Universidad
Rey Juan Carlos

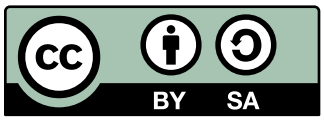
Apuntes

Asignatura: Diseño y Análisis de Algoritmos

Grado en Ingeniería Informática

13 de mayo de 2024

Autor: Manuel Rubio Sánchez



Prefacio

Este documento contiene apuntes relacionados con los siguientes temas de la asignatura Diseño y Análisis de Algoritmos, del Grado en Ingeniería Informática de la Universidad Rey Juan Carlos, impartida durante el curso 2023-24:

- Tema 1. Introducción a la Algoritmia (parcialmente)
- Tema 2.1. Preliminares Matemáticos
- Tema 2.2. Notaciones Asintóticas
- Tema 3.1. Análisis de Algoritmos Iterativos
- Tema 3.2. Análisis de Algoritmos Recursivos

Los temas 4, 5 y 6 se cubren en el libro:

- Introduction to Recursive Programming. Manuel Rubio-Sánchez. Taylor & Francis. 2018.

Índice de contenidos

Índice de figuras

Índice de códigos

1

Preliminares Matemáticos

Los algoritmos pueden entenderse como procedimientos para resolver problemas computacionales, los cuales a menudo son de carácter matemático, o cuyos enunciados vienen expresados en términos matemáticos. Este capítulo ofrece una visión general de fundamentos y notación matemática esencial. Ésta servirá no solo para poder comprender los enunciados de los problemas, sino porque es necesaria de cara al análisis de la eficiencia de algoritmos, el cual es principalmente teórico/matemático.

1.1 Recapitulación de conceptos y notación matemática

Este apartado describe varios conceptos matemáticos que pueden ser necesarios en algunos temas teóricos o en ejercicios prácticos dentro de la asignatura.

1.1.1 Potencias y logaritmos

Las Tablas 1.1 y 1.2 repasan varias propiedades fundamentales de potencias y logaritmos, respectivamente. Es importante tener en cuenta que a, b, x e y son números reales con las siguientes restricciones: (1) la base de un logaritmo debe ser positiva y distinta de 1, (2) un logaritmo sólo se define para valores positivos (es decir, su argumento debe ser positivo), y (3) el denominador de una fracción no puede ser 0. Por ejemplo, $\log_b x = \log_a x / \log_a b$ sólo es válida para $a > 0$ con $a \neq 1$, $b > 0$ con $b \neq 1$, y $x > 0$.

Potencias	
• $b^1 = b$	• $b^0 = 1$
• $b^x b^y = b^{x+y}$	• $(b^x)^y = b^{xy} = (b^y)^x$
• $b^{-x} = 1/b^x$	• $(ab)^x = a^x b^x$

Tabla 1.1: Propiedades básicas de potencias.

Logaritmos	
• $\log_b b = 1$	• $\log_b 1 = 0$
• $\log_b(xy) = \log_b(x) + \log_b(y)$	• $\log_b(x/y) = \log_b(x) - \log_b(y)$
• $\log_b(x^y) = y \log_b x$	• $\log_b x = \log_a x / \log_a b$
• $\log_b a = 1/\log_a b$	• $x^{\log_b y} = y^{\log_b x}$
• $\log_b(b^x) = x$	• $b^{\log_b a} = a$

Tabla 1.2: Propiedades básicas de logaritmos.

Las potencias y logaritmos con base mayor que 1, de números positivos, son funciones monótonas crecientes. Por tanto, si $0 < x \leq y$ entonces $\log_b x \leq \log_b y$, y $b^x \leq b^y$ (siempre que $b > 1$).

1.1.2 Coeficientes binomiales

Un coeficiente binomial, denotado como $\binom{n}{m}$, es el coeficiente del término x^{n-m} de la expansión polinómica de $(x+1)^n$. Por ejemplo, para $n = 4$:

$$(x+1)^4 = 1x^4 + 4x^3 + 6x^2 + 4x + 1 = \binom{4}{0}x^4 + \binom{4}{1}x^3 + \binom{4}{2}x^2 + \binom{4}{3}x + \binom{4}{4}. \quad (1.1)$$

Se puede definir como:

$$\binom{n}{m} = \begin{cases} 1 & \text{si } m = 0 \text{ o } n = m, \\ \dfrac{n!}{m!(n-m)!} & \text{en otro caso,} \end{cases}$$

donde n y m son enteros que satisfacen $n \geq m \geq 0$. Además, un coeficiente binomial se

puede definir de manera recursiva mediante:

$$\binom{n}{m} = \begin{cases} 1 & \text{si } m = 0 \text{ o } n = m, \\ \binom{n-1}{m-1} + \binom{n-1}{m} & \text{en otro caso.} \end{cases}$$

Los coeficientes binomiales juegan un papel importante en la combinatoria. En concreto, $\binom{n}{m}$ determina el número de *combinaciones* de n elementos tomados de m en m. Es decir, es el número de formas de en las que es posible seleccionar m elementos de un conjunto de n elementos diferentes ($0 \leq m \leq n$), donde el orden en que se seleccionan los elementos no importa.

1.1.3 Funciones de parte entera

Las funciones de parte entera toman un número real y devuelven un número entero próximo, sea por exceso o por defecto.

La *función suelo* de un número real x, denotada como $\lfloor x \rfloor$, devuelve el mayor número entero igual o menor que x. De manera similar, la *función techo* de x, denotada como $\lceil x \rceil$ retorna el número entero más próximo a x por exceso, es decir, el menor número entero igual o mayor que x. Formalmente se pueden definir mediante:

$$\lfloor x \rfloor = \text{máx}\{m \in \mathbb{Z} \mid m \leq x\},$$
$$\lceil x \rceil = \text{mín}\{m \in \mathbb{Z} \mid m \geq x\},$$

donde $\mathbb{Z}$ representa el conjunto de todos los números enteros, y $\mid$ debe interpretarse como "tal que". La siguiente lista incluye algunas propiedades de estas funciones:

- $\lfloor x \rfloor \leq x$
- $\lceil x \rceil \geq x$
- $\lfloor x + n \rfloor = \lfloor x \rfloor + n$
- $\lceil x + n \rceil = \lceil x \rceil + n$
- $\lfloor x \rfloor + \lfloor y \rfloor \leq \lfloor x + y \rfloor$
- $\lceil x \rceil + \lceil y \rceil \geq \lceil x + y \rceil$
- $n = \lfloor n/2 \rfloor + \lceil n/2 \rceil$
- $n - 2\lfloor n/2 \rfloor = 0 \Leftrightarrow n$ es par
- $n \texttt{ // } 2 = \lfloor n/2 \rfloor$
- $n \texttt{ >> } m = \lfloor n/2^m \rfloor$

Además:

- $\lfloor \log_{10} p \rfloor + 1 =$ el número de dígitos decimales de p
- $\lfloor \log_2 p \rfloor + 1 =$ el número de dígitos binarios (bits) de p

donde x es un número real, n es un entero, m un entero no negativo, y p un entero positivo. Además, `//` y `>>` representan operadores en Python (y en otros lenguajes de programación) que calculan el cociente de una división entera, y realizan un desplazamiento de bits a la derecha, respectivamente. Por último, $\Leftrightarrow$ denota "si y sólo si".

Otras funciones de parte entera usadas frecuentemente son el *truncamiento* o *parte entera*, y el *redondeo*. El truncamiento o parte entera de x, denotado como $[x]$, devuelve el número entero resultado de ignorar la parte decimal de x. Se puede definir de la siguiente manera:

$$[x] = \begin{cases} \lceil x \rceil & \text{si } x < 0, \\ \lfloor x \rfloor & \text{si } x \geq 0. \end{cases}$$

Finalmente la función redondeo (redon()) asigna a cada x real el valor entero más próximo a x, .donde si la primera cifra decimal de x es 5 o mayor el redondeo se hace por exceso, y si la primera cifra decimal es inferior a 5 el redondeo se hace por defecto:

$$\text{redon}(x) = \begin{cases} \lceil x - 0{,}5 \rceil & \text{si } x < 0, \\ \lfloor x + 0{,}5 \rfloor & \text{si } x \geq 0. \end{cases}$$

1.1.4 Vectores y matrices

Los vectores y matrices son elementos fundamentales en cualquier ingeniería y campo técnico, que facilitan enormemente la descripción y comunicación de expresiones y resultados matemáticos. Este apartado introduce algunas definiciones y notación, mientras que la Sección 1.1.8 muestra varias relaciones de vectores y matrices con conceptos de geometría.

Vectores

Un vector $\mathbf{v}$ de dimensión finita n se puede interpretar como una colección de n números escalares, ordenados en una lista. Si estos son reales diremos que $\mathbf{v} \in \mathbb{R}^n$. A veces denotaremos los vectores como $\mathbf{v} = (v_1, \ldots, v_n)$, y por defecto se considerarán vectores columna (en relación a las filas y columnas de matrices). Por eso, $\mathbf{v}^\intercal$ denotará un vector fila.

El vector nulo o cero es aquel que tiene todos sus elementos iguales a cero, y se suele escribir con $\mathbf{0}$. Por otro lado, $\mathbf{e}_j$ suele representar el vector con todos sus elementos iguales a cero excepto el j-ésimo, que toma el valor 1.

Geométricamente, un vector es cualquier ente matemático que se puede representar mediante un segmento de recta orientado dentro del espacio euclidiano. Aunque hay que recordar que la palabra *vector* se puede usar en términos más generales para denotar un elemento de un *espacio vectorial* (por ejemplo, un vector podría ser un polinomio), que no se explicará en este documento.

Por último, dado un conjunto finito de n vectores $\{\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n\}$ (pertenecientes a un espacio vectorial), se dice que son linealmente independientes si:

$$a_1\mathbf{v}_1 + a_2\mathbf{v}_2 + \cdots + a_n\mathbf{v}_n = \mathbf{0}$$

se satisface únicamente cuando todos los escalares $a_i = 0$. En caso contrario, se dice

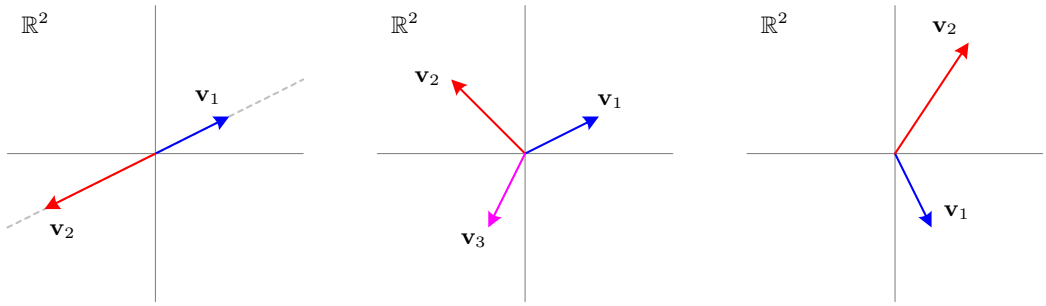


Figura 1.1: Ejemplos de vectores linealmente dependientes en (a) y (b), e independientes en (c), en $\mathbb{R}^2$.

que son linealmente dependientes. La Figura 1.1 muestra varios ejemplos de vectores linealmente dependientes e independientes. Obsérvese que la propiedad hace referencia a todo un conjunto de vectores. No tiene sentido decir que un sólo vector es linealmente dependiente o independiente.

Matrices

De manera similar, una matriz $\mathbf{A}$ de dimensiones $n \times m$ es una colección de nm escalares organizados en una cuadrícula de n filas y m columnas:

$$\mathbf{A} = \begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,m} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,m} \end{bmatrix}$$

donde el elemento $a_{i,j}$ de la matriz sería el ubicado en la fila i y columna j. Si los elementos de la matriz son reales diremos que $\mathbf{A} \in \mathbb{R}^{n \times m}$ (también podrían ser complejos, por ejemplo).

A continuación se describen brevemente varias definiciones y características de matrices con las que conviene estar familiarizado:

- Matriz cero ($\mathbf{0}$)
- Matriz identidad ($\mathbf{I}$)
- Matriz diagonal
- Matriz triangular superior/inferior
- Matriz permutación
- Rango: rg($\mathbf{A}$), número de columnas o filas linealmente independientes de $\mathbf{A}$

- Traza: $\text{tr}(\mathbf{A}) = \sum_{i=1}^{n} a_{i,i}$, donde $\mathbf{A} \in \mathbb{R}^{n \times n}$
- Determinante ($|\mathbf{A}|$ o $\det(\mathbf{A})$)
  - $|\mathbf{ABC}| = |\mathbf{BCA}| = |\mathbf{CAB}|$
- Matriz traspuesta ($\mathbf{A}^\intercal$)
  - $(\mathbf{ABC})^\intercal = \mathbf{C}^\intercal \mathbf{B}^\intercal \mathbf{A}^\intercal$
- Matriz simétrica: $\mathbf{A} = \mathbf{A}^\intercal$
- Matriz inversa: $\mathbf{A}^{-1}$, con $\mathbf{A}$ cuadrada, donde $\text{rg}(\mathbf{A}) = n$, $|\mathbf{A}| \neq 0$
- Matriz ortogonal: $\mathbf{Q}^\intercal \mathbf{Q} = \mathbf{Q}\mathbf{Q}^\intercal = \mathbf{I}$, con $|\det(\mathbf{Q})| = 1$ (y también $\mathbf{Q}^\intercal = \mathbf{Q}^{-1}$)
- Autovalores (λ) y autovectores ($\mathbf{v}$): $\mathbf{Av} = \lambda\mathbf{v}$, con $\mathbf{v} \neq 0$
  - Los autovalores son las raíces del *polinomio característico* $\det(\mathbf{A} - \lambda\mathbf{I}) = 0$
- Descomposiciones importantes:
  - LU: $\mathbf{A} = \mathbf{PLU}$, con $\mathbf{P}$ matriz permutación, $\mathbf{L}$ triangular inferior y $\mathbf{U}$ triangular superior
  - QR: $\mathbf{A} = \mathbf{QR}$, con $\mathbf{Q}$ ortogonal y $\mathbf{R}$ triangular superior
  - Descomposición de autovalores: $\mathbf{A} = \mathbf{V}\mathbf{\Lambda}\mathbf{V}^\intercal$, con $\mathbf{A}$ cuadrada e invertible, $\mathbf{V}$ es la matriz cuyas columnas son los autovectores de $\mathbf{A}$ y $\mathbf{\Lambda}$ es la matriz diagonal con los correspondientes autovalores
  - Descomposición de valores singulares: $\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^\intercal$, con $\mathbf{U}$ y $\mathbf{V}$ ortogonales y $\mathbf{\Sigma}$ diagonal

1.1.5 Límites y la regla de L'Hopital

En primer lugar,

$$\frac{c}{\infty} = 0 \qquad \text{y} \qquad \frac{\infty}{k} = \infty,$$

donde c es una constante cualquiera, k es otra constante positiva, y donde ∞ debe entenderse como el resultado de un límite. Por otro lado $c/0$ no está definido y sólo se puede considerar que es $\pm$infinito si el 0 del denominador también es el resultado de un límite:

$$\lim_{n \to 0^+} \frac{k}{n} = \infty \qquad \text{y} \qquad \lim_{n \to 0^-} \frac{k}{n} = -\infty,$$

donde k es una constante positiva.

Por otro lado, la mayoría de funciones que describen medidas de complejidad computacional son crecientes y carecen de asíntotas horizontales. Por tanto, el límite de estas funciones, cuando sus parámetros tienden a infinito, también es infinito. Por ejemplo:

$$\lim_{n \to \infty} \log_b n = \infty, \qquad \text{o} \qquad \lim_{n \to \infty} n^a = \infty,$$

para $b > 1$ y $a > 0$.

A menudo, los costes computacionales de diferentes algoritmos se pueden comparar a través de límites. Sean $f(n)$ y $g(n)$ dos funciones que expresan costes computacionales (p.e., número de operaciones a realizar por dos algoritmos, donde $f(n) > 0$ y $g(n) > 0$, para $n > 0$), el límite:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)}$$

permite comparar los *órdenes de crecimiento* de $f(n)$ y $g(n)$. Si dicho límite es 0 entonces el orden de crecimiento de $g(n)$ será mayor que el de $f(n)$. Si el límite fuera ∞ entonces $f(n)$ tendría un orden de crecimiento mayor que el de $g(n)$. Por último, si el límite es una constante mayor que 0 entonces $f(n)$ y $g(n)$ tendrán el mismo orden de crecimiento (no es necesario que el límite sea 1).

A veces, el límite da lugar a la indeterminación:

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

En ese caso se puede simplificar la fracción $f(n)/g(n)$ hasta que sea posible obtener un resultado que no sea una indeterminación. Una estrategia empleada con frecuencia consiste en aplicar la famosa regla de L'Hopital:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{f'(n)}{g'(n)},$$

donde $f'(n)$ y $g'(n)$ son las derivadas de $f(n)$ y $g(n)$, respectivamente. Formalmente, la regla de L'Hopital sólo es válida cuando el límite del cociente de derivadas existe, lo cual es habitual.

1.1.6 Derivadas

Esta sección repasa varios conceptos básicos de derivadas, tanto para funciones de una variable como de varias.

Definición y propiedades básicas

La derivada de una función $f(x)$ (donde x es una sola variable real), denotada como $f'(x)$, se define como el siguiente límite:

$$f'(x) = \lim_{h\to 0} \frac{f(x+h) - f(x)}{h},$$

el cual debe existir para que la función se considere derivable en x.

El valor concreto de $f'(x)$ indica la pendiente de la recta tangente a f en x, es

Función	Derivada	Función	Derivada
k	0	$ag(x)$	$ag'(x)$
x	1	$g(x) + h(x)$	$g'(x) + h'(x)$
x^n	$n \cdot x^{n-1}$	$g(x) \cdot h(x)$	$g'(x) \cdot h(x) + g(x) \cdot h'(x)$
$a^{g(x)}$	$g'(x) \cdot a^{g(x)} \cdot \ln(a)$	$\frac{g(x)}{h(x)}$	$\frac{g'(x) \cdot h(x) - g(x) \cdot h'(x)}{(h(x))^2}$
$\log_a(g(x))$	$\frac{g'(x)}{g(x) \cdot \ln(a)}$	$g(h(x))$	$g'(h(x)) \cdot h'(x)$

Tabla 1.3: Propiedades básicas de derivadas.

decir, en el punto $(x, f(x))$. La Tabla 1.3 muestra varias derivadas básicas, así como algunas propiedades de éstas. La *integración por partes* es una técnica útil para hallar expresiones de integrales. Existen muchas más que no se van a abordar en estos apuntes.

Múltiples variables: gradiente y matriz Hessiana

A menudo las funciones que surgen en problemas computacionales constan de varios parámetros o variables. En esos casos es habitual trabajar con las *derivadas parciales* de las funciones. Considérese una función $f(x_1, x_2, \ldots, x_n)$, donde cada x_i es una variable. La derivada parcial de f con respecto a x_i se denota como:

$$\frac{\partial f}{\partial x_i},$$

y consiste en la función f derivada con respecto a x_i, asumiendo que el resto de las variables son constantes. Para poder trabajar con todas las derivadas parciales simultáneamente surge el concepto de *gradiente* de una función, denotado por ∇f, que es el vector de las n derivadas parciales:

$$\nabla f(x_1, x_2, \ldots, x_n) = \nabla f(\mathbf{x}) = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_n} \right),$$

donde $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ simplemente representa el vector con las n variables.

Existen fórmulas concisas para representar derivadas de funciones de múltiples variables. La siguiente representa el gradiente de una función lineal ($\mathbf{a}^\intercal \mathbf{x} = \mathbf{x}^\intercal \mathbf{a} =$

$a_1x_1 + a_2x_2 \cdots\ a_nx_n$, que es simplemente el producto escalar entre $\mathbf{x}$ y un vector $\mathbf{a}$):

$$\frac{\partial \mathbf{x}^\intercal \mathbf{a}}{\partial \mathbf{x}} = \frac{\partial \mathbf{a}^\intercal \mathbf{x}}{\partial \mathbf{x}} = \mathbf{a}. \tag{1.2}$$

Como se puede ver, el resultado es simplemente el vector $\mathbf{a} = (a_1, a_2, \ldots, a_n)$. Por otro lado, para expresiones de tipo $\mathbf{x}^\intercal\mathbf{B}\mathbf{x}$, denominadas *formas cuadráticas*, donde $\mathbf{B}$ es una matriz de dimensiones $n \times n$, la derivada con respecto a $\mathbf{x}$ da lugar al siguiente gradiente:

$$\frac{\partial \mathbf{x}^\intercal \mathbf{B}\mathbf{x}}{\partial \mathbf{x}} = (\mathbf{B} + \mathbf{B}^\intercal)\mathbf{x}. \tag{1.3}$$

Obsérvese que las dos fórmulas anteriores son generalizaciones de la derivada, y serían válidas para $n = 1$.

Asimismo, la *matriz Hessiana* de f, denotada por $\nabla^2 f$, es una matriz cuadrada $n \times n$ (generalmente simétrica) que contiene las derivadas parciales de segundo orden:

$$\nabla^2 f(\mathbf{x}) = \begin{pmatrix} \dfrac{\partial^2 f(\mathbf{x})}{\partial x_1^2} & \dfrac{\partial^2 f(\mathbf{x})}{\partial x_1 \partial x_2} & \cdots & \dfrac{\partial^2 f(\mathbf{x})}{\partial x_1 \partial x_n} \\ \dfrac{\partial^2 f(\mathbf{x})}{\partial x_2 \partial x_1} & \dfrac{\partial^2 f(\mathbf{x})}{\partial x_2^2} & \cdots & \dfrac{\partial^2 f(\mathbf{x})}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \dfrac{\partial^2 f(\mathbf{x})}{\partial x_n \partial x_1} & \dfrac{\partial^2 f(\mathbf{x})}{\partial x_n \partial x_2} & \cdots & \dfrac{\partial^2 f(\mathbf{x})}{\partial x_n^2} \end{pmatrix}.$$

La derivada parcial de segundo orden:

$$\frac{\partial^2 f(\mathbf{x})}{\partial x_i \partial x_j}$$

es el resultado de derivar f primero con respecto a x_i y después con respecto a x_j. Es decir, es la derivada parcial con respecto a x_j de la derivada parcial de f con respecto a x_i.

1.1.7 Integrales

La integral $F(x)$ de una función $f(x)$ se puede describir informalmente como la *antiderivada* de $f(x)$. Es decir, $F(x)$ es una función tal que si se deriva el resultado sería $f(x)$. En otras palabras: $F'(x) = f(x)$. La notación más habitual para expresar $F(x)$ es:

$$F(x) = \int f(x)\, dx.$$

$\int f(x)\,dx = F(x)$	
$\int dx = x$	$\int af(x)\,dx = a\int f(x)\,dx$
$\int x^n\,dx = \frac{x^{n+1}}{n+1}$	$\int \big(f(x)+g(x)\big)\,dx = \int f(x)\,dx + \int g(x)\,dx$
$\int \frac{1}{x}\,dx = \ln\lvert x\rvert$	$\int u\,dv = uv - \int v\,du$ (integración por partes)
$\int a^x\,dx = \frac{a^x}{\ln a}$	$\int \ln x\,dx = x\ln x - x$

Tabla 1.4: Propiedades básicas de integrales.

Las integrales definidas, expresadas como:

$$\int_a^b f(x)\,dx = F(b) - F(a),$$

representan el área por debajo de la curva (función) $f(x)$ desde a hasta b. La Tabla 1.4 muestra varias integrales básicas, así como algunas propiedades de éstas (se ha omitido sumar una constante a las expresiones de $F(x)$).

1.1.8 Geometría

Este apartado contiene varias definiciones fundamentales asociadas a vectores y matrices, las cuales pueden interpretarse de manera geométrica, y no sólo algebraica.

Producto escalar y norma

Varios conceptos importantes en geometría están relacionados con el *producto escalar* entre dos vectores. En general, éste se puede definir para vectores de n componentes (o incluso para elementos más complejos de un espacio vectorial, como ciertas funciones). El producto escalar se puede definir y denotar de varias maneras:

$$\vec{\mathbf{a}} \cdot \vec{\mathbf{b}} = \langle \mathbf{a}, \mathbf{b} \rangle = \mathbf{a}^\intercal \mathbf{b} = \sum_{i=1}^{n} a_i b_i = |\mathbf{a}| \cdot |\mathbf{b}| \cdot \cos(\alpha),$$

donde α es el ángulo entre los vectores $\mathbf{a}$ y $\mathbf{b}$, y $|\cdot|$ denota el *módulo* o *norma* de un vector:

$$|\mathbf{a}| = \|\mathbf{a}\|_2 = \sqrt{a_1^2 + \cdots + a_n^2} = \sqrt{\mathbf{a}^\intercal \mathbf{a}}.$$

Esta definición ($\|\mathbf{a}\|_2$) se conoce como la norma euclídea de un vector, y refleja la noción clásica de distancia (euclídea). A menudo se emplea la norma euclídea al cuadrado ($\|\mathbf{a}\|_2^2$), que conduce a fórmulas más sencillas al cancelar la raíz cuadrada. Además, existen otros tipos de normas que definen la "longitud" de un vector de formas diferentes. Algunas de las más utilizadas son:

$$\|\mathbf{a}\|_1 = |a_1| + |a_2| + \cdots + |a_n|$$

y

$$\|\mathbf{a}\|_\infty = \max_{i=1,\ldots,n} \{a_1, a_2, \ldots, a_n\}.$$

Por último, también se puede definir el producto escalar entre dos matrices de dimensiones $n \times m$ de manera análoga. En concreto, es la suma de los productos de los elementos correspondientes (ubicados en las mismas filas y columnas) de ambas matrices:

$$\langle \mathbf{A}, \mathbf{B} \rangle = \sum_{i=1}^{n} \sum_{j=1}^{m} a_{i,j} \cdot b_{i,j} = \operatorname{tr}(\mathbf{B}\mathbf{A}^{\mathsf{T}}).$$

Además, existen varias normas sobre matrices. Por ejemplo, la norma Frobenius ($\|\cdot\|_F$) es la equivalente a la norma euclídea, y mide la raíz cuadrada de la suma de los elementos al cuadrado de la matriz:

$$\|\mathbf{A}\|_F = \sqrt{\sum_{i=1}^{n} \sum_{j=1}^{m} a_{i,j}^2}.$$

Ortogonalidad

Se dice que dos vectores son *ortogonales* si su producto escalar es 0, lo cual ocurre si son perpendiculares. Si además los vectores tiene módulo 1 decimos que son *ortonormales*. Un conjunto de vectores es ortogonal si cada par de vectores del conjunto son ortogonales. La ortogonalidad no es una propiedad de un sólo vector, sino de un conjunto de ellos. Las matrices ortogonales (también llamadas ortonormales) se caracterizan porque todas sus columnas son vectores ortonormales entre sí.

