Algoritmos Recursivos en Teoría de Grafos

1. De qué trata este libro

1.1. Antecedentes

Un algoritmo es un método para resolver una clase de problemas en una computadora. La complejidad de un algoritmo es el costo, medido en tiempo de ejecución, almacenamiento, o cualquier otra unidad relevante, de utilizar el algoritmo para resolver uno de esos problemas.

Este libro trata sobre algoritmos y complejidad y, por lo tanto, sobre los métodos para resolver problemas en computadoras y los costos (usualmente el tiempo de ejecución) de emplear dichos métodos.

Computar requiere tiempo. Algunos problemas toman muchísimo tiempo; otros pueden realizarse rápidamente. Algunos problemas parecen tardar mucho y, de pronto, alguien descubre una manera más veloz de resolverlos (un «algoritmo más rápido»). El estudio de la cantidad de esfuerzo computacional necesario para llevar a cabo ciertos tipos de cómputos es el estudio de la complejidad computacional.

Naturalmente, esperaríamos que un problema de cómputo para el cual se requieran millones de bits de datos de entrada tarde probablemente más que otro problema que solo necesita unos pocos elementos de entrada. Así, la complejidad temporal de un cálculo se mide expresando el tiempo de ejecución del cálculo como una función de alguna medida de la cantidad de datos necesaria para describir el problema a la computadora.

Por ejemplo, considere la siguiente afirmación: «Acabo de comprar un programa para invertir matrices y puede invertir una matriz $n \times n$ en tan solo $1{,}2\,n^3$ minutos». Aquí observamos una descripción típica de la complejidad de un algoritmo determinado: el tiempo de ejecución del programa se da como una función del tamaño de la matriz de entrada.

Un programa más rápido para la misma tarea podría ejecutarse en $0{,}8\,n^3$ minutos para una matriz $n \times n$. Si alguien realizara un descubrimiento realmente importante (véase la Sección 2.4), entonces tal vez podríamos reducir el exponente en lugar de mantenerlo en 3.

meramente reduciendo la constante multiplicativa. Así, un programa que invirtiese una matriz de $n \times n$ en tan solo $7n^{2,8}$ minutos supondría una mejora sorprendente del estado del arte.

A los efectos de este libro, una computación que está garantizada para tomar como máximo cn^3 tiempo para entrada de tamaño n se considerará una computación «fácil». Una que requiera como máximo n^{10} tiempo también es fácil. Si cierto cálculo sobre una matriz de $n \times n$ requiriese 2^n minutos, entonces sería un problema «difícil». Naturalmente, algunas de las computaciones que llamamos «fáciles» pueden tardar muchísimo en ejecutarse, pero aún así, desde nuestro punto de vista actual, la distinción importante será la garantía de tiempo polinómico o su ausencia.

La regla general es que si el tiempo de ejecución es como máximo una función polinómica de la cantidad de datos de entrada, entonces el cálculo es fácil; de lo contrario, es difícil.

Muchos problemas en informática son conocidos por ser fáciles. Para convencer a alguien de que un problema es fácil, basta con describir un método rápido para resolverlo. Convencer a alguien de que un problema es difícil es más complicado, pues habrá que demostrarle que es imposible encontrar una forma rápida de realizar el cálculo. No bastará con señalar un algoritmo particular y lamentar su lentitud. Al fin y al cabo, ese algoritmo puede ser lento, pero quizá exista uno más rápido.

La inversión de matrices es fácil. El método familiar de eliminación gaussiana puede invertir una matriz de $n \times n$ en tiempo como máximo cn^3.

Para dar un ejemplo de un problema computacional difícil, debemos alejarnos un poco. Uno interesante se llama el «problema del enlosado». Supongamos[1] que disponemos de infinitas baldosas idénticas, cada una con forma de hexágono regular. Entonces podemos enlosar todo el plano con ellas, es decir, cubrir el plano sin dejar espacios vacíos. Esto también puede hacerse si las baldosas son rectángulos idénticos, pero no si son pentágonos regulares.

En la Figura 0.1 mostramos un enlosado del plano mediante rectángulos idénticos, y en la Figura 0.2 un enlosado mediante hexágonos regulares.

Eso plantea ciertas cuestiones teóricas y computacionales. Una pregunta computacional es la siguiente: supongamos que nos dan un cierto polígono, no necesariamente regular ni convexo, y que tenemos infinitas baldosas idénticas con esa forma. ¿Podremos o no conseguir enlosar todo el plano?

Esa elegante cuestión ha sido demostrada[2] como computacionalmente insoluble. En otras palabras, no solo desconocemos un modo rápido de resolverla, sino que sabemos que tal modo no puede existir.

[1] Véase, por ejemplo, el artículo de Martin Gardner en *Scientific American*, enero de 1977, pp. 110–121.

[2] R. Berger, *The undecidability of the domino problem*, *Memoirs Amer. Math. Soc.* 66 (1966), Amer. Math. Soc., Providence, RI.

Cuando se presenta un problema de enlosado a una computadora, se ha demostrado que no existe ninguna forma de resolverlo, por lo que incluso buscar un algoritmo sería infructuoso. Eso no significa que la cuestión sea difícil para todo polígono. Los problemas difíciles pueden tener instancias fáciles. Lo que se ha demostrado es que no existe un método único que garantice que decidirá esta cuestión para cada polígono.

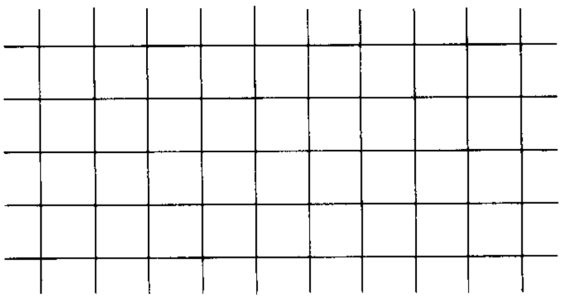

Figura 1: Enlosado con rectángulos.

El hecho de que un problema computacional sea difícil no significa que cada instancia del mismo deba ser difícil. El problema es difícil porque no podemos idear un algoritmo al que podamos otorgar una garantía de rendimiento rápido para todas las instancias.

Nótese que la cantidad de datos de entrada para la computadora en este ejemplo es bastante pequeña. Todo lo que necesitamos introducir es la forma del polígono básico. Sin embargo, no solo es imposible elaborar un algoritmo rápido para este problema, sino que se ha demostrado que es imposible idear cualquier algoritmo que esté garantizado a terminar con una respuesta Sí/No después de un número finito de pasos. ¡Eso sí que es difícil!

