}k^2 + \frac{6}{7}k - \frac{1}{7}.$$

- Para n impar, $n = 2k + 1$:

$$S_{opt}(n) = \frac{12}{7}k^2 + \frac{18}{7}k + \frac{5}{7}.$$

Solo queda comparar $S_{opt}(n)$ con $\tilde{S}(n)$ en cada caso, donde se observa que en el caso par, $S_{opt}(n) > \tilde{S}(n)$ para $k = 4$ (con n = 8) , y en el caso impar, la desigualdad se cumple para $k = 2$ (con $n = 5$). Queda aislado el caso $n = 6$ para $n_0 = 4$ o $n_0 = 5$ porque no se puede comparar directamente, así que simplemente se calcula el resultado de cada coste para dicho valor, donde se obtiene $S_{opt}(n) = 21$ y $\tilde{S}(n) = 18$.

□

Lema 4.1.4. $S_{opt}(n) \geq S_{opt}(\lceil \frac{n}{2} \rceil) + 2S_{opt}(\lfloor \frac{n}{2} \rfloor)$

Demostración. Veamos primero por inducción que $S_{opt}(n-1) \leq S_{opt}(n) < K(n)$ para $n \geq 2$.

Si $n \leq n_0$, tenemos que $K(n) - S_{opt}(n) = n^2 - n(n+1)/2 = n(n-1)/2 \ > 0$, y $S(n) = n(n+1)/2 \geq n(n-1)/2 = S(n-1)$.

Supongamos ahora que $n > n_0$, entonces $K(n) = 2K(\lceil \frac{n}{2} \rceil) + K(\lfloor \frac{n}{2} \rfloor) > K(\lceil \frac{n}{2} \rceil) + S_{opt}(\lceil \frac{n}{2} \rceil) + S_{opt}(\lfloor \frac{n}{2} \rfloor) \geq K(\lceil \frac{n}{2} \rceil) + 2S_{opt}(\lceil \frac{n}{2} \rceil) \geq S_{opt}(n)$, donde, desde la primera hasta la última, la cadena de igualdades se cumple por las siguientes razones:

1. La propiedad vista justo al principio de la demostración.

2. Propiedad vista en el **Lema 3.1.2**.

3. Por definición de $S_{opt}(n)$,

$$S_{opt}(n) = \min_{\frac{n}{2} \leq l \leq n} K(l) + S_{opt}(n-l),$$

que al buscar el mínimo es más pequeño que los términos con los que se compara.

Para continuar con la demostración y ver que $S_{opt}(n-1) \leq S_{opt}(n)$, distinguimos el caso $n = n_0 + 1$. En este caso, $S_{opt}(n-1) = n_0(n_0+1)/2$ y $S_{opt}(n) = \min_{\frac{n}{2} \leq l \leq n} K(l) + S_{opt}(n-l)$ que, por estar con valores menores o iguales que n_0, tenemos que $S_{opt}(n) = \min_{\frac{n}{2} \leq l \leq n} l^2 + (n-l)(n-l+1)$. El valor mínimo para $S_{opt}(n)$ se consigue para $l = \lceil \frac{n}{2} \rceil$, cuyo dicho valor es $n(n+1)/2$, luego $S_{opt}(n) = n(n+1)/2 \geq n(n-1)/2 = S_{opt}(n-1)$.

Para $n > n_0 + 1$, no puede ocurrir el caso de que el punto de corte elegido sea justamente n, es decir, $s(n) = n$, porque esto entraría en contradicción con $S_{opt}(n) < K(n)$ como hemos visto al principio de la demostración, por tanto, podemos aplicar el mismo razonamiento que para la segunda parte del **Lema 3.1.2**, llegando así a que $S_{opt}(n+1) \geq S_{opt}(n)$.

Finalmente vemos que se puede aplicar el **Lema 3.1.3** para $S := S_{opt}(n)$, pues su demostración tan solo utiliza que $S(n)$ es no decreciente, la definicion de $K(n)$ y el hecho de que $K(n) \geq S_{opt}(n)$. □

Con la ayuda de estos lemas, ya podemos finalmente demostrar el **Teorema 4.1.2**.

Demostración. Sea $T(n) = S_{opt}(n) - \tilde{S}(n)$. Tenemos que $T(n) = 0$ para $n \leq n_0$, $T(n) > 0$ para $n_0 < n \leq 2n_0$ por **Lema 4.1.3** y $T(n) \geq T(\lceil \frac{n}{2} \rceil) + 2T(\lfloor \frac{n}{2} \rfloor)$ para $n > 2n_0$ por el **Lema 4.1.4** junto a la definición de $\tilde{S}(n)$, y como $\lceil \frac{n}{2} \rceil > n_0$, por inducción se sigue automáticamente que $T(n) > 0$ para $n > n_0$. □

La conclusión a la que uno llega tras todos estos resultados es que, efectivamente, al combinar el producto de la escuela con el algoritmo de Mulders y su variante, este último calcula un menor número de multiplicaciones que el anterior, y dicho esto, la mejora se produce en los casos del **Lema 4.1.3** cuando $n_0 \leq n \leq 2n_0$ como se puede ver en el **Teorema 4.1.2**.

Capítulo 5

Cálculo del cociente y resto

En esta capítulo, vamos a estudiar otra aplicación que le podemos dar a los algoritmos estudiados en las secciones anteriores, el algoritmo de Mulders y su variante.

Para ello, hablaremos del algoritmo tradicional, o de la escuela, de división de dos polinomios con resto para poder compararlo con un algoritmo presentado en [7]. Nos centraremos en analizar dicho algoritmo y su coste computacional, pues una propiedad que cumple es que no es necesariamente cuadrático, sino que depende del algoritmo empleado para multiplicar polinomios dentro de la propia división. Aquí es donde cobran relevancia Mulders y su variante.

También hablaremos del Método de Newton, que aunque a primera vista pueda parecer que no tiene relación alguna con nuestro problema, será necesario su uso en el momento en el que necesitemos calcular la serie de potencias del polinomio inverso de otro dado módulo $x^l, l \in \mathbb{N}$.

Finalmente, veremos los resultados obtenidos al operar con un algoritmo u otro, y los compararemos para entonces poder ver cuál de ellos es más eficiente.

5.1. Método de Newton

Presentamos esta sección sin dar mayores detalles del método, pues no es la idea que queremos estudiar en profundidad, si bien es una herramienta de gran utilidad para este fin. Para ello, nos basamos en los apuntes de la asignatura Cálculo Numérico I, que se pueden ver con detalle en [5].

El método de Newton se trata de un algoritmo iterativo originalmente ideado para el cálculo de los ceros de un polinomio. Esto es, dado un polinomio f cualquiera, si tenemos x_n una aproximación suficientemente buena de la raíz que queremos aproximar, entonces el punto

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

es una mejor aproximación de la raíz. Habitualmente, se suele dar una representación geométrica de este resultado, pues uno puede ver que x^{n+1} es el punto en el cual la recta tangente a $f(x)$ en el punto x_n corta al eje x, y dicho punto se acerca más a la raíz buscada cuantas más iteraciones se realicen.