Matriz de rotación

Cuando multiplicamos un número x por otro y podemos interpretar que estamos escalando (que es una determinada transformación) x por y. Es decir, y escala (aumenta o reduce) el valor de x. De manera análoga, y hablando de vectores y matrices, es interesante pensar qué le hace una matriz $\mathbf{A}$ de dimensiones $n \times n$ cuando multiplica a un vector $\mathbf{v}$ de n componentes. Claramente, el resultado $\mathbf{w} = \mathbf{A}\mathbf{v}$ es otro vector de n elementos. Para $n = 2$, multiplicar un vector, el cual podemos representar en un plano, por una matriz de 2×2 simplemente genera un nuevo vector ($\mathbf{w}$) que también quedaría representado en el plano. Dependiendo de las características de la matriz (por ejemplo,

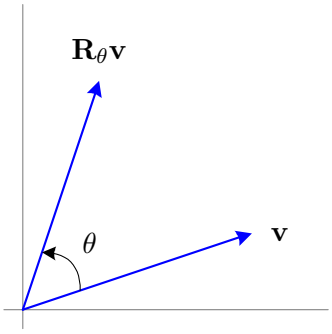


Figura 1.2: Efecto de multiplicar un vector $\mathbf{v}$ por una matriz de rotación $\mathbf{R}_\theta$.

su rango, si es diagonal, si es ortogonal, etc.) podemos determinar algunas relaciones del vector resultante $\mathbf{w}$ con respecto a $\mathbf{v}$. Por ejemplo, si la matriz es diagonal y tiene el mismo valor α en ambas componentes de su diagonal, entonces $\mathbf{w}$ simplemente será $\mathbf{v}$ escalado por α.

Otra matriz muy utilizada es la *matriz de rotación*, que se define de la siguiente manera:

$$\mathbf{R}_\theta = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix}.$$

Dado un vector inicial $\mathbf{v}$, el producto $\mathbf{R}_\theta\mathbf{v}$ es el resultado de rotar $\mathbf{v}$ θ radianes en sentido contrario a las agujas del reloj, como ilustra la Figura 1.2. Las matrices de rotación son ortogonales, lo cual significa que $\mathbf{R}_\theta^\intercal\mathbf{R}_\theta = \mathbf{R}_\theta\mathbf{R}_\theta^\intercal = \mathbf{I}$ es la matriz identidad. Esto implica que la transformación no altere el módulo del vector resultante. Es decir, $\|\mathbf{v}\| = \|\mathbf{R}_\theta\mathbf{v}\|$.

1.1.9 Sistemas de ecuaciones lineales

En este apartado veremos una conexión entre conceptos de álgebra lineal y geometría a través del análisis de sistemas de ecuaciones lineales.

Producto de matriz por vector

Existen dos maneras de calcular el producto de una matriz por un vector. La más conocida consiste en hallar una serie de productos escalares (filas de la matriz por el vector columna). Supongamos que deseamos multiplicar una matriz $\mathbf{A}$ de dimensiones $n \times m$, por un vector (columna) $\mathbf{x}$ de m elementos, donde $\mathbf{a}_{i,:}^\intercal$ es la i-ésima fila de $\mathbf{A}$. Podemos expresar el producto como:

$$\mathbf{A}\mathbf{x} = \begin{bmatrix} — \ \mathbf{a}_{1,:}^\intercal \ — \\ — \ \mathbf{a}_{2,:}^\intercal \ — \\ \vdots \\ — \ \mathbf{a}_{n,:}^\intercal \ — \end{bmatrix} \begin{bmatrix} | \\ \mathbf{x} \\ | \end{bmatrix} = \begin{bmatrix} \mathbf{a}_{1,:}^\intercal\mathbf{x} \\ \mathbf{a}_{2,:}^\intercal\mathbf{x} \\ \vdots \\ \mathbf{a}_{n,:}^\intercal\mathbf{x} \end{bmatrix}.$$

Aunque este procedimiento es sencillo (es como se suele explicar en clases de introducción a matrices), existe una segunda forma de calcular el producto que tiene una interpretación geométrica más clara, y por tanto puede ayudar a comprender algunos conceptos de álgebra lineal relacionados. En esta segunda forma la clave es considerar los vectores columnas de la matriz $\mathbf{A}$, ya que el producto es una simple combinación lineal de éstos, donde los coeficientes de dicha combinación son los valores del vector $\mathbf{x}$. En concreto:

$$\mathbf{Ax} = \begin{bmatrix} | & | & & | \\ \mathbf{a}_{:,1} & \mathbf{a}_{:,2} & \cdots & \mathbf{a}_{:,m} \\ | & | & & | \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix} = \begin{bmatrix} | \\ \mathbf{a}_{:,1} \\ | \end{bmatrix} x_1 + \begin{bmatrix} | \\ \mathbf{a}_{:,2} \\ | \end{bmatrix} x_2 + \cdots + \begin{bmatrix} | \\ \mathbf{a}_{:,m} \\ | \end{bmatrix} x_m,$$

donde $\mathbf{a}_{:,i}$ es el i-ésimo vector columna de $\mathbf{A}$.

Interpretaciones geométricas de un sistema de ecuaciones lineales

Un sistema de n ecuaciones lineales con m incógnitas (las variables x_i):

$$\left.\begin{array}{ccccccccc} a_{1,1}x_1 & + & a_{1,2}x_2 & + & \cdots & + & a_{1,m}x_m & = & b_1 \\ a_{2,1}x_1 & + & a_{2,2}x_2 & + & \cdots & + & a_{2,m}x_m & = & b_2 \\ \vdots & & \vdots & & \ddots & & \vdots & & \vdots \\ a_{n,1}x_1 & + & a_{n,2}x_2 & + & \cdots & + & a_{n,m}x_m & = & b_n \end{array}\right\}$$

se puede expresar de manera más compacta mediante:

$$\mathbf{Ax} = \mathbf{b}.$$

Si la matriz $\mathbf{A}$ es cuadrada e invertible entonces la solución es $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$.

Cada ecuación lineal representa una restricción sobre los valores que pueden tomar las incógnitas. Geométricamente representan *hiperplanos* (recta para 2 incógnitas, plano para 3, etc.). El sistema de ecuaciones tiene solución (se satisfarían todas las ecuaciones) si la intersección de todos estos hiperplanos no es vacía. Considérese el siguiente sistema ($\mathbf{Ax} = \mathbf{b}$) de 3 ecuaciones y 2 incógnitas:

$$\begin{bmatrix} -1 & 2 \\ 1 & 2 \\ 2 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} -2 \\ 6 \\ 6 \end{bmatrix}.$$

Podemos representarlo mediante 3 rectas en el plano, tal y como muestra la Figura 1.3. En este caso no existe una solución, ya que las tres rectas no se cortan en un punto. En problemas de ingeniería es habitual buscar una *solución aproximada*, que esté cerca de las rectas.

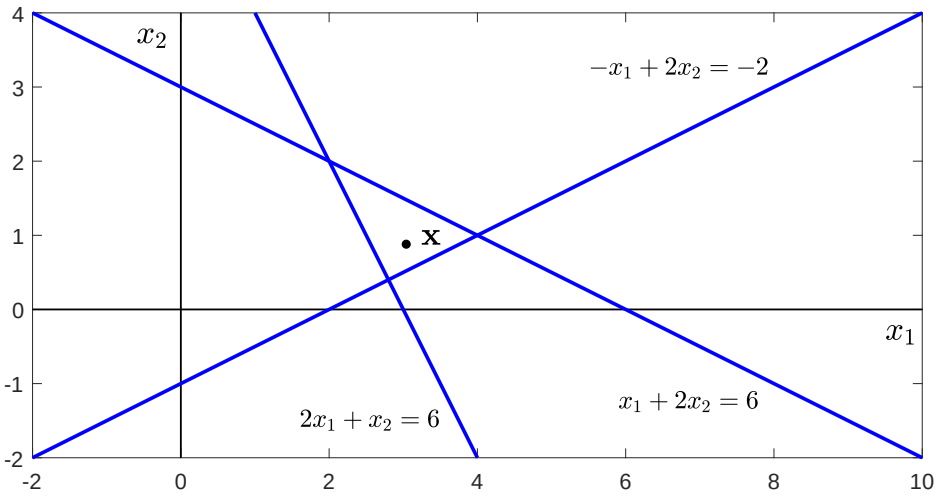


Figura 1.3: Interpretación en 2 dimensiones de un sistema con 3 ecuaciones (rectas) y 2 incógnitas (x_1 y x_2) incompatible $\mathbf{Ax} = \mathbf{b}$. El punto $\mathbf{x}$ es la solución aproximada.

También podemos representar la geometría del sistema en un espacio de n dimensiones. La Figura 1.4 ilustra los dos vectores columna de $\mathbf{A}$ en un espacio de $n = 3$ dimensiones. El producto $\mathbf{Ax}$ es un vector (o punto) que tiene que estar necesariamente en el plano generado por las dos columnas de $\mathbf{A}$, al ser una combinación lineal de éstas. Por tanto, el sistema de ecuaciones sólo podrá tener una solución si el vector (o punto) $\mathbf{b}$ se encuentra también en dicho plano. Como esto no se cumple en este ejemplo se puede buscar una solución aproximada. La estrategia más común es determinar los valores de $\mathbf{x}$ (es decir, x_1 y x_2) de manera que el punto $\mathbf{Ax}$, perteneciente al plano, esté lo más cerca de $\mathbf{b}$ como sea posible. Geométricamente, $\mathbf{Ax}$ sería la proyección ortogonal de $\mathbf{b}$ sobre el plano. Cuando $n \geq m$ y el rango de $\mathbf{A}$ es m esta proyección es $\mathbf{A}(\mathbf{A}^\intercal\mathbf{A})^{-1}\mathbf{A}^\intercal\mathbf{b}$. Por tanto, la solución aproximada es:

$$\mathbf{x} = (\mathbf{A}^\intercal\mathbf{A})^{-1}\mathbf{A}^\intercal\mathbf{b}. \tag{1.4}$$

Nota: esta solución aproximada no es, en general, el punto que minimiza la suma de distancias desde él a las regiones geométricas asociadas a las ecuaciones lineales (por ejemplo, a las rectas de la Figura 1.3). Para que esto ocurriese los módulos de las filas de $\mathbf{A}$ tendrían que ser iguales (obsérvese que esto ocurre en el ejemplo de la Figura 1.3). ¿Podrías explicar por qué?

1.2 Problemas de optimización

Los problemas de optimización constituyen una amplia e importante categoría de problemas computacionales.

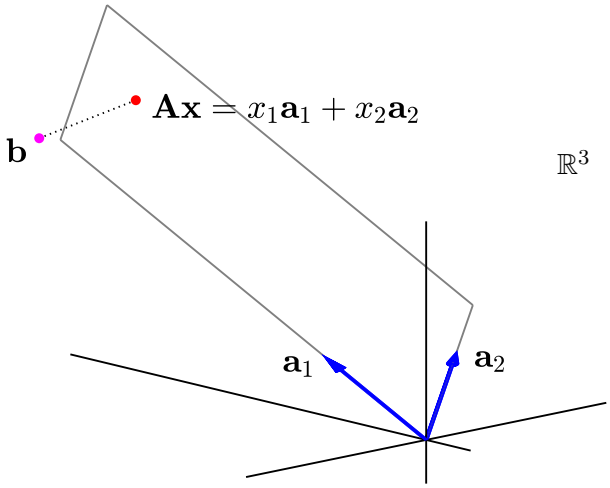


Figura 1.4: Interpretación en 3 dimensiones de un sistema con 3 ecuaciones y 2 incógnitas incompatible $\mathbf{Ax} = \mathbf{b}$, donde $\mathbf{a}_1$ y $\mathbf{a}_2$ son las columnas de $\mathbf{A}$.

1.2.1 Nomenclatura y notación

Dada una función matemática $f_0(x_1, \ldots, x_n)$ de n variables, que denominamos *función objetivo*, en estos problemas se busca valores específicos de las variables x_i que hagan que dicha función tome el mayor o menor valor posible. De esta manera, podemos hablar de problemas de maximización o minimización, o de maximizar o minimizar una función. Además, en algunos problemas las variables deben satisfacer una serie de *restricciones*, que definen una *región factible* donde se encontrará la solución.

Los puntos óptimos de las funciones son aquellos en los que se alcanza un mínimo o máximo en un entorno "local" alrededor de dicho punto. A estos puntos se les denomina *óptimos locales*. En cambio, un *óptimo global* (puede haber varios) es aquel en el que la función necesariamente alcanza su valor mínimo o máximo. La Figura 1.5(a) ilustra estos dos tipos de óptimos. Por otro lado, hay que tener en cuenta que las restricciones de un problema pueden conducir a soluciones óptimas en los que la derivada (o gradiente) no se anule, como se ilustra en la Figura 1.5(b).

En cuanto a notación, es habitual especificar un problema de optimización (en este caso de minimización) de la siguiente manera:

$$\begin{aligned} \underset{\mathbf{x}}{\text{minimiza}} \quad & f_0(\mathbf{x}) \\ \text{sujeto a} \quad & f_i(\mathbf{x}) \leq b_i, \quad i = 1, \ldots, m \end{aligned}$$

En este caso $\mathbf{x}$ engloba al conjunto de variables o parámetros de la función objetivo f_0, y las funciones f_i definen las restricciones sobre estas variables. Obsérvese que un problema de maximización se puede convertir fácilmente a uno de minimización simplemente cambiando el signo de f_0.

Como ejemplo, veremos el *Problema de la mochila 0/1*. Considérese un conjunto de

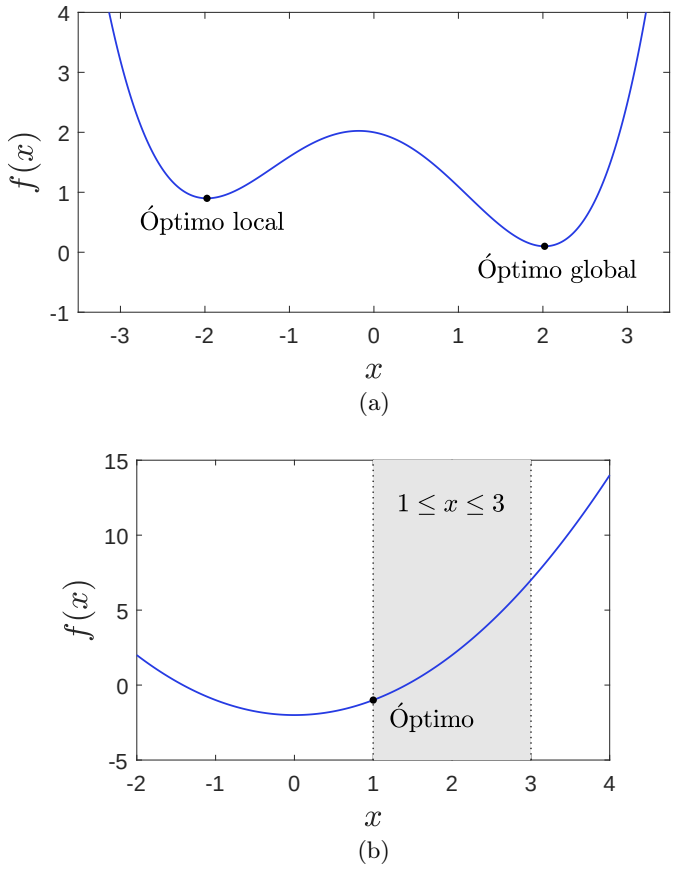


Figura 1.5: Diferencia entre óptimo local y global (a), y óptimo en una región factible limitada (b).

n objetos, cada uno con un peso p_i y un valor v_i, para $i = 1, \ldots, n$, y una mochila con capacidad C, que es el máximo peso que puede soportar la mochila. El objetivo consiste en seleccionar el subconjunto de objetos que puedan introducirse en la mochila, sin sobrepasar la capacidad C, tal que la suma de sus valores sea máxima. Empleando la notación anterior el problema se especificaría de la siguiente manera:

$$\begin{aligned} \underset{\mathbf{x}}{\text{maximiza}} \quad & \mathbf{v}^\intercal \mathbf{x} \\ \text{sujeto a} \quad & x_i \in \{0, 1\}, \quad i = 1, \ldots, n \\ & \mathbf{p}^\intercal \mathbf{x} \leq C \end{aligned}$$

donde $\mathbf{p}$ y $\mathbf{v}$ son vectores que contienen los pesos y los valores, mientras que $\mathbf{x}$ es un vector que define el subconjunto de objetos a introducir en la mochila. En concreto, $\mathbf{x}$ es otro vector de n variables binarias o *indicadoras* x_i donde:

$$x_i = \begin{cases} 1 & \text{si se introduce el objeto } i \text{ en la mochila,} \\ 0 & \text{en caso contrario.} \end{cases}$$

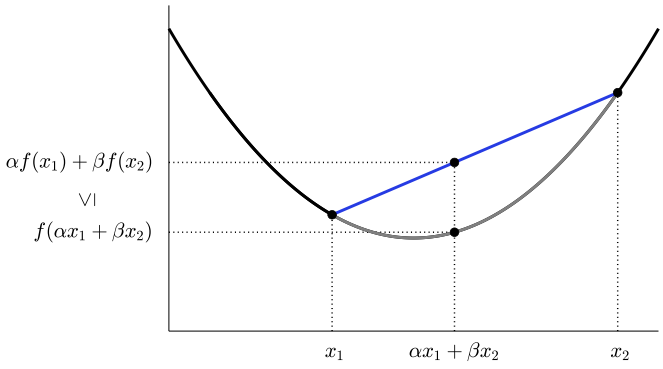


Figura 1.6: Función convexa de una variable.

Obsérvese que los productos escalares $\mathbf{p}^\intercal \mathbf{x} = \sum_{i=1}^{n} p_i x_i$ y $\mathbf{v}^\intercal \mathbf{x} = \sum_{i=1}^{n} v_i x_i$ reflejan el peso y el valor total, respectivamente, de los objetos introducidos en la mochila.

1.2.2 Tipos de problemas de optimización

Los problemas de optimización se pueden agrupar en varias categorías. Este apartado presenta algunas de ellas.

Problemas lineales

Los problemas lineales, también denominados problemas de *programación lineal*, se caracterizan porque tanto la función objetivo f_0, como las restricciones f_i, para $i = 1, \ldots, m$, son *funciones lineales*. Una función f es lineal si:

$$f(\alpha \mathbf{x}_1 + \beta \mathbf{x}_2) \;=\; \alpha f(\mathbf{x}_1) + \beta f(\mathbf{x}_2)$$

$$\forall\ \mathbf{x}_1, \mathbf{x}_2 \in \mathbb{R}^n$$

$$\forall\ \alpha, \beta \in \mathbb{R}$$

Problemas convexos

Los problemas de optimización convexos, se caracterizan porque tanto la función objetivo f_0, como las restricciones f_i, para $i = 1, \ldots, m$, son *funciones convexas*. Una función f es convexa si:

$$f(\alpha \mathbf{x}_1 + \beta \mathbf{x}_2) \;\leq\; \alpha f(\mathbf{x}_1) + \beta f(\mathbf{x}_2)$$

$$\forall\ \mathbf{x}_1, \mathbf{x}_2 \in \mathbb{R}^n$$

$$\forall\ \alpha, \beta \in [0, 1], \text{ con } \alpha + \beta = 1$$

La Figura 1.6 ilustra estas propiedades gráficamente empleando una función de una variable. El segmento entre dos puntos de la función no puede estar por debajo de la

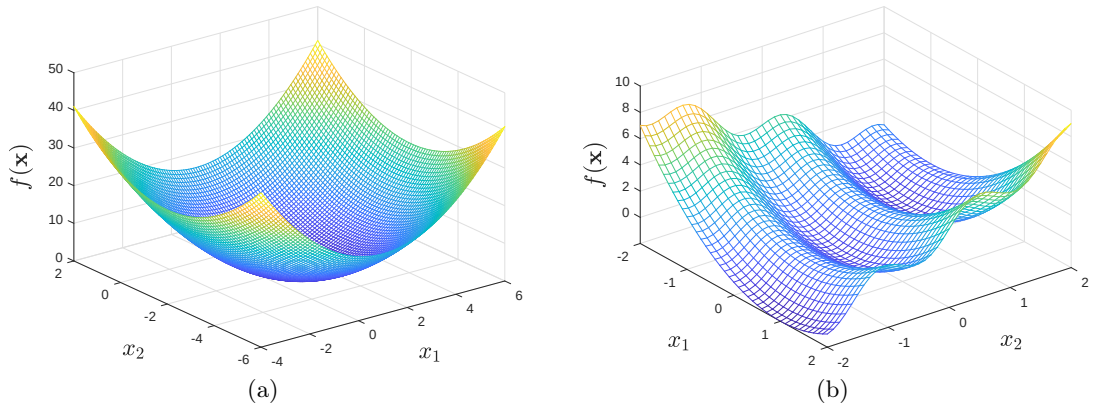


Figura 1.7: Función convexa (a) y no convexa (b) de dos variables.

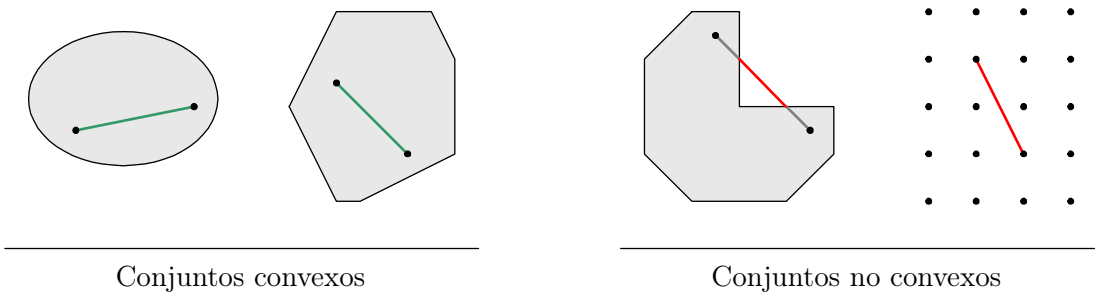


Figura 1.8: Ejemplos de conjuntos convexos y no convexos.

función. La Figura 1.7 ilustra una función convexa (a) y no convexa (b) de dos variables. Las funciones convexas tienen la ventaja de que cualquier óptimo local es necesariamente global.

Además, para que un problema de optimización sea convexo las restricciones deben definir un *conjunto convexo*, que es aquel tal que cualquier segmento entre dos elementos (puntos) cualquiera del conjunto debe estar completamente contenido en el conjunto. La Figura 1.8 muestra varios ejemplos de conjuntos convexos y no convexos. Finalmente, si el dominio de las variables es discreto las restricciones no darán lugar a conjuntos convexos, y por tanto los problemas no podrán ser convexos. Por ejemplo, los problemas de programación lineal son convexos por definición. Sin embargo, si se introducen restricciones discretas, como que algunas variables sólo puedan tomar valores enteros (como el problema de la mochila 0-1), los problemas dejarían de ser convexos. Esto puede hacer que el problema sea intratable desde un punto de vista de eficiencia computacional.

Mínimos cuadrados

Considérese el problema de resolver un sistema de ecuaciones lineales $\mathbf{Ax} = \mathbf{b}$, donde $\mathbf{A} \in \mathbb{R}^{n \times m}$ con $n \geq m$, y cuyo rango es m. En el apartado 1.1.9 vimos que cuando no

tiene solución se busca un vector $\mathbf{x}$ aproximado tal que $\mathbf{Ax}$ esté lo más cerca posible de $\mathbf{b}$ (véase la Figura 1.4). Esta estrategia se puede expresar planteando el siguiente problema de optimización, que resulta ser convexo:

$$\begin{array}{c}\text{minimiza} \\ \mathbf{x} \in \mathbb{R}^m\end{array} \quad \|\mathbf{Ax} - \mathbf{b}\|^2,$$

donde $\|\cdot\|^2$ es el módulo de un vector al cuadrado (la solución es la misma si no se eleva al cuadrado, pero se eleva a 2 para facilitar las operaciones matemáticas que conducen a la solución). En este caso, $\|\mathbf{Ax} - \mathbf{b}\|^2$ es la distancia (al cuadrado) entre $\mathbf{Ax}$ y $\mathbf{b}$. Obsérvese que este problema carece de restricciones.

Aprovechando la propiedad $\|\mathbf{u}\|^2 = \mathbf{u}^\intercal\mathbf{u}$, podemos reescribir la función objetivo de la siguiente manera:

$$\begin{aligned} f(\mathbf{x}) = \|\mathbf{Ax} - \mathbf{b}\|^2 &= (\mathbf{Ax} - \mathbf{b})^\intercal(\mathbf{Ax} - \mathbf{b}) \\ &= (\mathbf{x}^\intercal\mathbf{A}^\intercal - \mathbf{b}^\intercal)(\mathbf{Ax} - \mathbf{b}) \\ &= \mathbf{x}^\intercal\mathbf{A}^\intercal\mathbf{Ax} - \mathbf{x}^\intercal\mathbf{A}^\intercal\mathbf{b} - \mathbf{b}^\intercal\mathbf{Ax} + \mathbf{b}^\intercal\mathbf{b} \\ &= \mathbf{x}^\intercal\mathbf{A}^\intercal\mathbf{Ax} - 2\mathbf{b}^\intercal\mathbf{Ax} + \mathbf{b}^\intercal\mathbf{b}. \end{aligned}$$

Que es una generalización de una función cuadrática para m variables. Pues bien, si para hallar el mínimo de una parábola convexa podemos simplemente calcular su derivada, igualarla a 0 y despejar la incognita, para este problema multivariable podemos proceder de manera similar. En este caso, tendríamos que hallar el gradiente de la función, igualarlo al vector $\mathbf{0}$, y despejar el vector $\mathbf{x}$.