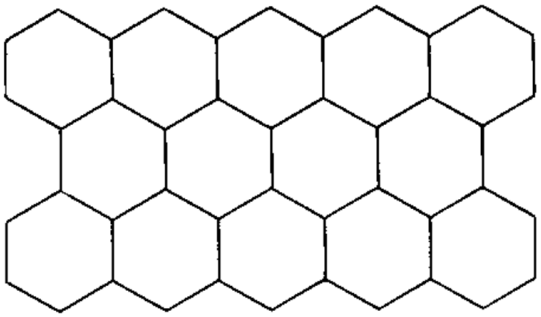

Figura 2: Enlosado con hexágonos.

2. Problemas difíciles frente a fáciles

Tomemos un momento más para decir, de otra manera, exactamente qué queremos decir con un cómputo «fácil» frente a uno «difícil».

Piensa en un algoritmo como una pequeña caja que puede resolver una cierta clase de problemas computacionales. En la caja entra la descripción de un problema particular de esa clase y entonces, tras cierta cantidad de tiempo o esfuerzo computacional, aparece la respuesta.

Ejemplo 0.1. Se garantiza que si el problema de entrada se describe con B bits de datos, entonces se obtendrá una respuesta después, como máximo, de $6B^3$ minutos.

Ejemplo 0.2. Se garantiza que todo problema cuya descripción requiera como máximo B bits de datos se resolverá en, como mucho, $0{,}7B^{15}$ segundos.

Una garantía de rendimiento, como las dos anteriores, a veces se denomina *estimación de complejidad del peor caso*, y es fácil ver por qué. Si tenemos un algoritmo que, por ejemplo, ordena cualquier secuencia dada de números en orden ascendente (ver Sección 2.2), puede ocurrir que algunas secuencias sean más fáciles de ordenar que otras.

Por ejemplo, la secuencia $1, 2, 7, 11, 10, 15, 20$ ya está casi ordenada, así que nuestro algoritmo podría, si aprovecha ese orden parcial, ordenarla muy rápidamente. Otras secuencias podrían ser mucho más difíciles de manejar y, por tanto, requerir más tiempo.

Así que, para algunos problemas cuya cadena de bits de entrada tiene B bits, el algoritmo podría operar en tiempo $6B$, en otros necesitaría, digamos, $10B \log B$ unidades de tiempo, y para aún otras instancias de longitud B bits, podría necesitar $5B^2$ unidades de tiempo para completar la tarea.

Entonces, ¿qué diría la tarjeta de garantía? Tendría que considerar la posibilidad más desfavorable; de lo contrario, la garantía no sería válida. Aseguraría al usuario que, si la instancia del problema de entrada puede describirse con B bits, entonces aparecerá una respuesta tras, como máximo, $5B^2$ unidades de tiempo. Por lo tanto, una garantía de rendimiento equivale a una estimación del peor escenario posible: el cálculo más largo que podría ocurrir si se introducen B bits al programa.

Los límites de peor caso son el tipo más común, pero existen otras clases de límites para el tiempo de ejecución. Podríamos dar, en cambio, un límite de caso promedio (ver Sección 5.7). Eso no garantizaría un rendimiento peor que tal o cual; declararía que si se promedia el rendimiento sobre todas las cadenas de bits posibles de B bits, entonces el tiempo de cálculo promedio será tal o cual (en función de B).

Ahora hablemos de la diferencia entre problemas computacionales fáciles y difíciles y entre algoritmos rápidos y lentos.

Una garantía que no asegurase un rendimiento «rápido» contendría alguna función de B que creciese más rápido que cualquier polinomio. Por ejemplo, e^B o $2^{\sqrt{B}}$, etc. Es la garantía de tiempo polinómico frente a la que no necesariamente lo es lo que marca la diferencia entre las clases de problemas fáciles y difíciles, o entre algoritmos rápidos y lentos. Es muy deseable trabajar con algoritmos tales que podamos proporcionar una garantía de rendimiento para su tiempo de ejecución que sea, como mucho, una función polinómica del número de bits de entrada.

Un algoritmo es lento si, para cualquier polinomio P que consideremos, existen valores arbitrariamente grandes de B y cadenas de datos de entrada de B bits que hacen que el algoritmo realice más de $P(B)$ unidades de trabajo.

Un problema computacional es *tratable* si existe un algoritmo rápido que resuelva todas sus instancias.

Un problema computacional es *intratable* si puede demostrarse que no existe ningún algoritmo rápido para él.

Ejemplo 0.3. Aquí hay un problema computacional familiar y un método, o algoritmo, para resolverlo. Veamos si el método tiene una garantía de tiempo polinómico.

El problema es el siguiente. Sea n un número entero dado. Queremos averiguar si n es primo. El método que elegimos es el siguiente. Para cada entero

$$m = 2, 3, \ldots, \lfloor \sqrt{n} \rfloor$$

preguntamos si m divide a n. Si todas las respuestas son «No», entonces declaramos que n es primo; en caso contrario, es compuesto.

Ahora analizaremos la complejidad computacional de este algoritmo. Eso significa que veremos cuántas operaciones implica la prueba. Para un entero dado n, el trabajo se puede medir en unidades de división de un número entero por otro. En esas unidades, obviamente realizaremos alrededor de $\sqrt{n}$ divisiones.

Parecería que este es un problema tratable, porque, al fin y al cabo, $\sqrt{n}$ crece de forma polinómica en n. Por ejemplo, hacemos menos de n divisiones, y eso sin duda es un polinomio en n. Según nuestra definición de algoritmos rápidos y lentos, la distinción se basó en el crecimiento polinómico frente al crecimiento más rápido que el polinómico del trabajo según el tamaño del problema, y, por tanto, este problema debería ser fácil. ¿Verdad? Pues no, en realidad no.

Conviene referirse a la distinción entre métodos rápidos y lentos para ver que debemos medir la cantidad de trabajo realizado en función del número de bits de entrada al problema. En este ejemplo, n no es el número de bits de la entrada. Por ejemplo, si $n = 59$, no necesitamos 59 bits para describir n, sino solo 6. En general, el número de dígitos binarios en la representación de un entero n es aproximadamente $\log_2 n$.

Así, en el problema de este ejemplo, el de probar la primalidad de un entero dado n, la longitud de la cadena de bits de entrada B es aproximadamente $\log_2 n$. Vista bajo esta perspectiva, el cálculo parece de pronto muy largo. Una cadena formada por tan solo $\log_2 n$ ceros y unos ha provocado que nuestro poderoso ordenador realice alrededor de $\sqrt{n}$ unidades de trabajo.

Si expresamos la cantidad de trabajo realizada como una función de B, encontramos que la complejidad de este cálculo es aproximadamente $2^{B/2}$, y eso crece mucho más rápido que cualquier función polinómica de B.

Por lo tanto, el método que acabamos de discutir para comprobar la primalidad de un entero dado es lento. La historia de este problema acaba de tomar un giro interesante. Tras muchos años en los que no se sabía si la primalidad podía probarse en tiempo polinómico, en 2002 se halló un nuevo algoritmo que lo hace exactamente. Véase el Capítulo 4 para una discusión más detallada de este problema.