Así como se habla de calcular los ceros de una función, este problema es análogo al cálculo de la inversa de dicha función, pues en realidad calcular la inversa de una función dada f no es más que calcular los valores de nuestro espacio de salida, llamémosle X, que tienen imagen por f, es decir, pertenecen al dominio de la función. Esto lo podemos ver como, dado un punto y cualquiera del dominio de la función, entonces queremos calcular los puntos x en los que se cumple que $f(x) = y$, de donde se puede concluir que se puede utilizar el método de Newton para calcular los ceros de $g = f(x) - y$.

Hay una serie de condiciones que se deben cumplir para poder asegurar la convergencia del método. Si estamos buscando una raíz del polinomio f en el intervalo $[a, b], a, b \in \mathbb{R}$, primero se debe dar que f tenga un cero en el intervalo; luego, la función f que se está estudiando debe ser derivable y no nula en ninguno de los puntos que encierra este intervalo, como es lógico.

5.2. División de la escuela

El algoritmo tradicional de división, o división de la escuela, indistintamente, sigue la idea que define la *División Euclídea*. Si el espacio en el que nos encontramos es un dominio euclídeo, entonces la división euclídea dice que, dados dos polinomios A y B, $B \neq 0$, de grados n y m respectivamente, y que cumplen que $m \leq n$, existen Q y R únicos, con $deg(R) < B$, de forma que

$$A = BQ + R.$$

Cada uno de los polinomios considerados en la igualdad tienen un nombre propio dentro de este problema. Se le llama dividendo a A, el polinomio a dividir, divisor a B, el polinomio que divide a A, cociente a Q y finalmente resto a R.

No siempre se puede asegurar que existen tales polinomios Q y R, dependerá del anillo en el que nos encontremos y los propios polinomios que se pretenda dividir. Por ejemplo, si quisiéramos dividir $A = x^3 + 2x$ entre $B = 5x$ en el anillo $R = \mathbb{Z}[x]$, no sería posible de ninguna forma, ya que no existe el inverso de 5 en $\mathbb{Z}$. Este problema no existe si estamos trabajando en un cuerpo.

Para poder calcular los polinomios Q y R y, por ende, dividir dos polinomios A y B, los pasos a seguir mediante la división euclídea son los siguiente:

1. Se calcula un término del cociente en cada paso, donde cada término será un monomio $q_i \in R[x], i \geq 0$ tal que al multiplicarlo por el divisor B, el término de mayor grado del polinomio obtenido sea igual al término de mayor grado del dividendo, de tal manera que al restarse, quede eliminado, de forma que el "nuevo" dividendo sea un polinomio de grado menor que el original.

2. El paso 1 se repite hasta que se obtenga un dividendo cuyo grado sea menor que el divisor, pues entonces no se podrá seguir dividiendo.

3. Cuando se haya finalizado la división, el último dividendo obtenido resulta ser el resto R, y el cociente Q será la suma de cada uno de los monomios calculados en el paso 1 y 2.

Si $R = 0$, es decir, el resto es el polinomio nulo, entonces se dice que A es divisible por B, que habitualmente se expresa como $B \mid A$, y en caso contrario, si $R \neq 0$, entonces se cumple que $\deg(R) = \deg(A) - \deg(B) = n - m$.

Para calcular la complejidad temporal de la división de la escuela, se deben tener en cuenta tanto el número de multiplicaciones realizadas como el número de sumas. En una iteración del paso 1, se realizan $m+1$ multiplicaciones al multiplicar el término calculado del cociente por B, y m sumas al sumar el resultado obtenido con el resto que se tenga en el momento. Al finalizar, el algoritmo habrá realizado $n - m + 1$ veces dichas operaciones, luego el número total será $(2m + 1)(n - m + 1)$, que es de orden $\mathcal{O}(n^2)$ cuando $m \sim n/2$. De nuevo, estamos ante un algoritmo con complejidad cuadrática, razón por la que nos interesa buscar una alternativa más eficiente.

Teniendo ya una idea clara de como funciona la división, podemos definir su pseudocódigo.

DivisionEscuela(A, B):

Entrada:
1. $A = \sum_{0 \leq i \leq n} a_i x^i, B = \sum_{0 \leq i \leq m} b_i x^i$, los polinomios que se quieren dividir, con $n \geq m$.
Salida:
(Q, R), cociente y resto de la división de A entre B.

if $n < m$ **then**
  return $(0, A)$
end if
$R \leftarrow A$, $u \leftarrow b_m^{-1}$
Para $i = n - m, n - m - 1, ..., 0$
if $k = \deg(R) = m + i$ **then**
  $q_i \leftarrow r_k \cdot u$, $R \leftarrow R - q_i x^i b$
  else $q_i \leftarrow 0$
end if
return $(Q = \sum_{i=0}^{n-m} q_i x^i, R)$

Pese a que sea un algoritmo de complejidad cuadrática, al igual que el producto de la escuela, lo sencillo que resulta el algoritmo hace que sea fácil de manejar e implementar.

Cabe destacar que la *División Euclídea* es la"herramienta" principal detrás de la Aritmética Modular, gracias a la cual podemos interpretar y calcular el anillo cociente $R[[x]]/(x^n)$ en el que estamos trabajando con series de potencias truncadas, y que será igualmente imprescindible para poder usar la división rápida que queremos ver más adelante.

5.3. Método de los coeficientes indeterminados

Dado un polinomio B cualquiera en el anillo $R[x]$, si b_0, el término independiente de B es una unidad de R, entonces B tiene inverso en el anillo de series de potencias $R[[x]]$. En caso

de que se verifique esta condición, el procedimiento más sencillo a seguir para el cálculo de este inverso es el de los coeficientes indeterminados.

El método de los coeficientes indeterminados utiliza el producto de convolución, es decir, la multiplicación de la escuela, tal que, como D es la inversa de B, $BD = 1$, que por definición viene a ser lo mismo que

$$(\sum_{i=0}^{n} b_i \cdot \sum_{i=0}^{n} d_i) = b_0 d_0 + (b_1 d_0 + b_0 d_1)x + ... + (\sum_{i=0}^{n} b_i d_{n-i})x^n = 1 + \mathcal{O}(x^{n+1})$$

Por tanto, conocido B, para calcular cada uno de los términos de D solo es necesario igualar coeficiente a coeficiente con el resultado, y despejar. Con este procedimiento, se obtiene la siguiente recursión:

$$b_0 d_0 = 1 \rightarrow d_0 = \frac{1}{b_0}$$
$$b_1 d_0 + b_0 d_1 = 0 \rightarrow b_1 = -\frac{b_1 d_0}{b_0}$$
$$...$$
$$\sum_{i=0}^{k} b_i d_{k-i} = 0 \rightarrow d_k = -\frac{\sum_{i=1}^{k} b_i d_{k-i}}{b_0}$$

Por tanto, el método de los coeficientes indeterminados sigue el siguiente esquema.

CoeficientesIndeterminados(B)

Entrada:
1. $B = \sum_{i=0}^{n} b_i x^i$, polinomio de grado n.
Salida:
$D = \sum_{i=0}^{n} d_i x^i$ tal que $B \cdot D = 1 + \mathcal{O}(x^{n+1})$.
if $n = 0$ **then**
    return $1/b_0$
end if
Para $k = 1, ..., n$
  $d_k = -\frac{1}{a_0}(\sum_{j=1}^{k} a_j b_{k-j})$
return $D = \sum_{k=0}^{n} d_k x^k$

Como hemos visto antes, el método de los coeficientes indeterminados emplea el algoritmo de la escuela para calcular D, lo que implica que el coste de este método sea también de $\mathcal{O}(n^2)$.

