Análisis y Diseño de Algoritmos

Héctor Jiménez Salazar
José de Jesús Lavalle Martínez
FCC, BUAP
Traducción de partes del libro
Fundamentals of Algorithmics de Brassard and Bratley
hjimenez@correo.cua.uam.mx jlavalle@cs.buap.mx

Otoño 2020

Índice general

1. Introducción **5**

2. Bases matemáticas **9**
2.1. Técnicas de demostración 11
2.2. Límites 24
2.3. Probabilidad 29

3. Introducción a la algoritmia **35**
3.1. Problemas y ejemplares 36
3.2. La eficiencia de los algoritmos 37
3.3. Análisis del peor caso y del caso promedio 41
3.4. Operación elemental 45
3.5. La importancia de la eficiencia 49
3.6. Un algoritmo lineal para ordenamiento 52
3.7. Especificación completa de un algoritmo 54

4. Clases de funciones **57**
4.1. Cota superior 57
4.2. Cota inferior 61
4.3. Clase de equivalencia 62
4.4. Cotas condicionales 64

5. Análisis de Algoritmos **67**
5.1. Análisis de estructuras de control 67
5.1.1. Secuencia 68
5.1.2. Ciclos for 68
5.1.3. Llamados recursivos 72
5.1.4. Ciclos while y repeat 73

5.2. Uso de un barómetro 76
5.3. Solución de recurrencias 80
5.3.1. Recurrencias homogéneas 80
5.3.2. Recurrencias no homogéneas 87
5.3.3. Cambio de variable 97

6. Algoritmos voraces 105
6.1. Dar cambio 106
6.2. Características generales de los algoritmos voraces 107
6.3. Grafos: árboles de expansión mínimos 110
6.3.1. Algoritmo de Kruskal 113

7. Divide y vencerás 117
7.1. Introducción: multiplicando enteros grandes 117
7.2. La plantilla general 123

8. Programación dinámica 129
8.1. Dos ejemplos simples 130
8.1.1. Cálculo del coeficiente binomial 130
8.1.2. La serie mundial 131

9. Complejidad computacional 135
9.1. Preliminares 135
9.2. La clase P 137
9.2.1. Ejemplos de problemas en P 139
9.3. La clase NP 146
9.3.1. Ejemplos de problemas en NP 151
9.3.2. ¿P = NP? 153
9.4. Completitud-NP 155
9.4.1. Reducibilidad en tiempo polinomial 156
9.4.2. Definición de Completitud-NP 160
9.4.3. El teorema de Cook-Levin 160

Capítulo 1

Introducción

El 10 de Agosto de 2002 murió E.W. Dijkstra, un físico dedicado a la computación hace más de cuarenta años. Él propuso el algoritmo de camino mínimo para recorrer un mapa, el sistema operativo de multiprogramación "THE", las bases de la programación estructurada, y participó en la definición del lenguaje de programación ALGOL, entre otras contribuciones.

El hecho es significativo no sólo por la pérdida humana sino por el coincidente fin del liderazgo de la Ciencia de la Computación en el manejo de la información, tal como J. Denning dio a entender en febrero de 2001: es una ironía que quien dio vida a la computación pase a ser una parte más de lo que hoy se llama la Tecnología de la Información (TI).

Así, ha terminado el ciclo del establecimiento de los principios fundamentales de una ciencia. La tecnología que ésta ha generado no sólo avanza con más rapidez, sino que impone la línea de desarrollo. Los profesionistas de la TI se apoyan indiscutiblemente en las bases desarrolladas por los científicos de la computación, aquéllos usan los sistemas operativos, protocolos de comunicación, lenguajes de programación, sistemas de bases de datos, sistemas de ayuda al diseño, etc, para muy diversos fines que la sociedad demanda, muy probablemente sin conocer detalles fundamentales que garantizan el funcionamiento de los sistemas construidos.

Es un hecho, al igual que la Ingeniería Hidráulica se apoya en la Física, la TI se apoya en la Ciencia de la Computación, aun la última con su núcleo en la Teoría de la Computación.

En el inicio de la Computación la TI era parte de la primera, sin diferenciar todavía la explosión de áreas que fueron surgiendo. Bien que la Matemática ha dado la pauta para la solución de problemas de muy diversas

disciplinas, en muchas aplicaciones había que desarrollar el algoritmo pues éste no existía.

Con el tiempo se reunió una cantidad considerable de algoritmos que abarcan las más variadas aplicaciones, o que pueden ser adaptados con facilidad. Al mismo tiempo se fueron planteando preguntas fundamentales de la Ciencia de la Computación como el asegurar que un programa termine (decidibilidad). O bien, si sabemos que en ciertas condiciones el algoritmo encontrará un resultado desearíamos saber si éste se obtiene en un tiempo razonable (tratabilidad).

