¿Qué tan rápido multiplicamos cuando multiplicamos rápido?

Un viaje desde la aritmética escolar hasta la transformada rápida de Fourier

Lautaro Yerimen Arias

Facultad de Ciencias Exactas - UNLP

"La ciencia es lo que entendemos suficientemente bien para explicarle a una computadora; el arte es todo lo demás"

Donald Knuth

Concurso de monografías UMA - 2025

Agradecimientos

Indudablemente, mi primer agradecimiento tiene que ir para el Mono y Juli, mis dos grandes amigos y compañeros en el increíble camino de aprender algoritmos desde cero, con quienes compartimos tantos años y tantas jornadas disfrutando combinar matemática con programación.
Tengo que agradecerle también a mi familia, por el aguante, la paciencia y el aliento constante para que avance con el trabajo.
A Alfonsina, la primera persona que escuchó mi idea para la monografía y me motivó a meterle para adelante, que también fue la primera persona en leer un borrador del trabajo.
A ella, a Yain, a Malen y a Valentín tengo que agradecerles sus valiosos comentarios sobre lo escrito para poder mejorar el trabajo.
Nuevamente a Valentín y también a Ezequiel por su invaluable compañía y aguante a lo largo de todo el proceso de escritura. A Carlos Salvador y a León por siempre bancarme en estos proyectos que me enganchan.
A cada persona que me escuchó a lo largo de estos meses contarles las ideas que tanto me gustan y que busco contar en este trabajo. A la UMA por la oportunidad del concurso.
A la OMA por ser la más bella puerta de entradas a la matemática y a la ICPC por ser la puerta de entrada más linda a la programación.
A toda aquella persona con la que haya tenido el placer de compartir un rato de matemática, pensando problemas, aprendiendo teoremas o simplemente filosofando sobre algún resultado o historia que nos haya llamado la atención.
A la UNLP, a la Facultad de Ciencias Exactas y a la educación pública de nuestro país que tantas oportunidades me dio y a la que estaré siempre agradecido.
En la memoria de Dora y Amor.

Prólogo

Este trabajo no busca ser autoconcluyente, de hecho su objetivo es exáctamente el opuesto: despertar el interés del lector en los temas que trata y que al terminarlo tenga ganas de profundizar más su conocimiento al respecto. En estas páginas se invita al lector a adentrarse en el increíble mundo de las matemáticas aplicadas a los algoritmos computacionales, buscando dar un pantallazo de algunas ideas que recuerdo como si hubiese sido ayer la fascinación que me generaron cuando las leí por primera vez, tanto por lo ingeniosas que me resultaron como por todo el provecho que les pude sacar aplicándolas a la resolución de problemas relacionados a la programación. Técnicas de lo más variadas y entrelazamiento de distintas ramas de la matemática que uno no está muy familiarizado con ver trabajar en conjunto se unen de manera natural. A veces, con el fin de generar una lectura fluida se optó por omitir demostraciones muy técnicas o extensas, no sin antes facilitar fuentes para todo aquel interesado en leerlas, o simplificar un poco algunas definiciones para hacerlas más naturales para el contexto donde iban a ser usadas, intentando conseguir el siempre difícil equilibrio entre la rigurosidad matemática y la motivación de las ideas y los pasos realizados. Muchos de los conceptos trabajados cuentan con una amplia variedad de aplicaciones, por lo que se optó por presentar algunos pero siempre invitando al lector interesado en continuar su formación sobre el tema con material extra. Espero que puedan disfrutar de la lectura de este trabajo.

Índice

1. Introducción.

"La vida es buena por solo dos cosas: descubrir matemáticas y enseñar matemáticas."

Simeón Denis Poisson, 1781–1840.

Tenía 8 años y estaba en tercer grado. Cecilia, la coordinadora de matemática de mi escuela, había organizado una competencia entre los chicos de mi grado donde ella sacaba dos números al azar entre el 1 y el 10 y había que decir rápido el resultado de su multiplicación. Nunca fui muy bueno para acordarme las tablas, así que la mejor idea que se me ocurrió para practicar fue hacer muchas multiplicaciones porque creía que de ese modo iba a poder hacer las cuentas cada vez más rápido. Gané varias rondas, mi método de práctica estaba dando sus frutos... Hasta que me tocó calcular 9 por 7. La tabla del 9 era mi peor enemigo, me parecía por mucho la más difícil, pero una compañera, Inés, casi automáticamente dijo 63. Quedé fascinado por lo que era, o una gran memoria para las tablas, o una grandísima velocidad de cálculo.

Algunos días después, rompiendo el código de honor de los magos, Inés me reveló su secreto: "en la tabla del 9 yo lo que hago es ponerle un 0 al final y después resto el numero, entonces 7 por 9 es 70 menos 7, que es más fácil de resolver". Me pareció simplemente increíble, una idea revolucionaria. Inés había encontrado la manera no de multiplicar más rápido, sino mejor aún, de hacerlo de manera más eficiente. Ella había aplicado, sin saberlo, la **propiedad distributiva**:

$$9x = (10 - 1)x = 10x - x$$

La intuición atrás de su idea tenía arraigado un principio muy profundo: optimizar no es hacer cálculos más rápidos, sino reestructurar el problema para reducir la cantidad de operaciones. Esta idea es el corazón de la teoría de la complejidad computacional. A lo largo de este trabajo, vamos a recorrer ideas increíbles, teoremas elegantes y sobre todo algoritmos basados en estas herramientas matemáticas que nos permitan multiplicar *más rápido*, o mejor dicho, de manera más eficiente. A cada herramienta nueva que presentemos vamos a acompañarla de algunos problemas donde pueda lucirse. Muchos resultados matemáticos y algoritmos de los que vamos a hablar tienen una belleza intrínseca por la elegancia de sus demostraciones, pero también por la sutileza con la que puedan ser usados en la resolución de distintos problemas planteados.

Capítulo a capítulo en esta monografía vamos a presentar algoritmos que nos van a permitir resolver problemas, que sin ellos llevaría cientos de años, en un puñado de segundos. Nuestro recorrido comenzará con un capítulo para discutir y sentar las bases sobre algunas ideas sobre cómo medir y comparar la eficiencia de algoritmos y qué es considerado una operación en el marco de la complejidad computacional. Tras esto, nos adentraremos de lleno en un caso particular de la multiplicación: el cálculo de potencias. Presentaremos la exponenciación binaria, un elegante algoritmo que reduce de manera sustancial el número de operaciones necesarias, y exploraremos tres sorpresivas aplicaciones: en tests de primalidad, en el cálculo de términos de sucesiones de recurrencias como Fibonacci e incluso en teoría de grafos, calculando potencias de algo más que números. A continuación, abordaremos el problema de multiplicar dos números enteros

de gran tamaño, donde el ingenioso algoritmo de Karatsuba representó el primer gran avance sobre el método escolar. Para terminar, llevaremos la multiplicación un paso más allá para multiplicar polinomios, con la Transformada Rápida de Fourier (FFT) como el gran recurso para hacerlo, e introduciremos las Funciones Generatrices como herramienta de modelado para resolver problemas de combinatoria.

2. Orden de complejidad

2.1. ¿Qué significa que un algoritmo sea más eficiente que otro?

Cuando se piensa en qué influye en una manera rápida de ejecutar un algoritmo, en lo primero que pensamos es en la potencia de la computadora en la que va a ser ejecutado. Pero, ¿existe una forma universal de medir la eficiencia, que trascienda implementaciones concretas o limitaciones de hardware?

Suele ser poco intuitivo al principio pensar en el poder de cómputo de un ordenador moderno o en qué optimizaciones pueden llegar a tener un impacto real en el tiempo de ejecución. Si estamos resolviendo cálculos a mano, ahorrarnos unas 1000 operaciones probablemente nos reduzca nuestra tarea en varios minutos. Sin embargo, una computadora promedio es capaz de resolver 10^8 *operaciones elementales* en unos 0.2 segundos, por lo que una optimización que reduzca en 1000 operaciones el proceso ahorraría 0.0002 segundos del tiempo de ejecución. Imperceptible.

Imaginemos que contamos con dos posibles beneficios para usar en una compra de supermercado. El primero que nos hace un descuento fijo de $4000, el segundo que nos hace un descuento del 50% del monto total. Es claro que en una compra de $5.000 el ahorro del primer beneficio es mayor que el del segundo: $4.000 de ahorro contra $2.500. Pero en una compra de $1.000.000, el ahorro del segundo beneficio es de $500.000, mientras que con el primero seguiría siendo de $4.000. La conclusión es clara: la elección que con montos pequeños es la mejor opción a medida que el gasto aumenta se va volviendo más y más desfavorable. Este cambio de paradigma, de buscar "descuentos fijos" a buscar "descuentos porcentuales", es la esencia de la teoría de la complejidad computacional: analizar el comportamiento de su eficiencia conforme los valores de entrada se van volviendo más y más grandes.

2.2. Dos definiciones necesarias: operación elemental y peor caso

La idea de decidir si un algoritmo es más eficiente que otro tiene implícito el contar, o al menos estimar, cuántas operaciones realiza cada uno para comparar estas cantidades. Es en este momento cuando surge la necesidad de formalizar un poco más algo que hasta ahora veníamos pasando por alto: *¿qué tipo de operaciones contamos?*. Aunque a la hora de escribir en la calculadora pueda parecer una única operación, es claro que calcular 3^{30} tiene de fondo muchas más operaciones que las que tiene el calcular $3 + 30$. Es entonces necesario hablar del concepto de **operación elemental** de manera un poco más formal.

En el modelo de computación *Random Access Machine* (RAM) [1], una **operación elemental** es cualquier operación que:

1. Se ejecuta en tiempo constante independientemente del tamaño de los operandos (más adelante diremos que el tiempo de ejecución es $O(1)$),

2. corresponde a instrucciones atómicas del procesador, es decir que no puede ser interrumpida ni dividida en pasos más pequeños,

3. pertenece a uno de estos tipos:

   - Operaciones aritméticas: $+$, $-$, $\times$, $/$, $\bmod$ sobre enteros de tamaño fijo (ej: 32/64 bits),
   - operaciones lógicas: AND, OR, NOT, XOR, desplazamientos de bits,
   - operaciones en memoria: lectura/escritura en posiciones de memoria,
   - operaciones de comparación: $<$, $>$, $=$, $\neq$ entre valores.

Además, para algoritmos que manipulan números arbitrariamente grandes (como multiplicación de enteros), una operación elemental es *la manipulación de un solo dígito* (sin importar cual sea la base) [2].

Ya habiendo formalizado un poco más la idea de operación elemental, podemos adentrarnos más en la discusión sobre la complejidad computacional. En la sección anterior discutimos sobre la importancia de analizar la eficiencia de los algoritmos a medida que los valores de entrada iban creciendo. Pero vamos a ver que con este análisis no alcanza. Supongamos que recibimos como dato de entrada una permutación del conjunto $(1, 2, 3, 4, 5, 6, 7, 8)$ y contamos con un algoritmo eficiente que tiene que dar una serie de pasos para ordenar los números de menor a mayor. Es claro que las entradas $(1, 2, 3, 4, 5, 6, 7, 8)$ y $(2, 5, 3, 8, 6, 1, 4, 7)$ no van a necesitar la misma cantidad de operaciones para ser ordenadas, porque en el primer caso los números ya se encuentran ordenados. Surge entonces la necesidad de definir tres tipos de casos a analizar al definir un algoritmo:

- **Peor caso:** Máximo tiempo posible para cualquier entrada de tamaño n.
  Ejemplo: Multiplicar $99\ldots9 \times 99\ldots9$ con el método escolar.
  Importancia: Garantiza cotas seguras.

- **Caso promedio:** Tiempo esperado con entradas aleatorias.
  Ejemplo: Multiplicar números con dígitos uniformemente distribuidos.
  Importancia: Predice comportamiento en escenarios reales.

- **Mejor caso:** Mínimo tiempo posible (no suele ser demasiado útil en la práctica).
  Ejemplo: Multiplicar por 10^k (basta con agregar k ceros al final del número).

En la práctica del análisis de algoritmos, el peor caso termina siendo el más usado como medida porque nos brinda cotas seguras e incondicionales: si un algoritmo es eficiente en el escenario más desfavorable, lo será en cualquier situación. En un mundo donde las entradas patológicas existen (como los hackers intentando colapsar sistemas o datos generados adversariamente), y donde hay sistemas críticos que se busca que no colapsen (como frenos automáticos o transacciones bancarias), esta garantía tiene un valor extra. ¿Significa esto que el caso promedio y el mejor caso son inútiles?. No, pero su rol es complementario y ayuda a refinar los algoritmos para datos típicos.

2.3. El concepto clave de la complejidad computacional: La notación O grande

Con un criterio unificado para medir esfuerzo computacional (operaciones elementales) y contexto (peor caso), estamos listos para introducir el lenguaje que sintetiza escalabilidad: la notación O grande[1].

El orden de crecimiento del tiempo de ejecución de un algoritmo ofrece una caracterización simple de su eficiencia y permite comparar el desempeño relativo de distintas alternativas. Aunque calcular tiempos exactos de ejecución es posible, la extrema precisión a la hora de contar cantidad de operaciones no suele justificar el costo analítico que tiene, y con poder hacer estimaciones es suficiente. La idea detrás de esto es que en cualquier expresión algebráica que represente la cantidad de operaciones de un algoritmo en función del tamaño de la entrada, a medida que el tamaño se vuelve arbitrariamente grande los términos de menor orden y los factores constantes van siendo eclipsados por el tamaño del término de mayor orden.