En este libro abordaremos algunos problemas fáciles y otros que aparentan ser difíciles. Es el "aparentan" lo que hace las cosas muy interesantes. Se trata de problemas para los que nadie ha encontrado un algoritmo rápido, pero tampoco nadie ha demostrado la imposibilidad de que exista. Debe añadirse que toda el área se investiga con vigor debido al atractivo y a la importancia de las numerosas preguntas sin respuesta que aún persisten.

Así pues, aunque todavía no sepamos muchas cosas que nos gustaría conocer en este campo, ¡no es por falta de intentarlo!

2.1. 0.3 A Preview

El Capítulo 1 contiene parte de la base matemática que se necesitará para nuestro estudio de los algoritmos. No se pretende que la lectura de este libro, o su uso como texto en un curso, deba comenzar necesariamente con el Capítulo 1. Probablemente sea mejor sumergirse directamente en el Capítulo 2 y, cuando se requieran habilidades o conceptos concretos, leer las secciones pertinentes del Capítulo 1. De lo contrario, las definiciones e ideas de ese capítulo podrían parecer carentes de motivación cuando, de hecho, la motivación —en gran cantidad— reside en los capítulos posteriores del libro.

El Capítulo 2 trata sobre algoritmos recursivos y el análisis de sus complejidades.

El Capítulo 3 versa sobre un problema que parece difícil, pero resulta ser sencillo: el problema de flujo en redes. Gracias a investigaciones muy recientes, existen algoritmos rápidos para los problemas de flujo en redes, y estos tienen muchas aplicaciones importantes.

En el Capítulo 4 estudiamos algoritmos en una de las ramas más antiguas de las matemáticas, la teoría de números. Es notable que las conexiones entre esta

Las conexiones entre este tema antiguo y la investigación más moderna en métodos computacionales son muy fuertes.

En el Capítulo 5 veremos que existe una gran familia de problemas, que incluye varias cuestiones computacionales muy importantes, unidas por una notable unidad estructural. No sabemos si son difíciles o fáciles. Sí sabemos que aún no hemos encontrado una forma rápida de resolverlos, y la mayoría de la gente sospecha que son difíciles. También sabemos que, si alguno de estos problemas resulta ser difícil, entonces todos lo son, y si alguno resulta ser fácil, entonces todos lo son.

Esperamos que, después de haber descubierto algo sobre lo que las personas saben y lo que no saben, el lector haya disfrutado del recorrido por este tema y pueda estar interesado en ayudar a descubrir un poco más.

3. Preliminares Matemáticos

3.1. 1.1 Órdenes de magnitud

En esta sección vamos a analizar las tasas de crecimiento de distintas funciones y a presentar los cinco símbolos de la asintótica que se utilizan para describir esas tasas de crecimiento.

En el contexto de los algoritmos, el propósito de esta discusión es disponer de un buen lenguaje que nos permita comparar la rapidez con la que diferentes algoritmos realizan la misma tarea, la cantidad de memoria que utilizan o cualquier otra medida de complejidad que estemos empleando.

Supongamos que disponemos de un procedimiento para invertir matrices cuadradas no singulares. ¿Cómo podríamos medir su velocidad? Lo habitual sería decir algo como: «si la matriz es de tamaño $n \times n$, entonces el método se ejecuta en un tiempo $16{,}8\,n^3$.» Con ello sabríamos que, si una matriz de 100×100 puede invertirse, con este método, en 1 minuto de tiempo de computador, entonces una matriz de 200×200 requeriría $2^3 = 8$ veces más tiempo, es decir, unos 8 minutos. La constante "16.8" no intervino en absoluto en este ejemplo; lo relevante fue únicamente que el trabajo crece como la tercera potencia del tamaño de la matriz.

Por tanto, necesitamos un lenguaje que nos permita afirmar que el tiempo de cómputo, como función de n, crece «del orden de n^3», o «a lo sumo tan rápido como n^3», o «al menos tan rápido como $n^5 \log n$», etcétera.

Los símbolos que se emplean en este lenguaje de comparación de tasas de crecimiento son los siguientes cinco: «o» (se lee «es pequeña o de»), «O» (se lee «es gran O de»), «Θ» (se lee «es theta de»), «$\sim$» (se lee «es asintóticamente igual a», o de manera menos formal, «twiddles») y «Ω» (se lee «es omega de»).

Ahora expliquemos qué significa cada uno de ellos. Sean $f(x)$ y $g(x)$ dos funciones de x. Cada uno de los cinco símbolos anteriores permite comparar la rapidez de crecimiento de f y g. Si decimos que

$$f(x) = o\big(g(x)\big),$$

entonces, de forma informal, estamos diciendo que f crece más lentamente que g cuando x es muy grande. De manera formal, la definición se enuncia así:

Definición 1.1. Decimos que $f(x) = o\big(g(x)\big)$ $(x \to \infty)$ si

$$\lim_{x\to\infty} \frac{f(x)}{g(x)} = 0.$$

Aquí van algunos ejemplos:

(a) $x^2 = o(x^5)$

(b) $\sin x = o(x)$

(c) $14{,}709\sqrt{x} = o\big(x/2 + 7\cos x\big)$

(d) $\dfrac{1}{x} = o(1)$ (?)

(e) $23 \log x = o\big(x^{0{,}02}\big)$

A partir de estos pocos ejemplos se puede ver que, a veces, demostrar que existe una relación de tipo 'o' puede ser sencillo y, en otras ocasiones, bastante difícil. El ejemplo (e), por ejemplo, requiere el uso de la regla de *L'Hôpital*.

Si disponemos de dos programas de ordenador y uno de ellos invierte matrices $n \times n$ en tiempo $635n^3$, mientras que el otro lo hace en tiempo $o\big(n^{2{,}8}\big)$, sabemos que, para valores suficientemente grandes de n, la garantía de rendimiento del segundo programa será superior a la del primero. Por supuesto, el primer programa podría ejecutarse más rápido en matrices pequeñas, digamos de hasta tamaño $10\,000 \times 10\,000$. Si cierto programa se ejecuta en tiempo $n^{2{,}03}$ y alguien produce otro programa para el mismo problema que corre en tiempo $o\big(n^2 \log n\big)$, entonces ese segundo programa supondría una mejora, al menos en el sentido teórico. La razón de la calificación "teórico", una vez más, es que el segundo programa sería superior solo si n fuera suficientemente grande.

El segundo símbolo del vocabulario de la asintótica es la 'O'. Cuando decimos que $f(x) = O\big(g(x)\big)$ queremos decir, de forma informal, que f ciertamente no crece más rápido que g; podría crecer a la misma velocidad o más despacio: ambas posibilidades las permite la 'O'. De manera formal, tenemos la siguiente definición:

Definición 1.2. Decimos que $f(x) = O\big(g(x)\big)$ $(x \to \infty)$ si existen C, x_0 tales que

$$|f(x)| < Cg(x) \quad \text{para todo } x > x_0.$$

El calificativo "$x \to \infty$" se omitirá normalmente, pues se entenderá que la mayoría de las veces estaremos interesados en valores grandes de las variables involucradas.