Y, aunque pueda ser razonable el tiempo en que un algoritmo termine, nos preguntamos si ese tiempo puede ser reducido (ubicarlo en una clase de complejidad menor). Éstas son cuestiones eminentemente de la Ciencia de la Computación pero, como dijimos, al proporcionar más piezas de diferentes características (algoritmos) podemos armar muy sofisticados e insospechados rompecabezas (sistemas), lo cual, con base en que una computadora es un sistema que habilita el funcionamiento de otros sistemas (máquina universal), este proceso demandó herramientas que ayudaran a la tarea de no únicamente ensamblar rompecabezas, además, acercándose a las preguntas fundamentales, se concibió asegurar que el rompecabezas construido funcionará como era esperado.

Lo anterior refiere a procesos de ingeniería, que, como sabemos, en computación se llama Ingeniería de Software. Particularmente, de la garantía de que un programa funcione se ocupa, con un fuerte sesgo teórico, la Especificación Formal.

Así como la Lógica Matemática creó PROLOG, fueron propuestos, para la Ingeniería de Software, lenguajes que permitieran especificar sistemas formalmente. Con este ejemplo tenemos una muestra del tejido entre Ciencias de la Computación y la TI.

Al parecer el Análisis y Diseño de Algoritmos estaría condenado a la extinción pues ya hay muchos algoritmos, e incluso código en clases para los lenguajes orientados a objetos, sin embargo cada día se reafirma la idea de C. Babbage que se puede expresar como: siempre habrá necesidad de crear un algoritmo más veloz para una nueva máquina.

Esto es, un científico de la Computación no puede prescindir de saber cómo se analiza un algoritmo, o cómo se adapta o se diseña uno para nuevas condiciones, ya que una de sus tareas es crear algoritmos. La sentencia de Babbage no pierde vigencia pues no debe olvidarse que aún cuando en el término de un año pueda triplicarse, con el Hardware, la velocidad de

cómputo el descenso de la complejidad de un algoritmo puede representar una mejora no en múltiplos de velocidad sino en órdenes de magnitud. Precisemos esto un poco más.

El aumento de velocidad de una computadora afecta a todos los programas que ésta ejecute, mientras que el mejoramiento "real" de un algoritmo solamente afecta a los programas que lo usen. Aclarado lo anterior, nos permitimos hacer la siguiente comparación: puede decirse que en 4 años la velocidad de cómputo aumenta por un factor de cuatro, en tanto puede afirmarse que como fruto del estudio de algunos algoritmos "sencillos" en cinco años la velocidad aumenta por un factor de cien.

Pero, a diferencia del aumento de la velocidad ganada por una nueva computadora (que es un múltiplo de la velocidad previa), la velocidad mejorada por un algoritmo no se debe a un factor fijo sino a uno variable o mejor dicho a una función. Esto es, si aumentamos nuestros datos de entrada al algoritmo diez veces más entonces, tendremos un factor de velocidad de mil y no de cien. Supuesto que cada vez se demanda el procesamiento de más datos.

En conclusión, el Análisis y Diseño de Algoritmos puede verse como una forma de mejorar parte del de la máquina con una "tecnología" que se desarrolla potencialmente en una pequeña fracción del tiempo que requiere el desarrollo de las nuevas computadoras (para nuestro ejemplo, el Hardware necesitaría aproximadamente 250 años para lograr la velocidad de un algoritmo mejorado).

Empecemos entonces a estudiar un tema que está en la frontera de la Ciencia Computacional y muchas áreas de la TI.

Capítulo 2

Bases matemáticas

Siempre una necesidad ha sido asegurar, bajo ciertas condiciones, un suceso. Este principio ha guiado el desarrollo de la ciencia, y también de otros sistemas. En términos prácticos debe asugurarse, por ejemplo, que un puente no se caiga, que no haya corrupción, que un programa obtenga resultados en cierto periodo de tiempo, etc.

Aunque estamos acostumbrados a vivir en un mundo "causa-efecto", es decir que para conseguir algo sabemos qué debe hacerse, no ha sido así todo el tiempo. Estas reglas han sido decubiertas o convenidas a lo largo de muchas experiencias.

En los mundos "desconocidos" debemos, antes de resolver problemas, experimentar y aprender. Para este propósito suele ser útil la experiencia ganada en problemas semejantes.

La matemática ha sido lo común en la solución de muchos problemas. A través de los modelos, la matemática ha proporcionado una forma de ver el problema y predecir su comportamiento en ciertas condiciones.