Vamos ahora a formalizar un poco más esto: decimos que una función $f(n)$ es de orden $O(g(n))$, y se lee "f es O grande de g", si existen constantes positivas c y n_0 tales que para todo $n \geq n_0$, se cumple que $|f(n)| \leq c \cdot g(n)$, que no es otra cosa que decir que $f(n)$ a partir de cierto punto n_0 no crece más rápido que $g(n)$, salvo por un factor constante c. De este modo decimos que, por ejemplo, $f(n) = 2n^2 + 70n + 3$ es de orden $O(n^2)$ porque para $n \geq 36$ y $c = 4$ vale que $f(n) \leq 4n^2$. Que un algoritmo sea $O(g(n))$ no significa que siempre haga $g(n)$ operaciones, sino que en el peor caso no crece más rápido que $g(n)$.

2.3.1 Álgebra de la notación O: Sumas y productos

Ya definida la notación, necesitamos establecer algunas reglas prácticas sobre cómo operar entre las complejidades de distintas operaciones para analizar algoritmos que resulten de combinarlas.

- **Suma de complejidades en bloques secuenciales:** Si un algoritmo realiza una tarea seguida de otra, la complejidad total está dominada por la tarea más costosa.

$$O(f(n)) + O(g(n)) = O(\max(f(n), g(n)))$$

  .
  Ejemplo: Si un algoritmo ejecuta una operación de complejidad $O(n^2)$ seguido de un bucle de complejidad $O(n)$, la complejidad del algoritmo será de $O(n^2)$.

- **Producto de complejidades en bloques anidados:** Si un algoritmo realiza una tarea dentro de un bucle, las complejidades se multiplican.

$$O(f(n)) \cdot O(g(n)) = O(f(n) \cdot g(n)).$$

[1]Aunque la notación de O grande es la más extendida para expresar cotas superiores, la teoría de la complejidad también utiliza las notaciones $\Omega(g(n))$ para cotas inferiores, y $\Theta(g(n))$ para cotas ajustadas (que son simultáneamente superiores e inferiores). Para mantener el foco de esta monografía, nos centraremos exclusivamente en la notación O grande, ya que es la herramienta fundamental y suficiente para comparar la eficiencia de los algoritmos que presentaremos.

Ejemplo: un bucle que se repite n veces es $O(n)$ y en cada iteración realiza un trabajo con una complejidad de $O(\sqrt{n})$, la complejidad total del algoritmo será $O(n\sqrt{n})$.

2.3.2 Jerarquía de complejidades más comunes

A continuación, presentamos una jerarquía de las clases de complejidad más comunes, ordenadas de la más eficiente a la menos eficiente. Esta lista sirve como guía para poder comparar rápidamente la eficiencia de distintas soluciones a un mismo problema.

$$O(1) < O(\log(n)) < O(\sqrt{n}) < O(n) < O(n\log(n)) < O(n^2) < O(2^n) < O(n!).$$

Nota:[2]

En general, los algoritmos cuya complejidad es polinómica, o mejor, se consideran eficientes o tratables. Mientras que aquellos con complejidad exponencial, o peor, se consideran ineficientes o intratables para entradas grandes.

Para tomar una dimensión real de lo que estas jerarquías significan en la práctica, analicemos en la siguiente tabla el tiempo de ejecución estimado para un algoritmo de cada clase de complejidad para distintos tamaños de entrada n. Para la elaboración de la tabla, asumimos una velocidad de procesamiento de unas 10^8 operaciones elementales por segundo.

Complejidad	**n = 20**	**n = 50**	**n = 500**	**n = 50,000**	**n = 1,000,000**
$O(\log(n))$	4.3×10^{-8} s	5.6×10^{-8} s	9.0×10^{-8} s	1.6×10^{-7} s	2.0×10^{-7} s
$O(\sqrt{n})$	4.5×10^{-8} s	7.1×10^{-8} s	2.2×10^{-7} s	2.2×10^{-6} s	1.0×10^{-5} s
$O(n)$	2.0×10^{-7} s	5.0×10^{-7} s	5.0×10^{-6} s	0.0005 s	0.01 s
$O(n\log(n))$	8.6×10^{-7} s	2.8×10^{-6} s	4.5×10^{-5} s	0.008 s	0.2 s
$O(n^2)$	4.0×10^{-6} s	2.5×10^{-5} s	0.0025 s	25 s	2.8 horas
$O(n^3)$	8.0×10^{-5} s	0.0012 s	1.25 s	14.5 días	317 años
$O(2^n)$	0.01 s	130 días	$> 10^{134}$ años	Mucho más que la edad del universo	
$O(n!)$	771 años	$> 10^{48}$ años	$> 10^{1118}$ años	Supera cualquier analogía física imaginable	

La tabla habla por sí sola. Un algoritmo considerado eficiente como $O(n\log n)$, resuelve un problema con un millón de elementos en una fracción de segundo. Un algoritmo cuadrático ya tarda horas para el mismo tamaño. A partir de la complejidad cúbica los tiempos se vuelven impracticables para entradas moderadamente grandes, y las complejidades que la intuición nos sugería que eran muy ineficientes, como $O(2^n)$ y $O(n!)$, se vuelven imposibles de calcular para valores de n sorprendentemente pequeños. Viendo esta tabla no quedan dudas: la optimización de los algoritmos muchas veces es la diferencia entre lo computable y lo inalcanzable.

[2]Es una convención en ciencias de la computación que, al hablar de complejidad, la base del logaritmo en $O(\log(n))$ se asume como base 2, ya que muchos algoritmos operan dividiendo el problema en mitades. Sin embargo, desde la perspectiva de la notación O-grande, la base es irrelevante, pues cambiar de base solo introduce un factor constante que la notación absorbe.

2.4. Una mención inevitable: P vs. NP.

Una vez presentada la jerarquía de complejidades, podemos clasificar los problemas en dos grandes familias: aquellos que podemos resolver eficientemente y aquellos que, en la práctica, son intratables para entradas de gran tamaño. En este contexto, una pregunta que surge es: ¿Hay problemas cuya solución podamos verificar rápidamente, pero que no sabemos encontrar una solución en tiempo polinómico? Esta pregunta, en la teoría de la complejidad computacional, recibió el nombre de P vs. NP y en el año 2000 fue elegido como uno de los siete Problemas del Milenio del Clay Mathematics Institute. Para entender de que se trata vamos a definir dos clases de problemas computacionales:

- **Clase P, o de tiempo polinómico:** son todos los problemas de decisión que pueden ser resueltos por un algoritmo en un tiempo polinómico. Es decir, los problemas "eficientemente resolubles".

- **Clase NP o de tiempo polinómico no determinista:** son todos los problemas de decisión donde, no necesariamente sabemos cómo encontrar una solución de manera eficiente, pero sí podemos verificar una posible solución en tiempo polinómico. Por ejemplo, dado un Sudoku, decidir si tiene solución es muy difícil. Pero dada una solución completa, es fácil verificar si es válida o no.

Es claro que todo problema en P también está en NP, porque si podemos encontrar una solución rápida, también podemos verificarla rápidamente. La pregunta que más interés genera es si resulta que todos los problemas en NP están también en P. Es decir, si todo problema cuya solución puede ser verificada rápidamente puede también ser resuelto rápidamente. La creencia mayoritaria es que $P \neq NP$, pero nadie ha logrado demostrarlo.

Dentro de la clase NP, existe una familia de problemas especialmente interesantes que son llamados **NP-completos**. Pero para hablar de ellos primero necesitamos el concepto de **reducibilidad**. Decimos que un problema A se puede reducir a un problema B, si podemos transformar cualquier instancia del problema A en una instancia del problema B en tiempo polinómico preservando la respuesta. Es decir, la instancia transformada es "sí" en B, si y solo si la original era "si" en A, en caso de que B sea un problema de decisión o en una solución óptima satisfaciendo cierta condición, en caso de que sea un problema de optimización. Intuitivamente esto significa que "B es al menos tan difícil como A".

Con este concepto nuevo, ahora podemos hablar de los problemas NP-duros, que son aquellos problemas (de decisión u optimización) a los cuales se puede reducir todo problema en NP. Por último, los problemas NP-completos son los problemas NP-duros que también están dentro de NP, y son considerados "los más difíciles" dentro de esta clase. Esta definición tiene implícito un resultado muy fuerte: encontrar una solución polinómica a cualquier problema NP-completo, sería encontrar una solución eficiente a todos los problemas de NP y demostraría que $P = NP$.

3. Exponenciación binaria.

3.1. Multiplicando a la velocidad del logaritmo

Nota importante: a lo largo de este capítulo, para no perder la rigurosidad de las afirmaciones, vamos a asumir siempre, se explicite o no en la sección, que estamos trabajando con aritmética modular con algún módulo primo arbitrario p relativamente grande, por ejemplo $10^9 + 7$, porque al trabajar con números arbitrariamente grandes las operaciones dejan de ser operaciones elementales por lo que dejan de ejecutarse en tiempo constante y el cálculo de las complejidades es un poco más detallado porque requiere considerar operaciones por dígitos, mientras que tomar módulo en cada paso es $O(1)$ y nos permite mantener acotado el tamaño de los operandos para seguir realizando operaciones aritméticas en $O(1)$. La idea del capítulo es ilustrar las ideas detrás de los algoritmos y no entrar en tecnicismos, más aún, varios de estos algoritmos son usados en el contexto de aritmética modular como ya veremos.

Vamos a comenzar con un caso particular de la multiplicación: las potencias. La expresión a^n, para exponentes enteros positivos, más allá de ser una manera compacta de expresar el producto entre un número y si mismo varias veces, parece requerir $n - 1$ multiplicaciones si usamos el método con el que estamos acostumbrados a trabajar:

$$a^n = \underbrace{a \times a \times \cdots \times a}_{n \text{ veces}}.$$

Esto es $O(n)$ operaciones, que para valores como $n = 10^{20}$ tardaría más de **3000 años**. Si quisiéramos un algoritmo más eficiente, deberíamos reducir considerablemente la complejidad. Veamos un ejemplo para motivar la idea que pronto presentaremos.
Vamos a buscar calcular 3^{16}, que según el método que conocemos requiere unas 15 multiplicaciones. Pero notemos que $3^{16} = 3^8 \times 3^8$ por lo que, en principio, calcular 3^8 nos llevaría 8 multiplicaciones y luego una multiplicación extra para multiplicarlo por si mismo, y ya obtendríamos el resultado buscado haciendo únicamente un total de 9 operaciones. Ahora bien, podríamos repetir el mismo procedimiento al notar que $3^8 = 3^4 \times 3^4$, y al final necesitaríamos unas 6 operaciones en total.
¿Qué está sucediendo en cada paso? Estamos dividiendo a la mitad el exponente y usando el hecho de que $k^{2n} = k^n \times k^n$, por lo que a la cantidad total de operaciones la estamos dividiendo por dos y le estamos sumando uno. Si volvemos al caso de 3^{16} y seguimos hasta que el exponente sea uno, notaremos que en total vamos a necesitar unas $\log_2(16) = 4$ operaciones:

$$3 \times 3 = 3^2 \to 3^2 \times 3^2 = 3^4 \to 3^4 \times 3^4 = 3^8 \to 3^8 \times 3^8 = 3^{16}$$

3.2. La idea y su implementación

Vamos a formalizar ahora la idea anterior. La elección del exponente 16 no fue casualidad: al ser una potencia de 2, en cada paso podíamos partir el exponente a la mitad y el único exponente impar que nos íbamos a topar era el 1. Pero, ¿qué sucede con otros números? Por ejemplo, si tuviésemos que calcular 3^{17}. Es el momento de extender la estrategia para cualquier exponente entero positivo n. Para eso, fijamos la base a y hacemos las siguientes dos observaciones:

- **Si el exponente n es par:** podemos escribir $a^n = \left(a^{n/2}\right)^2$, y para calcular a^n solo necesitamos calcular $a^{n/2}$ y elevarlo al cuadrado.
- **Si el exponente n es impar:** podemos escribir $a^n = a^{n-1} \times a$ y, como $n-1$ es par, lo redujimos al caso anterior.

Esta idea es la esencia del algoritmo conocido como **Exponenciación Binaria**, y podemos ver que en cada paso reducimos el problema a uno con un exponente de aproximadamente la mitad del tamaño. Para terminar de ilustrar como funciona, vamos a presentar el pseudocódigo de una función recursiva. `exp_bin`(base, exponente)

Algoritmo 1 Exponenciación Binaria Recursiva

Entrada: base, exponente ≥ 0 (enteros)
Salida: base$^{\text{exponente}}$

```
1: si exponente = 0
2:     retornar 1                                  ▷ Caso base
3: sino si exponente = 1
4:     retornar base                               ▷ Caso trivial
5: fin si
6: t ← exp_bin(base, ⌊exponente/2⌋)
7: si exponente mod 2 = 0                          ▷ exponente par
8:     retornar t × t
9: sino                                            ▷ exponente impar
10:    retornar base × t × t
11: fin si
```

Notamos que, en cada llamada recursiva a la función `exp_bin`, el valor del exponente se divide aproximandamente a la mitad. Pero esto no podríamos hacerlo más de $\log_2(n)$ veces. Ahora notamos que, dentro de cada llamada, realizamos un número acotado de operaciones: una o dos multiplicaciones y un chequeo de paridad del exponente, por lo que cada llamada es $O(1)$. Luego, la complejidad total del algoritmo es $O(\log(n)) \times O(1) = O(\log(n))$.