Apoyándonos en las fórmulas en (1.2) y (1.3) podemos hallar el gradiente de la función:

$$\nabla f(\mathbf{x}) = 2\mathbf{A}^\intercal\mathbf{Ax} - 2\mathbf{A}^\intercal\mathbf{b}.$$

Igualándolo al vector cero y despejando $\mathbf{x}$ tenemos:

$$\begin{aligned} 2\mathbf{A}^\intercal\mathbf{Ax} - 2\mathbf{A}^\intercal\mathbf{b} &= \mathbf{0}, \\ 2\mathbf{A}^\intercal\mathbf{Ax} &= 2\mathbf{A}^\intercal\mathbf{b}, \\ \mathbf{A}^\intercal\mathbf{Ax} &= \mathbf{A}^\intercal\mathbf{b}, \\ \mathbf{x} &= (\mathbf{A}^\intercal\mathbf{A})^{-1}\mathbf{A}^\intercal\mathbf{b}, \end{aligned}$$

que es precisamente la fórmula descrita en (1.4). Obsérvese que para que sea válida debe existir $\mathbf{A}^\intercal\mathbf{A}$ debe ser invertible. Por eso se requiere que $n \geq m$ y que el rango de $\mathbf{A}$ sea m.

1.3 Sumatorios

Los sumatorios son fórmulas compactas para expresar la suma de una colección de entidades matemáticas (números enteros, números reales, vectores, matrices, funciones,

etc.). Es importante estar familiarizado con ellos porque aparecen, no sólo en definiciones de funciones y fórmulas que tengamos que programar, sino también al analizar la eficiencia tanto de algoritmos iterativos como recursivos. Cuando los términos a sumar se pueden expresar mediante una función $f()$, evaluada en enteros consecutivos desde uno inicial m hasta uno final n, el sumatorio se escribe de la siguiente manera (*notación sigma*):

$$\sum_{i=m}^{n} f(i) = f(m) + f(m+1) + \cdots + f(n-1) + f(n). \tag{1.5}$$

Como se puede ver, el resultado es la suma de los términos que surgen tras sustituir i por los números desde m a n dentro de $f(i)$. Por ejemplo:

$$\sum_{i=0}^{4} ki^2 = k \cdot 0^2 + k \cdot 1^2 + k \cdot 2^2 + k \cdot 3^2 + k \cdot 4^2,$$

donde $f(i) = ki^2$.

El resultado de la suma no depende de la *variable contadora*, que en este caso era i. Por tanto, podríamos haber escogido cualquier otro nombre para dicha variable, como j o k:

$$\sum_{i=m}^{n} f(i) = \sum_{j=m}^{n} f(j) = \sum_{k=m}^{n} f(k).$$

Además, podemos expresar la misma suma de varias maneras:

$$\sum_{i=m}^{n} f(i) = \sum_{i=m-1}^{n-1} f(i+1) = \sum_{i=m}^{n} f(n-i+m),$$

lo cual equivale a realizar un cambio de variable. Obsérvese que en el último sumatorio lo único que cambia es el orden en el que se suman los términos (en sentido contrario con respecto al primer sumatorio).

Por último, si el límite inferior (m) es mayor que el superior (n) se suele considerar que el sumatorio vale 0.

1.3.1 Propiedades básicas de sumatorios

Las siguientes propiedades son útiles para simplificar y trabajar con sumatorios, y se pueden obtener fácilmente de propiedades básicas de sumas y multiplicaciones. En primer lugar,

$$\sum_{i=1}^{n} 1 = \underbrace{1 + 1 + \cdots + 1 + 1}_{n \text{ veces}} = n.$$

En este caso $f(i)$ es una constante (1) que no depende de la variable contadora i. De manera similar,

$$\sum_{i=1}^{n} k = \underbrace{k + k + \cdots + k + k}_{n \text{ veces}} = kn.$$

Este ejemplo ilustra que cualquier término multiplicativo que no dependa de la variable contadora puede extraerse fuera del sumatorio:

$$\sum_{i=1}^{n} k = \sum_{i=1}^{n} (k \cdot 1) = \sum_{i=1}^{n} k \cdot 1 = k \cdot \left(\sum_{i=1}^{n} 1 \right) = k \sum_{i=1}^{n} 1 = kn. \tag{1.6}$$

En este caso el término constante k, que podemos considerar que está multiplicado por 1, no depende de i, y se puede extraer fuera del sumatorio, donde aparecería multiplicándolo (en cuanto a notación, (1.6) muestra que no es necesario usar paréntesis dentro del sumatorio si f no se puede descomponer en la suma de varios términos aditivos). En general, la propiedad se puede expresar de la siguiente manera:

$$\sum_{i=m}^{n} k f(i) = k \sum_{i=m}^{n} f(i),$$

que no es otra cosa que extraer el factor común (propiedad distributiva de la multiplicación respecto de la suma), que en este caso es k. Naturalmente, el factor k puede ser el producto de varios términos que no dependan de la variable contadora, y podrían contener los límites del sumatorio:

$$\sum_{i=m}^{n} amn^2 i^3 = amn^2 \sum_{i=m}^{n} i^3,$$

donde a es una constante.

Por otro lado, para límites generales y un término k (constante) dentro del sumatorio que no dependa de la variable contadora tenemos:

$$\sum_{i=m}^{n} k = k(n - m + 1).$$

Nótese que no es $k(n - m)$.

Finalmente, si la función f contiene varios términos aditivos (que se suman o restan), se puede descomponer el sumatorio en varios más simples:

$$\sum_{i=m}^{n} \Big(f_1(i) + f_2(i) \Big) = \sum_{i=m}^{n} f_1(i) + \sum_{i=m}^{n} f_2(i).$$

Obsérvese que en este caso era necesario incluir paréntesis en el primer sumatorio.

1.3.2 Sumatorios de progresiones aritméticas

Una progresión aritmética es una sucesión de números tales que la diferencia de cualquier par de términos sucesivos de la secuencia es constante. Formalmente, si a_i es el i-ésimo término de la progresión, donde $i \geq 0$, entonces $a_i = a_{i-1} + d$, para $i > 0$, donde d es la diferencia entre términos consecutivos. Obsérvese que la definición previa de a_i es recursiva, pero también puede expresarse de manera no recursiva: $a_i = a_{i-1} + d = a_{i-2} + 2d = \cdots = a_0 + id$.

La *suma parcial* de una progresión aritmética corresponde al siguiente sumatorio:

$$\sum_{i=m}^{n} a_i.$$

A veces se denomina "serie aritmética", aunque técnicamente las series son sumas infinitas. Las sumas parciales de progresiones aritméticas se pueden simplificar para eliminar el sumatorio y trabajar con una fórmula más sencilla. En concreto:

$$S(m,n) = \sum_{i=m}^{n} a_i = \frac{1}{2}(a_m + a_n)(n - m + 1).$$

Para demostrar la validez de la fórmula anterior usaremos la siguiente propiedad de las progresiones geométricas: $a_{m+k} + a_{n-k} = a_m + a_n$. En concreto, sumaremos dos copias del sumatorio, una con términos en sentido "creciente" y otra en "decreciente":

$$\begin{array}{rrcccccccc} & S(m,n) & = & a_m & + & a_{m+1} & + \cdots + & a_{n-1} & + & a_n \\ + & & & & & & & & & \\ & S(m,n) & = & a_n & + & a_{n-1} & + \cdots + & a_{m+1} & + & a_m \\ \hline & 2S(m,n) & = & (a_m + a_n) & + & (a_m + a_n) & + \cdots + & (a_m + a_n) & + & (a_m + a_n) \end{array}$$

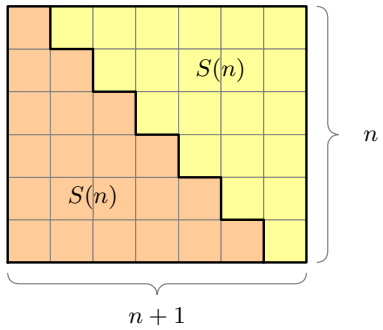
$$2S(m,n) = (a_m + a_n)(n - m + 1) \qquad \Rightarrow \qquad S(m,n) = \frac{1}{2}(a_m + a_n)(n - m + 1)$$

A continuación veremos algunos ejemplos de sumas parciales de progresiones aritméticas.

Suma de los primeros n números positivos

Se trata de un sumatorio que surge con frecuencia al analizar la eficiencia tanto de algoritmos iterativos como recursivos. Se puede entender como la suma parcial de n elementos de una progresión aritmética donde $s_0 = 1$ y $d = 1$:

$$S(n) \;=\; \sum_{i=0}^{n-1} s_i \;=\; \sum_{i=0}^{n-1} \big(s_0 + id\big) = \sum_{i=0}^{n-1} \big(1 + i\big)$$



$$2S(n) = n(n+1)$$

$$\Rightarrow \quad S(n) = \frac{n(n+1)}{2}$$

Figura 1.9: Demostración visual para la fórmula cuadrática de la suma de los n primeros números positivos.

$$= \sum_{i=0}^{n-1} 1 + \sum_{i=0}^{n-1} i \;=\; n + \sum_{i=0}^{n-1} i \;=\; \sum_{i=1}^{n} i.$$

La fórmula simplificada del sumatorio es:

$$S(n) \;=\; \sum_{i=1}^{n} i = 1 + 2 + \cdots + (n-1) + n \;=\; \frac{n(n+1)}{2}. \tag{1.7}$$

Esta fórmula se puede demostrar por inducción o mediante el procedimiento visto en el apartado 1.3.2:

$$\begin{array}{rrcccccccc} & S(n) & = & 1 & + & 2 & + \cdots + & (n-1) & + & n \\ + & S(n) & = & n & + & (n-1) & + \cdots + & 2 & + & 1 \\ \hline & 2S(n) & = & (n+1) & + & (n+1) & + \cdots + & (n+1) & + & (n+1) \end{array}$$

Lo cual implica que $2S(n) = n(n+1)$, ya que hay n términos (cada uno igual a $n+1$) a la derecha de la identidad, y dividiendo por 2 da lugar a (1.7).

Esta demostración también puede representarse de manera visual (véase la Figura 1.9) construyendo dos pirámides triangulares de $S(n)$ bloques cuadrados (de dimensiones 1×1), que pueden combinarse para formar un rectángulo con un lado de n bloques y otro de $n+1$, y por tanto con área $n(n+1)$. Esto implica que el área de cada pirámide triangular $S(n)$ debe ser $n(n+1)/2$.

Suma de los primeros n números impares

La suma de los n primeros números impares es igual a n^2 y se puede demostrar de manera análoga (por inducción, empleando el procedimiento que suma dos sumatorios en sentidos opuestos, visualmente, etc.). En este apartado veremos una demostración algebraica en la que sólo haremos uso de propiedades de sumatorios. En este caso el

sumatorio es:

$$S(n) \;=\; \sum_{i=1}^{n}(2i-1) = 1+3+\cdots+(2n-3)+(2n-1).$$

En primer lugar, el sumatorio se puede dividir en dos, los cuales se pueden simplificar:

$$S(n) = \sum_{i=1}^{n}(2i-1) = \sum_{i=1}^{n}2i - \sum_{i=1}^{n}1 = 2\sum_{i=1}^{n}i - n.$$

Finalmente, por (1.7):

$$S(n) = 2\frac{n(n+1)}{2} - n = n(n+1) - n = n^2 + n - n = n^2.$$

1.3.3 Sumatorios telescópicos

Un sumatorio telescópico se caracteriza porque la función dentro del sumatorio es la resta de dos términos consecutivos de algún tipo de progresión a:

$$S(n) = \sum_{i=1}^{n}(a_i - a_{i-1}) = a_n - a_0.$$

Es importante conocerlos ya que se pueden simplificar a una simple resta de dos términos, en este caso $a_n - a_0$:

$$S(n) = -a_0 + \underbrace{a_1 - a_1}_{0} + \underbrace{a_2 - a_2}_{0} \;\cdots\; + \underbrace{a_{n-1} - a_{n-1}}_{0} + a_n = a_n - a_0.$$

Por ejemplo, el siguiente sumatorio resulta ser telescópico:

$$\sum_{i=1}^{n-1}\frac{1}{i(i+1)} = \sum_{i=1}^{n-1}\left(\frac{1}{i} - \frac{1}{i+1}\right) = +1 - \frac{1}{2} + \frac{1}{2} - \frac{1}{3} + \ldots + \frac{1}{n-1} - \frac{1}{n} = 1 - \frac{1}{n},$$

donde $a_i = -1/(i+1)$:

$$\sum_{i=1}^{n-1}\frac{1}{i(i+1)} = \sum_{i=1}^{n-1}\left(+\left(-\frac{1}{i+1}\right) - \left(-\frac{1}{i}\right)\right) = -\frac{1}{n} - (-1) = 1 - \frac{1}{n}.$$

1.3.4 Sumatorios de potencias

En este apartado veremos sumas de tipo:

$$S_p(n) \;=\; \sum_{i=1}^{n} i^p,$$

que también aparecen con frecuencia en el análisis de algoritmos, tanto iterativos como recursivos. Como se ha visto previamente, $S_0(n) = n$ y $S_1(n) = n(n+1)/2$. Ahora veremos sumatorios para valores más grandes de p.

Suma de los primeros n cuadrados ($p = 2$)

La suma de los primeros n números positivos al cuadrado es:

$$S_2(n) \;=\; \sum_{i=1}^{n} i^2 = 1^2 + 2^2 + \cdots + (n-1)^2 + n^2 \;=\; \frac{(2n+1)n(n+1)}{6}. \qquad (1.8)$$

Para demostrar la identidad haremos uso del hecho de que un número al cuadrado se puede expresar como una suma de números impares. Así, podemos descomponer el sumatorio de la siguiente manera:

$$\begin{array}{rcccccccccccl}
S_2(n) = \sum_{i=1}^{n} i^2 & = & 1 & + & 4 & + & 9 & + & 16 & + & \cdots & + & n^2 \\
\hline
 & & 1 & + & 1 & + & 1 & + & 1 & + & & + & 1 & = 1(n) \\
 & & & + & 3 & + & 3 & + & 3 & + & & + & 3 & = 3(n-1) \\
 & & & & & + & 5 & + & 5 & + & & + & 5 & = 5(n-2) \\
 & & & & & & & + & 7 & + & & + & 7 & = 7(n-3) \\
 & & & & & & & & & & & & \vdots & \quad \vdots \\
 & & & & & & & & & & & + & 2n-1 & = (2n-1)1
\end{array}$$

A continuación sumamos los términos de la última columna:

$$\begin{aligned}
S_2(n) \;&=\; \sum_{i=1}^{n} i^2 = 1 \cdot n + 3 \cdot (n-1) + 5 \cdot (n-2) + \cdots + (2n-1) \cdot 1 \\
&=\; \sum_{i=1}^{n} (2i-1)(n-i+1) \;=\; \sum_{i=1}^{n} \left(2in - 2i^2 + 2i - n + i - 1\right) \\
&=\; \sum_{i=1}^{n} (2n+3)i - 2\underbrace{\sum_{i=1}^{n} i^2}_{S_2(n)} - \sum_{i=1}^{n} n - \sum_{i=1}^{n} 1 \\
&=\; (2n+3)\sum_{i=1}^{n} i - 2S_2(n) - n^2 - n
\end{aligned}$$

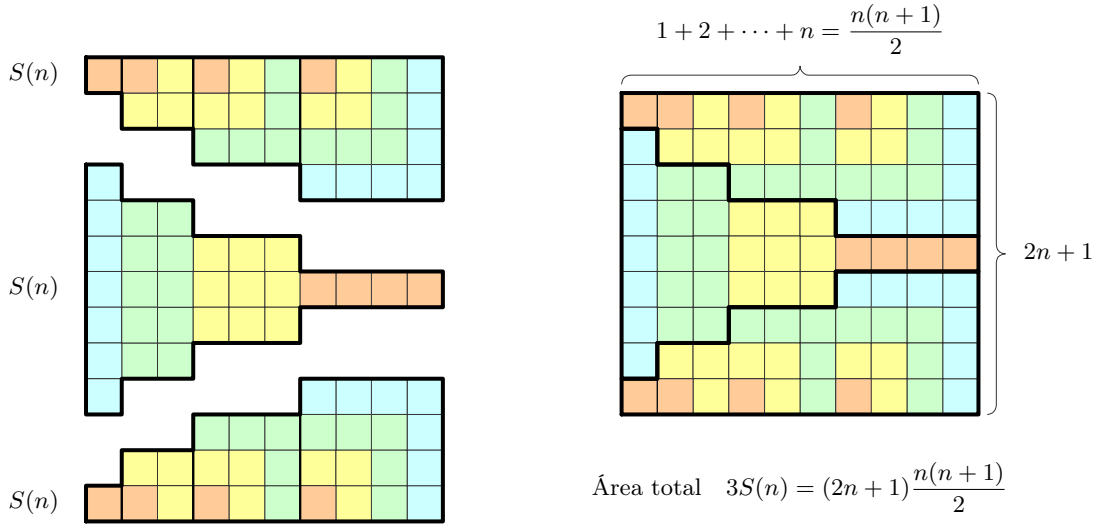


Figura 1.10: Demostración visual para la fórmula (polinomio cúbico) de la suma de los n primeros números positivos al cuadrado.

Continuando tenemos:

$$\begin{aligned} S_2(n) &= (2n+3)\frac{n(n+1)}{2} - 2S_2(n) - n^2 - n \\ 3S_2(n) &= \frac{(2n+3)n(n+1)}{2} - n^2 - n = \frac{n(2n^2+3n+1)}{2} \\ S_2(n) &= \frac{n(2n^2+3n+1)}{6} = \frac{(2n+1)n(n+1)}{6} \end{aligned}$$

Por último, la Figura 1.10 ofrece una demostración visual del sumatorio.

Estrategia general para mayores valores de p

La suma de los primeros cubos se puede expresar de la siguiente manera:

$$S_3(n) = \sum_{i=1}^{n} i^3 = \left(\sum_{i=1}^{n} i\right)^2 = \left(\frac{n(n+1)}{2}\right)^2. \tag{1.9}$$

Se puede demostrar de varias maneras (por ejemplo, inducción o visualmente). En este apartado derivaremos la fórmula a través de un procedimiento general para hallar cualquier sumatorio $S_p(n)$. Para ello, antes es necesario conocer $S_k(n)$ para $k < p$.

El primer paso consiste en plantear el siguiente sumatorio telescópico:

$$S = \sum_{i=1}^{n} \left[(i+1)^{p+1} - i^{p+1}\right].$$

Por tanto, para $p = 3$ consideraremos el siguiente sumatorio, el cual se puede simplificar a la resta de dos términos:

$$S = \sum_{i=1}^{n} \left[(i+1)^4 - i^4\right] = (n+1)^4 - 1^4.$$

Por otro lado, desarrollando la función dentro del sumatorio tenemos (véase (1.1)):

$$\begin{aligned} S &= \sum_{i=1}^{n} \left[i^4 + 4i^3 + 6i^2 + 4i + 1 - i^4\right] = \sum_{i=1}^{n} \left[4i^3 + 6i^2 + 4i + 1\right] \\ &= 4\sum_{i=1}^{n} i^3 + 6\sum_{i=1}^{n} i^2 + 4\sum_{i=1}^{n} i + \sum_{i=1}^{n} 1 = 4S_3(n) + 6S_2(n) + 4S_1(n) + S_0(n). \end{aligned}$$

Combinando los dos resultados para S tenemos:

$$(n+1)^4 - 1^4 = 4S_3(n) + 6S_2(n) + 4S_1(n) + S_0(n),$$

$$n^4 + 4n^3 + 6n^2 + 4n + 1 - 1 = 4S_3(n) + 6\frac{(2n+1)n(n+1)}{6} + 4\frac{n(n+1)}{2} + n.$$

Finalmente, despejando $S_3(n)$ y simplificando se obtiene (1.9).

1.3.5 Sumatorios de progresiones geométricas y variantes

En este apartado veremos sumatorios en los que la variable contadora aparece en el exponente de una potencia.

Suma parcial de una serie geométrica

Una progresión geométrica es aquella en la que un término es igual al anterior multiplicado por la misma constante r, denominada razón o factor. Formalmente se pueden definir con la siguiente regla, $a_i = r \cdot a_{i-1}$, para $i > 0$, y para algún valor inicial de a_0. De manera no recursiva tenemos:

$$a_i = r \cdot a_{i-1} = r^2 \cdot a_{i-2} = \cdots = r^i \cdot a_0.$$

La suma parcial de una serie o progresión geométrica es:

$$S(m,n) = \sum_{i=m}^{n} a_i = \sum_{i=m}^{n} r^i \cdot a_0 = a_0 \sum_{i=m}^{n} r^i = a_0 \frac{r^m - r^{n+1}}{1-r}, \tag{1.10}$$

para $r \neq 1$. Esta fórmula se puede obtener a través del siguiente enfoque, que usaremos para otros sumatorios. Se trata de considerar los dos sumatorios $S(m,n)$ y $rS(m,n)$, y restarlos. La ventaja reside en que se cancelan todos los términos excepto dos. El

proceso (ignorando a_0) se puede ilustrar de la siguiente manera:

$$\begin{array}{rlllllllll} & S(m,n) & = & r^m & + r^{m+1} + & r^{m+2} + & \ldots + & r^{n-1} + & r^n & \\ - & rS(m,n) & = & & + r^{m+1} + & r^{m+2} + & \ldots + & r^{n-1} + & r^n & + r^{n+1} \\ \hline & S(m,n) - rS(m,n) & = & r^m & & & & & & - r^{n+1} \end{array}$$

y despejando $S(m,n)$ se obtiene (1.10). Como ejemplo, un sumatorio que suele aparecer al analizar algoritmos es la suma de las primeras potencias de 2:

$$\sum_{i=0}^{n-1} 2^i = \frac{2^n - 1}{2 - 1} = 2^n - 1.$$

Por último, una serie geométrica converge a un valor constante si $|r| < 1$:

$$\sum_{i=0}^{\infty} r^i = \frac{1}{1 - r}.$$

Variante

En este apartado hallaremos una expresión simplificada para el siguiente sumatorio:

$$S(n) = \sum_{i=1}^{n} ir^i = 1 \cdot r^1 + 2 \cdot r^2 + 3 \cdot r^3 + \cdots + (n-1) \cdot r^{n-1} + n \cdot r^n. \tag{1.11}$$

Esta suma es más complicada que la suma parcial de una serie geométrica, pero se puede simplificar procediendo de manera análoga: restando $S(n)$ y $rS(n)$:

$$\begin{array}{rlllllllll} S(n) & = & 1 \cdot r^1 & + & 2 \cdot r^2 & + & 3 \cdot r^3 & + \cdots + & n \cdot r^n & \\ rS(n) & = & & + & 1 \cdot r^2 & + & 2 \cdot r^3 & + \cdots + & (n-1) \cdot r^n & + nr^{n+1} \\ \hline S(n) - rS(n) & = & r^1 & + & r^2 & + & r^3 & + \cdots + & r^n & - nr^{n+1} \end{array}$$

Ahora no se cancelan casi todos los términos pero se simplifican bastante. Teniendo en cuenta que $r + r^2 + \cdots + r^n$ es una suma parcial de una serie geométrica podemos usar (1.10) para simplificar el resultado. En concreto tenemos:

$$S(n) - rS(n) = \frac{r^{n+1} - r}{r - 1} - nr^{n+1},$$

$$S(n) = \frac{r - r^{n+1}}{(r-1)^2} + \frac{nr^{n+1}}{r-1} = \frac{r - r^{n+1} + (r-1)nr^{n+1}}{(r-1)^2} = \frac{r + r^{n+1}(nr - n - 1)}{(r-1)^2}.$$

Existen otros métodos para simplificar este sumatorio. Consideremos una suma par-

cial de una serie geométrica T con razón x y su derivada con respecto a x:

$$\begin{array}{ccccc} T & = & 1 + x + x^2 + \cdots + x^{m-1} & = & \dfrac{x^m - 1}{x - 1} \\ \downarrow & & \downarrow \text{ derivando} & & \downarrow \\ \dfrac{dT}{dx} & = & 1 + 2x + 3x^2 + \cdots + (m-1)x^{m-2} & = & \dfrac{mx^{m-1}(x-1) - 1 \cdot (x^m - 1)}{(r-1)^2} \end{array}$$

La expresión simplificada de la derivada surge de derivar la fórmula simplificada de T. Ahora bien, la derivada de T es muy parecida al sumatorio en (1.11) que pretendemos simplificar. Para obtener el mismo sumatorio sólo tenemos que multiplicar todo por x y después sustituir x por r, y m por $n+1$. Formalmente:

$$S(n) = x\frac{dT}{dx}\bigg|_{x=r,\ m=n+1} = r\frac{(n+1)r^n(r-1) - (r^{n+1} - 1)}{(x-1)^2} = \frac{r + r^{n+1}(nr - n - 1)}{(r-1)^2}.$$

1.3.6 Aproximación de sumatorios por integrales

De la misma manera que no se conocen expresiones analíticas (es decir, fórmulas) para muchas integrales, tampoco podremos hallar fórmulas simplificadas exactas para numerosos sumatorios. De todas formas, en ciertos contextos se puede trabajar con expresiones matemáticas que se aproximen al sumatorio. En este apartado veremos cómo obtener ciertas aproximaciones que resultarán ser *cotas* inferiores o superiores de sumatorios, cuando las funciones dentro de ellos sean monótonas crecientes o decrecientes (y no negativas).