En el algoritmo de división rápida que queremos presentar más adelante será necesario calcular la inversa de un polinomio, pero si buscamos que dicha división tenga un coste subcuadrático, resulta contraproducente utilizar este método para calcular un inverso, pero nos sirve para poder comparar el nuevo algoritmo con otro, tanto en lo teórico como en la parte práctica.

5.4. División con resto usando el método de Newton

Como ya hemos podido ver, la división de la escuela para polinomios resulta computacionalmente muy costosa, lo que nos invita a buscar una alternativa con la que poder obtener el mismo resultado pero con una complejidad temporal menor. Si hablamos de $M(n)$ como la complejidad temporal de un algoritmo como está definido en la **Definición 3.0.2**, en esta sección introduciremos un algoritmo para poder dividir dos series de potencias truncadas con complejidad $\mathcal{O}(M(n))$, por tanto, este algoritmo puede tener un coste computacional no necesariamente cuadrático. Es en este momento donde podremos utilizar los algoritmos presentados en los capítulos previos y así analizar las mejoras que presentan frente a los algoritmos clásicos.

Una de las claves del algoritmo que queremos definir reside en la aritmética modular. Suponiendo que estamos en un espacio euclídeo (en caso contrario no se puede verificar que se cumpla la División Euclídea), dado un polinomio cualquiera A y otro polinomio p, sabemos que podemos calcular el polinomio congruente a A módulo p, que no es más que el resto de la división euclídea entre A y p. Usualmente, si R es el resto de dividir A entre p, se dice que $A \cong R \mod p$, o, equivalentemente, A es congruente con R módulo p, cuyo significado no es otro que $A - R \mid p$.

El siguiente teorema, cuya demostración podemos encontrar en [7], es también importante para el correcto desarrollo de la idea que queremos explicar.

Teorema 5.4.1. *Sea R un dominio euclídeo, $a, m \in R$ y $S = R/mR$. Se cumple que $a \mod m \in S$ es una unidad de S si y solo si el máximo común divisor, que denotamos gcd, de a y m, es 1, es decir,* $\gcd(a, m) = 1$. *Si se verifica esta equivalencia, se puede calcular el inverso de a* mód m.

Ahora, suponemos que nuestro espacio R es un anillo conmutativo y unitario. Tomamos A y B dos series truncadas cualesquiera con $A \in R[[x]]/(x^n)$, $B \in R[[x]]/(x^m)$, el coeficiente de mayor grado de B una unidad de R y $n \geq m$. Como el coeficiente de mayor grado de B es unidad de R, se cumplen las condiciones de la División Euclídea, y por tanto sabemos que existen polinomios Q y R únicos de forma que $A = BQ + R$, con $\deg R < \deg B$. Si, tomando la expresión de la división dada, evaluamos cada serie en $\frac{1}{x}$, y a su vez multiplicamos toda la expresión por x^n, la igualdad que resulta es:

$$x^n A\left(\frac{1}{x}\right) = \left(x^{n-m} Q\left(\frac{1}{x}\right)\right) \cdot \left(x^m B\left(\frac{1}{x}\right)\right) + x^{n-m+1}\left(R\left(\frac{1}{x}\right)\right) \tag{5.1}$$

Si nos fijamos, para cada uno de los polinomios de la igualdad, no se ha hecho más modificación que cambiar los términos de posición, donde, por ejemplo para A, el término a_i se ha intercambiado por aquel en la posición $n - i$. Esta transformación de un polinomio y sus términos tiene nombre.

Definición 5.4.2. *Sea R un anillo y consideramos un polinomio A cualquiera con $A \in R[x]$ de grado n. Llamamos **polinomio recíproco de A**, o simplemente **recíproco** de A, al polinomio que se obtiene al evaluar A en $\frac{1}{x}$, y multiplicarlo por un factor x^k con $k \geq n$. Habitualmente, el recíproco de A se suele expresar como*

$$rev_k(A) = x^k A\left(\frac{1}{x}\right).$$

Si $k = n$, entonces $rev_n(A) = rev(A)$.

Si volvemos a fijarnos ahora la ecuación (5.1), la igualdad se puede leer como

$$rev_n(A) = rev_{n-m}(Q) \cdot rev_m(B) + x^{n-m+1} rev_{m-1}(R),$$

y por lo que hemos podido ver antes sobre aritmética modular, se tiene la relación de congruencia

$$rev_n(A) \cong rev_{n-m}(Q) \cdot rev_m(B) \mod x^{n-m+1}.$$

Como el coeficiente de mayor grado de B es una unidad de nuestro espacio, y estamos considerando un espacio euclídeo, sabemos que existe el inverso de $rev_m(B)$ en $R[x]$ gracias al **Teorema 5.4.1**, y que se puede calcular, por tanto la igualdad anterior resulta finalmente en

$$\mathrm{rev}_n(A) \cdot (rev_m(B))^{-1} \cong \mathrm{rev}_{n-m}(Q) \mod x^{n-m+1}.$$

De esta forma, el cociente de la división Q buscado se puede obtener simplemente calculando el recíproco del recíproco de Q, es decir $Q = \mathrm{rev}_{n-m}(\mathrm{rev}_{n-m}(Q))$, y una vez obtenido Q, el resto de la división R se puede calcular como $R = AB - Q$.

Por tanto, este algoritmo de división de polinomios se vale simplemente del uso del recíproco de un polinomio y las propiedades de la aritmética modular. Obtener el recíproco de un polinomio es una operación sencilla, no es computacionalmente costosa, así que a primera vista no es un factor a tener en cuenta al hablar de la eficiencia del algoritmo. Por otro lado, el producto $rev_n(A) \cdot (rev_m(B))^{-1}$ es un producto de polinomios que se puede calcular tanto con el algoritmo de Mulders como con la variante presentada en el capítulo anterior. Sin embargo, una cuestión a la que uno sí que debería prestar atención es al cálculo de la inversa, en este caso, la inversa de $rev_m(B)^{-1}$.

Para poder realizar el cálculo de la inversa de forma más eficiente respecto al método de los coeficientes indeterminados, será necesario utilizar el método de Newton visto en la sección 5.1, que, como ya se vió, su funcionalidad también permite calcular la inversa de una función (siempre y cuando se estén en las condiciones de convergencia adecuadas). Recordamos que, dada una función f y una aproximación suficientemente buena de la raíz a calcular x_0, la recursión con la que se calculan las siguientes aproximaciones es

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}.$$

En el problema que nos ocupa, dado un polinomio $f \in R[x]$ cuyo término independiente es una unidad de R, queremos calcular otro polinomio g de forma que $fg \cong 1 \mod x^l$, lo que es equivalente a calcular los ceros de la función $\gamma(g) = 1/g - f = 0$. Con estas variables, la iteración del método de Newton queda como

$$g_{i+1} = g_i - \frac{\gamma(g_i)}{\gamma'(g_i)} = g_i - \frac{1/g_i - f}{1 - g_i^2} = 2g_i - fg_i^2,$$

de donde la última igualdad se obtiene simplemente al operar los términos obtenidos en la anterior. Con este método, en principio vamos a ser capaces de calcular sucesivos términos de la inversa de f, pero con el siguiente teorema vamos a ver hasta qué precisión se va a dar esta afirmación.