Antes de pasar a las aplicaciones de este algoritmo, nos hacemos una última pregunta: ¿no sería lógico conjeturar que este algoritmo calcula a^n en la menor cantidad posible de pasos, si consiste en partir a la mitad el exponente cada vez que podemos hacerlo? Lo maravilloso de la matemática es que cuando todo parece estar dado para ser de cierta forma, sucede lo menos pensado. Para mostrar un contraejemplo, procedemos a calcular a^{15} con exponenciación binaria (notar que los pasos se cuentan de abajo hacia arriba porque al ser una función recursiva se va llamando hasta llegar al caso base):

Paso	Cálculo	Operaciones	Resultado
4	a^1	Caso base	a
3	$(a^1)^2 = a^2$	Cuadrado: $a \cdot a$	a^2
	$a \cdot a^2$	Multiplicación	a^3
2	$(a^3)^2 = a^6$	Cuadrado: $a^3 \cdot a^3$	a^6
	$a \cdot a^6$	Multiplicación	a^7
1	$(a^7)^2 = a^{14}$	Cuadrado: $a^7 \cdot a^7$	a^{14}
	$a \cdot a^{14}$	Multiplicación	a^{15}
Total: 6 operaciones			

Y ahora vamos a calcularlo de la siguiente manera:

Paso	Cálculo	Operaciones	Resultado
1	a^1	Caso base	a
2	$(a^1)^2 = a^2$	Cuadrado: $a \cdot a$	a^2
3	$(a^2)^2 = a^4$	Cuadrado: $a^2 \cdot a^2$	a^4
	$a \cdot a^4$	Multiplicación	a^5
4	$(a^5)^2 = a^{10}$	Cuadrado: $a^5 \cdot a^5$	a^{10}
	$a^5 \cdot a^{10}$	Multiplicación	a^{15}
Total: 5 operaciones			

Buscar la menor cantidad de operaciones necesarias para calcular a^n no es otra cosa que buscar la menor cantidad de operaciones necesarias para sumar n partiendo desde el 1 y bajo ciertas reglas, dado que en potencias de igual base los exponentes se suman. Definimos una **cadena de sumas** para calcular un número entero positivo n como una sucesión de números naturales que comienza en 1 y termina en n, y que cada término se obtiene sumando dos de los elementos anteriores de la lista (no necesariamente distintos). La longitud de una cadena de sumas es el número de sumas necesarias para obtener todos sus términos, es decir uno menos que su tamaño. Decimos que una cadena de sumas para calcular n es **minimal** si no existe otra que tenga una longitud menor que también termine en n. Lo que buscamos entonces para calcular a^n en la menor cantidad posible de operaciones es calcular una cadena minimal para n. Encontrar dicha cadena resulta ser un problema NP-duro [3], lo que implica que, bajo los modelos actuales de complejidad computacional, cualquier algoritmo exacto requerirá tiempo exponencial en el peor caso.

Esto justifica el uso de métodos heurísticos en aplicaciones que demandan extremada eficiencia.

3.3. Exponenciación binaria aplicada a test de primalidad

3.3.1 Test de primalidad de Fermat

Una aplicación natural de la exponenciación binaria modular son los tests de primalidad probabilísticos. Estos algoritmos, basados en propiedades de la aritmética modular, determinan si un número es primo con probabilidad controlada de error, pero con una eficiencia muy superior a los métodos determinísticos. Su velocidad, lograda mediante exponenciación binaria, los hace indispensables para trabajar con números de cientos de dígitos en criptografía.

Como un primer paso en este recorrido, recordemos el Pequeño Teorema de Fermat que dice que si a es un entero y p un número primo tales que $(a, p) = 1$ entonces $a^{p-1} \equiv 1 \pmod{p}$. El contrarrecíproco nos dice que si $n > 1$ es entero y existe un entero a tal que $(a, n) = 1$ y $a^{n-1} \not\equiv 1 \pmod{n}$, entonces n es compuesto. El test de primalidad de Fermat se basa en esta idea. A continuación presentamos un pseudo código del algoritmo:

Algoritmo 2 Test de Primalidad de Fermat

Entrada: $n > 1$ (entero a testear), $k \geq 1$ (número de iteraciones)
Salida: `"Compuesto"` o `"Probable primo"`

si $n \leq 1$
  retornar `"Compuesto"` ▷ *Trivialmente no primo*
sino si $n = 2$ **o** $n = 3$
  retornar `"Primo"` ▷ *Casos base*
fin si
para $i \leftarrow 1$ **hasta** k
  $a \leftarrow$ entero aleatorio en $[2, n-2]$ ▷ *Selección de base*
  $d \leftarrow \text{mcd}(a, n)$
  si $d \neq 1$
    retornar `"Compuesto"` ▷ *Factor encontrado*
  fin si
  $r \leftarrow a^{n-1} \mod n$ ▷ *Exponenciación binaria modular*
  si $r \neq 1$
    retornar `"Compuesto"` ▷ *Viola el Pequeño Teorema de Fermat*
  fin si
fin para
retornar `"Probable primo"` ▷ *Con probabilidad* $\geq 1 - 2^{-k}$

Vamos a demostrar que para un n compuesto **que no sea de Carmichael**, la probabilidad de que el algoritmo retorne "Probable primo" después de k iteraciones es $\leq 2^{-k}$. La limitación clave es que el test falla para los **números de Carmichael**: enteros compuestos que satisfacen $a^{n-1} \equiv 1 \pmod{n}$ para todo a coprimo con n. Estos números:

- Son infinitos [4]
- siempre son detectados erróneamente como primos por el algoritmo

Para probar este resultado, vamos a ir repasando algunos conceptos de teoría de grupos a medida que los vayamos usando. Empezamos por definir un **grupo**, que es un par $(G, *)$ formado por un conjunto G y una operación binaria

$$* : G \times G \ \rightarrow \ G$$

que satisface las siguientes propiedades:

1. **Clausura:** Para todo $a, b \in G$, el elemento $a * b$ también pertenece a G.
2. **Asociatividad:** Para todo $a, b, c \in G$,
$$(a * b) * c \ = \ a * (b * c).$$
3. **Elemento identidad:** Existe un elemento $e \in G$ tal que para todo $a \in G$,
$$e * a = a \quad \text{y} \quad a * e = a.$$
4. **Inversos:** Para cada $a \in G$ existe un elemento $a^{-1} \in G$ tal que
$$a * a^{-1} = e \quad \text{y} \quad a^{-1} * a = e.$$

Sea $n > 1$ entero compuesto que no es de Carmichael. Como cada iteración es independiente de las anteriores, con demostrar que la probabilidad de que el algoritmo retorne "probable primo" tras una iteración es a lo sumo $\frac{1}{2}$, bastará para demostrar que luego de k iteraciones la probabilidad es a lo sumo 2^{-k}. El test toma al azar un entero del rango $2 \leq a \leq n-2$, por lo que hace falta calcular cuántos valores de a cumplen que $(a, n) = 1$ y que $a^{n-1} \equiv 1 \pmod{n}$, que son los que harían que el algoritmo retorne "probable primo". Recordemos que $\mathbb{Z}_n$ es el conjunto de todos los restos en la división por n, que un elemento a en $\mathbb{Z}_n$ representa la clase de todos los enteros que tienen resto a en la división por n y que, dados a y b elementos de dicho conjunto, definimos $a + b = c$ donde c es el resto de dividir $a + b$ por n, y definimos $ab = c$ donde c es el resto de dividir ab por n. Consideremos ahora el conjunto $\mathbb{Z}_n^* = \{\, a \in \mathbb{Z}_n : \text{mcd}(a, n) = 1\}$. Es sabidos que tal conjunto es un grupo multiplicativo (con las operaciones heredadas de $\mathbb{Z}_n$). Definimos ahora $S = \{a \in \mathbb{Z}_n^* : a^{n-1} \equiv 1 \pmod{n}\}$. Veamos que S resulta ser un subgrupo:

1. **Clausura:** Sean a, b dos elementos de S, notemos que $(ab)^{n-1} \equiv a^{n-1}b^{n-1} \equiv 1 \pmod{n}$, luego $ab \in S$.
2. **Elemento identidad:** Es claro que $1^{n-1} \equiv 1 \pmod{n}$, luego $1 \in S$.
3. **Inversos:** Sea a un elemento de S, notamos que $(a^{-1})^{n-1} \equiv (a^{n-1})^{-1} \equiv 1^{-1} \equiv 1 \pmod{n}$, luego $a^{-1} \in S$.

Por otro lado, como n no es de Carmichael, existe al menos un elemento x en $\mathbb{Z}_n^*$ tal que $x^{n-1} \not\equiv 1 \pmod{n}$, por lo que $x \notin S$ y $S \neq \mathbb{Z}_n^*$ y S resulta ser un subgrupo propio.

Recordemos ahora el Teorema de Lagrange [5], que nos dice que si G es un grupo finito y H es un subgrupo suyo, entonces

$$|G| = |H|[G : H]$$

Donde $[G : H]$ es la cantidad de conjuntos de la forma gH con $g \in G$. Pero de esto, podemos deducir que $|H|$ es un divisor de $|G|$.

Volviendo a la demostración, como $S \neq \mathbb{Z}_n^*$, tenemos que $|S| < |\mathbb{Z}_n^*|$, pero como vimos que S era subgrupo, por el Teorema de Lagrange tenemos que $|S|$ divide a $|\mathbb{Z}_n^*|$. Luego $2 \leq \frac{|\mathbb{Z}_n^*|}{|S|}$, y llegamos a que $|S| \leq \frac{|\mathbb{Z}_n^*|}{2}$.
Para terminar, sea $A = \mathbb{Z}_n^* \setminus \{1, n-1\}$ (el conjunto de donde el algoritmo toma un elemento al azar), tenemos entonces que $|A| + 2 = |\mathbb{Z}_n^*|$. Por otro lado, notemos que $1 \in S$ pero $1 \notin A$, por lo que

$$|S \cap A| \leq |S| - 1 \leq \frac{|\mathbb{Z}_n^*|}{2} - 1 = \frac{|A| + 2}{2} - 1 = \frac{|A|}{2}$$

Ahora solo resta calcular la probabilidad de que dado n compuesto no de Carmichael, el algoritmo retorne probable primo:

$$\frac{\textbf{\#Casos favorables}}{\textbf{\#Casos totales}} = \frac{|S \cap A|}{|A|} \leq \frac{|A|/2}{|A|} = \frac{1}{2}$$

como queríamos demostrar.

3.3.2 Test de primalidad de Miller-Rabin

El Test de Fermat, elegante por su simplicidad, nos muestra cómo un resultado del siglo XVII se vuelve práctico gracias a la exponenciación binaria. Su debilidad ante los números de Carmichael motiva la necesidad de tests más fuertes como el de Miller-Rabin, que veremos a continuación.

Antes de comenzar con este nuevo algoritmo, recordemos que si p es un número primo, entonces $\mathbb{Z}_p$ es un dominio de integridad. ¿Qué significa esto? Que si $a, b \in \mathbb{Z}_p$ son tales que $ab \equiv 0 \pmod{p}$, entonces o bien $a \equiv 0 \pmod{p}$ o bien $b \equiv 0 \pmod{p}$. Dado $x \in \mathbb{Z}_p$, consideramos la ecuación $x^2 \equiv 1 \pmod{p}$. O equivalentemente, $(x+1)(x-1) \equiv 0 \pmod{p}$. Como $\mathbb{Z}_p$ es un dominio de integridad, o bien $x \equiv 1 \pmod{p}$ o bien $x \equiv -1 \pmod{p}$. Nuevamente, el contrarrecíproco de esto nos dice que si $n > 1$ es un entero tal que existe $x \in \mathbb{Z}_n$, $x \not\equiv 1 \pmod{n}$ y $x \not\equiv -1 \pmod{n}$ tal que $x^2 \equiv 1 \pmod{n}$, entonces n es un número compuesto. Esta es, junto al Pequeño Teorema de Fermat, la idea clave detrás del test de primalidad de Miller-Rabin.

Antes de presentar el pseudo código de este algoritmo, vamos a explicar un poco los pasos, suponiendo que se quiere analizar la primalidad de un número entero positivo impar n. El caso de que sea par es fácil de resolver porque si no es 2 entonces es compuesto. Llamaremos **Testigo fuerte** a todo número que dado un número compuesto n, nos de como salida en el Test de Miller-Rabin "Compuesto". Los pasos del algoritmo, con algunos comentarios sobre la motivación de cada uno, son los siguientes:

1. **Descomposición de $n-1$:** Factorizamos $n-1$ como $n-1 = 2^s \cdot d$ donde d es **impar** y $s \geq 1$. Esto se logra dividiendo sucesivamente por 2 hasta obtener un número impar. Es claro que esta representación es única para cada n.

2. **Elección de la base a:** Seleccionamos aleatoriamente un entero a en el rango $2 \leq a \leq n-2$

3. **Primera verificación (Test de Fermat):** Calculamos $x_0 \equiv a^d \pmod{n}$ (en el Test de Fermat clásico calculamos $a^{n-1} \pmod{n}$, en este primero a^d para filtrar rápidamente casos más sencillos). Si resulta que:

$$x_0 \equiv \pm 1 \pmod{n},$$

entonces a **no es testigo fuerte** y n pasa la prueba para esta base.