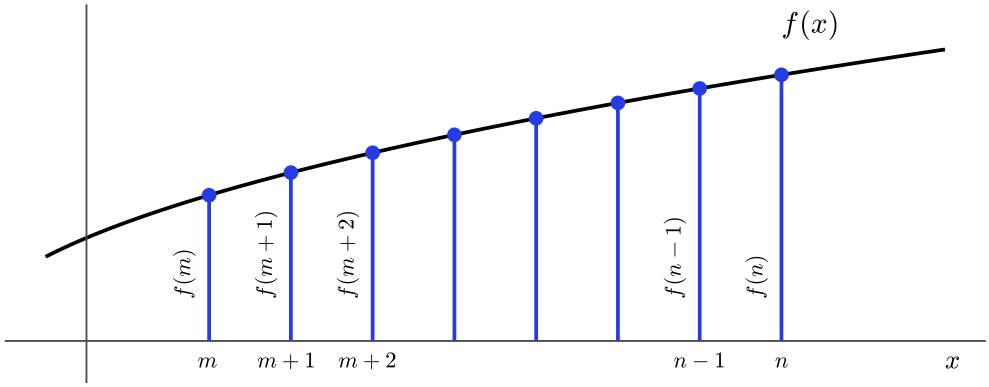
En concreto, para funciones monótonas crecientes las cotas son:

$$\int_{m-1}^{n} f(x)\,dx \quad \leq \quad \sum_{i=m}^{n} f(i) \quad \leq \quad \int_{m}^{n+1} f(x)\,dx,$$

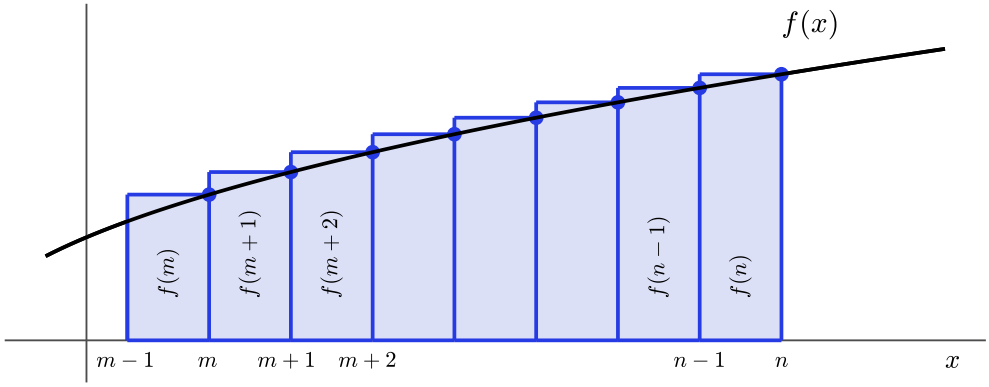
mientras que para funciones monótonas decrecientes las cotas son:

$$\int_{m}^{n+1} f(x)\,dx \quad \leq \quad \sum_{i=m}^{n} f(i) \quad \leq \quad \int_{m-1}^{n} f(x)\,dx$$

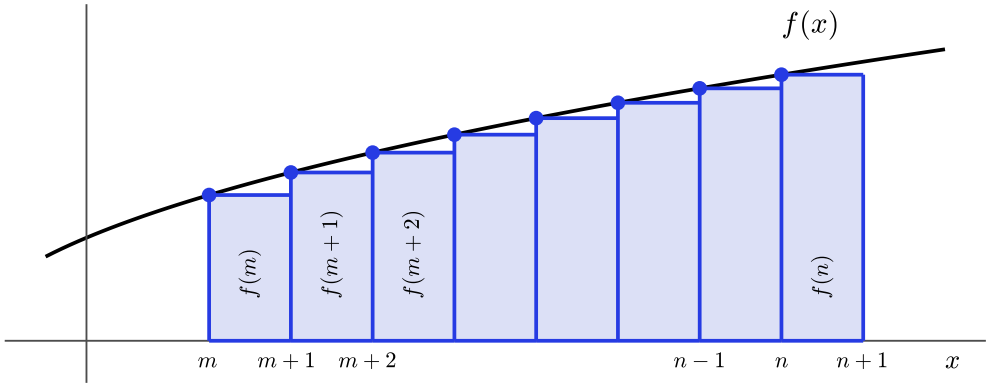
Las Figuras 1.11 y 1.12 muestran cómo surgen estas cotas. En primer lugar, el sumatorio de $f(i)$ evaluado desde m hasta n no es más que la suma de las alturas de la función $f(i)$ en los puntos $i = m, m+1, \ldots, n$. Por otro lado, como la diferencia entre puntos consecutivos es 1, se pueden construir rectángulos cuya área (base = 1, por altura = $f(i)$) sea igual a $f(i)$. De esta manera, la suma de las áreas de los rectángulos sombreados será igual al sumatorio. Por último, teniendo en cuenta que una integral definida mide áreas por debajo de la curva entre sus dos intervalos, se obtienen las desigualdades que dan lugar a las cotas del sumatorio. Lógicamente, para que estas



$$\sum_{i=m}^{n} f(i) \;=\; \text{suma de las longitudes de los segmentos}$$

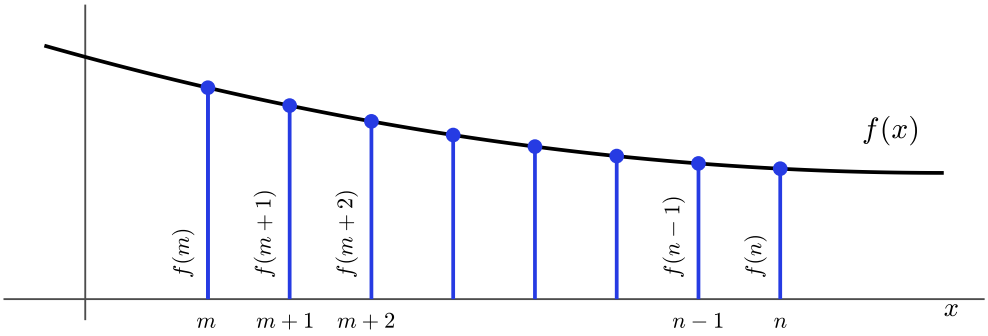


$$\text{Cota inferior} \;=\; \int_{m-1}^{n} f(x)\,dx \;=\; \text{suma de las áreas de los rectángulos} \;\leq\; \sum_{i=m}^{n} f(i)$$

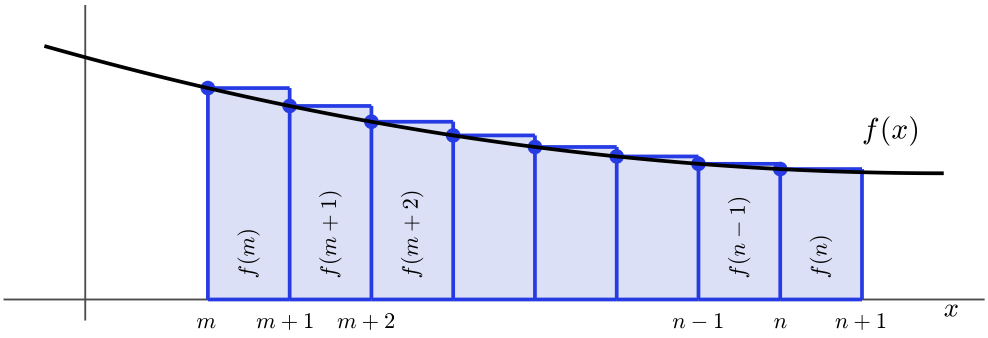


$$\sum_{i=m}^{n} f(i) \;=\; \text{suma de las áreas de los rectángulos} \;\leq\; \int_{m}^{n+1} f(x)\,dx \;=\; \text{Cota superior}$$

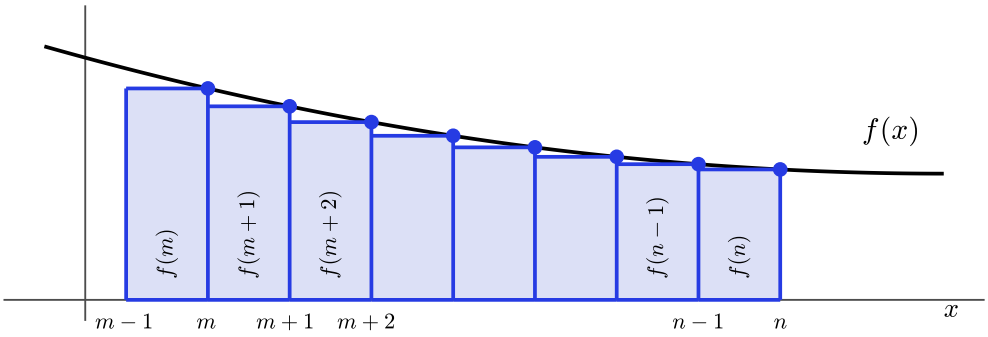
Figura 1.11: Derivación de cotas inferior y superior para sumatorios de funciones monótonas crecientes.



$$\sum_{i=m}^{n} f(i) \;=\; \text{suma de las longitudes de los segmentos}$$



$$\text{Cota inferior} \;=\; \int_{m}^{n+1} f(x)\,dx \;=\; \text{suma de las áreas de los rectángulos} \;\leq\; \sum_{i=m}^{n} f(i)$$



$$\sum_{i=m}^{n} f(i) \;=\; \text{suma de las áreas de los rectángulos} \;\leq\; \int_{m-1}^{n} f(x)\,dx \;=\; \text{Cota superior}$$

Figura 1.12: Derivación de cotas inferior y superior para sumatorios de funciones monótonas decrecientes.

cotas puedan usarse en la práctica, se deben conocer las fórmulas correspondientes a las integrales (ya que nuestro objetivo es sustituir un sumatorio por una fórmula, aunque sólo sea una aproximación).

1.3.7 Productorios

De manera análoga a los sumatorios, un *productorio* (también llamado simplemente producto) es la multiplicación de varios términos definidos mediante una función $f(i)$, evaluados en enteros consecutivos desde un índice inicial m hasta uno final n. Se expresa de la siguiente manera:

$$\prod_{i=m}^{n} f(i) = f(m) \cdot f(m+1) \cdot \ \cdots \ \cdot f(n-1) \cdot f(n),$$

donde se suele asumir que vale 1 si $m > n$. Por ejemplo, la función factorial se puede definir como:

$$n! = \prod_{i=1}^{n} i = 1 \cdot 2 \cdot \ \cdots \ \cdot (n-1) \cdot n,$$

donde además $0! = 1$.

Al igual que en los sumatorios, los términos multiplicativos que no dependan de la variable contadora se pueden extraer fuera del productorio. Sin embargo, si el productorio multiplica a n términos, esos términos deben elevarse a n:

$$\prod_{i=1}^{n} k f(i) = k^n \prod_{i=1}^{n} f(i).$$

Además, en general, el producto de una suma no es la suma de productos:

$$\prod_{i=m}^{n} \big(f_1(i) + f_2(i)\big) \neq \prod_{i=m}^{n} f_1(i) + \prod_{i=m}^{n} f_2(i).$$

Una operación muy habitual consiste en considerar el logaritmo de un productorio, el cual se puede expresar como un sumatorio de logaritmos:

$$\log\left(\prod_{i=m}^{n} f(i)\right) = \sum_{i=m}^{n} \log f(i),$$

Lo cual puede simplificar fórmulas matemáticas considerablemente.

En probabilidad suele ser habitual trabajar con productos de varias probabilidades. Pues bien, a la hora de comparar estos productos se puede emplear el resultado del productorio tal cual, o el logaritmo de dicho productorio. Esto es válido ya que un logaritmo (con base mayor que 1) es una función monótona creciente, y por tanto no afecta a las desigualdades (si $a < b$ entonces $\log(a) < \log(b)$).

Además, existe otra razón importante para aplicar logaritmos a productos de probabilidades. Si se multiplican muchos números en $[0, 1]$ el resultado puede ser muy pequeño, dando lugar a errores de subdesbordamiento o *underflow*. El uso de logaritmos puede resolver este problema (junto con el denominado procedimiento *log-sum-exp trick*).

2

Complejidad Computacional

El *análisis de algoritmos* es el campo que estudia cómo estimar teóricamente los recursos que necesitan los algoritmos para resolver problemas computacionales. Este capítulo se centrará en la *complejidad computacional* en tiempo (también se puede estudiar en espacio/memoria, en cuanto al ancho de banda, u otros recursos) de algoritmos. Ésta es una medida teórica de cuánto tiempo, o cuántas operaciones, se necesitan para resolver un problema (el Código 2.1 muestra una forma sencilla de medir tiempos de ejecución en Python, a través del módulo `time`). Se determina analizando una función, digamos T, del tamaño de la entrada que cuantifica este número de operaciones para una instancia particular del problema. En informática, la eficiencia de los algoritmos se estudia generalmente contemplando cómo se comporta T cuando el tamaño del problema es muy grande. Además, el factor clave es la velocidad a la que T crece a medida que el tamaño de la entrada tiende a infinito. En los siguientes apartados se explican estas ideas y la notación matemática que se suele utilizar para caracterizar la complejidad computacional. Por último, aunque el tamaño de un problema puede depender de varios factores, nos centraremos en analizar la complejidad temporal computacional de algoritmos que resuelven problemas cuyo tamaño viene determinado por un único factor (por ejemplo, la longitud de una lista). Así, el coste del tiempo de ejecución de los algoritmos tratados vendrá determinado por una función $T(n)$ de un parámetro, donde n suele representar el tamaño del problema.

Código 2.1: Medición de tiempos de ejecución a través del módulo `time` de Python.

```
import time

t = time.time()
# Ejecuta código aquí
tiempo_transcurrido = time.time() - t
print(tiempo_transcurrido)
```

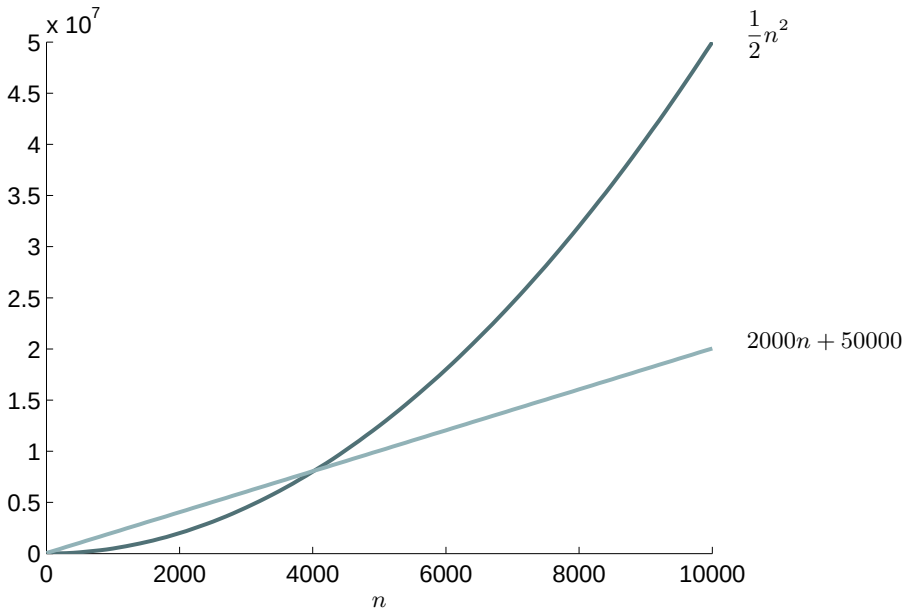


Figura 2.1: El término de orden superior determina el orden de crecimiento de una función. Para $T(n) = 0,5n^2 + 2000n + 50000$ el orden es cuadrático, ya que el término $0,5n^2$ domina claramente a los términos de orden inferior (incluso sumados) para valores grandes de n.

2.1 Orden de crecimiento de funciones

La función $T(n)$, que cuantifica el tiempo de ejecución de un algoritmo, puede contener varios términos aditivos que contribuyen de forma diferente a su valor para una entrada dada. Por ejemplo, sea $T(n) = 0,5n^2 + 2000n + 50000$. Para valores moderados de n todos los términos son relevantes respecto a la magnitud de $T(n)$. Sin embargo, a medida que n aumenta, el término principal $(0,5n^2)$ afecta al crecimiento de la función considerablemente más que los otros dos (incluso combinados). Por lo tanto, el orden de crecimiento de la función se caracteriza por su término de orden superior. La Figura 2.1 muestra un gráfico de $0,5n^2$ junto con $2000n+50000$. Para entradas grandes es evidente que el término cuadrático domina a los otros dos, y por lo tanto caracteriza el orden de

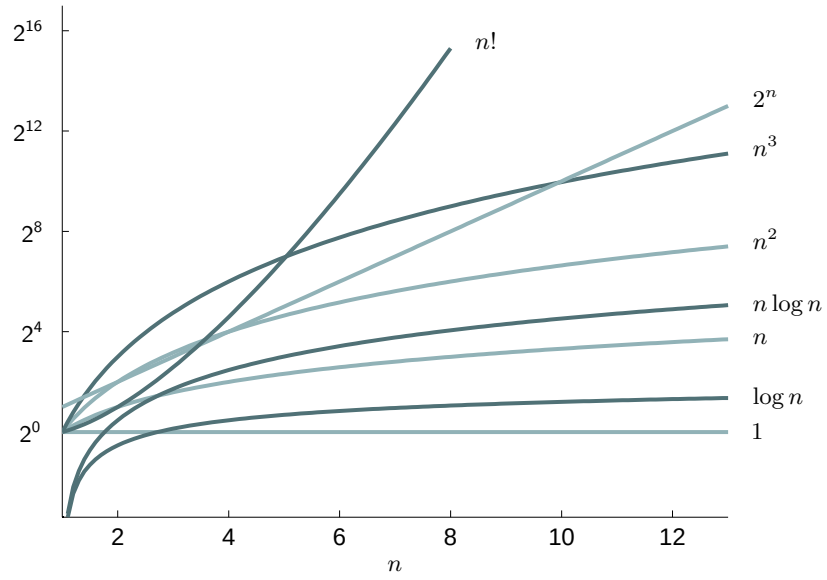


Figura 2.2: Órdenes de crecimiento frecuentes en complejidad computacional.

crecimiento de $T(n)$. Los coeficientes del polinomio se han elegido para ilustrar que no desempeñan un papel significativo en la determinación del orden de crecimiento de la función.

La Figura 2.2 dibuja varios órdenes de crecimiento que suelen aparecer al analizar complejidad computacional. Pueden ordenarse de la siguiente manera cuando se consideran valores grandes de n:

$$1 \quad < \quad \log n \quad < \quad n \quad < \quad n \log n \quad < \quad n^2 \quad < \quad n^3 \quad < \quad 2^n \quad < \quad n!,$$

donde, informalmente, $f(n) < g(n)$ si:

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

Los órdenes de crecimiento anteriores se denominan (de izquierda a derecha) "constante", "logarítmico", "lineal", "n-log-n", "cuadrático", "cúbico", "exponencial" y "factorial".

Dado que la escala del eje Y en la Figura 2.2 es logarítmica, las diferencias entre los órdenes de crecimiento pueden parecer mucho menores de lo que son en realidad (la diferencia entre marcas consecutivas significa que un algoritmo es 16 veces más lento/rápido). La Tabla 2.1 muestra valores concretos para las funciones, donde destacan claramente las rápidas tasas de crecimiento de las funciones exponenciales o factoriales. Los problemas que no pueden resolverse mediante algoritmos en tiempo polinómico suelen considerarse intratables, ya que los métodos tardarían demasiado en terminar incluso para problemas de tamaño moderado. Por el contrario, los problemas que pueden

Tabla 2.1: Valores concretos de funciones frecuentes en complejidad computacional.

1	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
1	0	1	0	1	1	2	1
1	1	2	2	4	8	4	2
1	2	4	8	16	64	16	24
1	3	8	24	64	512	256	40320
1	4	16	64	256	4096	65.536	$2{,}09 \cdot 10^{13}$
1	5	32	160	1024	32.768	4.295.967.296	$2{,}63 \cdot 10^{35}$

resolverse en tiempo polinómico se consideran abordables. Sin embargo, la línea que separa los problemas tratables de los intratables puede ser sutil. Si el tiempo de ejecución de un algoritmo se caracteriza por ser de orden polinómico con un grado grande, en la práctica podría tardar demasiado en obtener una solución o en que sus resultados intermedios fueran útiles.

2.1.1 Notación asintótica

Un detalle importante sobre los órdenes mencionados hasta ahora es la ausencia de términos multiplicativos constantes. Al igual que los términos de orden inferior, éstos se omiten, ya que son menos relevantes que el orden real de crecimiento a la hora de determinar la eficiencia computacional para entradas grandes. Además, no merece la pena el esfuerzo de especificar la eficiencia de un algoritmo con precisión exacta, ya que su tiempo de ejecución depende de numerosos factores que incluyen el hardware del ordenador, el lenguaje de programación, el compilador o intérprete, y muchos otros. Así pues, en la práctica basta con suponer que simplemente se tarda una cantidad de tiempo constante en ejecutar una instrucción básica, donde su valor es irrelevante.

El análisis teórico de algoritmos y problemas se basa en un tipo de notación denominada *notación asintótica*, que nos permite descartar los términos de orden inferior y las constantes multiplicativas cuando tratamos con entradas arbitrariamente grandes. En particular, la notación asintótica proporciona definiciones de conjuntos que podemos utilizar para especificar *cotas asintóticas*.

La notation "O-grande" define el siguiente conjunto:

$$\mathcal{O}(g(n)) = \Big\{ f(n) : \exists\ c > 0 \text{ y } n_0 > 0 \ /\ 0 \leq f(n) \leq c \cdot g(n), \forall n \geq n_0 \Big\}.$$

Si una función $f(n)$ pertenece a este conjunto, entonces $g(n)$ será una *cota superior asintótica* para $f(n)$. Esto significa que $g(n)$ será mayor que $f(n)$, pero la definición sólo requiere que esto sea cierto en un intervalo desde algún punto $n_0 > 0$ hasta el infinito, donde además podemos multiplicar $g(n)$ por una constante positiva c lo suficientemente grande. La Figura 2.3(a) ilustra gráficamente la idea. Si $g(n)$ es una cota superior asintótica para $f(n)$ entonces debe ser posible encontrar una constante positiva c tal

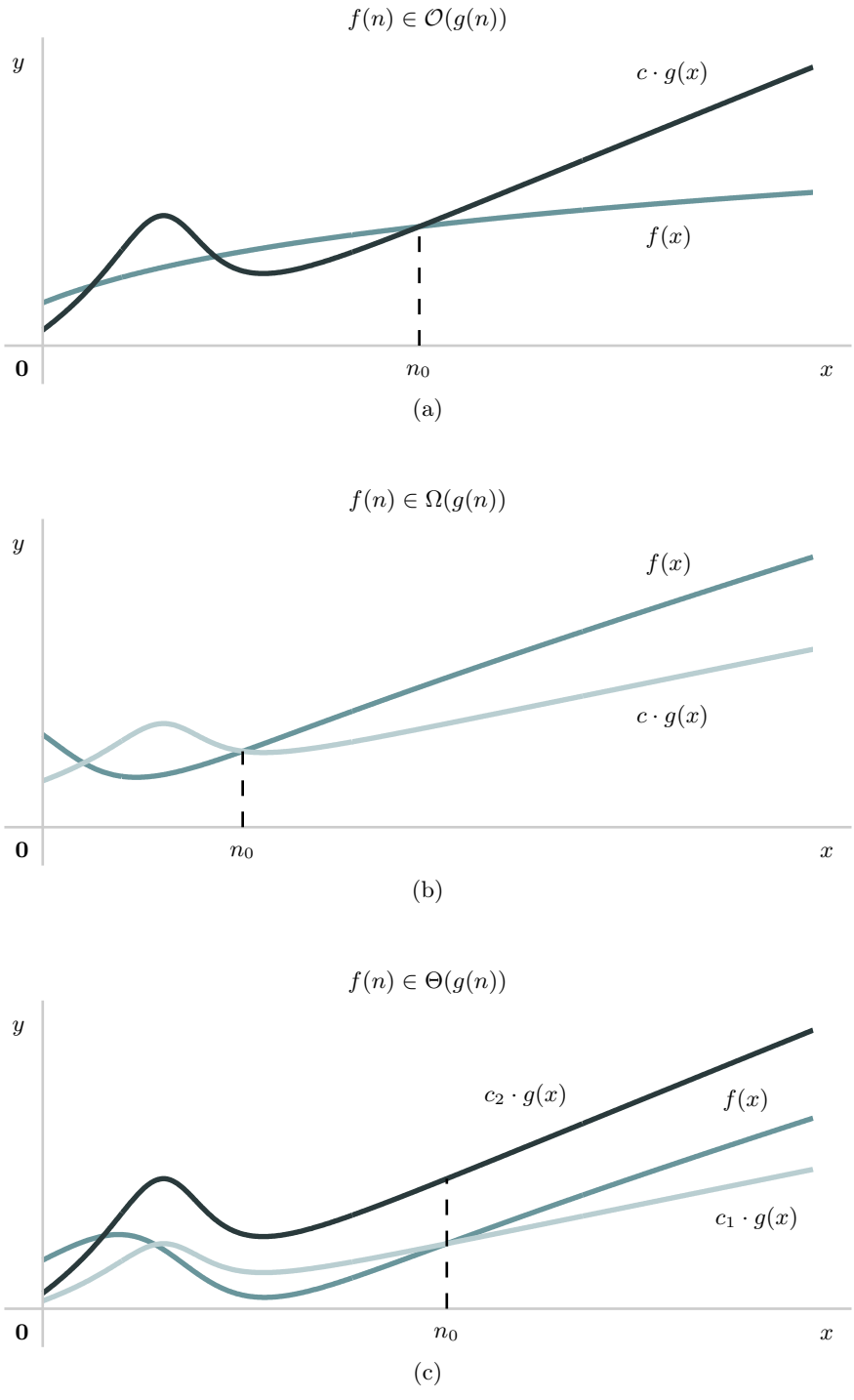


Figura 2.3: Ilustraciones gráficas de las definiciones de notación asintótica usadas en complejidad computacional.

que $cg(n) \geq f(n)$, pero a partir de algún valor positivo n_0 hasta el infinito (lo que ocurra para $n < n_0$ es irrelevante). Para demostrar que una función pertenece a $\mathcal{O}(g(n))$ basta con demostrar la existencia de un par de constantes c y n_0 que satisfagan la definición, ya que no son únicas. Por ejemplo, si la definición es cierta para una pareja c y n_0 particular, entonces también lo será para valores mayores de c y n_0. En este sentido, no es necesario proporcionar los valores más bajos de c y n_0 que satisfagan la definición $\mathcal{O}$ (esto también se aplica a las notaciones que mencionaremos a continuación).

Los algoritmos suelen compararse en función de su eficiencia en el peor caso, que corresponde a una instancia de un problema, entre todas las que comparten el mismo tamaño, para la que el algoritmo necesita más recursos (tiempo, almacenamiento, etc.). Dado que la notación O-grande especifica límites superiores asintóticos, puede utilizarse para garantizar que un algoritmo concreto necesitará como máximo una determinada cantidad de recursos, incluso en el peor de los casos, para entradas grandes. Por ejemplo, el tiempo de ejecución del algoritmo *quicksort* que ordena una lista o vector de n elementos pertenece a $\mathcal{O}(n^2)$ en general, ya que requiere realizar del orden de n^2 comparaciones en el peor de los casos. Sin embargo, el *quicksort* puede ejecutarse más rápidamente (en tiempo $n \log n$ en el caso mejor y promedio).