Teorema 5.4.3. *Sea R un anillo conmutativo y unitario, $f, g_0, g_1, ... \in R[x]$, donde f_0 es una unidad de R, $g_0 = \frac{1}{f_o}$ y cada g_{i+1} es una aproximación de un polinomio g calculada mediante el método de Newton como $g_{i+1} \cong 2g_i - fg_i^2 \mod x^{2^{i+1}}$. Entonces $fg_i \cong 1 \mod x^{2^i}$ para todo $i \geq 0$.*

Demostración. Realizaremos la prueba por inducción en el índice i. En el caso base, para $i = 0$, tenemos por definición que

$$fg_0 \cong f_0 g_0 \cong f_0 \frac{1}{f_0} \cong 1 \mod x^{2^0}.$$

Supongamos ahora que el resultado se cumple para i, veamos ahora que se cumple para $i+1$.

$$1 - fg_{i+1} \cong 1 - f(2g_i - fg_i^2) \cong 1 - 2fg_i + f^2 g_i^2 \cong (1 - fg_i)^2 \cong 0 \mod x^{2^{i+1}}$$

□

Por tanto, lo que hemos definido gracias al método de Newton es un algoritmo que calcula sucesivas aproximaciones de la inversa g de un polinomio f, donde el teorema arriba demostrado nos indica, primero, que el algoritmo es correcto, y segundo, que cada iteración duplica el número de términos correctos calculados de g. Con estas consideraciones, ya podemos definir el algoritmo.

InversoMedianteNewton(f, l)

Entrada:
1. f, polinomio del que se quiere calcular su inversa.
2. l, precisión que se quiere conseguir de la inversa de f.

$g_0 \leftarrow 1/f_0, \quad n \leftarrow \deg(f), \quad r \leftarrow \lceil \log_2(n) \rceil$
para i desde 1 hasta r:
  $g_i = 2g_{i-1} - fg_{i-1}^2 \text{ rem } x^{2^i}$
return g_r

Para calcular la complejidad temporal de este algoritmo y nos servimos del siguiente teorema enunciado en [7].

Teorema 5.4.4. *El algoritmo InversoMedianteNewton es correcto y calcula el polinomio inverso de un polinomio g en módulo x^l. Si l es una potencia de 2 cualquiera, $l = 2^r$, entonces el método calcula $3M(l) + l \in O(M(l))$ operaciones en nuestro anillo R.*

Demostración. La correctitud del algoritmo se comprueba gracias al **Teorema 5.4.3** y al hecho de que x^l divide a x^{2^r}. Para no realizar operaciones redundantes, en el paso iterativo del algoritmo podemos descartar automáticamente cualquier término de grado superior a x^{2^i} (ya lo hacen el algoritmo de Mulders y la variante), y, como por definición tenemos que $g_i \cong g_{i-1} \cdot (2 - fg_{i-1}) \cong g_{i-1} + \mathcal{O}(x^{2^{i-1}})$, ya tenemos los primeros 2^{i-1} términos, así que no es necesario recalcularlos en cada iteración y podemos descartarlos también. En cuanto el coste, cada una de las iteraciones

del algoritmo calculan $M(2^{i-1})$ operaciones con g_{i-1}^2, $M(2^i)$ para el producto $fg_{i-1}^2 + \mathcal{O}(x^{2^i})$ y como en cada iteración los nuevos términos calculados son únicamente la mitad superior del producto $fg_{i-1}^2 + \mathcal{O}(x^{2^i})$, esto añade 2^{i-1} operaciones.

Por tanto, sumando el coste de cada una de las operaciones obtenemos el coste total de una sola iteración del **Paso 2**, $M(2^i) + M(2^{i-1}) + 2^{i-1} \leq \frac{3}{2}M(2^i) + 2^{i-1}$, y como se ejecutan r iteraciones, obtendremos el coste total del algoritmo sumando los costes de cada iteración, luego

$$\sum_{1\leq i\leq r}\left(\frac{3}{2}M(2^i) + 2^{i-1}\right) \leq \left(\frac{3}{2}M(2^r) + 2^{r-1}\right)\sum_{1\leq i\leq r} < 3M(2^r) + 2^r = M(l) + l,$$

usando que $2M(n) \leq M(2n)\ \forall n \in \mathbb{N}$. □

Ejemplo 5.4.5. *Veamos con un sencillo ejemplo el funcionamiento de método para el cálculo del inverso de un polinomio. Supongamos que estamos trabajando en el espacio $R = \mathbb{Z}/5\mathbb{Z}$, y consideramos el polinomio $A = 2 + 3x + x^2$. Como A es un polinomio con 3 términos, tomemos $l = 4$ como parámetro con el que llamar al algoritmo.*

Así, la llamada que hacemos es InversoMedianteNewton$(A, 4)$:

- *El primer paso es definir como g_0 al polinomio inverso del término independiente de A, donde los polinomios g_i serán las sucesivas aproximaciones que se calculen del inverso. Así, g_0 se define como $g_0 = a_0^{-1} = 2^{-1} \cong 3$ en $\mathbb{Z}/5\mathbb{Z}$.*

  Ahora, calculamos desde $i = 1$ hasta r aproximaciones de g_i, donde r es simplemente $r = \lceil log_2(l)\rceil = 2$, siguiendo la recursión

$$g_i = 2g_{i-1} - fg_{i-1}^2 = g_{i-1}(2 - fg_{i-1}).$$

  1. *$i = 1$.*

     Calculamos cada uno de los términos de la ecuación de arriba por separado, teniendo en cuenta que estamos trabajando en $\mathbb{Z}/5\mathbb{Z}$:

$$A \cdot g_0 = (2 + 3x + x^2) \cdot 3 \cong 1 + 4x + 3x^2 \cong 1 + 4x + \mathcal{O}(x^2),$$

     *ya que, siguiendo las instrucciones del pseudocódigo y el **Teorema 5.4.4**, estamos trabajando con precisión hasta x^2. Continuando con las operaciones, tenemos*

$$2 - (1 + 4x + 3x^2) \cong 1 + x + 2x^2,$$

     y solo resta multiplicar por g_0, esto es,

$$g_1 = 3 \cdot (1 + x + 2x^2) \cong 3 + 3x + x^2$$

     de forma que se puede comprobar que $A \cdot g_1 \cong 1 \mod x^2$.