4. **Verificaciones sucesivas (raíces cuadradas):** Si no se cumplieron las condiciones anteriores, iteramos para $r = 1, 2, \ldots, s-1$:

$$\begin{aligned} x_r &\equiv x_{r-1}^2 \pmod{n} \\ &\equiv a^{2^r \cdot d} \pmod{n} \end{aligned}$$

Verificamos en cada paso:

- Si $x_r \equiv -1 \pmod{n}$: n pasa la prueba (a no es testigo fuerte)
- Si $x_r \equiv 1 \pmod{n}$: n es **compuesto** (a es testigo fuerte, pues x_{r-1} sería raíz no trivial de 1)
- Si $x_r \not\equiv \pm 1 \pmod{n}$: continuamos iterando

5. **Conclusión final:** Si después de todas las iteraciones ($r = 1$ hasta $s-1$) no apareció -1 y además $x_{s-1} \not\equiv -1 \pmod{n}$, entonces n es compuesto y a resulta testigo fuerte, pues:

$$a^{n-1} = a^{2^s \cdot d} \not\equiv 1 \pmod{n}$$

violando el Pequeño Teorema de Fermat.

El pseudo código del algoritmo es el siguiente:

Algoritmo 3 Test de Primalidad de Miller-Rabin

Entrada: $n > 3$ (impar), $k \geq 1$ (núm. iteraciones)
Salida: `"Compuesto"` o `"Probable primo"`

Factorizar $n - 1 = 2^s \cdot d$ ▷ *d impar, $s \geq 1$*
para $i \leftarrow 1$ **hasta** k
  $a \leftarrow$ entero aleatorio en $[2, n-2]$ ▷ *Selección de base candidata*
  $x \leftarrow a^d \mod n$ ▷ *Exponenciación binaria modular*
  si $x \equiv 1$ **o** $x \equiv n-1$
    continuar ▷ *No es testigo fuerte para esta iteración*
  fin si
  $r \leftarrow 1$
  mientras $r < s$ ▷ *Verificar raíces cuadradas*
    $x \leftarrow x^2 \mod n$
    si $x \equiv n-1$
      romper ▷ *Rompe el ciclo interior*
    sino si $x \equiv 1$
      retornar `"Compuesto"` ▷ *Raíz no trivial de 1*
    fin si
    $r \leftarrow r + 1$
  fin mientras
  si $x \not\equiv n-1$
    retornar `"Compuesto"` ▷ *Base a es testigo fuerte*
  fin si
fin para
retornar `"Probable primo"` ▷ *Error $\leq 4^{-k}$*

Pero, ¿por qué funciona? La genialidad del algoritmo de Miller-Rabin radica en cómo une estas comprobaciones. Si n es primo, siempre pasará la prueba de Miller-Rabin, no importa la base que se elija, a diferencia del test anterior que fallaba con los números de Carmichael. Esto se debe a que se cumplirá el Pequeño Teorema de Fermat y la única raíz cuadrada de 1 será 1 o -1.

Por otro lado, si n es compuesto, es muy probable que no pase la prueba. Para un número compuesto n, se ha demostrado que la probabilidad de que un número a elegido al azar del rango $2 \leq a \leq n-2$ sea un testigo fuerte es de al menos $\frac{3}{4}$, por lo que la probabilidad de fallo del test tras una iteración es a lo sumo $\frac{1}{4}$ [6].

Al repetir la prueba con diferentes bases aleatorias k veces, la probabilidad de que un número compuesto n sea erróneamente clasificado como "probable primo" disminuye exponencialmente. Si se realizan k pruebas, la probabilidad de un falso positivo es menor a 4^{-k}. Para tomar noción de la eficiencia del algoritmo, con solo 10 bases la probabilidad de error es menor a una en un millón.

La eficiencia del paso 3 (cálculo de $a^d \mod n$) depende críticamente de la exponenciación binaria modular, reduciendo su complejidad de $O(d)$ a $O(\log(d))$. Sin esta optimización, el test sería inviable para números de miles de cifras.

3.4. Pero... ¿sólo los números se pueden multiplicar?

Si repasamos atentamente las ideas de la exponenciación binaria, en realidad lo único que usamos de la multiplicación es la propiedad asociativa. Por lo que esta idea se podría aplicar a cualquier operación asociativa, y, en particular, al producto de matrices cuadradas.

Recordemos que el producto de dos matrices cuadradas $A = (a_{ij})$ y $B = (b_{jk})$, ambas de orden n, es una nueva matriz $C = (c_{ik})$ donde cada elemento se define como:

$$c_{ik} = \sum_{j=1}^{n} a_{ij} b_{jk} \quad \text{para} \quad 1 \leq i, k \leq n$$

Esto equivale a realizar n^3 multiplicaciones y $n^2(n-1)$ sumas en el método estándar. Como la exponenciación binaria nos permite calcular potencias k-ésimas en $O(\log(k))$, y el producto entre dos matrices cuadradas en $\mathbb{R}^{n\times n}$ es $O(n^3)$, podemos afirmar que la complejidad computacional de elevar a la k-ésima potencia una matriz cuadrada $A \in \mathbb{R}^{n\times n}$ es $O(n^3 \log(k))$.

A continuación, presentaremos dos ejemplos de problemas que pueden resolverse de manera más eficiente si utilizamos el método anterior.

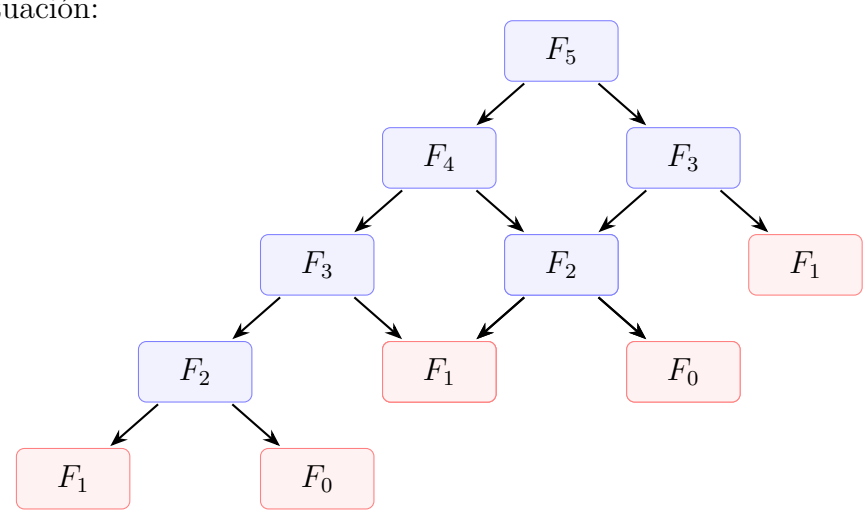
3.4.1 Recurrencias

Una **recurrencia lineal homogénea de orden k con coeficientes constantes** es una ecuación que define una secuencia $\{a_n\}_{n\geq 0}$ en un cuerpo $\mathbb{K}$ (generalmente $\mathbb{R}$ o $\mathbb{C}$) mediante:

$$a_n = c_1 a_{n-1} + c_2 a_{n-2} + \cdots + c_k a_{n-k}, \quad \forall n \geq k$$

En este tipo de recurrencias, decimos que k es el **orden de la recurrencia**, que $c_1, c_2, \ldots, c_k \in \mathbb{K}$ constantes con $c_k \neq 0$ son los coeficientes y que los valores $a_0, a_1, \ldots, a_{k-1} \in \mathbb{K}$ son dados y llamados **condiciones iniciales**.

El problema planteado es cómo calcular a_m para un m dado. Resolvamos este problema en un ejemplo, para entender la complejidad que puede llegar a tener un cálculo sin ninguna optimización. Consideremos la sucesión de Fibonacci, dada por $F_n = F_{n-1} + F_{n-2} \forall\, n \geq 2$ y $F_0 = 0$, $F_1 = 1$. Vamos a calcular F_5 con un diagrama para ilustrar la situación:



Observando el diagrama de cálculo, notamos que hacemos unas **15 llamadas recursivas**, de las cuales 9 resultan ser redundantes por calcular varias veces los mismos términos. En general, este método para calcular el n-ésimo número de Fibonacci tiene complejidad $O(2^n)$. Por lo que para calcular F_{40} ya se tardarían unos dieciocho minutos, y para F_{60} casi treinta y seis años: inviable. Finalmente, si la recurrencia es de orden k, entonces la complejidad de calcular el n-ésimo término es $O(k^n)$

Ahora bien, volviendo a Fibonacci, la primera optimización significativa podría ser ir almacenando los valores calculados. Así, cada F_i sería calculado una única vez, y calcular cada término nuevo sería únicamente sumar los dos términos anteriores. Por lo que calcular el n-ésimo término tendría una complejidad $O(n)$, ya que cada cálculo nuevo necesitaría únicamente sumar los dos términos previos. Mucho más efectivo que el método anterior, podríamos calcular F_{10^8} en menos de un segundo. Pero nuevamente el algoritmo volvería a ser muy lento para valores de n más grandes. En general, si la recurrencia es de orden k, calcular el n-ésimo término tiene complejidad $O(nk)$.

La última optimización que vamos a hacer surge de notar que podemos pensar el cálculo del siguiente Número de Fibonacci como un producto de matrices si usamos la siguiente identidad matricial:

$$\begin{pmatrix} F_{n+2} \\ F_{n+1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} F_{n+1} \\ F_n \end{pmatrix}.$$

Pero aplicando reiteradas veces esta identidad, llegamos a algo que tiene de fondo una potencia tremenda:

$$\begin{pmatrix} F_{n+1} \\ F_n \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n \begin{pmatrix} F_1 \\ F_0 \end{pmatrix}$$

Apareció la estrella de este capítulo, la potencia. Podemos aplicar entonces la exponenciación binaria. Como en este caso $k = 2$ está fijo, resolver el problema del n-ésimo Fibonacci tiene complejidad $O(\log(n))$. Así que podemos trabajar con valores de n de miles de cifras sin problema.

Formalizando un poco lo anterior, dada una recurrencia lineal homogénea $a_n = c_1 a_{n-1} + c_2 a_{n-2} + \cdots + c_k a_{n-k}, \quad \forall n \geq k$, definimos su **Matriz de Transformación** $M \in \mathbb{K}^{k \times k}$ [7] como:

$$M = \begin{pmatrix} c_1 & c_2 & \cdots & c_k \\ 1 & 0 & \cdots & 0 \\ 0 & \ddots & \ddots & \vdots \\ 0 & \cdots & 1 & 0 \end{pmatrix},$$

y, para cada $n \geq k - 1$, definimos el **Vector de Estado** $V_n \in \mathbb{K}^{k \times 1}$ como:

$$V_n = \begin{pmatrix} a_n \\ a_{n-1} \\ \vdots \\ a_{n-k+1} \end{pmatrix}.$$

Además V_k está dado por las condiciones iniciales. Es fácil notar que, por como fueron definidas, se cumple que $V_n = MV_{n-1} \ \forall \ n \geq k$. Si aplicamos varias veces esta igualdad, concluímos que $V_n = M^{n-(k-1)} V_{k-1}$. Luego podemos obtener V_n tras calcular con exponenciación binaria M^{n-k+1} y multiplicar el resultado por V_{k-1}. Realmente bellísimo.

3.4.2 Contando caminos en grafos

Un grafo es un objeto matemático muy interesante por sus propiedades matemáticas. Pero también muy versátil para modelar de manera abstracta distintas redes de relaciones en sistemas complejos: desde circuitos eléctricos o rutas entre ciudades hasta dinámicas sociales, preservando propiedades estructurales clave. Una primera aproximación informal a la definición de **Grafo Dirigido** es "un conjunto de puntos y flechas que unen algunos de esos puntos". A los puntos los llamaremos **nodos** y a las flechas que apuntan de un nodo a otro **aristas dirigidas**. Una definición un poco más rigurosa nos dice que un grafo dirigido es un par $G = (V, E)$ donde V es un conjunto no vacío de vértices y $E \subseteq V \times V$ es un conjunto de pares ordenados llamados aristas. Si $(u, v) \in E$ significa que hay una arista de u a v. Es importante notar que $(u, v) \neq (v, u)$. Un camino de longitud k desde un vértice u hasta un vértice v es una secuencia finita de vértices

$$p = (v_0, v_1, \ldots, v_k)$$

tal que para cada $i \in \{0, 1, \ldots, k-1\}$, $(v_i, v_{i+1}) \in E$. El problema que planteamos ahora es calcular cuantos caminos de longitud exáctamente k hay entre cada par posible de vértices del grafo.

Un primer abordaje a este problema con una estrategia intuitiva es resolver el problema de manera recursiva en cada nodo. Para esto, podemos definir una función `contar_caminos(origen, destino, longitud)` que funcione de la siguiente manera:

1. **Caso Base ($longitud = 0$):**

$$\texttt{contar\_caminos}(u, v, 0) = \begin{cases} 1 & \text{si } u = v \\ 0 & \text{si } u \neq v \end{cases}$$

   (Solo existe el camino trivial de longitud 0 si estamos en el mismo nodo)

2. **Paso Recursivo ($longitud > 0$):**

$$\texttt{contar\_caminos}(u, v, k) = \sum_{\substack{w \\ (u,w)\in E}} \texttt{contar\_caminos}(w, v, k-1)$$

   y obtenemos la respuesta sumando mientras w va recorriendo todos los vecinos de u (nodos conectados por una arista saliente).

Y para calcular la solución completa a nuestro problema ejecutaríamos esta función para cada posible par de nodos (u, v) en el grafo.

Este enfoque es análogo a la primera solución recursiva que vimos para los números de Fibonacci y, al igual que esa solución, a pesar de ser intuitiva y fácil de comprender, es extremadamente ineficiente. Si en nuestro grafo el grado de salida máximo[3] de cualquier nodo es d, el número de operaciones para calcular todos los caminos de longitud k que salen desde un vértice seleccionado u es $O(d^k)$.