En cambio, la notación "Omega-grande" define cotas inferiores asintóticas:

$$\Omega(g(n)) = \Big\{ f(n) : \exists\ c > 0 \text{ y } n_0 > 0 \ /\ 0 \leq c \cdot g(n) \leq f(n), \forall n \geq n_0 \Big\}.$$

La Figura 2.3(b) ilustra gráficamente la idea. La notación Omega-grande es útil para especificar un límite inferior en los recursos necesarios para resolver un problema, independientemente del algoritmo que se aplique. Por ejemplo, es posible demostrar teóricamente que cualquier algoritmo capaz de ordenar una lista de n números reales requerirá $\Omega(n \log n)$ comparaciones en el peor caso.

Finalmente, la notación "Theta-grande" define *cotas asintóticas ajustadas*:

$$\Theta(g(n)) = \Big\{ f(n) : \exists\ c_1 > 0,\ c_2 > 0, \text{ y } n_0 > 0\ / \\ 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n), \forall n \geq n_0 \Big\}.$$

Si $f(n) \in \Theta(g(n))$ entonces $f(n)$ y $g(n)$ compartirán el mismo orden de crecimiento. Así, eligiendo dos constantes apropiadas c_1 y c_2, $g(n)$ será tanto un límite asintótico superior como inferior de $f(n)$, como se muestra en la Figura 2.3(c). En otras palabras, $f(n) \in \mathcal{O}(g(n))$ y $f(n) \in \Omega(g(n))$. Por ejemplo, el algoritmo *merge sort* para ordenar una lista o vector de n elementos siempre requiere (en el mejor y peor de los casos) del orden de $n \log n$ comparaciones. Por tanto, decimos que su tiempo de ejecución pertenece a $\Theta(n \log n)$.

Al especificar un límite asintótico, las constantes y los términos de orden inferior se eliminan de la función $g(n)$. Por ejemplo, aunque es cierto que $3n^2 + 10n \in O(5n^2 + 20n)$, basta con indicar $3n^2 + 10n \in O(n^2)$, ya que estas notaciones indican órdenes de crecimiento. A este respecto, el lector puede haber notado la ausencia de la base

en logaritmos cuando se describe el orden de crecimiento de una función. La razón es que la diferencia entre dos logaritmos de bases distintas es un término multiplicativo constante. Sea ρ alguna cota asintótica ($\mathcal{O}$, Ω o Θ), y a y b dos bases de logaritmos. Las siguientes identidades indican que todos los logaritmos comparten el mismo orden, independientemente de su base:

$$\rho\left(\log_a g(n)\right) = \rho\left(\frac{\log_b g(n)}{\log_b a}\right) = \rho\left(\underbrace{\frac{1}{\log_b a}}_{\text{constante}} \log_b g(n)\right) = \rho\left(\log_b g(n)\right).$$

Así, la base de un logaritmo no se especifica al indicar su orden de crecimiento.

Demostraciones

En este apartado veremos cómo demostrar si $f(n) \in \mathcal{O}(g(n)$ o $f(n) \in \Omega(g(n)$ empleando las definiciones anteriores directamente (el Apartado 2.1.1 ilustra otro enfoque basado en límites matemáticos). El proceso de demostración variará dependiendo de si realmente se verifica la pertenencia (aquello que queramos demostrar). Si es cierto que se verifica, entonces deberemos encontrar una pareja de constantes c y n_0 tal que se cumpla la definición formal. En cambio, si $f(n) \notin \mathcal{O}(g(n)$ o $f(n) \notin \Omega(g(n)$, en lugar de buscar una pareja de constantes c y n_0 hay que demostrar que va a ser imposible encontrarla.

Veamos dos ejemplos para ilustrar ambas estrategias. Supongamos que queremos demostrar si $5n + 2 \in \mathcal{O}(n^2)$. Como la expresión es cierta debemos buscar una pareja de constantes que satisfagan la definición de $\mathcal{O}$. Para ello planteamos la siguiente desigualdad (usaríamos $\geq$ si la expresión a demostrar involucrara a Ω):

$$5n + 2 \leq cn^2.$$

Después, escogemos algún valor de $c > 0$ adecuado, que, intuitivamente, haga que el miembro derecho sea mayor que el izquierdo. Para este ejemplo concreto tomar cualquier valor de c serviría, por ejemplo $c = 1$, ya que un orden cuadrático siempre es mayor que uno lineal, independientemente de c (si el orden de ambos miembros fuera el mismo entonces sería imprescindible escoger c lo suficientemente grande para asegurarse de que el miembro derecho tomara valores mayores que el izquierdo). Habiendo escogido $c = 1$ el siguiente paso consiste en ver para qué valores de n se cumple la desigualdad resultante:

$$5n + 2 \leq n^2 \qquad \Leftrightarrow \qquad 0 \leq n^2 - 5n - 2.$$

La función $n^2 - 5n - 2$ es una parábola convexa (con un mínimo) con raíces en $(5 \pm \sqrt{33})/2$. Por tanto, la desigualdad se cumple para todo $n \geq (5 + \sqrt{33})/2$, y, para $c = 1$, cualquier valor para n_0 mayor o igual que $(5 + \sqrt{33})/2$ sería válido de cara a la definición de $\mathcal{O}$. Recordamos que los valores de estas constantes son irrelevantes, ya

que lo importante es que existan (que hayamos encontrado una pareja). Por ejemplo, si hubiésemos escogido $c = 5$ entonces $n_0 = 2$ sería válido, y si hubiésemos seleccionado $c = 8$, entonces $n_0 = 1$ sería suficiente.

En cambio, supongamos que queremos verificar si se cumple $3n^2+2n-2 \in \mathcal{O}(n)$. En este caso la expresión no es cierta ya que el orden de la función cuadrática es claramente mayor que lineal. Ahora, al plantear

$$3n^2 + 2n - 2 \leq cn$$

ya no debemos escoger un valor para c. En este caso debemos razonar por qué, cojamos la constante c que cojamos, la desigualdad nunca se va a cumplir en un intervalo $[n_0, \infty)$. En este ejemplo primero podemos pasar el término cn al miembro izquierdo:

$$3n^2 + (2 - c)n - 2 \leq 0,$$

que vuelve a ser una parábola convexa, sea cual sea el valor de c. Sin embargo, en este caso debemos buscar los valores de n que hagan que la parábola sea negativa. Claramente, esto solo puede ocurrir en un intervalo finito, y nunca desde un n_0 hasta infinito. Esto implica que no va a ser posible encontrar una pareja de constantes c y n_0, y por tanto podemos concluir que $3n^2 + 2n - 2 \notin \mathcal{O}(n)$.

Límites

También podemos utilizar límites para determinar el orden de las funciones, debido a las siguientes sentencias equivalentes que los relacionan con las definiciones de cotas asintóticas:

$$f(n) \in \mathcal{O}(g(n)) \;\Leftrightarrow\; \lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty \quad \text{(constante o cero)},$$

$$f(n) \in \Omega(g(n)) \;\Leftrightarrow\; \lim_{n\to\infty} \frac{f(n)}{g(n)} > 0 \quad \text{(constante} > 0, \text{ o infinito)},$$

$$f(n) \in \Theta(g(n)) \;\Leftrightarrow\; \lim_{n\to\infty} \frac{f(n)}{g(n)} = \text{constante} > 0.$$

Además de las definiciones descritas en el Apartado 2.1.1, existen otras cotas asintóticas relacionadas. Por ejemplo, si $f(n) \in \mathcal{O}(g(n))$ decimos que el orden de crecimiento de f menor o igual que el de g. Pues bien, la notación o ("o-pequeña") se usa para indicar una *cota superior estricta*. Así, si $f(n) \in o(g(n))$ decimos que el orden de crecimiento de f debe ser *necesariamente menor* que el de g. Análogamente, ω ("omega-pequeña") indica una *cota inferior estricta*. Por tanto, si $f(n) \in \omega(g(n))$ entonces el orden de f debe ser estrictamente mayor que el de g. Los límites asociados para estas definiciones

son:

$$f(n) \in o(g(n)) \;\Leftrightarrow\; \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0,$$

$$f(n) \in \omega(g(n)) \;\Leftrightarrow\; \lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty.$$

$\log n! \in \Theta(n \log n)$

Un resultado importante en algoritmia es $\log n! \in \Theta(n \log n)$, ya que se usa, por ejemplo, para demostrar que cualquier algoritmos de ordenación capaz de ordenar una lista de n números reales requiere del orden de $n \log n$ operaciones. Es decir, no es posible construir un algoritmo general de ordenación que corra en tiempo lineal (existen algoritmos con coste lineal, pero requieren que los datos a ordenar cumplan ciertas propiedades, por ejemplo, que sean enteros).

Para demostrar el resultado (donde podemos asumir que la base del logaritmo es el número e) debemos verificar que $\log n! \in \mathcal{O}(n \log n)$ y que $\log n! \in \Omega(n \log n)$. Para $\mathcal{O}$ planteamos la desigualdad:

$$\log n! \;\leq\; n \log n.$$

Como con $n!$ no podemos trabajar directamente (por ejemplo, no podemos despejar n), debemos expresar el miembro izquierdo de otro modo:

$$\sum_{i=1}^{n} \log i \;\leq\; n \log n,$$

que a su vez podemos reescribir como:

$$\log 1 + \log 2 + \cdots + \log n \;\leq\; c(\underbrace{\log n + \log n + \cdots + \log n}_{n \text{ veces}}).$$

En este caso podemos escoger $c = 1$, y como el logaritmo es una función creciente (asumiendo que la base es mayor que 1, tendríamos $\log 1 \leq \log n$, $\log 2 \leq \log n$, etc.), la desigualdad se cumpliría para todo $n \geq 1$, y por tanto podemos tomar $n_0 = 1$. De esta manera, habríamos verificado que $\log n! \in \mathcal{O}(n \log n)$.

La demostración relativa a $\log n! \in \Omega(n \log n)$ es algo más compleja. En este caso planteamos:

$$\sum_{i=2}^{n} \log i \;\geq\; n \log n,$$

donde el sumatorio puede empezar en $i = 2$ ya que $log1 = 0$. No obstante, como no se conoce una fórmula para el sumatorio, resulta difícil trabajar con él directamente. En cambio, ya que tenemos una desigualdad, podemos aplicar los conceptos de cotas descritos en el Apartado 1.3.6 para trabajar con un a fórmula en vez del sumatorio. En

este caso nos interesa encontrar una cota inferior del sumatorio:

$$\sum_{i=2}^{n} \log i \;\geq\; \underbrace{\qquad g(n) \qquad}_{\substack{\text{Cota inferior de} \\ \sum_{i=2}^{n} \log i}} \;\geq\; cn\log(n).$$

Si demostramos $g(n) \geq cn\log(n)$ en un intervalo $[n_0, \infty)$ entonces necesariamente $\sum_{i=2}^{n} \log i \geq cn \log n$ en ese intervalo.

Como el logaritmo es una función monótona creciente, podemos hallar una cota inferior utilizando la aproximación por integrales:

$$\sum_{i=2}^{n} \log i \;\geq\; \int_{1}^{n} \log x \; dx$$

Resolviendo la integral por partes:

$$\int_{1}^{n} \log x \; dx = \Big[x \log x - x\Big]_{1}^{n} = n \log n - n - 0 + 1$$

Por tanto, combinando estos resultados:

$$\log n! = \sum_{i=2}^{n} \log i \geq \int_{1}^{n} \log x \; dx = n \log n - n + 1 \geq cn \log n$$

Lo cual implica que si podemos demostrar que $n \log n$ es cota inferior de $n \log n - n + 1$, entonces necesariamente será cota inferior de $\log n!$. De esta manera, solo queda encontrar las constantes c y n_0 para:

$$n \log n - n + 1 \geq cn \log n$$

Escogiendo $c = 1/2$ tenemos:

$$n \log n - n + 1 \geq \frac{1}{2} n \log n$$

$$\frac{1}{2} n \log n - n + 1 \geq 0$$

$$n\big(\frac{1}{2} \log n - 1\big) + 1 \geq 0$$

Lo cual es cierto si

$$\frac{1}{2} \log n - 1 \geq 0 \quad \Leftrightarrow \quad \log n \geq 2,$$

y por tanto se cumple para

$$n \geq e^2.$$

Así que $\log n! \in \Omega(n \log n)$ habiendo escogido $c = 1/2$ y $n_0 = e^2$.

Por último, también podemos verificar $\log n! \in \Theta(n \log n)$ si demostramos que el siguiente límite:

$$\lim_{n\to\infty} \frac{\log n!}{n \log n}.$$

es una constante mayor que 0. Si procedemos a calcularlo veremos que el resultado es ∞/∞. No obstante, no podremos aplicar la regla de L'Hopital al no poder derivar el factorial. De todas formas, podemos usar la *aproximación de Stirling* al factorial:

$$n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

que es válida para valores de n muy grandes, y por tanto podemos sustituir el factorial por dicha fórmula en nuestro límite:

$$\begin{aligned}
&\lim_{n\to\infty} \frac{\log(\sqrt{2\pi n}\left(\frac{n}{e}\right)^n)}{n \log n} \\
&= \lim_{n\to\infty} \frac{\log\sqrt{2\pi} + \log\sqrt{n} + \log\left(\frac{n}{e}\right)^n}{n \log n} \\
&= \lim_{n\to\infty} \frac{\log\sqrt{2\pi} + \frac{1}{2}\log n + n\log\frac{n}{e}}{n \log n} \\
&= \lim_{n\to\infty} \frac{\log\sqrt{2\pi} + \frac{1}{2}\log n + n\log n - n\log e}{n \log n} \\
&= \lim_{n\to\infty} \frac{\log\sqrt{2\pi}}{n \log n} + \lim_{n\to\infty} \frac{\frac{1}{2}\log n}{n \log n} + \lim_{n\to\infty} \frac{n \log n}{n \log n} - \lim_{n\to\infty} \frac{n \log e}{n \log n} \\
&= 0 + 0 + 1 - 0 = 1
\end{aligned}$$

En efecto, como el límite es una constante distinta de cero, tenemos $\log(n!) \in \Theta(n \log n)$.

2.2 Análisis de algoritmos iterativos

Este apartado contiene una breve introducción al análisis de algoritmos iterativos. Se analizarán dos métodos de ordenación (el algoritmo de inserción y el burbuja) de forma diferente, y se introducirán algunos conceptos básicos como el caso mejor, peor y medio.

2.2.1 Análisis por líneas

El algoritmo de ordenamiento por inserción, también denominado *insertion sort* o *insert sort*, sigue una estrategia sencilla que se asemeja a cómo las personas ordena-

		Coste	Nº de veces
0	`def insert_sort(a):`		
1	`    for j in range(1,len(a)):`	C_1	n
2	`        val = a[j]`	C_2	$n-1$
3	`        # Inserta a[j] en la lista`		
	`        # ordenada a[0:j-1]`	0	$n-1$
4	`        i = j-1`	C_4	$n-1$
5	`        while (i>=0) and (a[i]>val):`	C_5	$\sum_{j=1}^{n-1}(t_j+1)$
6	`            a[i+1] = a[i]`	C_6	$\sum_{j=1}^{n-1} t_j$
7	`            i = i-1`	C_7	$\sum_{j=1}^{n-1} t_j$
8	`        a[i+1] = val`	C_8	$n-1$

Tabla 2.2: Pseudocódigo del algoritmo de ordenamiento por inserción, donde n es el número de elementos de la lista de entrada **a**, y donde t_j es el número de veces que la condición del bucle de la línea 5 es verdadera (que se ejecuta el cuerpo del bucle), para un determinado valor de j.

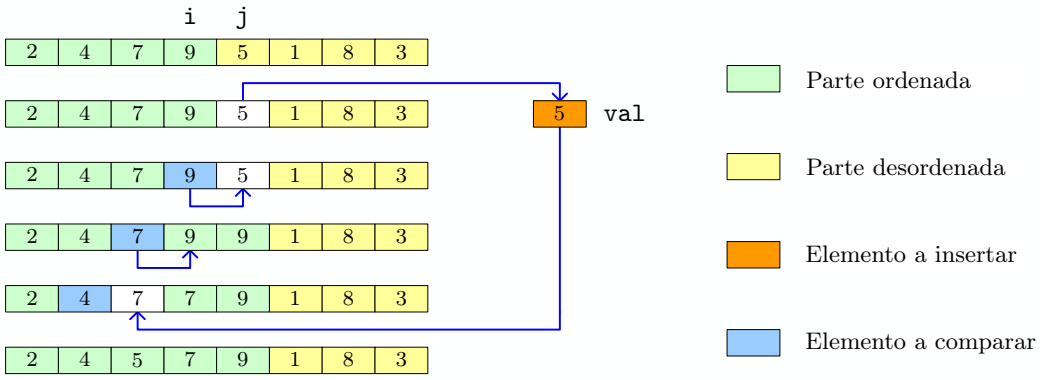


Figura 2.4: Ilustración de la inserción del elemento ubicado en la posición j dentro de la sublista ordenada desde el primer elemento hasta el $j-1$, para el algoritmo de ordenación por inserción de la Tabla 2.2. El proceso se corresponde con una iteración del bucle externo del algoritmo. Los desplazamientos hacia la derecha se realizan en el bucle interno.

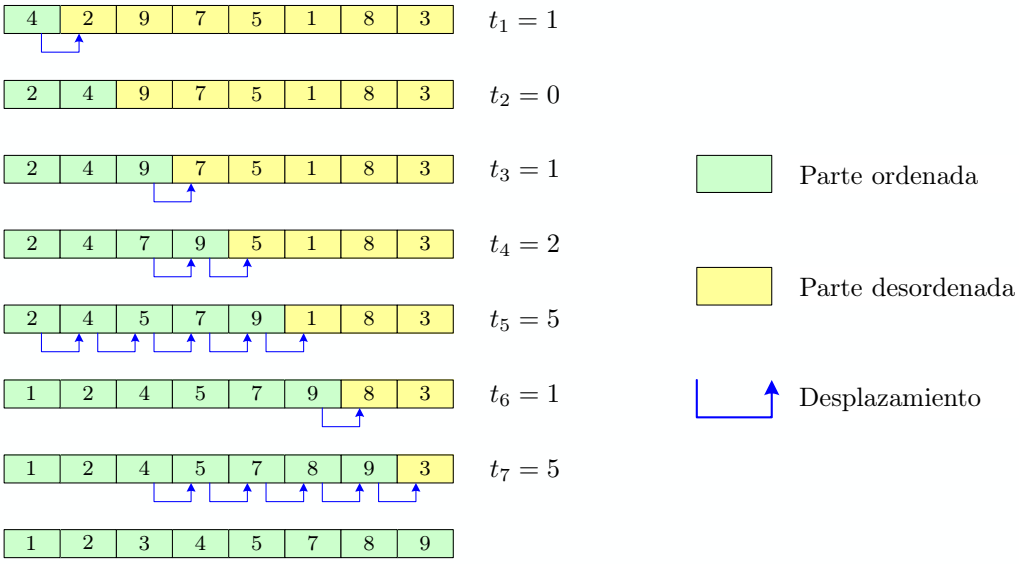


Figura 2.5: Explicación de las variables t_j del pseudocódigo en la Tabla 2.2 para una instancia concreta del problema de ordenación. En la iteración j-ésima del bucle externo el algoritmo debe realizar t_j desplazamientos de los elementos de la lista hacia la derecha, que corresponden con las evaluaciones verdaderas de la condición de la línea 5 del pseudocódigo (del bucle interno). Estos números t_j variarán en función de la lista inicial.

mos elementos en la vida real. Asumiendo que el objetivo es ordenar una lista de n elementos, los cuales podrían ser números reales, la idea consiste en dar $n - 1$ pasos, donde en cada uno se inserta un elemento no procesado con anterioridad, en una lista que contiene todos los elementos previamente procesados y ordenados. La Tabla 2.2 ilustra un posible pseudocódigo del algoritmo, mientras que la Figura 2.4 muestra la etapa j-esima del algoritmo para una instancia concreta del problema. En dicha etapa se inserta el elemento ubicado en la posición j, que previamente no se ha procesado, en la lista ordenada que queda a su izquierda (con índices menores que j), la cual ha sido ordenada previamente por el algoritmo. Naturalmente, al finalizar el proceso la sublista desde la primera posición hasta la j quedará ordenada, y el algoritmo iniciará una nueva iteración del bucle externo para insertar el siguiente elemento no procesado de la posición $j + 1$. El bucle interno desplaza los elementos de la sublista ordenada una posición hacia la derecha, hasta que se pueda insertar el elemento que estaba en la posición j, almacenado en la variable `val`, en la posición correcta.

Un enfoque para analizar el coste en tiempo del algoritmo consiste en determinar cuántas veces se ejecuta una línea del código. La última columna de la Tabla 2.2 muestra estas cantidades, mientras que la penúltima contiene constantes que se pueden interpretar como el número de operaciones o el tiempo que requiere ejecutar las líneas de código. Simplemente se indican constantes generales ya que estos valores dependen de muchos factores (procesador, datos en memoria caché, lenguaje de programación empleado, etc.) que no merece la pena considerar en un estudio teórico. Lo importante es que estos valores son constantes.

La Figura 2.5 muestra una explicación de las variables t_j, que inidican el número de desplazamientos de los elementos de la lista hacia la derecha para poder realizar una inserción, y que también son el número de veces que la condición de la línea 5 es verdadera. Estas cantidades son variables ya que dependen de la particular instancia del problema (es decir, de la lista de entrada inicial a ordenar). Teniendo en cuenta los productos de las constantes C_j por el número de veces que se ejecuta cada línea, el coste del algoritmo lo podemos expresar mediante:

$$\begin{aligned} T(n) = C_1 n + C_2(n-1) + C_4(n-1) + C_5 \sum_{j=1}^{n-1}(t_j + 1) \\ + C_6 \sum_{j=1}^{n-1} t_j + C_7 \sum_{j=1}^{n-1} t_j + C_8(n-1). \end{aligned} \tag{2.1}$$

No obstante, para que la fórmula en (2.1) sea útil necesitamos dar unos valores concretos a las variables t_j, para unas instancias del problema concretas. En el análisis de algoritmos se suelen estudiar tres casos: el *mejor*, el *peor*, y el caso *medio*.

El mejor caso consiste en la instancia para la que el algoritmo hace el menor número de operaciones, considerando un tamaño n general (el mejor caso no consiste en ordenar una lista de un elemento). Para este algoritmo el mejor caso ocurre cuando la lista de entrada ya está ordenada. Como no sería necesario hacer ninguna inserción, la condición de la línea 5 siempre sería falsa, y por tanto $t_j = 0$ para todo j. Sustituyendo $t_j = 0$ en (2.1) obtenemos:

$$\begin{aligned} T(n) &= C_1 n + C_2(n-1) + C_4(n-1) + C_5(n-1) + C_8(n-1) \\ &= (C_1 + C_2 + C_4 + C_5 + C_8)n - (C_2 + C_4 + C_5 + C_8) \\ &= K_1 n + K_2 \ \in \ \Theta(n). \end{aligned}$$

Por tanto, el coste en el mejor caso es lineal. Este ejemplo también deja patente la irrelevancia del valor de las constantes C_j de cara al orden asintótico de la función analizada.

Por otro lado, el peor caso ocurre cuando la lista está totalmente ordenada pero en sentido decreciente. Es el peor caso ya que cada vez que se analiza un nuevo elemento es necesario insertarlo en la primera posición de la lista. Eso implica que en la etapa j habría que hacer j desplazamientos de los elementos de la lista. Matemáticamente tendríamos $t_j = j$, y sustituyendo en (2.1) obtenemos:

$$\begin{aligned} T(n) = C_1 n + C_2(n-1) + C_4(n-1) + C_5\left(\frac{n(n+1)}{2} - 1\right) \\ + C_6\left(\frac{n(n-1)}{2}\right) + C_7\left(\frac{n(n-1)}{2}\right) + C_8(n-1) \end{aligned}$$

Código 2.2: Variante del algoritmo de ordenación "burbuja" en Python.

```
def burbuja(a):
    n = len(a)
    for i in range(0,n-1):                # i=0..n-2
        for j in range(n-1, i, -1):       # j=n-1..i+1
            if(a[j-1]>a[j]):
                aux = a[j-1]
                a[j-1] = a[j]
                a[j] = aux
```

$$
\begin{aligned}
&= \left(\frac{C_5}{2} + \frac{C_6}{2} + \frac{C_7}{2}\right) n^2 + \left(C_1 + C_2 + C_4 + \frac{C_5}{2} - \frac{C_6}{2} - \frac{C_7}{2} + C_8\right) n \\
&- (C_2 + C_4 + C_5 + C_8) \\
&= K_1 n^2 + K_2 n + K_3 \ \in \ \Theta(n^2)
\end{aligned}
$$

Por tanto, el coste del algoritmo en el caso peor es cuadrático. Finalmente, el caso medio para este algoritmo ocurre cuando los elementos se insertan en la mitad de las sublistas ordenadas. En este escenario tendríamos $t_j = j/2$, y el coste volvería a ser cuadrático.

2.2.2 Análisis detallado

En este apartado veremos un análisis más detallado de las operaciones que se realizan en los bucles. El resultado va ser idéntico, en términos de complejidad computacional, pero en este caso usaremos sumatorios para contabilizar el número de operaciones que realiza un algoritmo. En concreto, emplearemos la siguiente fórmula:

$$T_{\text{bucle}} = 1_{\text{inicialización}} + \sum^{n} \left(1_{\text{comparación}} + T_{\text{cuerpo}} + 1_{\text{incremento}}\right) + 1_{\text{última comparación}}$$

Consideraremos que se realizará una primera operación para inicializar la variable contadora del bucle. Después, asumiendo que el cuerpo del bucle se ejecuta n veces tendremos que sumar, por cada iteración: (a) una comparación cuyo resultado es verdadero, para entrar en el cuerpo del bucle, (b) el tiempo de ejecutar todo el cuerpo del bucle, y (c) una operación para incrementar o decrementar la variable contadora. Además, tendremos una última comparación cuyo resultado es falso, asociada a la salida del bucle.