  2. *$i = 2$. Repetimos el mismo procedimiento que en el paso anterior, pero esta vez con $g_1 = 3 + 3x + x^2$.*

Empezamos multiplicando A por g_1,

$$A \cdot g_1 = (2 + 3x + x^2)(3 + 3x + x^2) \cong 1 + 4x^2 + x^3 + x^4 \cong 1 + 4x^2 + x^3 + \mathcal{O}(x^4),$$

pues en este caso estamos trabajando con precisión hasta x^4. Continuando con el algoritmo

$$2 - A \cdot g_1 = 2 - (1 + 4x^2 + x^3) \cong 1 + x^2 + 4x^3.$$

Finalmente, multiplicando por g_1, obtenemos

$$g_2 = (3 + 3x + x^2)(1 + x^2 + 4x^3) \cong 3 + 3x + 4x^2.$$

Tras finalizar el algoritmo, podemos concluir que $g_2 = 3 + 3x + 4x^2$ es un polinomio tal que $Ag_2 \cong 1 \mod x^4$.

Si el parámetro l que pasemos es mayor que el grado del polinomio A pero no es una potencia de 2, este método calculará coeficientes de más de la inversa de f, pues la aproximación siempre se aproximará a la potencia de 2 más cercana a l desde arriba.

Una vez establecido el método a emplear para calcular la inversa del divisor, se puede definir al completo el algoritmo de la división rápida.

DivisionRápidaConResto(A, B):

Entrada:
1. $A = \sum_{i=0}^{n} a_i x^i, B = \sum_{i=0}^{m} b_i x^i$ polinomios que se van a dividir.
Salida:
(Q, R), cociente y resto de la división de A entre B.
if $\deg(A) < \deg(B)$ **then**:
    return $(A, 0)$.
end if
$m \leftarrow \deg(A) - \deg(B)$
$rev(A) \leftarrow rev_{\deg(A)}(A)$
$(rev(B))^{-1} \leftarrow$ InversoMedianteNewton$(rev_{\deg(B)}(B), m + 1)$
$Q^* = revA \cdot (rev(B))^{-1}$ rem x^{m+1}
Devuelve $(Q = rev_m(Q^*), R = A - BQ)$

Como se introdujo al principio del capítulo, el número de operaciones en R de este algoritmo es $\mathcal{O}(M(n))$, menor que la división de la escuela. Para demostrar que esto es cierto, simplemente enunciamos el siguiente teorema, cuyo mismo enunciado y demostración se pueden encontrar en [7].

Teorema 5.4.6. *Sea R un anillo conmutativo y unitario, A, B dos polinomios en $R[x]$ de grados $n + m$ y n, respectivamente, con $n \geq m \in \mathbb{N}$ y a_0 una unidad de R. El algoritmo DivisionRápidaConResto aplicado a A y B calcula $4M(m) + M(n) + O(n)$ productos en el anillo R.*

Ejemplo 5.4.7. *De nuevo, consideramos el espacio* $R = \mathbb{Z}/5\mathbb{Z}$ *y los polinomios* $A = 3x^3+2x^2+x+4$ *y* $B = x^2+1$. *Usaremos el algoritmo DivisionRapidaConResto para dividir los polinomimos A y B y calcular el cociente y resto correspondiente a dicha operación.*

Tenemos que $\deg(A) = 3$ *y* $\deg(B) = 2$, *por tanto podemos dividir A entre B sin ningún obstáculo aparente. Para empezar, vemos que* $m = 3 - 2 = 1$ *es el grado que tendrá el cociente obtenido al dividir A entre B. Calculamos ahora los recíprocos de ambos polinomios, de forma que* $rev(A) = 4x^3 + x^2 + 2x + 3 \cong 3 + 2x + \mathcal{O}(x^2)$, *ya que posteriormente trabajaremos con precisión* $m + 1$, *y* $rev(B) = 1 + x^2$ *de nuevo.*

Calculamos el inverso de $rev(B)$ *mediante el algoritmo InversoMedianteNewton. Seguimos el esquema del algoritmo:*

1. *Calculamos* $c_0 = 1/b_0$, *teniendo en cuenta que la llamada al algoritmo la hemos hecho con el recíproco de B aunque en este caso particular ambos polinomios sean iguales, luego* $c_1 = 1$.

2. *Calculamos la siguiente iteración de la aproximación del inverso siguiendo la igualdad*

$$c_1 = 2c_0 - rev(B)c_0^2 = c_0(2 - rev(B)c_0),$$

luego

$$c_1 = (2 - rev(B)) = 1 + 4x^2,$$

por tanto, $(rev(B))^{-1} = 1 + 4x^2$.

Una vez calculado el inverso del recíproco de B, calculamos este polinomio por el recíproco de A, de forma que obtenemos la expresión

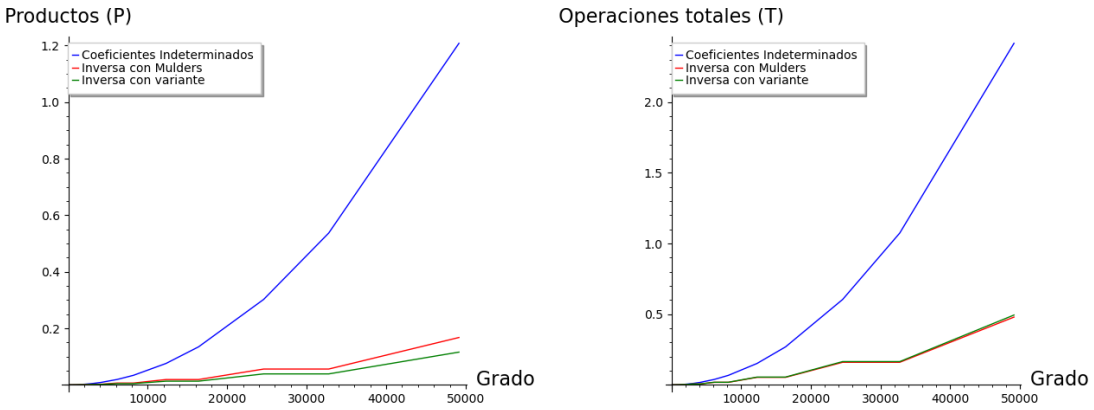
$$Q^* = rev(A) \cdot (rev(B))^{-1} \cong (3 + 2x)(1 + 4x^2) \cong 3 + 2x + \mathcal{O}(x^2).$$

Para finalizar, tenemos que el cociente de la división es $Q = rev(Q^*) = 2 + 3x$, *y el resto R lo calculamos como*

$$R = A - BQ = (4+x+2x^2+3x^3) - (1+x^2)(2+3x) \cong (4+x+2x^2+3x^3) - (2+3x+2x^2+3x^3)$$
$$= 2 + 3x.$$

Para terminar, veamos los resultados obtenidas con las implementaciones de cada algoritmo sobre el número de operaciones realizadas. En este caso, vamos a comparar el número de multiplicaciones y de operaciones totales calculadas. Primero para las implementaciones de Mulders y la variante sin tener en consideración el uso de algoritmos cuadráticos para polinomios de grado pequeño, y, a continuación, para los algoritmos InversaMedianteNewton y DivisionRapidaConResto, comparándolos con sus contrapartes "tradicionales" y consigo mismos cambiando el algoritmo de multiplicación utilizado en cada caso. Estas pruebas analizan de manera indirecta los algoritmos de multiplicación truncada vistos en los capítulos anteriores.