Por ejemplo, ciertamente $\sin x = O(x)$, pero aún se puede afirmar más, a saber, que $\sin x = O(1)$. Asimismo,

$$x^3 + 5x^2 + 77\cos x = O\big(x^5\big).$$

y $1/(1+x^2) = O(1)$. Ahora podemos ver cómo la 'o' aporta información más precisa que la 'O', pues podemos afinar el último ejemplo diciendo que $1/(1+x^2) = o(1)$. Esto es más preciso porque no solo nos indica que la función está acotada cuando x es grande, sino que también aprendemos que la función en realidad se aproxima a 0 cuando $x \to \infty$.

Esto es típico de la relación entre O y o. Con frecuencia, un resultado en 'O' es suficiente para una aplicación. Sin embargo, es posible que no sea así y que necesitemos la estimación más precisa en 'o'.

El tercer símbolo del lenguaje de la asintótica es el 'Θ'.

Definición 1.3. Decimos que $f(x) = \Theta(g(x))$ si existen constantes $c_1 > 0$, $c_2 > 0$, x_0 tales que para todo $x > x_0$ se cumple $c_1\, g(x) < f(x) < c_2\, g(x)$.

Entonces podemos decir que f y g tienen la misma tasa de crecimiento; sólo las constantes multiplicativas son inciertas. Algunos ejemplos del uso de 'Θ' son:

$$\begin{aligned} (x+1)^2 &= \Theta(3x^2), \\ \frac{x^2+5x+7}{5x^3+7x+2} &= \Theta\left(\frac{1}{x}\right), \\ \sqrt{3+\sqrt{2}\,x} &= \Theta\big(x^{1/4}\big), \\ \big(1+\tfrac{3}{x}\big)^x &= \Theta(1). \end{aligned}$$

El 'Θ' es mucho más preciso que el 'O' o el 'o'. Si sabemos que $f(x) = \Theta(x^2)$, entonces conocemos que $f(x)/x^2$ permanece entre dos constantes no nulas para valores de x suficientemente grandes. La tasa de crecimiento de f queda establecida: crece cuadráticamente con x.

El símbolo más preciso de la asintótica es el '$\sim$'. Nos dice que no sólo f y g crecen a la misma tasa, sino que en realidad f/g se aproxima a 1 cuando $x \to \infty$.

Definición 1.4. Decimos que $f(x) \sim g(x)$ si

$$\lim_{x\to\infty} \frac{f(x)}{g(x)} = 1.$$

A continuación se muestran algunos ejemplos:

$$\begin{aligned} x^2 + x &\sim x^2, \\ (3x+1)^4 &\sim 81x^4, \\ \sin\big(\tfrac{1}{x}\big) &\sim \tfrac{1}{x}, \\ \frac{2x^3+5x+7}{x^2+4} &\sim 2x, \\ 2x + 7\log x + \cos x &\sim 2x. \end{aligned}$$

Observemos la importancia de precisar las constantes multiplicativas cuando se usa el símbolo '$\sim$'. Si bien es cierto que $2x^2 = \Theta(x^2)$, no es cierto que $2x^2 \sim x^2$. Dicho sea de paso, también es cierto que $2x^2 = \Theta(17x^2)$, pero hacer tal afirmación es de mal estilo, pues el '17' no aporta información adicional.

El último símbolo del lenguaje asintótico que necesitaremos es el 'Ω'. En pocas palabras, 'Ω' es la negación de 'o'. Es decir, $f(x) = \Omega\big(g(x)\big)$ significa que no es cierto que $f(x) = o\big(g(x)\big)$. En el estudio de algoritmos para computadoras, se emplea 'Ω' cuando queremos expresar que cierto cálculo requiere al menos una cantidad dada de tiempo.

Por ejemplo, podemos multiplicar dos matrices $n \times n$ en tiempo $O(n^3)$. Más adelante veremos cómo hacerlo aún más rápido, en $O(n^{2{,}81})$. Existen métodos incluso más veloces, pero algo es seguro: nadie podrá escribir un programa que multiplique pares de matrices $n \times n$ empleando menos de n^2 pasos computacionales, porque cualquier programa debe leer los datos de entrada y las matrices contienen $2n^2$ entradas. Por lo tanto, un tiempo de cómputo de cn^2 es, sin duda, una cota inferior para la velocidad de cualquier algoritmo general de multiplicación de matrices. Podemos decir, entonces, que el problema de multiplicar dos matrices $n \times n$ requiere $\Omega(n^2)$ tiempo.

La definición exacta de 'Ω' dada arriba es bastante delicada: la presentamos como la negación de algo. ¿Podemos reformularla de manera positiva? Sí, con algo de trabajo (véanse los Ejercicios 6 y 7). Puesto que $f = o(g)$ significa que $f/g \to 0$, afirmar $f = \Omega(g)$ implica que f/g *no* tiende a cero. Si suponemos que g toma solamente valores positivos (lo habitual), decir que f/g no tiende a 0 equivale a afirmar que existe $\varepsilon > 0$ y una sucesión infinita de valores de x, con $x \to \infty$, para los cuales $|f|/g > \varepsilon$. Por tanto, no necesitamos mostrar que $|f|/g > \varepsilon$ para todo x grande, sino únicamente para infinitos valores grandes de x.

Definición 1.5. Decimos que $f(x) = \Omega\big(g(x)\big)$ si existen $\varepsilon > 0$ y una sucesión $x_1, x_2, x_3, \ldots$, que tiende a infinito, tal que para todo j se cumple

$$|f(x_j)| > \varepsilon\, g(x_j).$$

Ahora presentaremos una jerarquía de funciones según sus tasas de crecimiento cuando x es grande. Entre las funciones de uso común que tienden a infinito al crecer x, quizá las más lentas sean $\operatorname{loglog} x$ o incluso $(\operatorname{loglog} x)^{1{,}03}$ y expresiones similares. Es cierto que $\operatorname{loglog} x \to \infty$ cuando $x \to \infty$, pero lo hace muy despacio: cuando $x = 1\,000\,000$, por ejemplo, $\operatorname{loglog} x = 2{,}6$.

Un poco más rápida que estas "babosas" es $\log x$ en sí. Al fin y al cabo, $\log 1\,000\,000 = 13{,}8$. De modo que, si tuviéramos un algoritmo de computadora que

Se podría realizar n operaciones en tiempo $\log n$ y alguien encontrara otro método que hiciera el mismo trabajo en tiempo $O(\log \log n)$; entonces, el segundo método, *ceteris paribus*, sería ciertamente una mejora, pero n tendría que ser extremadamente grande para que se notara dicha ventaja.