Considérese el Código 2.2, que es una variante del algoritmo de ordenación "burbuja" o *bubble sort* (que tiene el mismo coste que el algoritmo original). El coste del algoritmo viene determinado por la longitud n de la lista **a**. Por tanto, vamos a construir una función $T(n)$ para indicar el número de operaciones que realiza el código. Considerando

la primera asignación de la línea 2, y el bucle de la línea 3 tendríamos:

$$T(n) = \underbrace{1 + 1 + \sum_{i=0}^{n-2} \big(1 + T_{\text{cuerpo bucle externo}} + 1\big) + 1}_{\text{bucle externo}}.$$

Se recomienda usar el nombre de la variable contadora en el código como variable contadora en el sumatorio (en este caso i). Además, para los límites superior e inferior usamos los propios valores que tomaría la variable en el código.

Por otro lado, como el cuerpo del bucle externo es otro bucle, podemos definir su coste a través de la fórmula para los sumatorios:

$$T_{\text{cuerpo bucle externo}} = 1 + \sum_{j=i+1}^{n-1} \big(1 + T_{\text{cuerpo bucle interno}} + 1\big) + 1.$$

En este caso, la variable j toma valores desde $n-1$ bajando hasta $i+1$. Para no considerar que el sumatorio vale 0, usamos $i+1$ como límite inferior y $n-1$ como superior.

Por último, el coste asociado al cuerpo del bucle interno (del `if`) sería una operación en el caso mejor (si la lista de entrada estuviera ordenada en sentido creciente), o cuatro en el caso peor (la lista inicial estaría ordenada en sentido decreciente). En ambos casos es una constante, y por eso el algoritmo va a requerir del orden del mismo número de operaciones tanto en el caso mejor como el peor. Analicemos el caso mejor, en el cual:

$$T_{\text{cuerpo bucle interno}} = 1.$$

Este valor se puede sustituir para hallar el coste del cuerpo del bucle externo (que es el coste del bucle interno):

$$T_{\text{cuerpo bucle externo}} = 1 + \sum_{j=i+1}^{n-1} \big(1+1+1\big) + 1 = 2 + \sum_{j=i+1}^{n-1} 3 = 2 + 3(n-1-i) = 3n - 1 - 3i.$$

Esta fórmula se puede sustituir en la expresión de $T(n)$:

$$\begin{aligned} T(n) &= 1 + 1 + \sum_{i=0}^{n-2} \big(1 + 3n - 1 - 3i + 1\big) + 1 = 3 + \sum_{i=0}^{n-2} \big(3n + 1 - 3i\big) \\ &= 3 + \sum_{i=0}^{n-2} 3n + \sum_{i=0}^{n-2} 1 - \sum_{i=0}^{n-2} 3i = 3 + 3n(n-1) + (n-1) - 3\frac{(n-2)(n-1)}{2} \\ &= 3 + 3n^2 - 3n + n - 1 - \frac{3}{2}(n^2 - 3n + 2) = \frac{3}{2}n^2 + \frac{5}{2}n - 1 \ \in \ \Theta(n^2). \end{aligned}$$

Código 2.3: Función en Python para sumar los n primeros números enteros positivos.

```
def suma_primeros_naturales(n):
    if n == 1:
        return 1  # Caso base
    else:
        return suma_primeros_naturales(n - 1) + n  # Caso recursivo
```

Por tanto, el algoritmo de ordenación burbuja corre (para cualquier lista de entrada) en tiempo cuadrático.

2.3 Análisis de algoritmos recursivos

El tiempo de ejecución o el número de operaciones realizadas por un algoritmo recursivo se especifica a través de una *relación de recurrencia* (o simplemente, *recurrencia*), que es una función matemática recursiva, digamos T, que describe su coste computacional. Considere el Código 2.3 para sumar los n primeros enteros positivos. En primer lugar, el número de operaciones que debe realizar depende claramente del parámetro de entrada n. Así, T será una función de n.

En el caso base (cuando $n = 1$) el método realiza las operaciones básicas que se muestran en la Figura 2.6. Antes de ejecutar las instrucciones, el programa necesita almacenar información de bajo nivel (por ejemplo, sobre los parámetros o la dirección de retorno). Digamos que esto requiere a_0 unidades de tiempo de computación, donde a_0 es una simple constante. La siguiente operación básica evalúa la condición, tomando a_1 unidades de tiempo. Dado que el resultado es `True`, la siguiente operación es un "salto" a la tercera línea del método, que requiere a_2 unidades de tiempo. Finalmente, el método puede devolver el valor 1 en el último paso, lo que requiere a_3 unidades de tiempo. En total, el método requiere $a = a_0 + a_1 + a_2 + a_3$ unidades de tiempo para $n = 1$. Así, podemos definir $T(1) = a$. El valor exacto de a es irrelevante de cara a la complejidad computacional asintótica del método. Lo importante es que a es una cantidad constante que no depende de n.

Alternativamente, la Figura 2.7 muestra las operaciones realizadas en el caso recursivo (cuando $n > 1$). Sea $b = \sum_{i=0}^{5} b_i$ el tiempo total de computación necesario para realizar las operaciones básicas (almacenar información de bajo nivel, evaluar la condición, saltar al caso recursivo, restar una unidad a n, sumar n al resultado de la llamada recursiva, y devolver el resultado), que también es una constante cuyo valor exacto es irrelevante de cara a la complejidad computacional asintótica. Además de b, debemos considerar el tiempo requerido por la llamada recursiva. Dado que resuelve un problema completo de tamaño $n-1$, podemos definirlo como $T(n-1)$. Así, la relación de recurrencia $T(n)$ completa puede especificarse de la siguiente manera:

$$T(n) = \begin{cases} a & \text{si } n = 1, \\ T(n-1) + b & \text{si } n > 1, \end{cases} \tag{2.2}$$

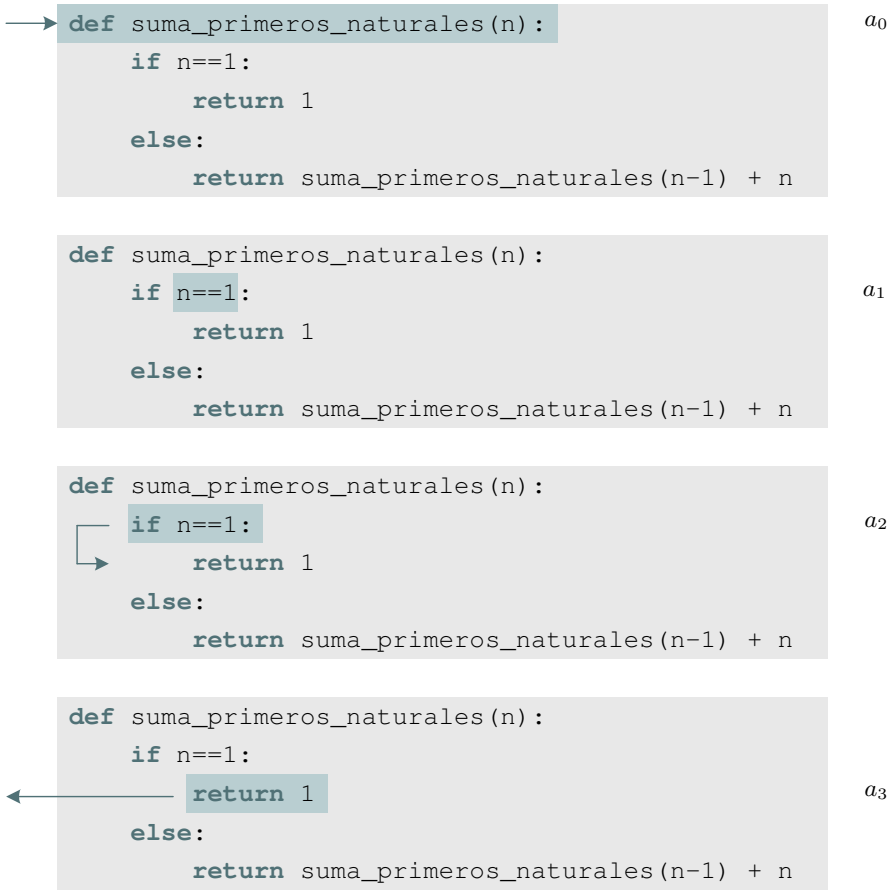


Figura 2.6: Secuencia de operaciones realizadas por la función del Código 2.3 en el caso base.

donde, por ejemplo, $T(3) = T(2) + b = T(1) + b + b = a + 2b$.

Aunque T describe correctamente el coste computacional del algoritmo, no es trivial averiguar su orden de crecimiento, ya que la definición es recursiva. Por lo tanto, el siguiente paso del análisis consiste en transformar la función en una fórmula no recursiva equivalente. A este proceso se le denomina *resolver* la relación de recurrencia. En este ejemplo, $T(n)$ es una función lineal de n:

$$T(n) = b(n-1) + a = bn - b + a \ \in \ \Theta(n).$$

En las siguientes secciones veremos cómo resolver esta recurrencia y otras comunes, empleando dos procedimientos diferentes.

Además, en estos apuntes simplificaremos las relaciones de recurrencia para facilitar el análisis. Considere el Código 2.4, que surge de la descomposición del problema de sumar los primeros n enteros positivos según ilustra la Figura 2.8. Su función de coste

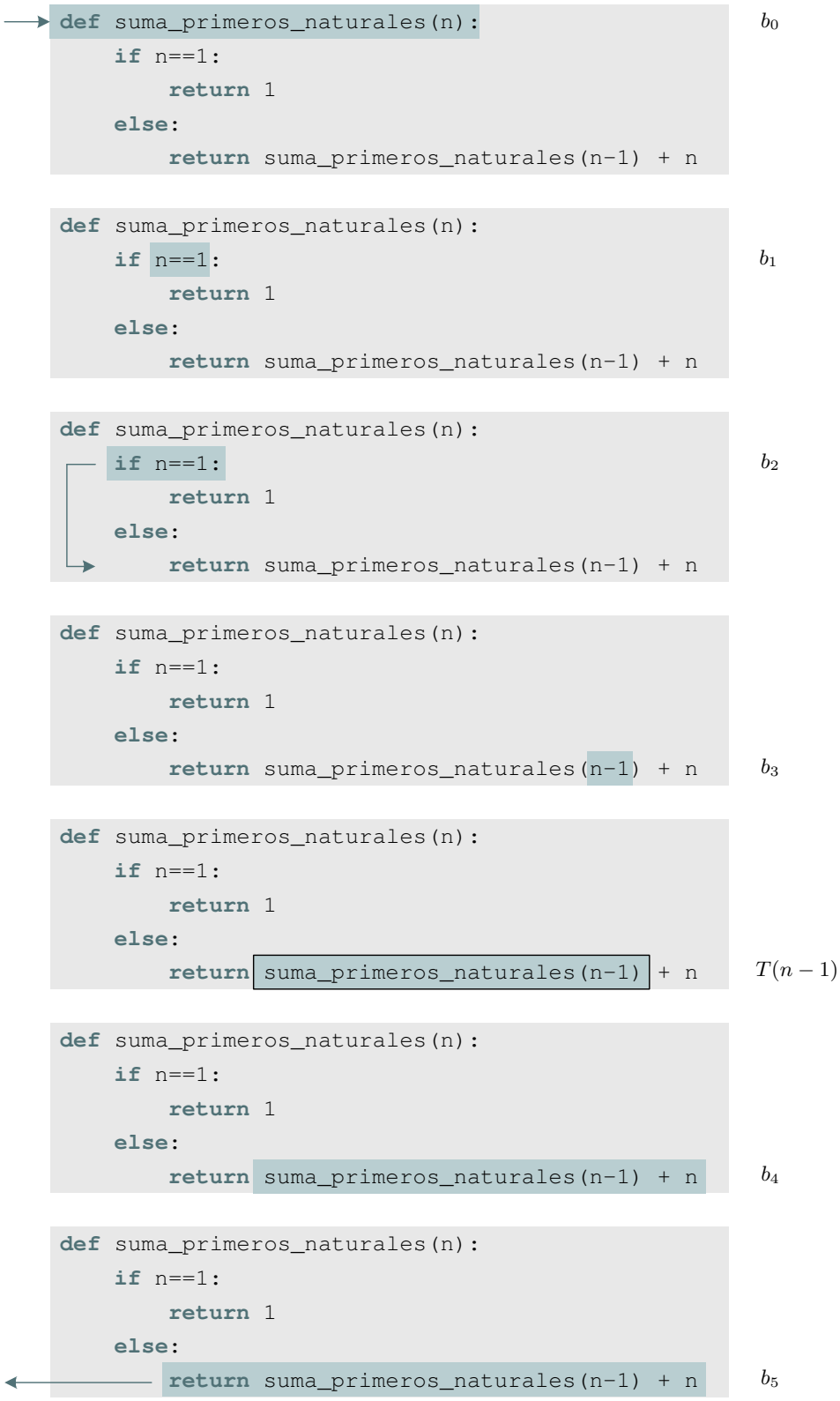


Figura 2.7: Secuencia de operaciones que realiza la función del Código 2.3 en el caso recursivo.

Código 2.4: Función en Python para sumar los n primeros enteros positivos, empleando subproblemas de (aproximadamente) la mitad del tamaño que el original.

```
def suma_primeros_naturales_alt_5(n):
    if n == 1:
        return 1
    elif n % 2 == 0:
        return 2 * suma_primeros_naturales_alt(n / 2) + (n / 2)**2
    else:
        return (2 * suma_primeros_naturales_alt((n - 1) / 2)
                + ((n + 1) / 2)**2)
```

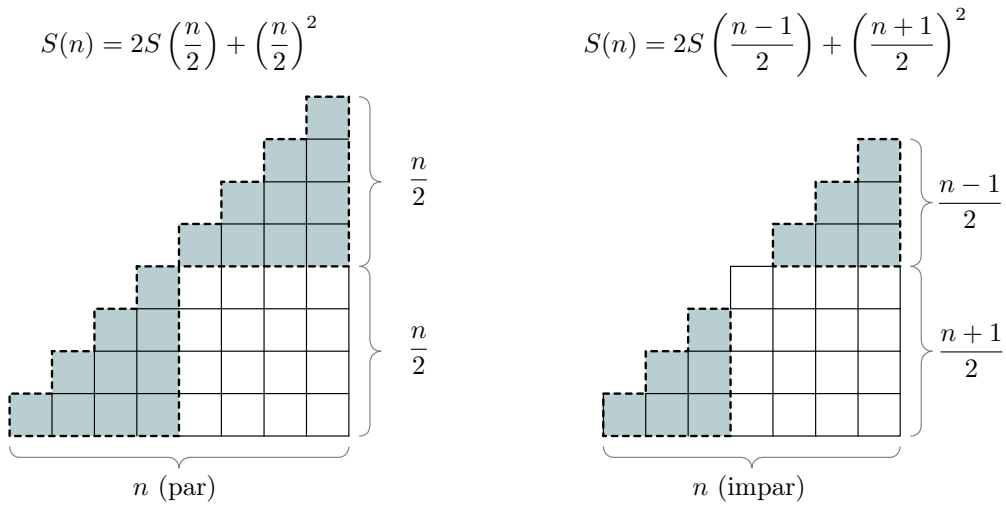


Figura 2.8: Diagrama ilustrando la descomposición en subproblemas empleada en el Código 2.4.

de tiempo de ejecución asociado se puede definir como:

$$T(n) = \begin{cases} a & \text{si } n = 1, \\ T\left(\frac{n}{2}\right) + b & \text{si } n > 1 \text{ y } n \text{ es par}, \\ T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + c & \text{si } n > 1 \text{ y } n \text{ es impar}. \end{cases} \tag{2.3}$$

Esta relación de recurrencia es difícil de analizar por dos razones. Por un lado, contiene más de un caso recursivo. Por otro, aunque es posible trabajar con la función suelo, está sujeta a tecnicismos y requiere matemáticas más complejas (por ejemplo, trabajar con desigualdades). Además, la complejidad adicional que se deriva de separar los casos para n par e impar, y de tratar con una recurrencia con más nivel de detalle, es innecesaria en lo que respecta al orden de crecimiento de la función. Dado que el algoritmo se basa en resolver subproblemas de (aproximadamente, en el caso impar) la mitad del tamaño, podemos trabajar con la siguiente relación de recurrencia en su lugar:

$$T(n) = \begin{cases} a & \text{si } n = 1, \\ T\left(\frac{n}{2}\right) + b & \text{si } n > 1. \end{cases} \tag{2.4}$$

En la práctica, trabajar con esta función es análogo a utilizar (2.3) y asumir que el tamaño del problema (n) será una potencia de dos, que puede ser arbitrariamente grande.

Por tanto, cubriremos las relaciones de recurrencia con un solo caso recursivo, evitando el uso de las funciones suelo o techo. Esto nos permitirá determinar definiciones exactas no recursivas de relaciones de recurrencia cuyo orden de crecimiento se puede caracterizar por la cota asintótica ajustado Θ.

2.3.1 Método de expansión de recurrencias

El método de *expansión de recurrencias*, también conocido como "método iterativo" o "método de sustitución hacia atrás", se puede utilizar principalmente para resolver relaciones de recurrencia cuyo caso recursivo contiene una referencia a la función recursiva (en algunos casos se puede aplicar a recurrencias en las que la función recursiva aparece varias veces en las definiciones recursivas). La idea consiste en simplificar la relación recursiva progresivamente paso a paso, hasta identificar un patrón general en la i-ésima etapa. Posteriormente, la función puede tomar valores concretos considerando que el caso base se alcanza en esa i-ésima etapa. Los siguientes ejemplos ilustran el procedimiento.

Relaciones de recurrencia básicas

Consideremos la función definida en (2.2). Su caso recursivo:

$$T(n) = T(n-1) + b \tag{2.5}$$

1. Escribir el caso recursivo de la relación de recurrencia
2. Expandir los términos recursivos (T) del lado derecho varias veces hasta detectar un patrón general para el i-ésimo paso
3. Determinar el valor de i que nos permite llegar a un caso base
4. Sustituir i en el patrón general y simplificar
5. Determinar el orden de crecimiento de la función T

Figura 2.9: Resumen de los pasos del método de expansión de recurrencias.

puede aplicarse repetidamente (a argumentos de menor tamaño) para *expandir* el término T del lado derecho. Por ejemplo, $T(n-1) = T(n-2) + b$, donde lo único que hemos hecho es sustituir n por $(n-1)$ en (2.5). Así, llegamos a

$$T(n) = \underbrace{[T(n-2)+b]}_{T(n-1)} + b = T(n-2) + 2b,$$

donde la expresión dentro de los corchetes es la expansión de $T(n-1)$. La idea se puede aplicar de nuevo expandiendo $T(n-2)$, que es $T(n-3) + b$. Así, en un tercer paso obtenemos

$$T(n) = \Big[\underbrace{T(n-3)+b}_{T(n-2)}\Big] + 2b = T(n-3) + 3b.$$

Después de varias de estas expansiones deberíamos ser capaces de detectar un patrón general correspondiente al i-ésimo paso. Para esta función es:

$$T(n) = T(n-i) + ib. \tag{2.6}$$

Finalmente, para algún valor de i el proceso alcanzará un caso base. Para la función (2.2) está definida para $T(1)$. Por tanto, el término $T(n-i)$ corresponderá al caso base cuando $n - i = 1$, o equivalentemente, cuando $i = n - 1$. Sustituyendo este resultado en (2.6) podemos eliminar la variable i de la fórmula, y proporcionar una definición completa no recursiva de $T(n)$:

$$T(n) = T(1) + (n-1)b = a + (n-1)b = bn - b + a \ \in \ \Theta(n).$$

Por tanto, el Código 2.3 corre en tiempo lineal con respecto a n.

La Figura 2.9 presenta un resumen del método de expansión que ahora aplicaremos a otras relaciones de recurrencia básicas. Considérese la función definida en (2.4). El

proceso de expansión es:

$$\begin{aligned} T(n) &= T(n/2) + b && \text{(paso 1)} \\ &= [T(n/4) + b] + b = T(n/4) + 2b && \text{(paso 2)} \\ &= [T(n/8) + b] + 2b = T(n/8) + 3b && \text{(paso 3)} \\ &= [T(n/16) + b] + 3b = T(n/16) + 4b && \text{(paso 4)} \\ &\vdots \end{aligned}$$

donde el patrón general para la relación de recurrencia en el paso i es:

$$T(n) = T(n/2^i) + ib. \tag{2.7}$$

El caso base $T(1)$ se alcanza cuando $n/2^i = 1$. Por tanto, se da cuando $n = 2^i$, o equivalentemente, cuando $i = \log_2 n$. Sustituyendo en (2.7) obtenemos:

$$T(n) = T(1) + b \log_2 n = a + b \log_2 n \; in \; \Theta(\log n).$$

Dado que el orden de crecimiento es logarítmico, el Código 2.4 es más rápido que el Código 2.3, cuyo orden es lineal. Esto tiene sentido intuitivo, ya que el primero descompone el problema original dividiendo su tamaño por dos, mientras que el segundo se basa en decrementar el tamaño del problema en una unidad. Así, el Código 2.4 necesita menos llamadas a funciones recursivas para llegar al caso base.

Un detalle sutil sobre las relaciones de recurrencia que se derivan de dividir el tamaño de un problema por una constante entera $k \geq 2$ es que no deben contener un solo caso base para $n = 0$. Desde un punto de vista matemático, nunca se alcanzaría, y no podríamos encontrar un valor para i en el tercer paso del método. El escollo es que el argumento de $T(n)$ debe ser un número entero. Así, la fracción en $T(n/k)$ corresponde en realidad a una división entera. Nótese que después de alcanzar $T(1)$ la siguiente expansión correspondería a $T(0)$ en lugar de $T(1/k)$. Por lo tanto, para estas relaciones de recurrencia debemos incluir un caso base adicional, normalmente para $n = 1$, con el fin de aplicar el método correctamente.

En los ejemplos anteriores fue bastante sencillo detectar el patrón recursivo general para la etapa i-ésima del proceso de expansión. Para las siguientes relaciones de recurrencia el paso es algo más complejo, ya que implicará calcular sumas como las presentadas en el apartado 1.3.

Considérese la siguiente relación de recurrencia:

$$T(n) = \begin{cases} a & \text{si } n = 0, \\ T(n-1) + bn + c & \text{si } n > 0. \end{cases} \tag{2.8}$$

El proceso de expansión es:

$$\begin{aligned}
T(n) &= T(n-1) + bn + c \\
&= [T(n-2) + b(n-1) + c] + bn + c \\
&= T(n-2) + 2bn - b + 2c \\
&= [T(n-3) + b(n-2) + c] + 2bn - b + 2c \\
&= T(n-3) + 3bn - b(1+2) + 3c \\
&= [T(n-4) + b(n-3) + c] + 3bn - b(1+2) + 3c \\
&= T(n-4) + 4bn - b(1+2+3) + 4c \\
&\vdots \\
&= T(n-i) + ibn - b(1+2+3+\cdots+(i-1)) + ic,
\end{aligned}$$

donde los corchetes indican expansiones particulares de términos que asociados a T. Además, el último paso contiene el patrón recursivo general, que se puede escribir como:

$$T(n) = T(n-i) + ibn - b\sum_{j=1}^{i-1} j + ic = T(n-i) + ibn - b\frac{(i-1)i}{2} + ic. \tag{2.9}$$

Dos conceptos erróneos frecuentes al utilizar sumas en el patrón en la etapa i-ésima son: (1) utilizar i como variable índice del sumatorio, y (2) elegir n como su límite superior. Es importante tener en cuenta que $i-1$ es el límite superior del sumatorio en este caso, lo que implica que la variable contadora no puede ser i.

Finalmente, el caso base $T(0)$ se alcanza cuando $i = n$. Por lo tanto, sustituyendo en (2.9) tenemos:

$$T(n) = bn^2 - \frac{b}{2}n(n-1) + cn + a = \frac{b}{2}n^2 + \left(c + \frac{b}{2}\right)n + a \ \in \ \Theta(n^2),$$

que es un polinomio cuadrático.

La siguiente relación de recurrencia aparece en algoritmos de divide y vencerás como el *mergesort*:

$$T(n) = \begin{cases} a & \text{si } n = 1, \\ 2T(n/2) + bn + c & \text{si } n > 1. \end{cases} \tag{2.10}$$

El proceso de expansión es:

$$\begin{aligned}
T(n) &= 2T(n/2) + bn + c \\
&= 2[2T(n/4) + bn/2 + c] + bn + c \\
&= 4T(n/4) + 2bn + 2c + c \\
&= 4[2T(n/8) + bn/4 + c] + 2bn + 2c + c \\
&= 8T(n/8) + 3bn + 4c + 2c + c \\
&= 8[2T(n/16) + bn/8 + c] + 3bn + 4c + 2c + c \\
&= 16T(n/16) + 4bn + 8c + 4c + 2c + c \\
&\vdots \\
&= 2^i T(n/2^i) + ibn + c(1 + 2 + 4 + \cdots + 2^{i-1}).
\end{aligned}$$

De nuevo, los corchetes indican expansiones de términos asociados a T. En este caso son especialmente útiles, ya que cada uno de los términos expandidos debe multiplicarse por dos. Se recomienda a los estudiantes que utilicen corchetes, ya que omitirlos es una fuente común de errores.

En este caso, el patrón recursivo contiene una suma parcial de una serie geométrica.

$$T(n) = 2^i T(n/2^i) + ibn + c\sum_{j=0}^{i-1} 2^j = 2^i T(n/2^i) + ibn + c(2^i - 1). \tag{2.11}$$

Por último, se alcanza el caso base $T(1)$ cuando $n/2^i = 1$, lo cual implica que $i = \log_2 n$. Por tanto, sustituyendo en (2.11) da lugar a:

$$T(n) = nT(1) + bn\log_2 n + c(n-1) = bn\log_2 n + n(a+c) - c \ \in \ \Theta(n\log n),$$

cuyo término de mayor orden es $bn\log_2 n$.