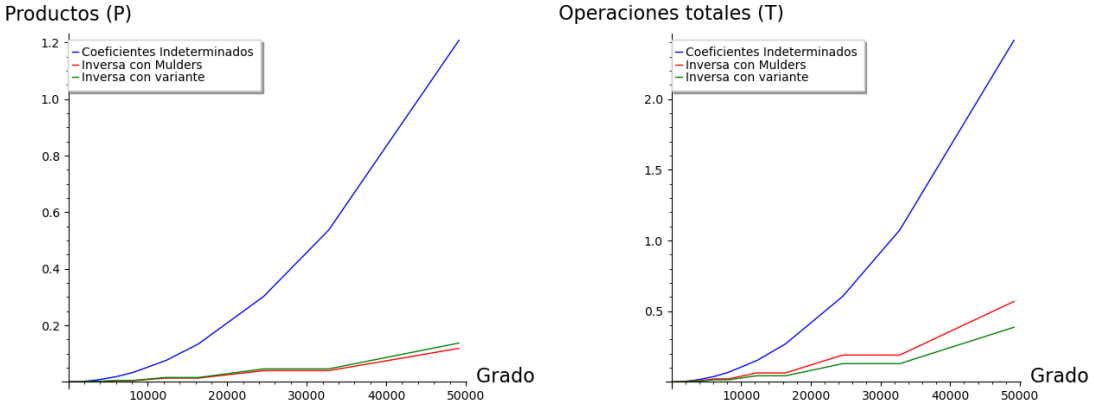
Para las pruebas sobre el método del inverso de un polinomio, se han considerado polinomios de tamaño una potencia de 2, 2^i, desde $i = 7$, y de tamaño $3 \cdot 2^i$, es decir, sumarle la mitad 2^{i-1} a 2^i, hasta $i = 15$, incluido.



(a) Número de productos calculados por cada método

(b) Número de operaciones totales calculadas por cada método

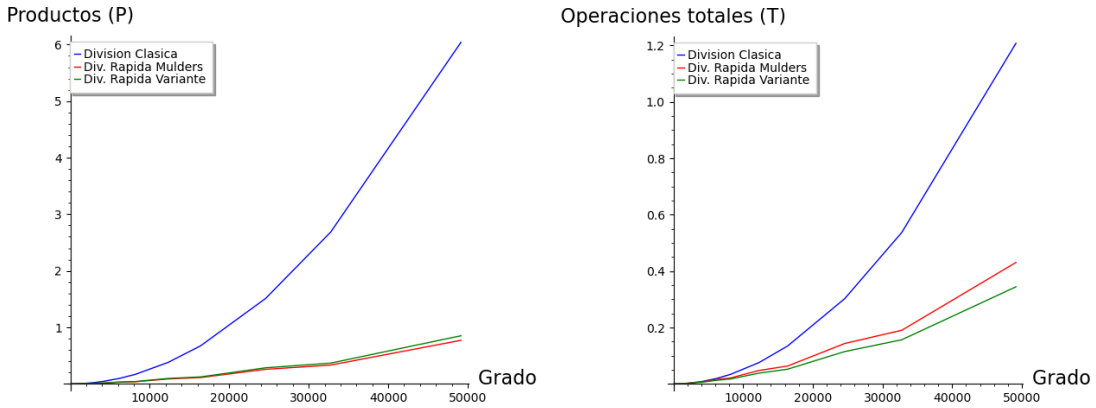
Figura 5.1: Operaciones calculadas por Mulders, la variante y el producto de la escuela



(a) Número de productos calculados por InversoMedianteNewton

(b) Número de operaciones totales calculadas por InversoMedianteNewton

Figura 5.2: Operaciones calculadas por el método del inverso



(a) Número de productos calculados por DivisionRapidaConResto

(b) Número de operaciones totales calculadas por DivisionRapidaConResto

Figura 5.3: Operaciones calculadas por el método de la división

Sobre el algoritmo de división rápida, las pruebas se han hecho, de nuevo, tomando como dividendo de la división polinomios de la misma forma a los escogidos para las pruebas sobre el algoritmo InversoMedianteNewton, y como divisor, polinomios de la mitad de tamaño que el dividendo. Los resultados obtenidos se muestran en la Figura 5.3:

La conclusión que podemos extraer de los datos de la Figura 5.1 es que, efectivamente, la variante de Mulders realiza un número menor de multiplicaciones que el algoritmo de Mulders, a cambio de una pequeña ganancia en el número de operaciones total. En concreto, el número de multiplicaciones computadas por la variante es entre un 20 % y un 30 % menor respecto a Mulders, mientras que la diferencia entre número de operaciones totales es de un 10 % mayor para la variante, diferencia que se hace menor a mayor grado de los polinomios.

Sobre los resultados para los métodos del inverso y la división, consideramos ya el uso de algoritmos cuadráticos con Mulders y la variante. En ambos casos, la conclusión que se puede extraer es que los algoritmos presentados a lo largo del trabajo son más eficientes que los algoritmos tradicionales, como hemos discutido en la memoria.

Centrándonos en el número de operaciones del algoritmo de Mulders y la variante analizando primero los resultados del método InversoMedianteNewton en la Figura 5.2, en cuanto al número de multiplicaciones resulta que la última calcula un número mayor de estas respecto a la primera, una diferencia de entre un 10 % y un 15 % más operaciones, pero sin embargo, se obtiene una ganancia en cuanto a las operaciones totales mucho mayor, pues aquí el número de operaciones totales calculadas por el algoritmo de Mulders es entre un 40 % y hasta un 50 % mayor que la variante.

Esta diferencia se hace menor a la hora de utilizar el algoritmo DivisionRapidaConResto, como se puede ver en la Figura 5.3 donde la conclusión general es idéntica a la anterior, pero ambas diferencias en el número de operaciones se ven reducidas, siendo que el número de productos calculados usando la variante de Mulders es un 10 % mayor prácticamente en cada momento, y el número de operaciones totales calculados por Mulders es de poco más del 20 %.

Cabe mencionar que no hemos hecho un análisis de tiempo debido a que las estructuras de datos escogidas imponen un coste adicional de tiempo, como puede ser la construcción y destrucción de listas o el uso de la clase auxiliar *Element* que, al llevar el conteo de operaciones hace que el coste en tiempo aumente de manera significativa.

Todos estos cambios deberían estar presentes en una implementación que pueda ser usada en producción. Además, para los anillos más usuales, programas como Sage tienen implementado algoritmos específicos, por lo que no es evidente que este tipo de implementaciones genéricas sean de relevancia en la práctica.

Bibliografía

[1] G. Hanrot and P. Zimmermann: *A long note on Mulders' short product*. Journal of Symbolic Computation, Volume 37, Issue 3, 2004.

[2] T. Mulders: *On short multiplication and division*, AAECC 11, 1, 69-68, 200.

[3] C. Palazuelos: *Apuntes de la asignatura Algorítmica y complejidad*, Universidad de Cantabria, 2025.

[4] L.M. Pardo: *Manual de Iniciación al Algebra Conmutativa. Parte Primera*, Universidad de Cantabria, 2024.

[5] J. Segura: *Apuntes de la asignatura Cálculo Numérico I*, Universidad de Cantabria, 2022.

[6] The Sage Developers: *SageMath, the Sage Mathematics Software System (Version* 10,6*)*, 2025, https://www.sagemath.org.