Para tomar dimensión de la ineficiencia de este método, podemos tomar un grafo completo (para cada par de nodos u, v existe una arista entre ambos en cada dirección) con $|V| = 10$ nodos ($d_{\max} = 9$) y longitud $k = 20$:
$9^{20} \approx 1.21 \times 10^{19}$ llamadas recursivas $\implies$ más de $38,000$ años en hardware moderno.

Es claro que, nuevamente, parte de la ineficiencia radica en la enorme cantidad de veces que resolvemos cada subproblema. Por ejemplo, si fijamos dos nodos v y w, `contar_caminos`(w, v, k-1) será calculado tantas veces como caminos de un paso lleguen a w. En este caso, y recordándonos nuevamente lo bello de la matemática y como a veces distintas ramas que pueden parecer alejadas terminan cruzándose, surgen herramientas del álgebra lineal para ayudarnos a "almacenar" los resultados parciales y no calcular varias veces lo mismo.

Dado un grafo de n nodos, definimos su **Matriz de Adyacencia** $A \in \mathbb{Z}^{n\times n}$ dada por

$$A_{ij} = \#\{\text{aristas desde i hacia j}\} \quad \forall\, i, j \in V$$

Vamos a demostrar ahora un resultado sobre las matrices de adyacencia que nos va a servir para nuestro problema. Supongamos que tenemos un grafo dirigido $G = (V, E)$ y $|V| = n$, y sea $A \in \mathbb{Z}^{n\times n}$ su matriz de adyacencia. Entonces, para todo entero positivo k, se cumple que $A^k_{ij} = \#\{\text{caminos de longitud k desde i hacia j}\} \quad \forall\, i, j \in V$ [8]. Para esto, vamos a proceder por inducción:

1. **Caso Base ($k = 1$):** tenemos que $A^1 = A$ y, por como está definida la matriz de adyacencia, vale que $A_{ij} = \#\{\text{aristas desde i hacia j}\} \quad \forall\, i, j \in V$, y las aristas son exáctamente los caminos de longitud uno.

[3] El **grado de salida** de un nodo u, notado $\deg^+(u)$, es el número de aristas que salen de u. El **grado de salida máximo** es $d = \max_{u\in V} \deg^+(u)$. El análisis $O(d^k)$ considera el peor caso, donde cada nodo tiene exactamente d vecinos salientes.

2. **Paso inductivo:** Supongamos que nuestro resultado es cierto para cierto $k \geq 1$ entero positivo. Tenemos entonces $A^{k+1} = A^k A$ y, por definición de producto de matrices, resulta que:

$$A_{ij}^{k+1} = \sum_{m=1}^{n} A_{im}^{k} \cdot A_{mj}$$

Pero notemos que, por definición de matriz de adyacencia, A_{mj} cuenta la cantidad de caminos de longitud uno desde m hasta j, y por la hipótesis inductiva A_{im}^k cuenta la cantidad de caminos de longitud k de i a m. Entonces cada término de la pinta $A_{im}^k \cdot A_{mj}$ cuenta

$$\begin{aligned}&(\text{caminos de } i \text{ a } m \text{ de longitud } k) \ \times \ (\text{caminos de } m \text{ a } j \text{ de longitud } 1)\\ &= \ \text{caminos de } i \text{ a } j \text{ de longitud } k+1 \text{ que pasan por } m.\end{aligned}$$

Ahora bien, como todo camino de longitud $k+1$ desde i hasta j se descompone en un camino de longitud k desde i hasta algún m seguido de un camino de longitud 1 de ese m hasta j, A_{ij}^{k+1} resulta contar exáctamente la cantidad de caminos de longitud $k+1$ desde i hasta j como queríamos demostrar.

Después de incorporar este resultado a nuestro repertorio matemático, es claro que resolver el problema propuesto se puede sintetizar en calcular A^k. Ya que en cada entrada de esta matriz tendríamos la respuesta a la pregunta *¿cuántos caminos de longitud k hay desde i hasta j?*. Por lo demostrado previamente, sabemos que elevar una matriz de orden n a su k-ésima potencia tiene una complejidad computacional $O(n^3 \log(k))$ por lo que, retomando el ejemplo del grafo completo con $|V| = 10$ y longitud $k = 20$, resulta que necesitaríamos del orden de $10^3 \times \log_2(20) \approx 4300$ operaciones $\implies$ 0.0001 segundos. La representación matricial del grafo y la exponenciación binaria tendieron un puente entre lo infinito y lo instantáneo: la potencia de las buenas ideas en su máximo esplendor.

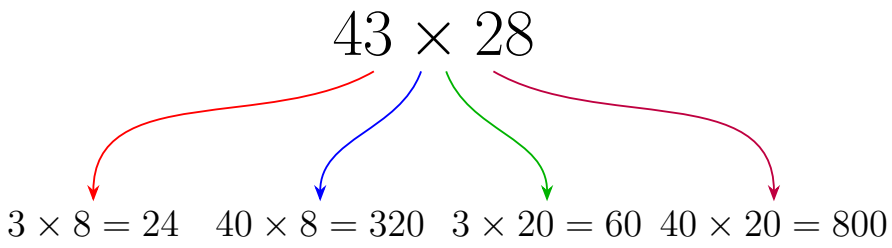
4. Karatsuba

> *"El gran obstáculo para el descubrimiento no es la ignorancia, es la ilusión del conocimiento."*
>
> Daniel J. Boorstin, 1914–2004.

Nota importante: como mencionamos en el capítulo de orden de complejidad, cuando hagamos referencia a algoritmos para manipular números arbitrariamente grandes, una operación elemental es la manipulación de un solo dígito (sin importar cual sea la base b elegida).

4.1. El joven que desafió la historia de la multiplicación

Hasta mediados del siglo XX, la comunidad matemática consideraba el método de multiplicación escolar, ilustrado en el siguiente ejemplo:



$$24 + 320 + 60 + 800 = 1204$$

con su complejidad de $O(n^2)$ para multiplicar dos números de n dígitos no solo como el estándar, sino como el mejor posible. Tal era el consenso al respecto que en 1960, en un seminario en la Universidad Estatal de Moscú, Andrey Kolmogorov, uno de los grandes matemáticos del siglo XX, conjeturó que ningún algoritmo podría multiplicar números de n dígitos en menos de $O(n^2)$ operaciones, considerando el método escolar como asintóticamente óptimo. Lamentablemente para la conjetura de Kolmogorov, pero afortunadamente para las matemáticas, se encontraba entre la audiencia un joven de 23 años: Anatoly Karatsuba, quien por ese entonces era estudiante de posgrado de Kolmogorov.

Una semana. Una semana le llevó a Karatsuba rebatir la conjetura de Kolmogorov. En tan solo una semana desarrolló un algoritmo basado en la estrategia de "divide y vencerás" que rompía con la barrera cuadrática. El algoritmo de Karatsuba multiplicaba dos enteros de tamaño n con una complejidad $O\left(n^{\log_2(3)}\right)$ [9], que tiene un exponente aproximado de 1.585. Esto no solo fue revolucionario por la mejora sustancial al tiempo de ejecución, sino que también abrió un campo de investigación completamente nuevo en la búsqueda de algoritmos cada vez más eficientes para multiplicar.

4.2. La idea de Karatsuba y cómo implementarla

La estrategia del "divide y vencerás" consiste en tres pasos: primero dividir el problema en subproblemas más pequeños del mismo tipo, segundo resolver recursivamente cada subproblema (si son muy pequeños, resolverlos directamente) y, por último, combinar las soluciones de los subproblemas para obtener la solución global. Vamos a aplicar esta idea al problema de multiplicar dos números de tamaño n. Por simplicidad en los cálculos, asumamos que n es una potencia de 2. Supongamos que los números que queremos multiplicar son x e y. Es claro que podemos reescribirlos a cada uno como $x = a \cdot 10^{n/2} + b$ e $y = c \cdot 10^{n/2} + d$, siendo a y c enteros positivos de $n/2$ cifras y b y d enteros no negativos de a lo sumo $n/2$ cifras. Operando llegamos a que:

$$xy = ac \cdot 10^n + (ad + bc) \cdot 10^{n/2} + bd$$

Para calcular el producto original, necesitamos resolver cuatro multiplicaciones de tamaño $n/2$: ac, ad, bc y bd. Un algoritmo recursivo basado en esta descomposición realizaría 4 llamadas a sí mismo con problemas de la mitad de tamaño, lo que nos lleva a una complejidad final de $O(n^2)$. El "divide y vencerás" no es suficiente para mejorar el algoritmo.

Pero donde todos vieron un callejón sin salida, Karatsuba pudo ver más allá y con una simple pero ingeniosa identidad algebraica logró reducir la cantidad de multiplicaciones necesarias en cada paso recursivo. Si definimos $\alpha_1 = ac$, $\alpha_2 = bd$ y $\alpha_3 = (a + b)(c + d)$,

cada uno de ellos es un producto de dos números del orden de $n/2$ dígitos, y si notamos que $\alpha_3 - \alpha_2 - \alpha_1 = ad + bc$ resulta que

$$xy = \alpha_1 \cdot 10^n + (\alpha_3 - \alpha_2 - \alpha_1) \cdot 10^{n/2} + \alpha_2$$

Y ahora para calcular el producto original necesitamos calcular solamente tres multiplicaciones de tamaño $n/2$. Esta mejora en cada paso recursivo resulta en el algoritmo de Karatsuba que tiene una complejidad computacional de $O\left(n^{\log_2(3)}\right)$. El caso base de la recursividad es cuando $n = 1$ y simplemente se retorna el producto de dos números de un dígito cada uno con el método tradicional. Vale la pena notar que todo el proceso anterior puede ser reemplazado por cualquier base de numeración B, no solamente 10. En efecto, basta reemplazar $x = a \cdot B^{n/2} + b$ e $y = c \cdot B^{n/2} + d$ y los pasos funcionan exactamente igual, con la misma complejidad computacional.

A continuación, presentaremos un pseudocódigo de la implementación del algoritmo. Pero antes es necesario abordar dos detalles prácticos: ¿qué sucede si el número de dígitos no es par? ¿Y si los números a multiplicar no tienen la misma longitud? La solución en ambos casos es la misma: completamos el número más corto con ceros a la izquierda hasta que ambos tengan la misma longitud, digamos n. Si n resulta ser impar, añadimos un cero más a la izquierda y la nueva longitud será par. Notemos que la operación de igualar los tamaños, en caso de ser necesaria, lo será únicamente en la primera llamada al algoritmo, y por otro lado, en cada llamado se añade a lo sumo un dígigo a cada número, por lo que no afecta la complejidad total del algoritmo. Ahora podemos aplicar el algoritmo correctamente. Sin embargo, por simplicidad de lectura, el pseudocódigo que presentaremos asumirá que los números de entrada ya han sido procesados como se describió, es decir, que tienen la misma longitud n y que n es par.

Algoritmo 4 Multiplicación de Karatsuba

Entrada: x, y: enteros no negativos con igual longitud n (par)
Salida: $x \times y$

1: **si** $n \leq 2$ ▷ *Umbral para multiplicación escolar*
2: **retornar** $x \times y$
3: **fin si**
4: $m \leftarrow n/2$ ▷ *Mitad de dígitos (entero por ser n par)*
5: $B \leftarrow 10^m$ ▷ *Base posicional*
6: $a \leftarrow \lfloor x/B \rfloor$ ▷ *Mitad superior de x*
7: $b \leftarrow x \mod B$ ▷ *Mitad inferior de x*
8: $c \leftarrow \lfloor y/B \rfloor$ ▷ *Mitad superior de y*
9: $d \leftarrow y \mod B$ ▷ *Mitad inferior de y*
10: $\alpha_1 \leftarrow \textsc{Karatsuba}(a, c)$ ▷ ac
11: $\alpha_2 \leftarrow \textsc{Karatsuba}(b, d)$ ▷ bd
12: $\alpha_3 \leftarrow \textsc{Karatsuba}(a + b, c + d)$ ▷ $(a + b)(c + d)$
13: $z \leftarrow \alpha_3 - \alpha_1 - \alpha_2$ ▷ $ad + bc$
14: **retornar** $\alpha_1 \times B^2 + z \times B + \alpha_2$ ▷ *Ensamblaje final*

4.3. ¿Por qué funciona en $O\left(n^{\log_2(3)}\right)$?

Que el algoritmo de Karatsuba es sustancialmente más eficiente que el algoritmo que se usaba hasta ese entonces es claro, porque descompone cada producto de tamaño n en tres productos de tamaño $n/2$. Pero, ¿cómo podemos demostrar que es $O\left(n^{\log_2(3)}\right)$?

Empecemos por llamar $T(n)$ a la cantidad de operaciones elementales que lleva multiplicar dos números de tamaño n con el algoritmo de Karatsuba. Denotamos también por $f(n)$ la cantidad de operaciones elementales que demandan las tareas no recursivas en cada llamada: dividir los números, sumar y restar enteros de hasta n dígitos y los desplazamientos (multiplicar por 10^n y $10^{n/2}$, que no es otra cosa que añadir ceros), que dado el tamaño de los números en los que aplicamos esas operaciones, podemos afirmar que el costo es lineal. Es decir que $f(n)$ es $O(n)$. Notemos que en cada llamada recursiva resolvemos tres subproblemas de tamaño $n/2$ y tenemos además el costo no recursivo extra de $f(n)$, por lo que el tiempo de ejecución satisface la siguiente relación de recurrencia:

$$T(n) = 3T(n/2) + f(n)$$

Pero la pregunta que todos nos hacemos es ¿cómo se resuelve una recurrencia como esta en análisis de complejidad? O más aún, ¿qué significa resolverla? La idea es buscar la complejidad que tiene que tener $T(n)$ para cumplir esa recurrencia. Para eso vamos a presentar una herramienta fundamental en el análisis de complejidad de algoritmos, el Teorema Maestro [10], que permite resolver recurrencias de la forma:

$$T(n) = a \cdot T(n/b) + f(n)$$

donde a representa la cantidad de subproblemas recursivos en los que descomponemos nuestro problema, n/b es el tamaño de cada subproblema (se asume que todos los subproblemas tienen el mismo tamaño), y por último $f(n)$ representa el costo de dividir el problema original y de combinar los resultados obtenidos al resolver cada subproblema. Además, el teorema tiene como hipótesis que $f(n)$ sea $O\left(n^d\right)$ para cierto $d \geq 0$. La complejidad computacional del algoritmo que concluye en teorema se define según tres casos, dependiendo si $d < \log_b(a)$, $d = \log_b(a)$ o $d > \log_b(a)$. En el caso en que $d < \log_b(a)$, la solución es que $T(n)$ sea $O\left(n^{\log_b(a)}\right)$.