Teorema Maestro

El *Teorema Maestro* es un resultado que puede utilizarse como receta rápida para determinar la complejidad computacional en tiempo de algoritmos basados en la estrategia de diseño *divide y vencerás*. En particular, se puede aplicar a las siguientes relaciones de recurrencia:

$$T(n) = \begin{cases} c & \text{si } n = 1, \\ aT(n/b) + f(n) & \text{si } n > 1, \end{cases}$$

donde $a \geq 1$, $b > 1$, $c \geq 0$, y f es una función asintóticamente positiva. Así, estos algoritmos resuelven a subproblemas cuyo tamaño es igual al del original dividido por b. Además, el procesamiento posterior o la combinación de las subsoluciones requiere $f(n)$ operaciones. En función de la contribución relativa de los términos $aT(n/b)$ y $f(n)$ al coste de ejecución del algoritmo, el teorema maestro establece, en su definición más general, que es posible determinar una cota ajustada asintótica para T en los tres casos siguientes:

1. Si $f(n) = \mathcal{O}(n^{\log_b a - \epsilon})$ para alguna constante $\epsilon > 0$, entonces:

$$T(n) \in \Theta(n^{\log_b a}).$$

2. Si $f(n) = \Theta\left(n^{\log_b a}(\log n)^k\right)$, con $k \geq 0$, entonces:

$$T(n) \in \Theta\left(n^{\log_b a}(\log n)^{k+1}\right).$$

3. Si $f(n) = \Omega(n^{\log_b a + \epsilon})$ con $\epsilon > 0$, y $f(n)$ satisface la condición de regularidad ($af(n/b) \leq df(n)$ para alguna constante $d < 1$, y para todo n lo suficientemente grande), entonces:

$$T(n) \in \Theta(f(n)).$$

Por ejemplo, para

$$T(n) = \begin{cases} 1 & \text{si } n = 1, \\ T(n/2) + 2^n & \text{si } n > 1, \end{cases}$$

es posible encontrar un $\epsilon > 0$ tal que el orden de $f(n) = 2^n$ sea mayor que el de $n^{log_2 1+\epsilon} = n^\epsilon$. De hecho, en este ejemplo cualquier valor $\epsilon > 0$ sería válido ya que el orden exponencial de 2^n es siempre mayor que el orden de un polinomio. Así pues, esta relación de recurrencia cae en el tercer caso del Teorema Maestro, lo que implica que $T(n) \in \Theta(2^n)$.

Cuando $f(n)$ es un polinomio de grado k podemos aplicar la siguiente versión más sencilla del Teorema Maestro:

$$T(n) \in \begin{cases} \Theta(n^k) & \text{si } \frac{a}{b^k} < 1 \\ \Theta(n^k \log n) & \text{si } \frac{a}{b^k} = 1 \\ \Theta(n^{\log_b a}) & \text{si} \frac{a}{b^k} > 1 \end{cases} \tag{2.12}$$

Este resultado puede obtenerse mediante el método de expansión de recurrencias.

Consideremos la siguiente relación de recurrencia:

$$T(n) = \begin{cases} c & \text{if } n = 1, \\ aT(n/b) + dn^k & \text{if } n > 1, \end{cases} \tag{2.13}$$

donde $a \geq 1$, $b > 1$, $c \geq 0$, y $d \geq 0$. El proceso de expansión es:

$$\begin{aligned} T(n) &= aT(n/b) + dn^k \\ &= a\left[aT(n/b^2) + d(n/b)^k\right] + dn^k \\ &= a^2T(n/b^2) + dn^k(1 + a/b^k) \\ &= a^2\left[aT(n/b^3) + d(n/b^2)^k\right] + dn^k(1 + a/b^k) \\ &= a^3T(n/b^3) + dn^k(1 + a/b^k + a^2/b^{2k}) \\ &\vdots \\ &= a^iT\Big(\frac{n}{b^i}\Big) + dn^k\sum_{j=0}^{i-1}\Big(\frac{a}{b^k}\Big)^j. \end{aligned}$$

El caso base es $T(1) = c$, que se alcanza cuando $i = \log_b n$. Sustituyendo los resultados:

$$T(n) = ca^{\log_b n} + dn^k\sum_{j=0}^{\log_b n-1}\Big(\frac{a}{b^k}\Big)^j = cn^{\log_b a} + dn^k\sum_{j=0}^{\log_b n-1}\Big(\frac{a}{b^k}\Big)^j,$$

donde hay tres casos dependiendo de los valores de a, b y k:

1. Si $a < b^k$ entonces $\sum_{j=0}^{\log_b n-1}\big(a/b^k\big)^j$ será un término constante (no puede ser infinito). Obsérvese que la suma infinita $\sum_{i=0}^{\infty} r^i = 1/(1-r)$ es una constante (es decir, no diverge) para $r < 1$. Por lo tanto, en este escenario:

   $$T(n) = cn^{\log_b a} + dKn^k,$$

   para alguna constante K. Además, como $a < b^k$ implica que $\log_b a < k$, el término de orden superior es n^k. Por tanto,

   $$T(n) \in \Theta(n^k).$$

2. Si $a = b^k$ entonces $T(n) = cn^k + dn^k\sum_{j=0}^{\log_b n-1} 1 = cn^k + dn^k\log_b n$, lo cual implica que:

   $$T(n) \in \Theta(n^k\log n).$$

3. Si $a > b^k$, entonces, resolviendo la suma parcial geométrica:

$$\begin{aligned} T(n) &= cn^{\log_b a} + dn^k \frac{\left(\frac{a}{b^k}\right)^{\log_b n} - 1}{\left(\frac{a}{b^k}\right) - 1} \\ &= cn^{\log_b a} + dn^k \frac{\frac{n^{\log_b a}}{n^k} - 1}{K} \\ &= \left(c + \frac{d}{K}\right) n^{\log_b a} - \frac{d}{K} n^k, \end{aligned}$$

donde $K = a/b^k - 1$ es una constante. Finalmente, como $a > b^k$ implica que $\log_b a > k$, tenemos:

$$T(n) \in \Theta(n^{\log_b a}).$$

Ejemplos adicionales

Considérese la siguiente relación de recurrencia (que describiría, por ejemplo, el tiempo de ejecución de un algoritmo de divide y vencerás para hallar el máximo de una lista):

$$T(n) = \begin{cases} 1 & \text{if } n = 1, \\ 2T(n/2) + 1 & \text{if } n > 1, \end{cases} \tag{2.14}$$

donde hemos supuesto que las constantes multiplicativas son iguales a 1, y que el tamaño de los subproblemas es exactamente la mitad que el del problema original. La recurrencia multiplica $T(n/2)$ por dos ya que el algoritmo necesita invocarse a sí mismo dos veces en los casos recursivos, con argumentos diferentes en cada llamada. Además, añade la constante 1 ya que los resultados de los subproblemas necesitan ser sumados y devueltos por el método. Por último, para simplificar, no necesitamos especificar un caso base para $n = 2$. Estas suposiciones no afectarán al orden de crecimiento del tiempo de ejecución.

Aplicando el método de expansión tenemos:

$$\begin{aligned} T(n) &= 2T(n/2) + 1 \\ &= 2[2T(n/4) + 1] + 1 = 4T(n/4) + 2 + 1 \\ &= 4[2T(n/8) + 1] + 2 + 1 = 8T(n/8) + 4 + 2 + 1 \\ &= 8[2T(n/16) + 1] + 4 + 2 + 1 = 16T(n/16) + 8 + 4 + 2 + 1 \\ &\vdots \\ &= 2^i T(n/2^i) + \sum_{j=0}^{i-1} 2^j = 2^i T(n/2^i) + 2^i - 1. \end{aligned}$$

El caso base $T(1) = 1$ ocurre cuando $n = 2^i$. Sustituyendo tenemos:

$$T(n) = n + n - 1 = 2n - 1 \ \in \ \Theta(n),$$

donde $T(n)$ es una función lineal de n. Naturalmente, este resultado está de acuerdo con el Teorema Maestro (véase (2.12)), ya que la relación de recurrencia es un caso especial de (2.13), donde $a = 2$, $b = 2$, y $k = 0$. Por tanto, $T(n) \in \Theta(2^{log_2 n}) = \Theta(n)$.

Considere la siguiente relación de recurrencia (que podría describir, por ejemplo, el número de operaciones que realizaría un algoritmo, no basado en divide y vencerás, para hallar la suma de los elementos de una lista):

$$T(n) = \begin{cases} 1 & \text{si } n = 1, \\ T(n-1) + 1 & \text{si } n > 1, \end{cases}$$

que es un caso especial de (2.5), donde $T(n) \in \Theta(n)$. En cambio, el tercer método descompone el problema original en dos de la mitad de tamaño, donde las subsoluciones no necesitan ser procesadas más. Por lo tanto, la relación de recurrencia correspondiente puede ser idéntica a la de (2.14).

Por último, considérese la siguiente relación de recurrencia (que describiría, por ejemplo, el coste de una función recursiva básica para hallar el número de dígitos de un número entero no negativo):

$$T(n) = \begin{cases} 1 & \text{si } n < 10, \\ T(n/10) + 1 & \text{si } n \geq 10. \end{cases} \tag{2.15}$$

Es diferente de las recurrencias anteriores, ya que el caso base se define en un intervalo. No obstante, al suponer que la entrada será una potencia de 10, podemos utilizar una definición alternativa de la recurrencia:

$$T(n) = \begin{cases} 1 & \text{si } n = 1, \\ T(n/10) + 1 & \text{si } n > 1. \end{cases} \tag{2.16}$$

Ambas funciones son equivalentes para $n = 10^p$ (y $p \geq 0$).

La segunda recurrencia es un caso especial de (2.13), donde $a = 1$, $b = 10$, y $k = 0$. Así, el Teorema Maestro indica que su complejidad es logarítmica, ya que $T(n) \in \Theta(n^k \log n) = \Theta(\log n)$. Podemos obtener este resultado de la siguiente ma-

nera aplicando el método de expansión:

$$\begin{aligned} T(n) &= T(n/10) + 1 \\ &= [T(n/100) + 1] + 1 = T(n/10^2) + 2 \\ &= [T(n/1000) + 1] + 2 = T(n/10^3) + 3 \\ &= [T(n/10000) + 1] + 3 = T(n/10^4) + 4 \\ &\vdots \\ &= T(n/10^i) + i. \end{aligned}$$

El caso base se obtiene cuando $n/10^i = 1$, es decir, cuando $i = \log_{10} n$. Sustituyendo obtenemos:

$$T(n) = T(1) + \log_{10} n = 1 + \log_{10} n \in \Theta(\log n).$$

2.3.2 Método general para resolver ecuaciones en diferencias

El método de expansión es eficaz para resolver relaciones de recurrencia en las que la función recursiva sólo aparece una vez en las definiciones recursivas. Esta sección describe un potente enfoque, que denominaremos "método general para ecuaciones en diferencias", que nos permite resolver recurrencias que pueden "llamarse" a sí mismas varias veces. En particular, el método se puede utilizar para resolver recurrencias de la siguiente forma:

$$T(n) = \underbrace{-a_1 T(n-1) - \cdots - a_k T(n-k)}_{\text{términos } T \text{ "en diferencias"}} + \underbrace{P_1^{d_1}(n) b_1^n + \cdots + P_s^{d_s}(n) b_s^n}_{\text{términos "polinomio} \times \text{exponencial"}}, \qquad (2.17)$$

donde a_i y b_i son constantes, y $P_i^{d_i}(n)$ son polinomios (de n) de grado d_i. Los términos asociados a T pueden aparecer varias veces en el lado derecho de la definición. Además, sus argumentos son necesariamente n menos alguna constante entera. Así, estas relaciones de recurrencia también se conocen como "ecuaciones en diferencias". En estos apuntes llamaremos a estos términos en los que interviene T "en diferencias", para enfatizar que los argumentos no pueden tomar la forma n/b, donde b es una constante. Cabe mencionar que el método de todas formas es general, ya que se puede aplicar para resolver recurrencias con términos del tipo $T(n/b)$, y otras más complejas, realizando transformaciones, como cambios de variable. Además, el lado derecho de la definición puede contener varios términos que consistan en un polinomio multiplicado por una potencia de n (es decir, una exponencial).

En lugar de proporcionar directamente un procedimiento general, los siguientes apartados explicarán el método de forma progresiva, comenzando con recurrencias sencillas e introduciendo nuevos elementos a medida que aumente la complejidad de las recurrencias.

Recurrencias homogéneas y raíces diferentes

Las relaciones de recurrencia homogéneas sólo contienen términos T en diferencias:

$$T(n) = -a_1 T(n-1) - \ \cdots \ - a_k T(n-k).$$

El primer paso para resolverlas consiste en pasar cada término al lado izquierdo de la ecuación:

$$T(n) + a_1 T(n-1) + \ \cdots \ + a_k T(n-k) = 0.$$

Posteriormente, definimos su *polinomio característico* asociado aplicando el cambio $x^{k-z} = T(n-z)$ para $z = 0, \ldots, k$:

$$x^k + a_1 x^{k-1} + \ \cdots \ + a_{k-1} x + a_k.$$

Lo único que hemos hecho es sustituir $T(n)$ por x^k, $T(n-1)$ por x^{k-1}, y así sucesivamente. El coeficiente (a_k) asociado al término T que tenga el argumento más pequeño será la constante del polinomio. El siguiente paso consiste en encontrar las k raíces del polinomio característico. Si r_i es su raíz i-ésima entonces se puede expresar en forma factorizada como:

$$(x - r_1)(x - r_2) \ \cdots \ (x - r_k).$$

Si todas las raíces son diferentes, entonces la expresión no recursiva para T será:

$$T(n) = C_1 r_1^n + \ \cdots \ + C_k r_k^n. \tag{2.18}$$

Finalmente, los valores de las constantes dependerán de los casos base de T, y se hallarán resolviendo un sistema de k ecuaciones lineales con k variables desconocidas (las constantes C_i). Necesitaremos encontrar k valores iniciales (casos base, para valores pequeños de n) de T para construir las k ecuaciones.

El siguiente ejemplo ilustra el enfoque aplicado a la función Fibonacci básica:

$$F(n) = \begin{cases} 1 & \text{si } n = 1, \\ 1 & \text{si } n = 2, \\ F(n-1) + F(n-2) & \text{si } n > 2, \end{cases} \tag{2.19}$$

que se puede reescribir como:

$$T(n) = \begin{cases} 0 & \text{si } n = 0, \\ 1 & \text{si } n = 1, \\ T(n-1) + T(n-2) & \text{si } n > 1, \end{cases} \tag{2.20}$$

donde hemos incluido un caso base extra para $n = 0$ que será útil en breve. El primer

paso consiste en escribir cada término T en el lado izquierdo de la identidad recursiva:

$$T(n) - T(n-1) - T(n-2) = 0.$$

Es una recurrencia homogénea ya que hay un 0 en el lado derecho. Formamos entonces el polinomio característico:

$$x^2 - x - 1.$$

Las raíces de esta función cuadrática son:

$$r_1 = \frac{1+\sqrt{5}}{2}, \qquad \text{and} \qquad r_2 = \frac{1-\sqrt{5}}{2},$$

que son diferentes. Por tanto, podemos expresar la recurrencia de manera no recursiva como:

$$T(n) = C_1 r_1^n + C_2 r_2^n = C_1 \left(\frac{1+\sqrt{5}}{2}\right)^n + C_2 \left(\frac{1-\sqrt{5}}{2}\right)^n. \tag{2.21}$$

El último paso consiste en encontrar valores para las constantes C_i resolviendo un sistema de ecuaciones lineales. Cada ecuación se forma eligiendo un valor pequeño para n correspondiente a un caso base, y aplicándolo a (2.21). La ecuación más sencilla se obtiene para $n = 0$, que es la razón por la que consideramos el caso base $n = 0$ en (2.20). Para la segunda ecuación podemos utilizar $n = 1$. Esto proporciona el siguiente sistema de ecuaciones lineales:

$$\left.\begin{array}{rcl} C_1 + C_2 = 0 = T(0) \\ \left(\frac{1+\sqrt{5}}{2}\right) C_1 + \left(\frac{1-\sqrt{5}}{2}\right) C_2 = 1 = T(1) \end{array}\right\}.$$

Las soluciones son:

$$C_1 = \frac{1}{\sqrt{5}}, \qquad \text{and} \qquad C_2 = -\frac{1}{\sqrt{5}}.$$

Por tanto, la función de Fibonacci se puede expresar de la siguiente manera:

$$T(n) = F(n) = \frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2}\right)^n - \frac{1}{\sqrt{5}} \left(\frac{1-\sqrt{5}}{2}\right)^n \in \Theta\left(\frac{1+\sqrt{5}}{2}\right)^n, \tag{2.22}$$

que es una función exponencial. Obsérvese que r_1^n domina claramente el crecimiento de la función. Por un lado, $r_1 > r_2$. Por otro, $|r_2| < 1$, lo que significa que r_2^n se aproximará a 0 a medida que n se acerque a infinito (es una función decreciente). Por último, a pesar de su apariencia compleja, el resultado es obviamente un número entero.

Para el siguiente ejemplo consideremos las siguientes funciones mutuamente recursivas:

$$A(n) = \begin{cases} 0 & \text{si } n = 1, \\ A(n-1) + B(n-1) & \text{si } n > 1, \end{cases} \tag{2.23}$$

$$B(n) = \begin{cases} 1 & \text{si } n = 1, \\ A(n-1) & \text{si } n > 1. \end{cases} \tag{2.24}$$

Para poder aplicar el método debemos redefinirlas exclusivamente en términos de sí mismas. Por un lado, $B(n) = A(n-1)$ implica que $B(n-1) = A(n-2)$. Además, $A(2) = 1$. Así, podemos redefinir $A(n)$ como:

$$A(n) = \begin{cases} 0 & \text{if } n = 1, \\ 1 & \text{if } n = 2, \\ A(n-1) + A(n-2) & \text{if } n \geq 3. \end{cases} \tag{2.25}$$

Por otra parte, como $A(n-1) = B(n)$, y $A(n) = B(n+1)$, podemos sustituir en (2.23) para obtener $B(n+1) = B(n) + B(n-1)$. Además, como $B(2) = 0$, podemos definir $B(n)$ como:

$$B(n) = \begin{cases} 1 & \text{if } n = 1, \\ 0 & \text{if } n = 2, \\ B(n-1) + B(n-2) & \text{if } n \geq 3. \end{cases} \tag{2.26}$$

Ambas funciones tienen la forma en (2.21). Por lo tanto, la única diferencia entre ellas es el valor de las constantes. En particular:

$$A(n) = \left(\frac{1}{2} - \frac{1}{2\sqrt{5}}\right)\left(\frac{1+\sqrt{5}}{2}\right)^n + \left(\frac{1}{2} + \frac{1}{2\sqrt{5}}\right)\left(\frac{1-\sqrt{5}}{2}\right)^n, \tag{2.27}$$

y

$$B(n) = \left(-\frac{1}{2} + \frac{3}{2\sqrt{5}}\right)\left(\frac{1+\sqrt{5}}{2}\right)^n + \left(-\frac{1}{2} - \frac{3}{2\sqrt{5}}\right)\left(\frac{1-\sqrt{5}}{2}\right)^n, \tag{2.28}$$

donde se puede suponer que $A(0) = 1$, y $B(0) = -1$. Por último, no es difícil ver que $A(n) + B(n)$ son números de Fibonacci $F(n)$, ya que sumando (2.27) y (2.28) se obtiene (2.22).

Raíces con multiplicidad mayor que uno

El apartado anterior mostró que si la multiplicidad de una raíz r_i es 1, correspondiente a un término $(x - r_i)^1$ en la factorización del polinomio característico, entonces la versión no recursiva de $T(n)$ contendrá el término $C_i r_i^n$. En cambio, cuando la multiplicidad m de una raíz r es mayor que 1, resultante de $(x-r)^m$, entonces $T(n)$ incorporará m términos de la forma: polinomio $\times$ constante $\times$ r^n. En particular, los polinomios serán diferentes potencias de n, que van de 1 a n^{m-1}. Por ejemplo, un término $(x-2)^4$ en la factorización del polinomio característico daría lugar a los cuatro términos siguientes en la versión no recursiva de $T(n)$:

$$C_1 2^n + C_2 n 2^n + C_3 n^2 2^n + C_4 n^3 2^n,$$

para algunas constantes C_i. Por tanto, $T(n)$ se puede expresar en general como:

$$T(n) = C_1 P_1(n) r_1^n + \cdots + C_k P_k(n) r_k^n, \tag{2.29}$$

donde $P_i(n)$ es un polinomio de tipo n^c, para alguna c.

Por ejemplo, considérese la siguiente recurrencia:

$$T(n) = 5T(n-1) - 9T(n-2) + 7T(n-3) - 2T(n-4),$$

con $T(0) = 0$, $T(1) = 2$, $T(2) = 11$ y $T(3) = 28$. Se puede expresar como $T(n) - 5T(n-1) + 9T(n-2) - 7T(n-3) + 2T(n-4) = 0$, cuyo polinomio característico es:

$$x^4 - 5x^3 + 9x^2 - 7x + 2.$$

Se puede factorizar (por ejemplo, aplicando la regla de Ruffini) como:

$$(x-1)^3(x-2),$$

lo cual implica que la recurrencia se podrá reescribir como:

$$\begin{aligned} T(n) &= C_1 \cdot 1 \cdot 1^n + C_2 \cdot n \cdot 1^n + C_3 \cdot n^2 \cdot 1^n + C_4 \cdot 1 \cdot 2^n \\ &= C_1 + C_2 n + C_3 n^2 + C_4 2^n, \end{aligned}$$

donde hay tres términos asociados a la raíz $r = 1$. Finalmente, las constantes se pueden obtener resolviendo el siguiente sistema de ecuaciones lineales:

$$\left.\begin{array}{rcl} C_1 + C_4 = 0 & = & T(0) \\ C_1 + C_2 + C_3 + 2C_4 = 2 & = & T(1) \\ C_1 + 2C_2 + 4C_3 + 4C_4 = 11 & = & T(2) \\ C_1 + 3C_2 + 9C_3 + 8C_4 = 28 & = & T(3) \end{array}\right\}.$$

Las soluciones son $C_1 = -1$, $C_2 = -2$, $C_3 = 3$, y $C_4 = 1$ (el Código 2.5 muestra cómo resolver el sistema de ecuaciones lineales, expresado como $\mathbf{Ax} = \mathbf{b}$, con el paquete NumPy). Por último,

$$T(n) = -1 - 2n + 3n^2 + 2^n \in \Theta(2^n),$$

cuyo orden de crecimiento es exponencial.

Código 2.5: Resolución de un sistema de ecuaciones lineales, $\mathbf{Ax} = \mathbf{b}$, en Python.

```
import numpy as np

A = np.array([[1, 0, 0, 1], [1, 1, 1, 2],
              [1, 2, 4, 4], [1, 3, 9, 8]])
b = np.array([0, 2, 11, 28])
x = np.linalg.solve(A, b)
```

Recurrencias no homogéneas

Una recurrencia no homogénea contiene términos no recursivos en su lado derecho. Podremos resolver estas recurrencias cuando dichos términos consistan en un polinomio por una exponencial, como se muestra en (2.17). El procedimiento es exactamente el mismo, pero para cada término $P_i^{d_i}(n)b_i^n$ tendremos que incluir $(x - b_i)^{d_i+1}$ en el polinomio característico, donde d_i es el grado del polinomio $P_i(n)$.

Considérese la siguiente relación de recurrencia:

$$T(n) = 2T(n-1) - T(n-2) + 3^n + n3^n + 3 + n + n^2.$$

Como en ejemplos anteriores, el primer paso consiste en pasar los términos T en diferencias al lado izquierdo de la recurrencia:

$$T(n) - 2T(n-1) + T(n-2) = 3^n + n3^n + 3 + n + n^2.$$

Para el siguiente paso es útil expresar los términos del lado derecho como el producto de un polinomio por una exponencial. Naturalmente, si un término sólo contiene una exponencial, entonces se puede multiplicar por un 1 (que es un polinomio). Del mismo modo, podemos multiplicar polinomios, que no aparezcan inicialmente multiplicados por una potencia, por 1^n. Por lo tanto, la recurrencia se puede escribir como:

$$T(n) - 2T(n-1) + T(n-2) = 1 \cdot 3^n + n \cdot 3^n + 3 \cdot 1^n + n \cdot 1^n + n^2 \cdot 1^n.$$

Además, aunque una determinada exponencial aparezca varias veces, debemos considerar que multiplica a un único polinomio. Así, la recurrencia debe considerarse como:

$$T(n) - 2T(n-1) + T(n-2) = (1+n) \cdot 3^n + (3 + n + n^2) \cdot 1^n.$$

El siguiente paso consiste en determinar el polinomio característico. Del lado izquierdo tenemos $(x^2 - 2x + 1) = (x-1)^2$. Del término $(1+n) \cdot 3^n$ tenemos que incluir $(x-3)^2$, donde el 3 es la base de la exponencial, y el 2 es el grado del polinomio (1) más uno. Del mismo modo, $(3 + n + n^2) \cdot 1^n$ proporciona el nuevo término $(x-1)^3$. Por lo tanto, el polinomio característico es;

$$(x-1)^2(x-3)^2(x-1)^3 = (x-1)^5(x-3)^2,$$

y $T(n)$ tendría la siguiente forma:

$$T(n) = C_1 + C_2 n + C_3 n^2 + C_4 n^3 + C_5 n^4 + C_6 3^n + C_7 n 3^n.$$

El siguiente ejemplo ilustra el enfoque con la siguiente relación de recurrencia no homogénea:

$$T(n) = \begin{cases} 1 & \text{si } n = 0, \\ 2T(n-1) + n + 2^n & \text{si } n > 0. \end{cases}$$

En este caso, podemos escribir el caso recursivo como $T(n) - 2T(n-1) = n \cdot 1^n + 1 \cdot 2^n$. El polinomio característico factorizado correspondiente es:

$$\underbrace{(x-2)}_{\text{de } T(n)-2T(n-1)} \cdot \underbrace{(x-1)^2}_{\text{de } n \cdot 1^n} \cdot \underbrace{(x-2)}_{\text{de } 1 \cdot 2^n} = (x-1)^2(x-2)^2,$$

por lo que la recurrencia tendrá la siguiente forma:

$$T(n) = C_1 + C_2 n + C_3 2^n + C_4 n 2^n.$$

Puesto que hay cuatro constantes desconocidas, necesitamos los valores de T evaluados en cuatro entradas diferentes. A partir del caso base $T(0) = 1$, podemos calcular $T(1)$, $T(2)$ y $T(3)$ utilizando $T(n) = 2T(n-1) + n + 2^n$. En concreto, $T(1) = 5$, $T(2) = 16$, y $T(3) = 43$. Con esta información podemos construir el siguiente sistema de ecuaciones lineales:

$$\left.\begin{array}{rcrcrcrcl} C_1 & & + & C_3 & & = & 1 & = & T(0) \\ C_1 & + & C_2 & + 2C_3 & + & 2C_4 & = & 5 & = T(1) \\ C_1 & + & 2C_2 & + 4C_3 & + & 8C_4 & = & 16 & = T(2) \\ C_1 & + & 3C_2 & + 8C_3 & + & 24C_4 & = & 43 & = T(3) \end{array}\right\}.$$

Las soluciones son $C_1 = -2$, $C_2 = -1$, $C_3 = 3$, and $C_4 = 1$. Por tanto, la expresión no recursiva de $T(n)$ es:

$$T(n) = -2 - n + 3 \cdot 2^n + n 2^n \ \in \ \Theta(n 2^n).$$

Argumentos fraccionarios

El método general para ecuaciones en diferencias sólo puede aplicarse cuando las entradas de los términos asociados a T son diferencias de la forma $n-b$, donde b es alguna constante entera. Sin embargo, es posible resolver algunas relaciones de recurrencia que contienen términos en los que las entradas de T son fracciones de la forma n/b, para alguna constante b. La clave reside en transformar la fracción en una diferencia mediante el cambio de variable $n = b^k$, y por tanto suponiendo que n es una potencia de b.