En la escala de rapidez de crecimiento, a continuación de los logaritmos, encontramos las potencias de x. Por ejemplo, consideremos $x^{0,01}$. Esta función crece más rápido que $\log x$, aunque no lo parecería si sustituyéramos algunos valores de x y comparáramos los resultados (véase el Ejercicio 1 al final de esta sección).

¿Cómo demostrar que $x^{0,01}$ crece más rápido que $\log x$? Mediante la regla de L'Hôpital.

Ejemplo 1.1. Consideremos el límite de $x^{0,01}/\log x$ cuando $x \to \infty$. A medida que $x \to \infty$, el cociente adopta la forma indeterminada ∞/∞, de modo que podemos aplicar la regla de L'Hôpital: derivamos el numerador, derivamos el denominador y volvemos a tomar el límite $x \to \infty$. Así, en lugar del cociente original obtenemos

$$\frac{0{,}01\, x^{-0,99}}{1/x} = 0{,}01\, x^{0,01},$$

el cual, evidentemente, crece sin cota superior cuando $x \to \infty$. Por lo tanto, el cociente original $x^{0,01}/\log x$ también crece sin límite. Hemos demostrado, con precisión, que $\log x = o\big(x^{0,01}\big)$; en ese sentido, podemos afirmar que $x^{0,01}$ crece más rápido que $\log x$.

Para continuar ascendiendo en la escala de tasas de crecimiento, nos encontramos con $x^{0,2}$, x, x^{15}, $x^{15}\log^2 x$, etcétera. Luego aparecen funciones que crecen más rápido que cualquier potencia fija de x, del mismo modo que $\log x$ crece más lento que toda potencia fija de x.

Consideremos $e^{\log^2 x}$. Como esto es lo mismo que $x^{\log x}$, crecerá más rápido que x^{1000}; de hecho, será mayor que x^{1000} tan pronto como $\log x > 1000$, es decir, tan pronto como $x > e^{1000}$ (¡no contengas la respiración!).

Así pues, $e^{\log^2 x}$ es un ejemplo de función que crece más rápido que toda potencia fija de x. Otro ejemplo es $e^{\sqrt{x}}$ (¿por qué?).

Definición 1.6. Decimos que una función presenta *crecimiento exponencial moderado* si crece más rápido que x^a, para toda constante a, pero más lento que c^x, para toda constante $c > 1$. Más precisamente, $f(x)$ es de crecimiento exponencial moderado si para todo $a > 0$ se cumple $f(x) = \Omega\big(x^a\big)$ y para todo $\varepsilon > 0$ se cumple $f(x) = o\big((1+\varepsilon)^x\big)$.

Más allá del rango de crecimiento exponencial moderado están las funciones que crecen exponencialmente rápido. Ejemplos típicos de tales funciones son $(1{,}03)^x$, 2^x, x^{97x}, y así sucesivamente. Formalmente, tenemos:

Definición 1.7. Una función f es de *crecimiento exponencial* si existe $c > 1$ tal que $f(x) = \Omega(c^x)$ y existe d tal que $f(x) = O(d^x)$.

Si "adornamos" una función de crecimiento exponencial con funciones menores, no cambiaremos el hecho de que sigue siendo de crecimiento exponencial. Así,

$$e^{\sqrt{x}+2x/(x^{49}+37)}$$

continúa siendo de crecimiento exponencial, porque e^{2x} ya lo es por sí misma y resiste los intentos de las funciones menores por alterar su comportamiento.

Más allá de las funciones de crecimiento exponencial existen funciones que crecen tan rápido como se desee. Por ejemplo, $n!$, que crece más rápido que c^n para toda constante fija c, y 2^{n^2}, que crece mucho más rápido que $n!$. Los rangos de crecimiento que más preocupan a los científicos de la computación están "entre" las funciones logarítmicas, muy lentas, y las de crecimiento exponencial. La razón es simple: si un algoritmo requiere más de una cantidad exponencial de tiempo para realizar su tarea, probablemente no se usará, o se usará solo en circunstancias muy inusuales. En este libro todos los algoritmos que veremos caen dentro de ese rango intermedio.

Ya hemos discutido los distintos símbolos asintóticos que se emplean para comparar tasas de crecimiento de pares de funciones y la jerarquía de rapidez de crecimiento, de modo que contamos con un pequeño catálogo de funciones que crecen lentamente, medianamente rápido, rápido y muy rápido. A continuación, estudiaremos el crecimiento de sumas que involucran funciones elementales, con el fin de descubrir a qué ritmo crecen dichas sumas.

Pensemos en la siguiente expresión:

$$f(n) = \sum_{j=0}^{n} j^2 = 1^2 + 2^2 + 3^2 + \cdots + n^2. \tag{1.1}$$

Es decir, $f(n)$ es la suma de los cuadrados de los primeros n enteros positivos.

¿Qué tan rápido crece $f(n)$ cuando n es grande?

Observemos de inmediato que, entre los n términos de la suma que define $f(n)$, el mayor es el último, a saber, n^2. Como hay n términos y el mayor vale apenas n^2, es evidente que $f(n) = O(n^3)$, e incluso que $f(n) \leq n^3$ para todo $n \geq 1$.

Supongamos que deseamos información más precisa sobre el crecimiento de $f(n)$, por ejemplo una afirmación del estilo $f(n) \sim ?$. ¿Cómo podríamos lograr una estimación mejor?

La mejor manera de empezar es visualizar la suma en (**??**), como se muestra en la Figura 1.1. En la figura se ve la curva $y = x^2$ en el plano x–y. Además, se dibuja un rectángulo sobre cada intervalo de longitud unitaria desde $x = 1$ hasta $x = n$. Todos los rectángulos quedan bajo la curva.

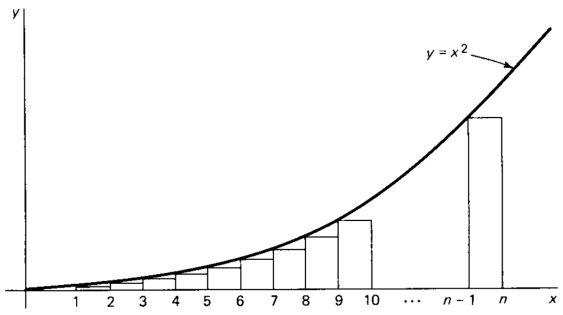

Figura 3: Cómo sobreestimar una suma.