Si aplicamos el Teorema Maestro a nuestra recurrencia en particular, tenemos que $a = 3$, $b = 2$ y $d = 1$, y como $1 < \log_2(3) \approx 1.585$, por lo que $T(n)$ resulta ser $O\left(n^{\log_2(3)}\right)$ como queríamos demostrar.

5. La Transformada Rápida de Fourier.

> *"El camino más corto entre dos verdades en el dominio real pasa a través del dominio complejo."*
>
> Daniel J. Boorstin, 1865–1963.

5.1. Funciones Generatrices

Gran parte de la combinatoria se dedica al desarrollo de herramientas para el conteo. Algunas de esas herramientas provienen de ramas de la matemática que nuestra intuición,

a priori, nos diría que no tienen mucha relación con la combinatoria. En esta sección vamos a presentar una herramienta de conteo que, a mí parecer, es de las más increíbles jamás desarrolladas: las funciones generatrices [11].

El problema que nos va a acompañar a lo largo de esta sección es el siguiente: se tienen algunas bananas y algunas manzanas, todas con pesos entero entre 1 y k. Ahora la conisgna es contar: para cada w entero entre 2 y $2k$, la cantidad de maneras de elegir una banana y una manzana de manera que su peso combinado sea exactamente w. Este tipo de problemas de conteo, aparentemente complejos, pueden resolverse de forma elegante usando la herramienta que vamos a presentar en este capítulo.

Antes de hablar de funciones generatrices, hace falta definir otro concepto clave: las **series formales de potencias**. Una serie formal de potencias es una expresión de la forma

$$a_0 + a_1 x + a_2 x^2 + \cdots = \sum_{n=0}^{\infty} a_n x^n$$

Donde la sucesión $(a_n)_{n\geq 0}$ es llamada la sucesión de coeficientes y toma valores en un cuerpo $\mathbb{K}$, aunque para nuestros propósitos en combinatoria, estos coeficientes serán casi siempre enteros que representarán cantidades. A pesar del parecido estético con las series de potencias trabajadas en el contexto del análisis matemático, las series formales de potencias son un objeto algebraico, no una función numérica. Por lo que cualquier tipo de análisis sobre la convergencia para distintos valores de x carece de utilidad. La variable x no es un número al que le asignaremos valores sino una suerte de indicador de posiciones que vamos a usar, por lo que el exponente n en x^n señala la posición del coeficiente a_n. Así, por ejemplo, aunque la serie formal de potencias $\sum_{n=0}^{\infty} n! x^n$ asociada a la sucesión $a_n = n!$ no genera ningún tipo de interés en términos analíticos, porque su radio de convergencia es 0, en el contexto de las series de potencias formales puede tener un rol importante.

Además, hace falta formalizar el álgebra de las series formales de potencia. Así que vamos a hablar sobre algunas definiciones de cómo operan entre sí, dejando algunas otras por fuera por exceder los propósitos de este trabajo, como son la derivada formal o la existencia del inverso multiplicativo para ciertos tipos de series. Dadas $\sum_{n=0}^{\infty} a_n x^n$ y $\sum_{n=0}^{\infty} b_n x^n$ dos series formales, definimos:

- La igualdad $\sum_{n=0}^{\infty} a_n x^n = \sum_{n=0}^{\infty} b_n x^n$, como la igualdad coeficiente a coeficiente, es decir $a_n = b_n \ \forall n \geq 0$
- La suma $\sum_{n=0}^{\infty} a_n x^n + \sum_{n=0}^{\infty} b_n x^n = \sum_{n=0}^{\infty} c_n x^n$ donde $c_n = a_n + b_n$, $\forall n \geq 0$
- El producto $\left(\sum_{n=0}^{\infty} a_n x^n\right) \cdot \left(\sum_{n=0}^{\infty} b_n x^n\right) = \sum_{n=0}^{\infty} c_n x^n$ donde
  $c_n = a_0 b_n + a_1 b_{n-1} + \cdots + a_n b_0 = \sum_{k=0}^{n} a_k b_{n-k}$, $\forall n \geq 0$

Cabe destacar que un caso importante de las series formales de potencias son los **polinomios**,

que son series formales donde solo finitos coeficientes no son nulos, en ese caso la igualdad, la suma y el producto se definen de la manera usual con la que se suele trabajar.

Ya habiendo definido este objeto algebraico y algunas de sus operaciones, vamos a hablar sobre las **funciones generatrices**. Las funciones generatrices no introducen un nuevo objeto, sino más bien podríamos decir que es un cambio conceptual sobre la percepción de las series formales de potencias a las que ahora vamos a asignarles un significado combinatorio a sus coeficientes, donde cada a_n representará ahora la respuesta a un problema de conteo de tamaño n, y a sus operaciones, donde ahora la suma y producto tendrán significado de operaciones combinatorias.

Tras definir los conceptos necesarios, vamos a resolver el problema que planteamos al principio, para ilustrar el poder de las funciones generatrices viéndolas en acción:
Lo primero que vamos a hacer es definir $M(x) = a_0 + a_1x + a_2x^2 + \cdots + a_kx^k$ como la función generatriz que cumple que cada coeficiente a_i es la cantidad de manzanas que tienen un peso de i, y $B(x) = b_0 + b_1x + b_2x^2 + \cdots + b_kx^k$ como la función generatriz que cumple que cada coeficiente b_i es la cantidad de bananas que tienen un peso de i (es claro que $a_0 = b_0 = 0$). ¿De qué maneras podríamos formar un peso combinado de exactamente 4? Podríamos tomar una manzana de peso 1 y una banana de peso 3, una manzana de peso 2 y una banana de peso 2, o una manzana de peso 3 y una banana de peso 1. Ahora nos preguntamos, ¿de cuántas maneras distintas podemos tomar una manzana de peso 1 y una banana de peso 3? Bueno, resulta que esto no es otra cosa que a_1b_3. De igual modo, a_2b_2 nos dice de cuántas maneras podemos hacerlo con una manzana de peso 2 y una banana de peso 2, y a_3b_1 nos dice lo mismo con manzanas de peso 3 y bananas de peso 1. Entonces, el resultado total para un peso combinado de exactamente 4 es $a_1b_3 + a_2b_2 + a_3b_1$. Consideremos ahora el producto

$$M(x) \cdot B(x) = \left(\sum_{n=0}^{k} a_n x^n\right) \cdot \left(\sum_{n=0}^{k} b_n x^n\right) = \sum_{n=0}^{2k} c_n x^n$$

Y recordando que $c_n = \sum_{j=0}^{n} a_j b_{n-j}$, si tomamos el caso $n = 4$ nos queda $c_4 = a_0b_4 + a_1b_3 + a_2b_2 + a_3b_1 + a_4b_0 = a_1b_3 + a_2b_2 + a_3b_1$, que es la respuesta que habíamos calculado hoy.

Resulta que multiplicar ambas funciones generatrices, en términos combinatorios es el equivalente a separar en dos conjuntos independientes (en este caso manzanas y bananas) y contar de cuántas maneras se puede tomar un elemento de cada uno de manera que la suma de los pesos de un número determinado. De este modo, si analizamos la expresión $c_n = \sum_{j=0}^{n} a_j b_{n-j}$ resulta que cada sumando es la cantidad de maneras de obtener n como peso combinado con una manzana de peso j y una banana de peso $n-j$. Así, si llamamos $C(x) = \sum_{n=0}^{2k} c_n x^n$, el coeficiente que acompañe a x^n para cada valor de n entre 2 y $2k$ será la respuesta a la pregunta "¿de cuántas maneras se puede formar un peso combinado de n con una manzana y una banana?"

Por último, un simple análisis de complejidad computacional nos dice que multiplicar ambos polinomios es $O(k^2)$, porque cada coeficiente de $M(x)$ se multiplica con cada coeficiente de $B(x)$. Ahora, la pregunta que nos surge es ¿podremos multiplicar polinomios de manera más rápida? Y de eso vamos a hablar en la siguiente sección.

5.2. Multiplicando polinomios

5.2.1 Una aproximación al problema con una estrategia conocida

Las funciones generatrices son una herramienta muy potente. Pero al final de la sección pasada nos topamos con que el producto de polinomios, tal y como lo aprendimos en el colegio, es $O(n^2)$. Al igual que con el producto de números grandes, es fácil conjeturar al principio que esa complejidad es la óptima. Si hay que multiplicar todos los coeficientes entre sí, ¿cómo podríamos hacerlo con una complejidad asintóticamente mejor? El lector atento tal vez recordará la idea de Karatsuba. Hay un parecido cuanto menos estético en las cosas que queremos multiplicar. Si dijimos que su algoritmo valía para cualquier base que eligiésemos, no únicamente para base 10, ¿no podríamos modificar el algoritmo para que la base se convierta en una variable?

Sean $A(x) = \sum_{k=0}^{n-1} a_k x^k$ y $B(x) = \sum_{k=0}^{n-1} b_k x^k$ ambos polinomios de grado $n-1$, que por simplicidad suponemos que n es una potencia de dos nuevamente. Si llamamos $M(x) = \sum_{k=0}^{n/2-1} a_k x^k$ y $N(x) = \sum_{k=0}^{n/2-1} a_{n/2+k} x^k$, notamos que

$$A(x) = \underbrace{a_0 + a_1 x + a_2 x^2 + \cdots + a_{n/2-1} x^{n/2-1}}_{M(x)} + \underbrace{a_{n/2} x^{n/2} + a_{n/2+1} x^{n/2+1} + \cdots + a_{n-1} x^{n-1}}_{x^{n/2} N(x)}$$

De manera análoga, definimos $Q(x)$ y $R(x)$ tales que $B(x) = Q(x) + x^{n/2} R(x)$. Ahora bien, si llamamos $P_1(x) = M(x) \cdot Q(x)$, $P_2(x) = N(x) \cdot R(x)$ y $P_3(x) = (M(x) + N(x)) \cdot (Q(x) + R(x))$ entonces llegamos a que

$$A(x) \cdot B(x) = P_1(x) + x^{n/2} \left((P_3(x) - P_2(x) - P_1(x)) + x^n P_2(x) \right)$$

. Y nuevamente, descompusimos el problema de un producto de dos polinomios de grado $n-1$ en tres productos de grado $n/2-1$, más el costo de unir los tres resultados con algunas operaciones extra. Un análisis similar al ya desarrollado en el capítulo anterior nos perimitiría concluir que este algoritmo resulta ser $O\left(n^{\log_2(3)}\right)$, por lo que una abstracción del algoritmo de Karatsuba nos permite multiplicar polinomios de manera más eficiente que cuadrática.

5.2.2 El nuevo enfoque que cambió todo

En este momento, cuando uno se ve tentado por la idea de buscar algún otro artilugio algebraico reacomodando términos para poder reducir la cantidad de operaciones, resulta que el próximo salto cualitativo en la eficiencia de algoritmos de producto se da con un cambio total del enfoque. Sabemos que dos polinomios $A(x)$ y $B(x)$ son iguales si y solo si son iguales coeficiente a coeficiente. Por simplicidad, supongamos que $A(x)$, $B(x) \in \mathbb{C}[x]$ pero que sus coeficientes son reales. A cada polinomio de grado $n-1$ se le puede asignar de manera biyectiva un vector en $\mathbb{C}^n$ mediante $A(x) = \sum_{k=0}^{n-1} a_k x^k \longrightarrow (a_0, a_1, \ldots, a_{n-1})$, que llamaremos **representación por coeficientes**. Luego, dos polinomios son iguales si y solo si tienen la misma representación por coeficientes (en particular, si dos vectores tienen

distinto tamaño claramente son distintos). Esta concepción de la igualdad de polinomios es con la que uno suele estar más familiarizado y, además, esta asignación es simple de deshacer. Es decir, a partir de ese vector es inmediato reconstruir el polinomio original. Esta correspondencia biyectiva es tan natural que, de ahora en adelante, identificaremos un polinomio con su vector de coeficientes. Este isomorfismo nos permite, con un leve pero útil abuso de notación, hablar de operaciones sobre A, donde A puede ser considerado tanto el polinomio $A(x)$ como su vector de coeficientes, según el contexto lo requiera.

Antes de seguir, hay una observación clave que realizar. Si $A(x)$ y $B(x)$ son polinomios de grado a lo sumo $n-1$, su producto tendrá grado a lo sumo $2n-2$. Para determinar este nuevo polinomio, necesitaremos su valor en al menos $2n-1$ puntos, por lo que antes de empezar debemos ampliar las representaciones de A y B. Lo que se suele hacer es rellenear con ceros los vectores de coeficientes de los dos polinomios hasta que tengan una tamaño de N, donde N es la menor potencia de dos, mayor o igual a $2n-1$, dado que los algoritmos que vamos a trabajar son más eficientes con tamaños potencia de dos. A partir de ahora, este N será nuestro tamaño de trabajo: todos los vectores de coeficientes se rellenarán con ceros hasta tener longitud N y todas las evaluaciones se realizarán en N puntos.