Considérese la relación de recurrencia en (2.14), cuyo caso recursivo es:

$$T(n) = 2T(n/2) + 1,$$

y donde podemos suponer que la entrada es una potencia de dos. Como el argumento en el término T del lado derecho es $n/2$ tenemos que aplicar el cambio de variable $n = 2^k$ para obtener:

$$T(2^k) = 2T(2^k/2) + 1 = 2T(2^{k-1}) + 1.$$

En esta nueva definición $T(2^k)$ es función de k. Por tanto, podemos sustituirla por el término $t(k)$ (un simple cambio de notación):

$$t(k) = 2t(k-1) + 1,$$

que ahora podemos resolver mediante el método general para ecuaciones en diferencias. En particular, la expresión puede escribirse como $t(k) - 2t(k-1) = 1 \cdot 1^n$. Por tanto, el polinomio característico es $(x-2)(x-1)$, y la función tendrá la siguiente forma:

$$t(k) = C_1 + C_2 2^k. \tag{2.30}$$

Deshaciendo el cambio de variable obtenemos una solución general a la recurrencia en términos de la variable original:

$$T(n) = C_1 + C_2 n. \tag{2.31}$$

El último paso consiste en determinar las constantes C_1 y C_2. Esto puede hacerse mediante (2.30) o (2.31). Para T podemos utilizar los casos base $T(1) = 1$ y $T(2) = 3$. Los casos base análogos para t son: $t(0) = T(2^0) = T(1) = 1$, y $t(1) = T(2^1) = T(2) = 3$. De cualquier manera, el sistema de ecuaciones lineales es:

$$\left.\begin{array}{rcl} C_1 + C_2 = 1 = T(1) = t(0) \\ C_1 + 2C_2 = 3 = T(2) = t(1) \end{array}\right\}.$$

Las soluciones son $C_1 = -1$ y $C_2 = 2$, y la expresión no recursiva de $T(n)$ es:

$$T(n) = 2n - 1 \in \Theta(n).$$

En la siguiente relación de recurrencia podemos suponer que la entrada es una potencia de cuatro. En particular, sea:

$$T(n) = \begin{cases} 1 & \text{if } n = 1, \\ 2T(n/4) + \log_2 n & \text{if } n > 1. \end{cases}$$

En este caso aplicando el cambio de variable $n = 4^k$ se llega a:

$$T(4^k) = 2T(4^k/4) + \log_2 4^k = 2T(4^{k-1}) + k \log_2 4 = 2T(4^{k-1}) + 2k.$$

El siguiente paso consiste en realizar la sustitución $t(k) = T(4^k)$:

$$t(k) = 2t(k-1) + 2k.$$

Llamaremos a esta operación "cambio de función" para destacar que sólo se sustituyen los términos T en la expresión (por los términos t). Por tanto, el cambio no afecta al término $2k$. La recurrencia puede reescribirse como $t(k) - 2t(k-1) = 2k \cdot 1^k$, donde el polinomio característico asociado es $(x-2)(x-1)^2$. Así, la recurrencia tiene la forma:

$$t(k) = C_1 2^k + C_2 + C_3 k.$$

Para deshacer el cambio de variable podemos utilizar $k = \log_4 n$, pero necesitamos una expresión para 2^k. En particular, $n = 4^k = (2^2)^k = (2^k)^2$. Por lo tanto, $2^k = \sqrt{n}$, lo que conduce a:

$$T(n) = C_1\sqrt{n} + C_2 + C_3 log_4 n.$$

Finalmente, a partir del caso base $T(1) = 1$ podemos calcular $T(4) = 2T(1) + \log_2 4 = 4$, y $T(16) = 2T(4) + \log_2 16 = 12$. Esto nos permite calcular las constantes C_i resolviendo el siguiente sistema de ecuaciones lineales:

$$\left.\begin{array}{rcl} C_1 + C_2 = 1 & = & T(1) \\ 2C_1 + C_2 + C_3 = 4 & = & T(4) \\ 4C_1 + C_2 + 2C_3 = 12 & = & T(16) \end{array}\right\}.$$

Las soluciones son $C_1 = 5$, $C_2 = -4$, y $C_3 = -2$, y la expresión no recursiva de $T(n)$ es:

$$T(n) = 5\sqrt{n} - 4 - 2\log_4 n \ \in \ \Theta(n^{1/2}).$$

Cuando el argumento de T en la expresión recursiva es una raíz cuadrada, podemos utilizar el cambio de variable $n = 2^{(2^k)}$. Consideremos la siguiente relación de recurrencia:

$$T(n) = 2T(\sqrt{n}) + \log_2 n \tag{2.32}$$

donde $T(2) = 1$ y $n = 2^{(2^k)}$ (nótese que $2^{(2^k)} \neq 4^k$). Esta última restricción sobre n garantiza que el procedimiento recursivo terminará en el caso base para $n = 2$. Aplicando el cambio de variable, tenemos:

$$T(2^{(2^k)}) = 2T(2^{(2^k/2)}) + \log_2 2^{(2^k)} = 2T(2^{(2^{k-1})}) + 2^k.$$

Utilizando el cambio de función $t(k) = T(2^{(2^k)})$, la recurrencia es $t(k) = t(k-1) + 2^k$,

cuyo polinomio característico es $(x-2)(x-2)$. Por tanto, la nueva recurrencia tendrá la forma

$$t(k) = C_1 2^k + C_2 k 2^k.$$

Para deshacer el cambio de variable podemos utilizar $k = \log_2(\log_2 n)$, y $2^k = \log_2 n$. La recurrencia en función de n es por tanto:

$$T(n) = C_1 \log_2 n + C_2(\log_2(\log_2 n)) \log_2 n. \tag{2.33}$$

Por último, podemos utilizar los casos base $T(2) = 1$ y $T(4) = 4$ para encontrar las constantes. En particular, tenemos que resolver el siguiente sistema de ecuaciones lineales:

$$\left.\begin{array}{rcl} C_1 & = 1 = & T(2) \\ 2C_1 + 2C_2 & = 4 = & T(4) \end{array}\right\}.$$

Las soluciones son $C_1 = C_2 = 1$. Por tanto, la fórmula no recursiva final de $T(n)$ es:

$$T(n) = \log_2 n + (\log_2(\log_2 n)) \log_2 n \ \in \ \Theta((\log(\log n)) \log n). \tag{2.34}$$

Multiples cambios de variable o función

La recurrencia anterior en (2.32) también puede resolverse aplicando dos cambios consecutivos de variable (y función). En primer lugar, el cambio $n = 2^k$ conduce a:

$$T(2^k) = 2T(\sqrt{2^k}) + \log_2 2^k = T(2^{k/2}) + k.$$

Sustituyendo $T(2^k)$ por $t(k)$, obtenemos la siguiente recurrencia:

$$t(k) = 2t(k/2) + k,$$

que sigue sin poder resolverse mediante el método, ya que no es una ecuación en diferencias. Sin embargo, podemos aplicar un nuevo cambio de variable para transformarla en una. Con el cambio $k = 2^m$ obtenemos:

$$t(2^m) = 2t(2^{m-1}) + 2^m,$$

que se puede escribir como una ecuación en diferencias realizando el cambio de función $u(m) = t(2^m)$:

$$u(m) = 2u(m-1) + 2^m.$$

Su polinomio característico es $(x-2)^2$, lo que implica que la recurrencia tiene la siguiente forma:

$$u(m) = C_1 2^m + C_2 m 2^m.$$

Deshaciendo los cambios de variable se llega a:

$$t(k) = C_1 k + C_2(\log_2(k))k,$$

y finalmente:

$$T(n) = C_1 \log_2 n + C_2(\log_2(\log_2 n)) \log_2 n,$$

que es idéntico a (2.33). Por tanto, la solución aparece en (2.34).

La estrategia de utilizar varios cambios de variables o funciones puede utilizarse para resolver recurrencias más complejas, como la siguiente no lineal:

$$T(n) = \begin{cases} 1/3 & \text{si } n = 1, \\ n\,[T(n/2)]^2 & \text{si } n > 1, \end{cases}$$

donde n es una potencia de dos. En primer lugar, podemos aplicar el cambio de variable $n = 2^k$, lo que conduce a:

$$T(2^k) = 2^k \left[T(2^k/2)\right]^2 = 2^k \left[T(2^{k-1})\right]^2.$$

Además, con el cambio de función $t(k) = T(2^k)$ obtenemos:

$$t(k) = 2^k \left[t(k-1)\right]^2,$$

que todavía no podemos resolver. Sin embargo, podemos tomar logaritmos a ambos lados de la definición:

$$\log_2 t(k) = \log_2 \left(2^k \left[t(k-1)\right]^2\right) = k + 2\log_2 t(k-1),$$

y aplicar el más complejo cambio de función $u(k) = \log_2 t(k)$ para obtener:

$$u(k) = k + 2u(k-1),$$

que podemos resolver mediante el método. En concreto, su polinomio característico es $(x-2)(x-1)^2$, lo que implica que $u(k)$ tiene la siguiente forma:

$$u(k) = C_1 2^k + C_2 + C_3 k.$$

Deshaciendo los cambios, tenemos primero:

$$t(k) = 2^{C_1 2^k + C_2 + C_3 k},$$

y finalmente:

$$T(n) = 2^{C_1 n + C_2 + C_3 log_2 n}.$$

El último paso consiste en determinar las constantes. Utilizando la recurrencia inicial con $T(1) = 1/3$, obtenemos $T(2) = 2[T(1)]^2 = 2/9$, y $T(4) = 4[T(2)]^2 = 4(2/9)^2 = 16/81$. Podemos utilizar estos valores para construir el siguiente sistema de ecuaciones (no lineales):

$$\left.\begin{array}{rcrcl} 2^{C_1+C_2} & = & 1/3 & = & T(1) \\ 2^{2C_1+C_2+C_3} & = & 2/9 & = & T(2) \\ 2^{4C_1+C_2+2C_3} & = & 16/81 & = & T(4) \end{array}\right\},$$

que puede transformarse en un sistema de ecuaciones lineales tomando logaritmos a ambos lados de las ecuaciones:

$$\left.\begin{array}{rcl} C_1 + C_2 & = & -log_2 3 \\ 2C_1 + C_2 + C_3 & = & 1 - \log_2 9 = 1 - 2\log_2 3 \\ 4C_1 + C_2 + 2C_3 & = & 4 - \log_2 81 = 4 - 4\log_2 3 \end{array}\right\}.$$

Las soluciones son: $C_1 = 2 - \log_2 3 = \log_2(4/3)$, $C_2 = -2$, y $C_3 = -1$. Por lo tanto, $T(n)$ se puede expresar como:

$$\begin{aligned} T(n) &= 2^{(\log_2(4/3))n - 2 - \log_2 n} = \left[2^{\log_2(4/3)}\right]^n \cdot 2^{-2} \cdot 2^{\log_2(1/n)} \\ &= \left[\frac{4}{3}\right]^n \cdot \frac{1}{4n} \in \Theta\left(\left[\frac{4}{3}\right]^n \cdot \frac{1}{n}\right). \end{aligned}$$

2.3.3 Análisis en espacio/memoria

Este apartado contiene una breve introducción al análisis de algoritmos en espacio (es decir, en cuanto a memoria). Lógicamente, la cantidad de memoria que requiere un programa dependerá del tamaño de las variables y estructuras de datos que necesite a lo largo de su ejecución. En este sentido, es útil tener en cuenta el "modelo de memoria" de un programa, que simboliza los datos e información que se almacenan en memoria al ejecutar un programa. La Figura 2.10 ilustra una versión simplificada del modelo de memoria (se suele estudiar en mayor detalle en asignaturas relacionadas con sistemas operativos), el cual consta de cuatro bloques principales:

- Código. La gran mayoría de ordenadores emplea la arquitectura de computadoras Von Neumann, en la que se almacenan tanto datos como instrucciones en la misma unidad de memoria. Por ello a estos ordenadores también se denominan "computadores de programa almacenado".
- Memoria estática. En este bloque se almacena variables globales y datos estáticos, los cuales pueden estar inicializados o no.
- Montículo o *heap*. En esta zona se almacena la memoria dinámica, que se va

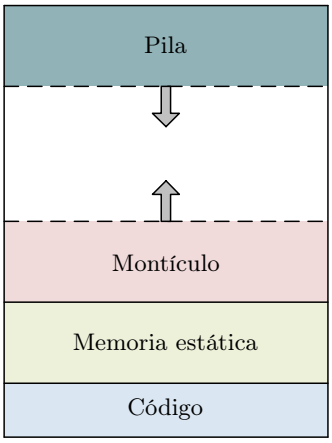


Figura 2.10: Modelo de memoria simplificado.

Código 2.6: Código con tres métodos que se llaman sucesivamente.

```
1  ...
2  A(..)
3  ...
4
5  def A(...)
6      ...
7      B(..)
8      ...
9
10 def B(...)
11     ...
12     C(..)
13     ...
14
15 def C(...)
16     ...
```

reservando y liberando a medida que se ejecuta el programa. Por esto crece y decrece. En algunos lenguajes como C el programador es el principal responsable de hacer un uso adecuado y eficiente de la memoria dinámica. Todo bloque de memoria dinámica que se reserve debe liberarse posteriormente.

- Pila o *stack*. Es la región de memoria asociada a los datos que almacenan los subprogramas (es decir, funciones, procedimientos, subrutinas, métodos, etc.). Por cada llamada a un subprograma se almacena un nuevo "registro de activación" que contiene los parámetros formales y las variables locales del subprograma, además de una serie de datos de bajo nivel (por ejemplo, la dirección de retorno). La pila también crece cuando se llama a un nuevo subprograma, y decrece cuando se retorna del mismo.

Veamos un primer ejemplo sencillo de lo que ocurre en la pila al ejecutar el Código 2.6

Código 2.7: Código Python para hallar en n-ésimo número de Fibonacci según (2.19).

```
def fibonacci(n):
    if n == 1 or n == 2:
        return 1
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)
```

con tres métodos. Desde el programa principal se hace una primera llamada al método `A`, el cual llama al `B`, que a su vez llama al `C`. El correspondiente proceso de llamadas y retornos de los métodos se ilustra en la Figura 2.11(a). En los tres primeros pasos se invocan los métodos, mientras que en los tres últimos se retorna de ellos. La información que se almacena en la pila del programa se muestra en (b). Como se puede apreciar, en el primer paso se guardan los datos asociados al método `A`. Cuando en el segundo paso `A` llama a `B` se crea un nuevo registro de activación asociado a `B`, pero no se borra la información sobre `A`. En el tercer paso se almacenan los datos relacionados con `C`, y cuando finalmente retorna (en el paso 4) el método se "borran" de la pila (técnicamente no se borran, sino que se modifica el puntero de pila, que indica donde termina ésta). Este borrado se repite en los pasos 5 y 6 cuando retornan `B` y `A`, respectivamente. El detalle importante a recordar de este ejemplo básico es que la cantidad de memoria necesaria para ejecutar el código (independientemente de la memoria estática general y la dinámica almacenada en el montículo) es la asociada a los tres métodos, ya que en un momento de la ejecución del programa los registros de activación de los tres métodos se encuentran almacenados en la pila.

La pila juega un papel fundamental al analizar algoritmos recursivos, ya que éstos pueden llegar a realizar multitud de llamadas recursivas, y por tanto pueden dar lugar a multitud de registros de activación. Consideremos la función de Fibonacci en (2.19), codificada en el Código 2.7, cuyo árbol de llamadas recursivas se ilustra en la Figura 2.12(a) para $n = 5$. Por un lado, en el Apartado 2.3.2 vimos que su complejidad en tiempo, especificada por (2.20), y expresada de manera no recursiva en (2.22), era exponencial. En concreto, $T(n) \approx 1{,}618^n$. Obsérvese que un algoritmo con un árbol de llamadas binario donde la profundidad de las hojas fuera la misma (n) tendría $2^n - 1$ nodos. Asumiendo una cantidad constante de operaciones por cada llamada, el coste en tiempo del algoritmo sería $\Theta(2^n)$. En el caso de esta función de Fibonacci el árbol de llamadas es también binario, pero está "podado" (el subárbol de la izquierda tiene más nodos que el de la derecha, para $n \geq 4$). El coste sigue siendo exponencial, pero la base de la potencia es inferior a dos.

Por otro lado, la Figura 2.12(b) muestra el estado de la pila del programa al ejecutar la función para $n = 5$. La clave de esta ilustración es enfatizar que la complejidad en espacio depende de la altura del árbol de llamadas recursivas, que es la profundidad máxima de cualquiera de sus hojas, y que también se puede interpretar como el máximo número de registros de activación en cualquier instante durante la ejecución del programa. En este caso, esta profundidad es $n - 1$, que es una función lineal de n. Además,

los registros de activación sólo requieren almacenar el parámetro de entrada (y otros datos de bajo nivel). Por tanto, el espacio que se requiere por cada llamada o registro de activación es constante, que podemos llamar K. Si el número máximo de registros de activación en la pila es $n-1$, entonces la cantidad de memoria requerida por el método será $K(n-1) \in \Theta(n)$.

Cabe mencionar que si la pila y el montículo se solaparan se produciría un error en tiempo de ejecución denominado *desbordamiento de pila* (*stack overflow*). En Python se limita el número de llamadas recursivas a 1000, aproximadamente. En caso de sobrepasar este límite saltaría un error indicando que se ha excedido de la máxima profundidad de recursión (*maximum recursion depth exceeded*). Esta filosofía de limitar el número de llamadas recursivas tiene mucho sentido. En general, los programas recursivos que podrían sobrepasar esta profundidad son los que solo hacen una llamada recursiva, ya que el correspondiente árbol sería lineal. En cambio, si el proceso de cómputo está distribuido "a lo ancho", donde el método realiza más de una llamada recursiva en sus casos recursivos (los nodos internos del árbol de llamadas tendrían más de un nodo hijo), la altura del árbol de llamadas será generalmente mucho menor que el límite por defecto en Python de unas 1000 llamadas recursivas (es posible aumentar este límite con la sentencia `sys.setrecursionlimit(n)`). Todo esto nos enseña que si el método recursivo solo hace una llamada recursiva entonces merece la pena sustituirlo por una versión iterativa. Ésta sería normalmente más eficiente que la recursiva (ya que no tiene que almacenar información en la pila del sistema), y además se evitarían errores asociados a desbordamientos de pila.

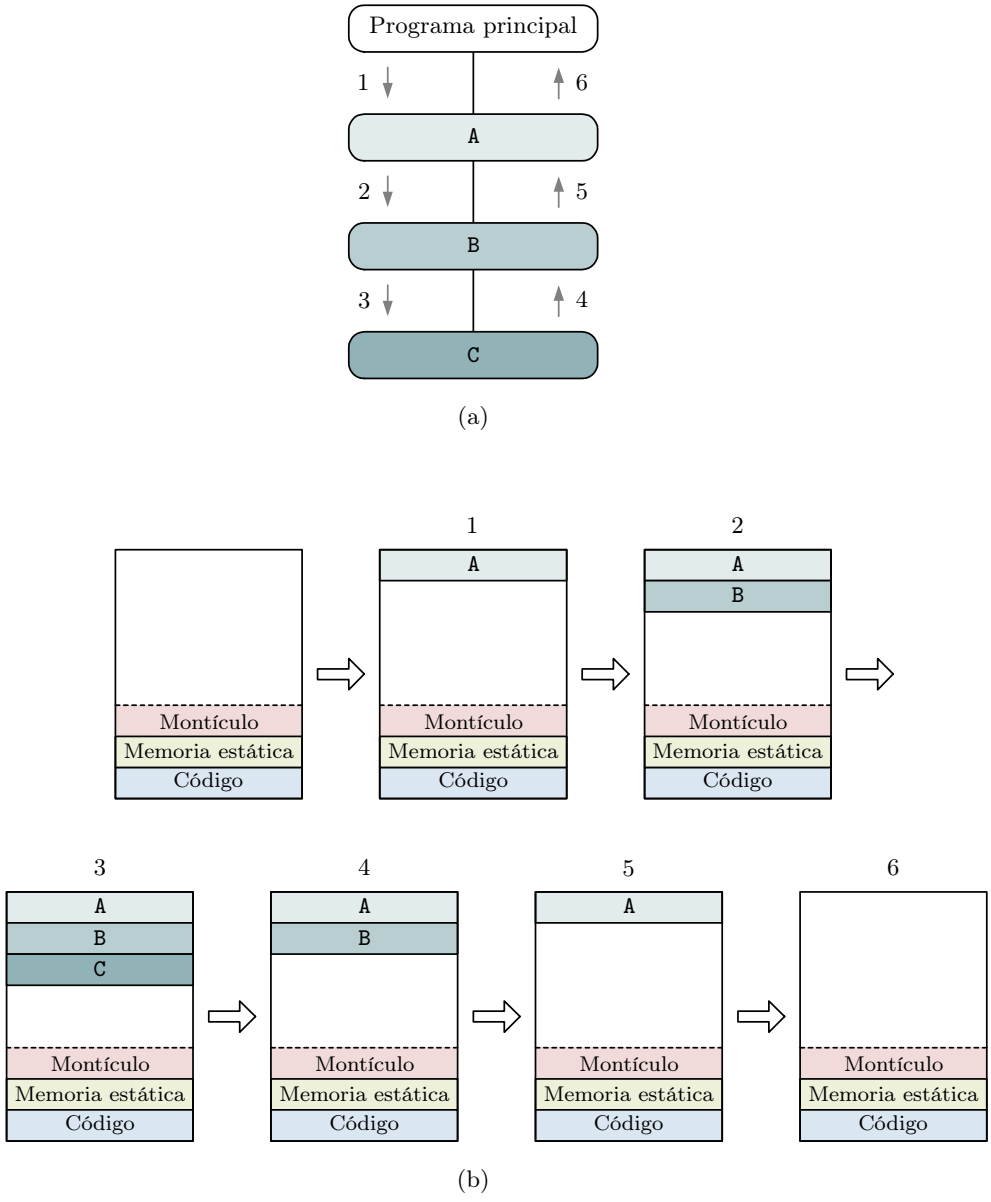
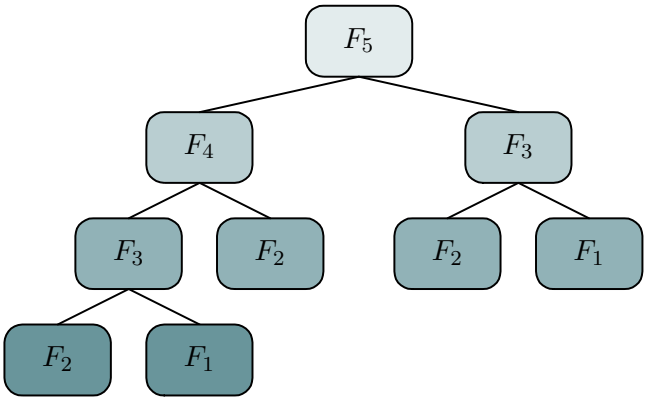
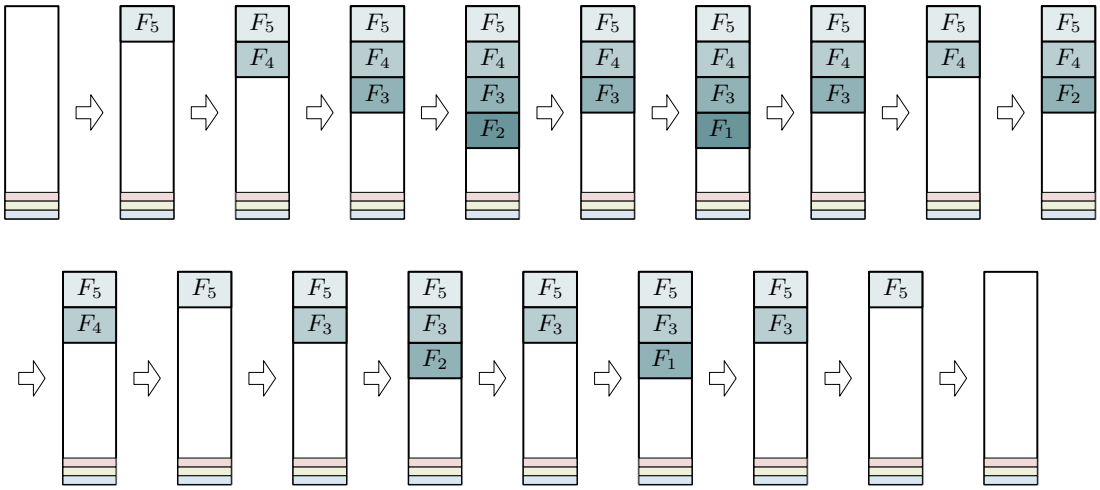


Figura 2.11: Proceso de llamadas y retornos del Código 2.6 en (a), mientras que en (b) se ilustra el estado de la pila dentro del modelo de memoria asociado a la ejecución del código.



(a)



(b)

Figura 2.12: Árbol de llamadas para la función recursiva de Fibonacci en (2.19) e implementada en el Código 2.7, en (a), mientras que en (b) se ilustra el estado de la pila dentro del modelo de memoria asociado a la ejecución del código. En ambas ilustraciones $F(i) = F_i$.