En consecuencia, el área total de todos los rectángulos es menor que el área bajo la curva, lo cual se expresa como:

$$\sum_{j=1}^{n-1} j^2 \;\leq\; \int_1^n x^2\,dx \;=\; \frac{n^3-1}{3}. \tag{1.2}$$

Si comparamos (**??**) con (**??**), observamos que hemos demostrado que

$$f(n) \leq \frac{(n+1)^3-1}{3}.$$

Ahora obtendremos una cota inferior para $f(n)$ de la misma manera. Esta vez empleamos la configuración de la Figura 1.2, donde nuevamente aparece la curva $y = x^2$, pero los rectángulos se han dibujado *encima* de la curva. A partir de la imagen vemos de inmediato que

$$1^2 + 2^2 + \cdots + n^2 \;\geq\; \int_0^n x^2\,dx \;=\; \frac{n^3}{3}. \tag{1.3}$$

Hemos acotado nuestra función $f(n)$ por ambos lados, y de forma bastante estrecha. Sabemos que

$$\forall n \geq 1: \quad \frac{n^3}{3} \;\leq\; f(n) \;\leq\; \frac{(n+1)^3-1}{3}.$$

De aquí obtenemos inmediatamente que $f(n) \sim n^3/3$, lo que nos brinda una idea bastante precisa de la tasa de crecimiento de $f(n)$ cuando n es grande. El lector habrá notado también que el símbolo '$\sim$' ofrece una estimación de crecimiento mucho más satisfactoria que la de 'O'.

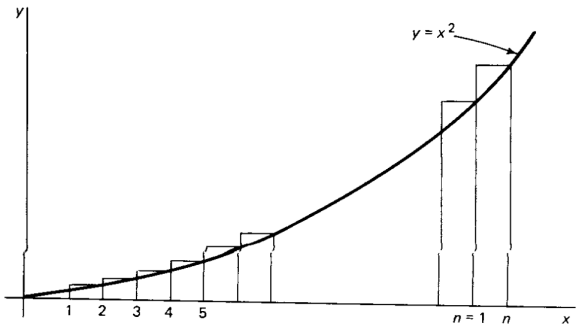

Figura 4: Cómo subestimar una suma.

Formulemos un principio general, para estimar el tamaño de una suma, que produzca estimaciones como las anteriores sin exigirnos cada vez visualizar figuras como las Figuras 1.1 y 1.2. La idea general es que, cuando se enfrenta la tarea de estimar las tasas de crecimiento de las sumas, se debe intentar comparar las sumas con integrales, porque normalmente son más fáciles de manejar.

Sea una función $g(n)$ definida para valores enteros no negativos de n, y supongamos que $g(n)$ es no decreciente. Queremos estimar el crecimiento de la suma:

$$G(n) = \sum_{j=1}^{n} g(j) \qquad (n = 1, 2, \ldots). \tag{1.4}$$

Considérese un diagrama que luce exactamente como la Figura 1.1 salvo que la curva mostrada es ahora la curva $y = g(x)$. La suma de las áreas de los rectángulos es exactamente $G(n-1)$, mientras que el área bajo la curva entre 1 y n es $\int_1^n g(t)\,dt$. Puesto que los rectángulos quedan totalmente debajo de la curva, sus áreas combinadas no pueden exceder el área bajo la curva, y tenemos la desigualdad:

$$G(n-1) \leq \int_1^n g(t)\,dt \qquad (n \geq 1). \tag{1.5}$$

Por otro lado, si consideramos la Figura 1.2, donde la gráfica es de nuevo la gráfica de $y = g(x)$, el hecho de que las áreas combinadas de los rectángulos no sean ahora menores que el área bajo la curva produce la desigualdad:

$$G(n) \geq \int_0^n g(t)\,dt \qquad (n \geq 1). \tag{1.6}$$

Si combinamos (1.5) y (1.6) vemos que hemos completado la demostración de:

Teorema 1. *Sea $g(x)$ no decreciente para x no negativos. Entonces*

$$\int_0^n g(t)\,dt \;\le\; \sum_{j=1}^{n} g(j) \;\le\; \int_1^{n+1} g(t)\,dt. \tag{1.7}$$

El teorema anterior es capaz de producir estimaciones bastante satisfactorias con muy poco esfuerzo, como muestra el siguiente ejemplo.

Sea $g(n) = \log n$ y sustitúyase en (1.7). Tras efectuar las integrales, obtenemos:

$$n\log n - n \;\le\; \sum_{j=1}^{n} \log j \;\le\; (n+1)\log(n+1) - n. \tag{1.8}$$

Reconocemos que el término central anterior es $\log n!$, y por lo tanto, al exponentiar (1.8) tenemos

$$\left(\frac{n}{e}\right)^n \;\le\; n! \;\le\; \frac{(n+1)^{\,n+1}}{e^{\,n}}. \tag{1.9}$$

Esta es una estimación bastante buena del crecimiento de $n!$, pues el término derecho es solo aproximadamente ne veces mayor que el izquierdo (¿por qué?), cuando n es grande.

Mediante el uso de herramientas ligeramente más precisas puede demostrarse una mejor estimación del tamaño de $n!$ llamada *fórmula de Stirling*, que establece que:

$$x! \;\sim\; \left(\frac{x}{e}\right)^x \sqrt{2\pi x}. \tag{1.10}$$

3.1.1. Ejercicios 1.1.1

1. Calcule los valores de $x^{0,01}$ y de $\log x$ para $x = 10,\ 1000,\ 1\,000\,000$. Encuentre un valor único de $x > 10$ para el cual $x^{0,01} > \log x$, y demuestre que su respuesta es correcta.

2. Algunas de las siguientes afirmaciones son verdaderas y otras falsas. ¿Cuáles son cuáles?

   (a) $(x^2 + 3x + 1)^3 \sim x^6$

   (b) $\dfrac{(\sqrt{x}+1)^3}{x^2+1} = o(1)$

(a) $e^{1/x} = \Theta(1)$

(b) $\dfrac{1}{x} \sim 0$

(c) $x^3(\log\log x)^2 = o(x^3 \log x)$

(d) $\sqrt{\log x} + 1 = \Omega(\log\log x)$

(e) $\sin x = \Omega(1)$

(f) $\dfrac{\cos x}{x} = O(1)$

(g) $\displaystyle\int_4^x \frac{dt}{t} \sim \log x$

(h) $\displaystyle\int_0^x e^{-t^2}\, dt = O(1)$

(i) $\displaystyle\sum_{j \le x} \frac{1}{j^2} = o(1)$

(j) $\displaystyle\sum_{j \le x} 1 \sim x$

3. Cada una de las tres sumas siguientes define una función de x. Bajo cada suma aparece una lista de cinco afirmaciones sobre la tasa de crecimiento, cuando $x \to \infty$, de la función que la suma define. En cada caso, indique cuál (o cuáles) de las cinco opciones es verdadera *(puede haber más de una verdadera)*.