Como siguiente paso, recordemos que todo polinomio de grado a lo sumo $N-1$ queda unívocamente determinado por su evaluación en N puntos distintos. Sean $x_0, x_1, \cdots, x_{N-1}$ números complejos distintos. Supongamos que $A(x_i) = B(x_i)$ para todo i, y que son ambos polinomios de grado a lo sumo $N-1$. Luego $P(x_i) = (A-B)(x_i) = 0$ para todo i, por lo que tenemos un polinomio de grado a lo sumo $N-1$ con N raíces distintas, lo cuál solo es posible si $(A-B)(x)$ es el polinomio nulo. Es decir, si los polinomios son iguales. Entonces, si fijamos el grado de los polinomios $N-1$, y N complejos distintos $x_0, x_1, \cdots, x_{N-1}$, a cada polinomio de grado a lo sumo $N-1$ se le puede asignar de manera biyectiva un vector en $\mathbb{C}^N$ mediante $A(x) \longrightarrow (A(x_0), A(x_1), \ldots, A(x_{N-1}))$, que llamaremos **representación por valores**. Luego, nuevamente, dos polinomios son iguales si y solo si tienen la misma representación por valores, aunque no parece haber una manera que no demande $O(N^2)$ operaciones para reconstruir el polinomio original, que es plantear un sistema de N ecuaciones con N incógnitas para despejar los coeficientes.

Vamos a definir ahora una función entre vectores de igual tamaño en $\mathbb{C}^N$, a la que notaremos $*$, pero por comodidad llamaremos producto, que será multiplicar coordenada a coordenada ambos vectores:

$$(u_0, u_1, \cdots, u_{N-1}) * (v_0, v_1, \cdots, v_{N-1}) = (u_0v_0, u_1v_1, \cdots, u_{N-1}v_{N-1})$$

Para mayor fluidez de la escritura definamos algunas notaciones extra también. Sea $A(x) = \sum_{k=0}^{N-1} a_k x^k$ un polinomio en $\mathbb{C}[x]_{\leq N-1}$, entonces:
$A = (a_0, a_1, \cdots, a_{N-1}) \in \mathbb{C}^N$ es su representación por coeficientes, donde los coeficientes entre a_n y a_{N-1} fueron completados con ceros.
Si $B(x)$ es también un polinomio en $\mathbb{C}[x]_{\leq N-1}$, consideramos el polinomio $A(x)B(x) \in \mathbb{C}[x]_{\leq N-1}$ y denotamos $A \cdot B = (c_0, c_1, \cdots, c_{N-1}) \in \mathbb{C}^N$ donde los c_i son los coeficientes de $A(x)B(x)$ rellenados con ceros hasta llegar al tamaño N, a la representación en coeficientes de $A(x)B(x)$.

Además, si fijamos los N puntos distintos en los que vamos a evaluar, definimos $f : \mathbb{C}[x]_{\leq N-1} \longrightarrow \mathbb{C}^N$ como la representación por valores de los polinomios de grado a lo sumo

$N-1$. Es fácil notar que si llamamos $C(x) = A(x)B(x)$ entonces $C(x_i) = A(x_i)B(x_i)$ para todo i, por lo que

$$f(A(x)B(x)) = f(C(x)) =$$
$$(C(x_0), C(x_1), \cdots, C(x_{N-1})) =$$
$$(A(x_0)B(x_0), A(x_1)B(x_1), \cdots, A(x_{N-1})B(x_{N-1})) =$$
$$(A(x_0), A(x_1), \cdots, A(x_{N-1})) * (B(x_0), B(x_1), \cdots, B(x_{N-1})) =$$
$$f(A(x)) * f(B(x))$$

Y resulta que el producto de la representación por valores de dos polinomios, es igual a la representación por valores del producto de ambos polinomios. El producto habitual entre polinomios es $O(N^2)$, pero el producto entre las representaciones es $O(N)$. El problema es que asignar a cada polinomio su representación por valores es $O(N^2)$ y una vez que tenemos la representación por valores, reconstruir el polinomio también es $O(N^2)$. Por lo que, hasta ahora, no hemos mejorado la eficiencia del producto, solo movimos la complejidad de lugar: ahora lo costoso es obtener la representación y deshacerla, y la complejidad sigue siendo cuadrática. Aunque parezca que nos encontramos en un callejón sin salida, esto que acabamos de presentar es la idea con la que trabaja la Transformada Rápida de Fourier (FFT por sus siglas en inglés) y que vamos a complementar con algunas propiedades de los números complejos para volver más eficiente nuestro algoritmo.

5.2.3 La clave de la eficiencia: las raíces de la unidad

Antes de continuar con el algoritmo, vamos a hacer un breve repaso sobre números complejos: la ecuación $x^N = 1$ tiene exáctamente N soluciones complejas para todo N entero positivo, a las que llamamos **raíces N-ésimas de la unidad**. Además, sabemos que cada una de las N raíces son de la forma $\omega_{N,k} = e^{\frac{2k\pi i}{N}}$ con $0 \leq k \leq N-1$. Por último notamos que $\omega_{N,1}$, a la que llamaremos ω_N, puede ser utilizada para describir a todas las demás raíces, ya que $\omega_{N,k} = e^{\frac{2k\pi i}{N}} = \left(e^{\frac{2\pi i}{N}}\right)^k = (\omega_N)^k$ para cada k.

Dado un polinomio $A(x)$ en $\mathbb{C}[x]_{\leq N-1}$, definimos la **Transformada Discreta de Fourier** como $DFT : \mathbb{C}^N \longrightarrow \mathbb{C}^N$ dada por

$$DFT(A) = (A(\omega_{N,0}), A(\omega_{N,1}), \cdots, A(\omega_{N,k-1})) =$$
$$(A\left(\omega_N^0\right), A\left(\omega_N^1\right), \cdots, A\left(\omega_N^{N-1}\right)) = (y_0, y_1, \cdots, y_{N-1})$$

Como la DFT es un caso particular de representación por valores, sabemos que $DFT(A \cdot B) = DFT(A) * DFT(B)$. Además sabemos que cualquier representación por valores es biyectiva, por lo que podemos definir la **Transformada discreta inversa de Fourier** $IDFT : \mathbb{C}^N \longrightarrow \mathbb{C}^N$ como la operación inversa de la DFT: $IDFT(y_0, y_1, \cdots, y_{N-1}) = A$, es decir que $IDFT$ reconstruye exáctamente el vector de coeficientes original A a partir de su representación por valores obtenida en la DFT. Ahora, por lo dicho en la sección anterior, si pudiéramos calcular $DFT(A)$ y $IDFT(A)$ de manera más eficiente que $O(N^2)$, nuestro algoritmo sería más eficiente, porque podemos calcular $A \cdot B = IDFT(DFT(A) * DFT(B))$.

5.2.4 La Transformada rápida de Fourier

Ahora presentaremos la **Transformada rápida de Fourier**[12], que es un algoritmo que nos permite calcular la DFT de manera eficiente aplicando divide y vencerás. Sea $A(x)$ un polinomio de grado $N-1$:

$$A(x) = a_0 + a_1 x + \cdots + a_{N-1}x^{N-1}$$

Ahora construimos dos polinomios de grado a lo sumo $N/2-1$, uno con los coeficientes en posiciones pares y otro con los coeficientes en posiciones impares:

$$P(x) = a_0 + a_2 x + \cdots + a_{N-2}x^{N/2-1}$$

$$I(x) = a_1 + a_3 x + \cdots + a_{N-1}x^{N/2-1}$$

Ahora podemos notar que

$$A(x) = P(x^2) + xI(x^2)$$

. Los polinomios $P(x)$ e $I(x)$ tienen la mitad de coeficientes que $A(x)$. Si llamamos $f(N)$ a la cantidad de operaciones elementales necesarias para calcular $DFT(A)$, a partir de $DFT(P)$, y $DFT(I)$) a la cantidad de operaciones elementales que lleva calcular $DFT(A)$, entonces tenemos que dicha cantidad cumple la recurrencia:

$$T_{DFT}(N) = 2T_{DFT}(N/2) + f(N)$$

Veamos que $f(N)$ resulta ser $O(N)$. En efecto, supongamos que ya calculamos

$$(b_0, b_1, \cdots, b_{N/2-1}) = \left(P\left(\omega_{N/2}^0\right), P\left(\omega_{N/2}^1\right), \cdots, P\left(\omega_{N/2}^{N/2-1}\right)\right) = DFT(P)$$

y

$$(c_0, c_1, \cdots, c_{N/2-1}) = \left(I\left(\omega_{N/2}^0\right), I\left(\omega_{N/2}^1\right), \cdots, I\left(\omega_{N/2}^{N/2-1}\right)\right) = DFT(I)$$

. Primero notemos que, para $0 \leq k \leq N/2-1$, vale que $\omega_N^{2k} = e^{\frac{4k\pi i}{N}} = e^{\frac{2k\pi i}{N/2}} = \omega_{N/2}^k$. Usando que $A(x) = P(x^2) + xI(x^2)$, tenemos entonces:

$$y_k = A\left(\omega_N^k\right) = P\left(\left(\omega_N^k\right)^2\right) + \omega_N^k I\left(\left(\omega_N^k\right)^2\right) =$$

$$P\left(\omega_N^{2k}\right) + \omega_N^k I\left(\omega_N^{2k}\right) = P\left(\omega_{N/2}^k\right) + \omega_N^k I\left(\omega_{N/2}^k\right) = b_k + \omega_N^k c_k$$

Por otro lado, notemos que también para $0 \leq k \leq N/2-1$ vale que $\omega_N^{2k+N} = \omega_N^{2k}\omega_N^N = \omega_N^{2k} = \omega_{N/2}^k$ y que $\omega_N^{k+N/2} = \omega_N^k \omega_N^{N/2} = -\omega_N^k$. Usando nuevamente que $A(x) = P(x^2) + xI(x^2)$ tenemos que:

$$y_{k+N/2} = A\left(\omega_N^{k+N/2}\right) = P\left(\left(\omega_N^{k+N/2}\right)^2\right) + \omega_N^{k+N/2} I\left(\left(\omega_N^{k+N/2}\right)^2\right) =$$

$$P\left(\omega_N^{2k+N}\right) + \omega_N^{k+N/2} I\left(\omega_N^{2k+N}\right) = P\left(\omega_{N/2}^k\right) - \omega_N^k I\left(\omega_{N/2}^k\right) = b_k - \omega_N^k c_k$$

Por lo que logramos reconstruir $DFT(A)$ en tiempo lineal. Tenemos entonces la recurrencia:

$$T_{DFT}(N) = 2T_{DFT}(N/2) + f(N)$$

Que como $f(N)$ es $O(N)$ podemos resolverla por el Teorema Maestro[10] que introdujimos en el capítulo anterior. Si indentificamos los parámetros, tenemos que $a = 2$ (dos subproblemas), $b = 2$ (de tamaño la mitad que el original) y un costo lineal para combinar ambos subproblemas. Por lo que el exponente de $O(N^d)$ es 1. Al comparar d con el valor de $\log_b(a)$ nos queda $\log_2(2) = 1$, por lo que $d = \log_b(a)$ y nos encontramos en el segundo caso del Teorema Maestro, cuya solución es que $T_{DFT}(N)$ sea $O\left(N^d \log(N)\right)$. Como en este caso $d = 1$, tenemos que calcular $DFT(A)$ tiene una complejidad $O(N \log(N))$.