[7] J. von zur Gathen and G. Juergen Gerhard: *Modern Computer Algebra*. Cambridge Univ. Press, 1999.

Apéndice A

Implementación de los algoritmos utilizados

Dejamos en este apéndice las implementaciones en SageMath de cada uno de los algoritmos que hemos explicado a lo largo de la memoria, y con los cuáles hemos realizado todas las pruebas prácticas.

Empezamos viendo la clase Elemento definida, junto a los atributos gracias a los cuales hemos podido llevar la cuenta de cada operación efectuada. Además, como nuestros polinomios son listas, hemos tenido que definir dos métodos que simulan las operaciones de suma y resta de polinomios.

```
class Elemento:
     def __init__(self, a):
          self.valor = a

     def __add__(self, other):
          global TFG_SUMA
          TFG_SUMA += 1
          return Elemento(self.valor + other.valor)

     def __mul__(self, other):
          global TFG_PRODUCTO
          TFG_PRODUCTO += 1
          return Elemento(self.valor * other.valor)

     def __truediv__(self, other):
          global TFG_PRODUCTO
          TFG_PRODUCTO += 1
          return Elemento(self.valor / other.valor)

     def __neg__(self):
          global TFG_PRODUCTO
          TFG_PRODUCTO += 1
          return Elemento(-self.valor)

```

```
25      def __sub__(self, other):
26          global TFG_SUMA
27          TFG_SUMA += 1
28          return Elemento(self.valor - other.valor)
29
30      def __repr__(self):
31          return repr(self.valor)
32
33      def __eq__(self, other):
34          if other == None:
35              return False
36          return self.valor == other.valor
37
38  def reinicia():
39      global TFG_SUMA, TFG_PRODUCTO
40      TFG_SUMA = 0
41      TFG_PRODUCTO = 0
42
43  def suma_polinomios(x: list, y: list):
44      degx = len(x)-1
45      degy = len(y)-1
46      x = copy(x)
47      y = copy(y)
48      if degx == 0:
49          y[0] = y[0] + x[0]
50          return y
51      if degy == 0:
52          x[0] = x[0] + y[0]
53          return x
54      if degx < degy:
55          for i in range(degx + 1):
56              y[i] = y[i] + x[i]
57          return y
58      if degy <= degx:
59          for i in range(degy + 1):
60              x[i] = x[i] + y[i]
61          return x
62
63  def resta_polinomios(x: list, y: list):
64      degx = len(x)-1
65      degy = len(y)-1
66      x = copy(x)
67      y = copy(y)
68      if degx == 0:
69          y[0] = y[0] - x[0]
70          return y
71      if degy == 0:
72          x[0] = x[0] - y[0]
73          return x
74      if degx < degy:
75          for i in range(degx + 1):
```

```
            y[i] = y[i] - x[i]
        return y
    if degy <= degx:
        for i in range(degy + 1):
            x[i] = x[i] - y[i]
        return x
```

Listing A.1: Clase Elemento y operaciones con polinomios

Ahora, veremos los algoritmos de multiplicación de polinomios definidos a lo largo de la memoria: por un lado los algoritmos clásicos, donde tenemos el producto de la escuela y el algoritmo de Karatsuba, y, por otro, los algoritmos estudiados, Mulders y su variante. Destacamos que el parámetro n de Mulders y la variante indican la precisión $\mathcal{O}(x^n)$ hasta la que se quiere obtener el resultado, mientras que el parámetro n de los algoritmos clásicos indican el grado de los polinomios que se pasan también como parámetro.

En este caso los métodos do_schoobook_product, do_karatsuba y do_karatsuba_different_size son los métodos que están implementados en Sage para el producto de polinomios con coeficientes en un anillo arbitrario.

```
from math import ceil, floor, log2

TFG_SUMA = 0
TFG_PRODUCTO = 0

def do_schoolbook_product(x: list, y: list, deg: int = -1):
    d1 = len(x) - 1; d2 = len(y) - 1
    if deg < 0 or deg >  d1 + d2 + 1:
        deg = d1 + d2 + 1
    if d1 == -1:
        return x
    elif d2 == -1:
        return y
    elif d1 == 0:
        return [x[0] * a for a in y[:deg]]
    elif d2 == 0:
        return [a * y[0] for a in x[:deg]]
    coeffs = [None] * deg
    for k in range(deg):
        start = 0 if k <= d2 else k - d2
        end = k if k <= d1 else d1
        sum = x[start] * y[k - start]
        for i in range(start + 1, end + 1):
            sum = sum + x[i] * y[k-i]
        coeffs[k] = sum
    return coeffs

def do_karatsuba(left, right, K_threshold, start_l, start_r, num_elts):
    if num_elts == 0:
        return []
    if num_elts == 1:
        return [left[start_l]*right[start_r]]
```

```
33      if num_elts <= K_threshold:
34          if num_elts == 2:
35              b, a = left[start_l], left[start_l + 1]
36              d, c = right[start_r], right[start_r + 1]
37              return [b*d, a*d+b*c, a*c]
38          return do_schoolbook_product(left[start_l: start_l + num_elts], right[
    start_r: start_r + num_elts])
39      if num_elts == 2:
40          b, a = left[start_l], left[start_l + 1]
41          d, c = right[start_r], right[start_r + 1]
42          ac = a*c
43          bd = b*d
44          return [bd, (a+b)*(c+d)-ac-bd, ac]
45      e = num_elts // 2
46      ne = num_elts - e
47      start_le = start_l + e
48      start_re = start_r + e
49      ac = do_karatsuba(left, right, K_threshold, start_le, start_re, ne)
50      bd = do_karatsuba(left, right, K_threshold, start_l, start_r, e)
51      a_m_b = left[start_le:start_le + ne]
52      c_m_d = right[start_re:start_re + ne]
53      for i in range(e):
54          a_m_b[i] += left[start_l + i]
55          c_m_d[i] += right[start_r + i]
56      tt1 = do_karatsuba(a_m_b, c_m_d, K_threshold, 0, 0, ne)
57      for i in range(len(bd)):
58          tt1[i] -= (ac[i] + bd[i])
59      for i in range(len(bd), len(ac)):
60          tt1[i] -= ac[i]
61      for i in range(e-1):
62          bd[e+i] += tt1[i]
63      bd.append(tt1[e-1])
64      for i in range(len(ac) - e):
65          ac[i] += tt1[e+i]
66      return bd + ac
67
68  def do_karatsuba_different_size(left, right, K_threshold):
69      n, m = len(left), len(right)
70      if n == 0 or m == 0:
71          return []
72      if n == 1:
73          return [left[0] * a for a in right]
74      if m == 1:
75          return [right[0] * a for a in left]
76      if n <= K_threshold or m <= K_threshold:
77          return do_schoolbook_product(left, right)
78      if n == m:
79          return do_karatsuba(left, right, K_threshold, 0, 0, n)
80      if n > m:
81          q = n // m; r = n % m
82          output = do_karatsuba(left, right, K_threshold, 0, 0, m)
```

```
        for i in range(1, q):
            mi = m * i
            carry = do_karatsuba(left, right, K_threshold, mi, 0, m)
            for j in range(m-1):
                output[mi + j] += carry[j]
            output.extend(carry[m-1:])
        if r:
            mi = m*q
            carry = do_karatsuba_different_size(left[mi:], right, K_threshold)
            for j in range(m-1):
                output[mi + j] += carry[j]
            output.extend(carry[m-1:])
        return output
    else:
        q = m // n; r = m % n
        output = do_karatsuba(left, right, K_threshold, 0, 0, n)
        for i in range(1, q):
            mi = n * i
            carry = do_karatsuba(left, right, K_threshold, 0, mi, n)
            for j in range(n-1):
                output[mi + j] += carry[j]
            output.extend(carry[n-1:])
        if r:
            mi = n*q
            carry = do_karatsuba_different_size(left, right[mi:], K_threshold)
            for j in range(n-1):
                output[mi + j] += carry[j]
            output.extend(carry[n-1:])
        return output

def mulders_short_product(f, g, n):
    if n == 0:
        return []
    if n == 1:
        return [f[0]*g[0]]
    k = ceil(0.694 * n)
    P1, Q1 = f[:k], f[k:]
    P2, Q2 = g[:k], g[k:]
    C1 = do_karatsuba(P1, P2, 0, 0, 0, k)
    C2 = mulders_short_product(P1[:n-k], Q2, n-k)
    C3 = mulders_short_product(P2[:n-k], Q1, n-k)
    C1 = C1[:n]
    for i in range(n-k):
        C1[k+i] += C2[i] + C3[i]
    return C1

def Mulders_variant(f, g, n):
    if n == 0:
        return []
    if n == 1:
        return [f[0]*g[0]]
```

```
    n0 = floor(n/2); n1 = ceil(n/2)
    f_even, f_odd = f[::2], f[1::2]
    g_even, g_odd = g[::2], g[1::2]
    l = Mulders_variant(f_even, g_even, n1)
    h = Mulders_variant(f_odd, g_odd, n0)
    fe_m_fo = [f_even[i] + f_odd[i] for i in range(n0)]
    ge_m_go = [g_even[i] + g_odd[i] for i in range(n0)]
    if n % 2 == 1:
        fe_m_fo.append(f_even[-1])
        ge_m_go.append(g_even[-1])
    m = Mulders_variant(fe_m_fo, ge_m_go, n0)
    for i in range(n0):
        m[i] -= l[i] + h[i]
    if n % 2 == 1:
        m.append(-l[-1])
    total = [None]*(n+1)
    for i in range(n0):
        total[2*i] = l[i]
        total[2*i+1] = m[i]
    if n % 2 == 1:
        total[2*n0] = l[-1]
    for i in range(n0-1):
        total[2+2*i] += h[i]
    if total[2*n0] is None:
        total[2*n0] = h[n0-1]
    else:
        total[2*n0] += h[n0-1]
    return total[:n]
```