   - $h_1(x) = \displaystyle\sum_{j \le x} \left(\frac{1}{j} + \frac{3}{j^2} + \frac{4}{j^3} \right)$
     (a) $\sim \log x$
     (b) $= O(x)$
     (c) $\sim 2 \log x$
     (d) $= \Theta(\log x)$
     (e) $= \Omega(1)$
   - $h_2(x) = \displaystyle\sum_{j \le \sqrt{x}} (\log j + j)$
     (a) $\sim x/2$
     (b) $= O(\sqrt{x})$
     (c) $= \Theta(\sqrt{x} \log x)$
     (d) $= \Omega(\sqrt{x})$
     (e) $= o(\sqrt{x})$
   - $h_3(x) = \displaystyle\sum_{j \le \sqrt{x}} \frac{1}{\lceil j \rceil}$

(a) $= O(\sqrt{x})$

(b) $= \Omega(x^{1/4})$

(c) $= o(x^{1/4})$

(d) $\sim 2x^{1/4}$

(e) $= \Theta(x^{1/4})$

4. De los cinco símbolos de asintótica O, o, $\sim$, Θ, Ω, ¿cuáles son transitivos? *(por ejemplo, si $f = O(g)$ y $g = O(h)$, entonces $f = O(h)$?)*

5. El objetivo de este ejercicio es mostrar que, si f crece más lentamente que g, siempre podemos encontrar una tercera función h cuya tasa de crecimiento se sitúe entre la de f y la de g. Demuestre lo siguiente: si $f = o(g)$, entonces existe una función h tal que $f = o(h)$ y $h = o(g)$. Dé una construcción explícita de h en términos de f y de g.

6. *Calentamiento para el ejercicio 7.* A continuación aparecen varias proposiciones matemáticas. En cada caso, escriba una proposición que sea la negación de la dada.

Además, en la negación, no utilices la palabra «no» ni ningún símbolo de negación. En cada caso la pregunta es: "Si esto no es verdadero, ¿entonces qué es verdadero?"

(a) $\exists x > 0 \ \ni \ f(x) = 0$

(b) $\forall x > 0, \ f(x) > 0$

(c) $\forall x > 0, \ \exists \varepsilon > 0 \ \ni \ f(x) < \varepsilon$

(d) $\exists x = 0 \ \ni \ \forall y < 0, \ f(y) < f(x)$

(e) $\forall x \ \exists y \ \ni \ \forall z : \ g(x) < f(y)f(z)$

(f) $\forall \varepsilon > 0 \ \exists x \ \ni \ \forall y > x : \ f(y) < \varepsilon$

¿Puedes formular un método general para negar tales proposiciones? Dada una proposición que contenga «∀», «∃», «∋», ¿qué regla aplicarías para negarla y dejar el resultado en forma positiva (sin símbolos de negación ni «no»)?

1. En este ejercicio elaboraremos la definición de 'Ω'.

(a) Escribe la definición precisa de la afirmación $\lim\limits_{x\to\infty} h(x) = 0$ (usa ε's).

(b) Escribe la negación de tu respuesta al apartado (a) como una afirmación positiva.

(c) Utiliza tu respuesta al apartado (b) para dar una definición positiva de la afirmación $f(x) = o\big(g(x)\big)$ y, con ello, justifica la definición del símbolo 'Ω' dada en el texto.

2. Ordena las siguientes funciones de menor a mayor tasa de crecimiento para n grande, de modo que cada una sea 'little-oh' de su sucesora: $2^{\sqrt{n}}$, $e^{\log n^3}$, $n^{3,01}$, 2^{n^2}, $n^{1,6}$, $\log n^3 + 1$, $\sqrt{n!}$, $n^3 \log n$, $n^{\log n}$, $(\log\log n)^3$, $n^{0,5}2^n$, $(n+4)^{12}$.

3. Encuentra una función $f(x)$ tal que $f(x) = \mathcal{O}\big(x^{1+\varepsilon}\big)$ sea verdadera para todo $\varepsilon > 0$, pero para la cual no sea cierto que $f(x) = \mathcal{O}(x)$.

4. Demuestra que la afirmación $f(n) = \mathcal{O}\big((2+\varepsilon)^n\big)$ para todo $\varepsilon > 0$ es equivalente a la afirmación $f(n) = o\big((2+\varepsilon)^n\big)$ para todo $\varepsilon > 0$.

3.2. Sistemas numéricos posicionales

En esta sección se ofrece una breve revisión de la representación de números en distintas bases. El sistema decimal habitual representa los números utilizando los

dígitos $0, 1, \ldots, 9$. Con el propósito de representar números enteros, podemos imaginar que las potencias de 10 se muestran ante nosotros de la siguiente forma:

$$\ldots, 100000, 10000, 1000, 100, 10, 1.$$

A continuación, para representar un número entero, podemos especificar cuántas copias de cada potencia de 10 deseamos. Si escribimos 237, por ejemplo, significa que queremos 2 centenas, 3 decenas y 7 unidades.

En general, si escribimos la cadena de dígitos que representa un número en el sistema decimal, como $d_m d_{m-1} \ldots d_1 d_0$, entonces el número representado por dicha cadena es

$$n = \sum_{i=0}^{m} d_i 10^i.$$

Probemos ahora con el sistema binario. En lugar de usar 10, utilizaremos 2. Por lo tanto, imaginamos que las potencias de 2 se muestran ante nosotros como:

$$\ldots, 512, 256, 128, 64, 32, 16, 8, 4, 2, 1.$$

Para representar un número, especificaremos cuántas copias de cada potencia de 2 queremos. Por ejemplo, si escribimos 1101, entonces necesitamos un 8, un 4 y un 1; por lo tanto, este debe ser el número decimal 13. Escribiremos

$$(13)_{10} = (1101)_2$$

para indicar que el número 13 en base 10 es el mismo que 1101 en base 2.

En el sistema binario (base 2) los únicos dígitos necesarios son 0 y 1. Esto significa que, si usamos solo 0 y 1, podremos representar cada número n de manera única. La representación única de cada número es, después de todo, lo que debemos esperar y exigir de cualquier sistema propuesto.

Profundicemos en este último punto. Si se nos permitiera usar más dígitos que solo 0 y 1, entonces podríamos representar el número $(13)_{10}$ en binario de muchas maneras. Por ejemplo, podríamos cometer el error de permitir los dígitos $0, 1, 2, 3$. Entonces 13 sería representable por $3 \cdot 2^2 + 1 \cdot 2^0$ o por $2 \cdot 2^2 + 2 \cdot 2^1 + 1 \cdot 2^0$, etc.

Si permitiéramos demasiados dígitos distintos, los números serían representables de más de una forma mediante una cadena de dígitos.

Si, en cambio, permitiéramos muy pocos dígitos, descubriríamos que algunos números no tendrían representación alguna. Por ejemplo, si usáramos el sistema decimal con solo los dígitos $0, 1, \ldots, 8$, entonces infinitos números no podrían representarse, así que será mejor mantener el 9.