5.2.5 La FFT Inversa

Ya demostramos que podemos calcular $DFT(A)$ en $O(N \log(N))$, pero optimizar ese proceso sería en vano si no podemos calcular también de manera eficiente $IDFT(Y)$, es decir, reconstruir a partir de la representación por valores la representación por coeficientes. Notemos que, por como la definimos, podemos escribir la $DFT(A)$ de manera matricial:

$$\begin{pmatrix} \omega_N^0 & \omega_N^0 & \omega_N^0 & \omega_N^0 & \cdots & \omega_N^0 \\ \omega_N^0 & \omega_N^1 & \omega_N^2 & \omega_N^3 & \cdots & \omega_N^{N-1} \\ \omega_N^0 & \omega_N^2 & \omega_N^4 & \omega_N^6 & \cdots & \omega_N^{2(N-1)} \\ \omega_N^0 & \omega_N^3 & \omega_N^6 & \omega_N^9 & \cdots & \omega_N^{3(N-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ \omega_N^0 & \omega_N^{N-1} & \omega_N^{2(N-1)} & \omega_N^{3(N-1)} & \cdots & \omega_N^{(N-1)(N-1)} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_{N-1} \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{N-1} \end{pmatrix}$$

Este tipo especial de matrices, donde cada fila es una progresión geométrica con primer término y razón no nulos se llama **Matrices de Vandermonde** y es una propiedad conocida que si las razones de las progresiones son distintas dos a dos, entonces la matriz es invertible [13]. Veamos que la inversa de esta matriz es exáctamente:

$$\frac{1}{N} \begin{pmatrix} \omega_N^0 & \omega_N^0 & \omega_N^0 & \omega_N^0 & \cdots & \omega_N^0 \\ \omega_N^0 & \omega_N^{-1} & \omega_N^{-2} & \omega_N^{-3} & \cdots & \omega_N^{-(N-1)} \\ \omega_N^0 & \omega_N^{-2} & \omega_N^{-4} & \omega_N^{-6} & \cdots & \omega_N^{-2(N-1)} \\ \omega_N^0 & \omega_N^{-3} & \omega_N^{-6} & \omega_N^{-9} & \cdots & \omega_N^{-3(N-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ \omega_N^0 & \omega_N^{-(N-1)} & \omega_N^{-2(N-1)} & \omega_N^{-3(N-1)} & \cdots & \omega_N^{-(N-1)(N-1)} \end{pmatrix}$$

Si llamamos C al producto de ambas matrices tenemos, para todo $0 \leq i, j \leq N-1$, que

$$C_{i+1j+1} = \frac{1}{N} \cdot \sum_{k=0}^{N-1} \omega_N^{-ik} \cdot \omega_N^{kj} = \frac{1}{N} \cdot \sum_{k=0}^{N-1} \omega_N^{k(j-i)} = \frac{1}{N} \cdot \sum_{k=0}^{N-1} \left(\omega_N^{j-i}\right)^k$$

, que no es otra cosa que una progresión geométrica de primer término 1 y de razón ω_N^{j-i}, por lo que nos queda:

$$C_{ij} \begin{cases} \frac{1}{N} \cdot \sum_{k=0}^{N-1} \left(\omega_N^{j-i}\right)^k = \frac{1}{N} \cdot \sum_{k=0}^{N-1} 1 = 1 & \text{si } i = j \\ \frac{1}{N} \cdot \sum_{k=0}^{N-1} \left(\omega_N^{j-i}\right)^k = \frac{1}{N} \cdot \frac{\left(\omega_N^{j-i}\right)^N - 1}{\omega_N^{j-i} - 1} = \frac{1}{N} \cdot \frac{\left(\omega_N^N\right)^{j-i} - 1}{\omega_N^{j-i} - 1} = \frac{1}{N} \cdot \frac{1^{j-i} - 1}{\omega_N^{j-i} - 1} = 0 & \text{si } i \neq j \end{cases},$$

por lo que C resulta ser la identidad, como queríamos demostrar. Despejando nos queda entonces:

$$\begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_{N-1} \end{pmatrix} = \frac{1}{N} \begin{pmatrix} \omega_N^0 & \omega_N^0 & \omega_N^0 & \omega_N^0 & \cdots & \omega_N^0 \\ \omega_N^0 & \omega_N^{-1} & \omega_N^{-2} & \omega_N^{-3} & \cdots & \omega_N^{-(N-1)} \\ \omega_N^0 & \omega_N^{-2} & \omega_N^{-4} & \omega_N^{-6} & \cdots & \omega_N^{-2(N-1)} \\ \omega_N^0 & \omega_N^{-3} & \omega_N^{-6} & \omega_N^{-9} & \cdots & \omega_N^{-3(N-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ \omega_N^0 & \omega_N^{-(N-1)} & \omega_N^{-2(N-1)} & \omega_N^{-3(N-1)} & \cdots & \omega_N^{-(N-1)(N-1)} \end{pmatrix} \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{N-1} \end{pmatrix}$$

La belleza de este resultado es que la matriz de la $IDFT$ es muy parecida a la matriz de la DFT, con la diferencia de que los exponentes de ω_N son negativos y hay un factor de escala $\frac{1}{N}$. Pero esta similitud que podría considerarse de índole estética tiene intrínseca una similitud algorítmica: ¿qué propiedades de ω_N usamos para el algoritmo de la FFT? Que $\omega_N^N = 1$, que $\omega_N^{N/2} = -1$, y que a medida que k recorre los enteros entre 0 y $N-1$ ω_N^k va recorriendo las N raíces N-ésimas de la unidad, que $\omega_N^{2k} = \omega_{N/2}^k$, pero ω_N no es la única raíz N-ésima de la unidad que cumple todas estas propiedades. Una raíz **raíz primitiva** de orden N de la unidad es un complejo z tal que $z^N = 1$ y $z^m \neq 1$ para todo valor de m entre 1 y $N-1$. Son propiedades conocidas que $\omega_{N,k}$ es raíz primitiva de la unidad de orden N si y solo si $(N,k) = 1$, y además que cualquier raíz primitiva de la unidad cumple las propiedades que nombramos sobre ω_N. Solo basta notar que $\omega_N^{-1} = \omega_N^N \omega_N^{-1} = \omega_N^{N-1} = \omega_{N,N-1}$. Pero como $(N-1, N) = 1$, resulta entonces ser también raíz primitiva de la unidad y podemos repetir todos los pasos que hicimos para calcular DFT en $O(N\log(N))$

5.3. El algoritmo final

Ya con todas las herramientas presentadas para poder multiplicar dos polinomios, dados $A(x)$ y $B(x)$ dos polinomios de grado a lo sumo $n-1$, el algoritmo final es el siguiente:

1. **Rellenar los coeficientes:** El polinomio que resulta del producto $C(x) = A(x)B(x)$ será de grado a lo sumo $2n-2$, por lo que necesitaremos $2n-1$ coeficientes para determinarlo. Elegimos N como la menor potencia de dos tal que $N \geq 2n-1$. Luego, extendemos los vectores de coeficientes de $A(x)$ y $B(x)$ con ceros hasta que ambos tengan longitud N. A estas extensiones las llamaremos A_r y B_r.

2. **FFT:** Calculamos las representaciones por valores de los polinomios A_r y B_r aplicando la FFT a sus vectores de coeficientes:

   $$Y_A = \mathrm{DFT}(A_r) \quad , \quad Y_B = \mathrm{DFT}(B_r)$$

   Este paso tiene una complejidad de $O(N\log(N))$.

3. **Producto entre las representaciones:** Calculamos el producto de las dos representaciones por valores para obtener la representación por valores del polinomio producto $C(x)$.

   $$DFT(C) = Y_C = Y_A * Y_B$$

Este paso tiene una complejidad de $O(N)$.

4. **FFT Inversa:** Aplicamos la FFT inversa al vector Y_C para reconstruir el vector de coeficientes del polinomio producto $C(x)$.

$$C = \text{IDFT}(Y_C)$$

Este paso también tiene una complejidad de $O(N \log(N))$. El vector C contiene los coeficientes de $A(x)B(x)$.

La complejidad total del algoritmo es la suma de las complejidades de cada paso, quedando dominada por la FFT y la FFT inversa, por lo que resulta ser $O(N \log(N))$.

6. Epílogo

> *"En matemáticas, no se entienden las cosas. Simplemente uno se acostumbra a ellas."*
>
> John von Neumann, 1903–1957.

Nuestra formación académica en el colegio suele tener un enfoque constructivista en muchas disciplinas: construyamos juntos un ejemplo donde esto sucede, o demos un proceso para hacerlo. La lógica es simple pero potente: si se como hacerlo, si tengo la receta exacta, o más aún, si ya lo hice, es claro que dicho ejemplo existe. Todavía me acuerdo cuando en la práctica de Análisis I teníamos que demostrar que la ecuación x=cos(x) tenía al menos una solución, estuvimos un buen rato tanteando hasta que Hernán nos explicó como hacerlo. Pasaron muchos teoremas, muchos problemas, muchas materias, pero Tomi, Marquito y yo todavía nos acordamos con total claridad la fascinación que nos produjo la solución con el Teorema de Bolzano. ¿Cómo podía ser posible que pudiéramos afirmar que algo existía si no podíamos saber en efecto quien era la solución? Más aún, tal vez era imposible obtener una expresión cerrada para dicha solución, pero la afirmación seguía siendo cierta. Los teoremas de existencia me parecen algo increíble, y me conmueve esa incertidumbre de que tal vez nunca sepamos como construir ese objeto que tenemos la certeza de que existe.

Paradójicamente, la existencia de los teoremas de existencia me hizo valorar más cuando un resultado matemático incluía ejemplos que podíamos construir. Sin embargo,a medida que me adentraba en el mundo de la programación me surgió una pregunta que hasta ese momento siempre creí tener saldada: *¿qué significa construir un ejemplo?* Si yo tengo una receta para cocinar una torta pero uno de los pasos necesarios implica utilizar un horno a una temperatura mayor que la superficie solar, y otro de los pasos implica que la torta pase 50.000 años en el horno, entonces... ¿tengo una receta para cocinar una torta?

Esta pregunta aparentemente trivial sintetiza la esencia de la complejidad computacional. En nuestro viaje desde la multiplicación escolar hasta la FFT, hemos visto cómo algoritmos teóricamente correctos pueden ser inútiles en la práctica cuando su complejidad es exponencial. Un algoritmo que requiere más operaciones que átomos hay en el universo para procesar una entrada modesta no es realmente un método constructivo, sino una curiosidad teórica.

La línea que divide la eficiencia de los algoritmos a veces se vuelve difusa: constantes realmente grandes pueden comprometer la practicidad. Consideremos un algoritmo que requiere $10^{10^{10}} \cdot n$ operaciones. ¿A partir de qué tamaño de entrada se vuelve mejor que un algoritmo cuadrático? El concepto de **algoritmo galáctico** busca formalizar esta paradoja, definiendo aquellos algoritmos que, a pesar de tener complejidad polinómica, solo son óptimos para tamaños de entrada que superan cualquier aplicación práctica concebible (incluso a escala galáctica).

En este constante diálogo entre dos mundos, el desarrollo de algoritmos para aplicaciones inmediatas y la ciencia básica donde la optimización es puramente teórica, la matemática emerge como lenguaje común. Ella nos permite navegar esa frontera sutil entre lo computable en teoría y lo construible en la práctica.

En ese sentido, el verdadero poder de la matemática computacional radica en encontrar esos atajos elegantes que transforman lo imposible en posible. Cada paso presentado a lo largo de esta monografía representó un salto cualitativo que nos permitió realmente construir soluciones. La notación $O(\cdot)$ es nuestra brújula en este territorio: nos dice qué recetas son viables y cuáles son meras fantasías culinarias.

Muchas de las ideas presentadas son una puerta de entrada hacia un mundo nuevo donde se continua investigando. Lo más revolucionario de la idea de Karatsuba no fue la complejidad obtenida, sino como rompió con una creencia que llevaba siglos siendo aceptada sin cuestionamientos, porque detrás de Karatsuba vinieron avances constantes en ese campo, el último algoritmo desarrollado para producto de números grandes data del 2019 [14]. Por otro lado, más allá del increíble poder de la FFT, el algoritmo tal cual lo presentamos tiene sus limitaciones. Pero donde nace un problema con un algoritmo nace un problema matemático nuevo por resolver. Buscando optimizaciones, es que surge la implementación iterativa de la FFT en lugar de la recursiva que planteamos, o la NTT [15] para producto de polinomios con aritmética modular. Mismo el producto de matrices se puede hacer de manera más eficiente con el algoritmo de Strassen [16]. Se invita al lector interesado también a ahondar en las increíbles aplicaciones de los algoritmos presentados, la FFT es para muchos el algoritmo más importante de la historia de la humanidad, su aplicación constante en distintas ramas de la computación le dan una impronta que excede su mera belleza matemática.

Al fin y al cabo, para ir cerrando, espero que podamos irnos con la idea de que al final, multiplicar rápido no es solo sobre velocidad. Es sobre hacer posible lo que antes era impensable. Es la diferencia entre saber que existe una solución y poder sostenerla en nuestras manos, aunque sea en forma de bits. En ese cruce entre lo abstracto y lo tangible, entre la existencia y la construcción eficiente, es donde la magia de estos algoritmos realmente brilla.

Referencias

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974. Definición canónica del modelo RAM y operaciones elementales (Capítulo 1).

[2] Donald E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley Professional, 3 edition, 1997. Modelo extendido para

operaciones con grandes enteros (Sección 4.3.1).

[3] Richard Schroeppel. Addition chains and the NP-completeness of minimal chain sequences. Technical Report MIT-LCS-TM-054, Massachusetts Institute of Technology, 1975.

[4] W. R. Alford, Andrew Granville, and Carl Pomerance. There are infinitely many carmichael numbers. *Annals of Mathematics*, 139(3):703–722, 1994.

[5] David S. Dummit and Richard M. Foote. *Abstract Algebra*. John Wiley & Sons, 3 edition, 2004. Teorema de Lagrange (Sección 3.2).

[6] Michael O. Rabin. Probabilistic algorithm for testing primality. *Journal of Number Theory*, 12(1):128–138, 1980.

[7] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics: A Foundation for Computer Science*. Addison-Wesley, 2 edition, 1994.

[8] Reinhard Diestel. *Graph Theory*. Springer, 5 edition, 2018.

[9] Anatolii A. Karatsuba and Yevgenii M. Ofman. Multiplication of many-digit numbers by automatic computers. *Doklady Akademii Nauk SSSR*, 145:293–294, 1962. In Russian; English translation in Soviet Physics Doklady, 7 (1963), 595–596.

[10] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 3rd edition, 2009. Sección que presenta el Teorema Maestro.

[11] Herbert S. Wilf. *Generatingfunctionology*. Academic Press, 2nd edition, 1994. Online PDF accessed via University of Pennsylvania.

[12] J. W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of Computation*, 19(90):297–301, 1965.

[13] Roger A. Horn and Charles R. Johnson. *Topics in Matrix Analysis*. Cambridge University Press, 1991. Ver sección 0.12 sobre matrices de Vandermonde.

[14] David Harvey and Joris van der Hoeven. Integer multiplication in time $o(n \log n)$. *arXiv preprint arXiv:1907.09577*, 2019.

[15] Joachim von zur Gathen and Jürgen Gerhard. *Modern Computer Algebra*. Cambridge University Press, 3 edition, 2013.

[16] Volker Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13(4):354–356, 1969.