Listing A.2: Implementación de algoritmos de multiplicación de polinomios

Para continuar, mostramos la implementación de los algoritmos para calcular la inversa, tanto del método basado en la iteración de Newton, como el método de los coeficientes indeterminados.

```
def inv_mulders(g, k = -1):
    if k == -1:
        k = log(len(g), 2).n().ceil()
    inv_g = [Elemento(1)/g[0]]
    g = g + [Elemento(0)] * (2**k - len(g))
    for i in range(k):
        gi2 = do_karatsuba(inv_g, inv_g, 8, 0, 0, 2**i) + [Elemento(0)]
        g_por_inv = mulders_short_product(g, gi2, 2**(i+1))
        inv_g = inv_g + [-foo for foo in g_por_inv[2**i:2**(i+1)]]
    return inv_g

def inv_variante(g, k = -1):
    if k == -1:
        k = log(len(g), 2).n().ceil()
    inv_g = [Elemento(1)/g[0]]
    g = g + [Elemento(0)] * (2**k - len(g))
    for i in range(k):
        gi2 = do_karatsuba(inv_g, inv_g, 8, 0, 0, 2**i) + [Elemento(0)]
```

```
        g_por_inv = Mulders_variant(g, gi2, 2**(i+1))
        inv_g = inv_g + [-foo for foo in g_por_inv[2**i:2**(i+1)]]
    return inv_g

def coefs_indeterminados(x: list, tam = -1):
    """
    Algoritmo para calcular la inversa de un polinomio mediante
    el metodo de los coeficientes indeterminados, es decir, usando
    la multiplicacion de la escuela y despejando cada coeficiente.

    Algoritmo pensado para usar con listas de tipo Elemento.
    """
    if tam == -1:
        tam = len(x)
    if tam > len(x):
        x = x + [Elemento(0)] * (tam - len(x))
    if len(x) == 1:
        return [Elemento(1) / x[0]]
    y = []
    for i in range(tam):
        gk = Elemento(0)
        if i == 0:
            gk = Elemento(1) / x[0]
        else:
            for j in range(1, i+1):
                gk = gk + x[j] * y[i - j]
            gk = -gk / x[0]
        y.append(gk)
    return y
```

Listing A.3: Cálculo de la inversa de un polinomio

Por último, mostramos la implementación de los métodos de división, tanto de la división clásica como de la división rápida.

```
def division_rapida_mulders(f, g):
    df = len(f) - 1
    dg = len(g) - 1
    if df < dg:
        return []
    m = df - dg
    rg = list(reversed(g))
    rf = list(reversed(f))
    g1 = inv_mulders(rg)
    if m >= len(g1):
        g1 = g1 + [Elemento(0)] * (m - len(g1) + 1)
    q1 = mulders_short_product(rf, g1, m + 1)
    q3 = q1 + [Elemento(0)] * (m + 1 - len(q1))
    q3.reverse()
    return (q3, suma_polinomios(f, [-k for k in mulders_short_product(
        g + [Elemento(0)] * (len(f) - len(g)),
        q3 + [Elemento(0)] * (len(f) - len(q3)),
```

```
        len(f)
    )]))

def division_rapida_variante(f, g):
    df = len(f) - 1
    dg = len(g) - 1
    if df < dg:
        return []
    m = df - dg
    rg = list(reversed(g))
    rf = list(reversed(f))
    g1 = inv_variante(rg)
    if m >= len(g1):
        g1 = g1 + [Elemento(0)] * (m - len(g1) + 1)
    q1 = Mulders_variant(rf, g1, m + 1)
    q3 = q1 + [Elemento(0)] * (m + 1 - len(q1))
    q3.reverse()
    return (q3, suma_polinomios(f, [-k for k in mulders_short_product(
        g + [Elemento(0)] * (len(f) - len(g)),
        q3 + [Elemento(0)] * (len(f) - len(q3)),
        len(f)
    )]))

def division_clasica(f: list, g: list):
    """
    Algoritmo tradicional para la division de polinomios.
    """
    degf = len(f) - 1
    deg_g = len(g) - 1
    r = copy(f)
    u = Elemento(1) / g[deg_g]
    q = []
    for i in range(degf - deg_g, -1, -1):
        if len(r) - 1 == deg_g + i:
            qi = u * r[len(r) - 1]
            for j in range(deg_g, -1, -1):
                r[i + j] = r[i + j] - g[j] * qi
            r = r[:len(r) - 1]
            q = [qi] + q
    return (q, r)
```

Listing A.4: División de polinomios: clásica y rápida con Mulders y su variante