La proposición general es la siguiente:

Teorema 1.9. Sea $b > 1$ un entero positivo (la *base*). Entonces todo entero positivo n puede escribirse de una y sólo una forma como

$$n = d_0 + d_1 b + d_2 b^2 + d_3 b^3 + \dots$$

si los dígitos $d_0, d_1, \dots$ satisfacen $0 \leq d_i \leq b - 1$ para todo i.

Observación. El teorema afirma, por ejemplo, que en base 10 necesitamos los dígitos $0, 1, \dots, 9$; en base 2 necesitamos sólo 0 y 1; en base 16 necesitamos dieciséis dígitos, etc.

Demostración. Fijada la base b, la prueba es por inducción sobre n, el número que se representa. Es claro que el número 1 puede representarse de una y sólo una forma con los dígitos disponibles (¿por qué?). Supongamos inductivamente que cada entero $1, 2, \dots, n-1$ es representable de manera única. Consideremos ahora el entero n. Definamos

$$d = n \text{ mód } b.$$

Entonces d es uno de los b dígitos permitidos. Por hipótesis de inducción, el número

$$n_0 = \frac{n - d}{b}$$

es representable de manera única; esto es,

$$\frac{n - d}{b} = d_0 + d_1 b + d_2 b^2 + \dots$$

Por lo tanto,

$$n = d + \frac{n - d}{b}\, b = d + d_0 b + d_1 b^2 + d_2 b^3 + \dots$$

es una representación de n que utiliza sólo los dígitos permitidos.

Supongamos, finalmente, que n posee otra representación de esta forma. Entonces

$$n = a_0 + a_1 b + a_2 b^2 + \cdots = c_0 + c_1 b + c_2 b^2 + \dots$$

Como a_0 y c_0 son ambos iguales a n mód b, concluimos que $a_0 = c_0$. En consecuencia,

$$n_0 = \frac{n - a_0}{b}$$

tendría dos representaciones diferentes, contradiciendo la hipótesis inductiva, pues hemos supuesto la veracidad del resultado para todo $n_0 < n$. □

Los valores típicos de b más utilizados, además del 10, son 2 (*sistema binario*), 8 (*sistema octal*) y 16 (*sistema hexadecimal*).

El sistema binario es extremadamente sencillo porque utiliza solo dos dígitos. Esto es muy conveniente si eres una computadora o un diseñador de computadoras, pues los dígitos pueden determinarse según que un componente esté 'encendido' (dígito 1) o 'apagado' (dígito 0). Los dígitos binarios de un número se llaman sus *bits* o su *cadena de bits*.

El sistema octal es popular porque ofrece una buena manera de recordar y manejar las largas cadenas de bits que produce el sistema binario. De acuerdo con el teorema, en el sistema octal los dígitos necesarios son $0, 1, \ldots, 7$. Por ejemplo:

$$(735)_8 = (477)_{10}.$$

La característica atractiva del sistema octal es la facilidad con la que podemos convertir entre octal y binario. Si se nos da la cadena de bits de un entero n, para convertirla a octal basta con agrupar los bits en grupos de tres, empezando por el bit menos significativo, y luego convertir cada grupo de tres bits, de forma independiente, en un único dígito octal. A la inversa, si se nos da la forma octal de n, la forma binaria se obtiene convirtiendo cada dígito octal, de manera independiente, en los tres bits que lo representan en el sistema binario. Por ejemplo, dado $(1101100101)_2$. Para convertir este número binario a octal, agrupamos los bits de tres en tres,

$$(1)(101)(100)(101),$$

comenzando desde la derecha, y luego convertimos cada triple en un único dígito octal, obteniendo así:

$$(1101100101)_2 = (1545)_8.$$

Si eres un programador en activo, resulta muy práctico usar las cadenas octales más cortas para recordar o anotar las cadenas binarias más largas, debido al ahorro de espacio unido a la facilidad de conversión en ambos sentidos.

El sistema hexadecimal (base 16) es como el octal, pero aún más. La conversión de ida y vuelta a binario ahora utiliza grupos de *cuatro* bits, en lugar de tres. En hexadecimal necesitaremos, según el teorema anterior, 16 dígitos. Disponemos de nombres prácticos para los primeros 10, pero ¿cómo denominaremos a los 'dígitos 10 a 15'? Los nombres que se usan convencionalmente para ellos son 'A', 'B', ..., 'F'. Por ejemplo:

$$(A52C)_{16} \;=\; 10(4096) + 5(256) + 2(16) + 12 \;=\; (42284)_{10}.$$

Por otro lado, si convertimos cada dígito hexadecimal, de manera independiente, a base 2, encontramos que

$$(A52C)_{16} = (1010)_2(0101)_2(0010)_2(1100)_2 = (1010010100101100)_2.$$

Finalmente, si **agrupamos los bits en grupos de tres**, podemos convertir a octal del modo siguiente:

$$(1010010100101100)_2 = (1)(010)(010)(100)(101)(100) = (122454)_8.$$

3.2.1. Ejercicios 1.2.1

1. Demuestre que la conversión de octal a binario se realiza correctamente convirtiendo cada dígito octal en un trío binario y concatenando los tríos resultantes. Generalice este teorema a otros pares de bases.

2. Efectúe las conversiones indicadas.

   (a) $(737)_{10} = ?_{\ 3}$

   (b) $(101100)_2 = ?_{\ 16}$

   (c) $(3377)_8 = ?_{\ 16}$

   (d) $(\text{ABCD})_{16} = ?_{\ 10}$

   (e) $(\text{BEEF})_{16} = ?_{\ 8}$

3. Escriba un procedimiento `convert (n, b:integer, digitstr:string)` que devuelva la cadena de dígitos que representa n en la base b.

3.3. Manipulaciones con series

En esta sección estudiaremos operaciones con series de potencias, incluyendo su multiplicación y la obtención de sus sumas en forma sencilla. Comenzamos con un pequeño catálogo de series de potencias que conviene conocer. Primero tenemos la *serie geométrica finita*:

$$\frac{1-x^n}{1-x} = 1 + x + x^2 + \cdots + x^{n-1}. \tag{1.11}$$

Esta ecuación es válida, sin duda, para todo $x \neq 1$, y sigue siendo cierta cuando $x = 1$ si tomamos el límite indicado en el lado izquierdo.

¿Por qué es cierta (1.11)? Basta multiplicar ambos lados por $1 - x$ para eliminar fracciones. El resultado es

$$\begin{aligned} 1 - x^n &= (1 + x + x^2 + x^3 + \cdots + x^{n-1})(1 - x) \\ &= (1 + x + x^2 + \cdots + x^{n-1}) - (x + x^2 + x^3 + \cdots + x^n) \\ &= 1 - x^n, \end{aligned}$$

y la demostración queda concluida.

Página 35

Página 36

Página 37

Página 38

Página 39

Página 40

Página 41

Página 42

Página 43

Página 44

Página 45

Página 46

Página 47

Página 48

Página 49

Página 50