La modelación nos permite aplicar un método a un problema que satisface ciertas condiciones. Suficiente para muchos problemas resulta aplicar algún método matemático. Sin embargo, tendrá que conocerse, al menos, el fundamento de dicho método para aplicarlo a nuevas situaciones. Esto conlleva a ejercer el razonamiento matemático.

El razonamiento matemático es una vivencia de convencimiento sobre abstracciones. El ejercitador matemático no sólo asiente sus expresiones, tiene argumentos, que se basan en las convenciones establecidas, para sostenerlas.

Esta explicación del "por qué" se afirma algo se llama demostración. El lector de matemáticas habrá de consentir igualmente la explicación del

escritor o, en su caso, refutarla en el mismo sistema de convenciones.

Estableceremos un modelo que represente el *comportamiento* de un algoritmo para llevar a cabo su análisis. Esto lo haremos en la sección dos, por lo pronto presentaremos las herramientas necesarias que permiten llevar a cabo el razonamiento matemático en el modelo que vamos a proponer.

Un conjunto se representa extensionalmente, por ejemplo $\{3, 6, 9, 12\}$ o intensionalmente, $\{x|x$ es un múltiplo de 3 menor que $15\}$. Es común el empleo de los conjuntos de números: naturales, enteros, racionales y reales, denotados por $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$ y $\mathbb{R}$, respectivamente.

Dados dos conjuntos A y B, el conjunto de elementos comunes a A y B o intersección, se expresa por $A \cap B$. Empleando la relación de pertenencia "$\in$", entre un elemento a y un conjunto B, $a \in B$, $A \cap B$ se escribe $\{x|x \in A \text{ y } x \in B\}$. Asimismo, se llama unión de A y B al conjunto $\{x|x \in A \text{ o } x \in B\}$ y producto de A por B, a $A \times B = \{(x, y)|x \in A \text{ y } y \in B\}$.

Se dice que el conjunto A está contenido en el conjunto B si todo elemento de A está en B: $A \subset B$. Algunas definiciones vienen a complementar lo anterior.

Sea P un conjunto. Una *relación binaria* R definida en P es un subconjunto de $P \times P$. Así, $x, y \in P$ se dicen relacionados por R, xRy, si $(x, y) \in R$. Con más generalidad podemos concebir una relación R de aridad $m > 0$ definida en una colección de m conjuntos $\{P_i\}_i$ como un subconjunto del *universo* $U = \Pi_{i=1}^{m} P_i = P_1 \times \ldots \times P_m$. Si $m = 1$ suele llamarse a R *propiedad*.

Es también usual que una relación binaria vista como un subconjunto de $P \times Q$ sea llamada *correspondencia*. Particularmente, cuando una correspondencia $R \subset P \times Q$ es tal que para cada $x \in P$ existe un único elemento $y \in Q$ tal que xRy, tenemos una *función* $P \to Q$. Dada una función $f : P \to Q$, a P se le llama *dominio*, a Q *imagen*, y a $f(P) = \{y \in Q|y = f(x), \text{ para algún } x \in P\}$ el *rango*. Si $f(P) = Q$, f se llama *suprayectiva*, y si para cualesquier $x_1, x_2 \in P, x_1 \neq x_2$ se cumple que $f(x_1) \neq f(x_2)$ a f se le llama *inyectiva*. Si $f : P \to Q$ es *biyectiva* (inyectiva y suprayectiva) existe una función $f^{-1} : Q \to P$ que se llama la *inversa* de f, i.e. $f^{-1}(f(x)) = x$ para toda $x \in P$.

Sean P un conjunto, R una relación en P y x, y, z cualesquier elementos de P. R es una relación: *reflexiva* si xRx, *simétrica* si xRy implica yRx, *transitiva* si xRy y yRz implica xRz, *antisimétrica* si xRy y yRx implica $x = y$.

Sean P un conjunto y R una relación definida en P: R es de *equivalencia* si es reflexiva, simétrica y transitiva, R es de *orden* (más precisamente, *orden*

parcial) si es reflexiva, antisimétrica y transitiva. Si para toda pareja $x, y \in P$ sucede que xRy o yRx, se dice que R es un *orden total*.

2.1. Técnicas de demostración

En esencia requerimos de definiciones y propiedades para explicar algún hecho dentro de la matemática.

Por ejemplo: afirmamos que el producto de dos impares es impar, pues si observamos cualesquier pareja de impares, digamos $a = 2i + 1$ y $b = 2j + 1$, expresamos su producto: $ab = 4ij + 2i + 2j + 1 = 2(2ij + i + j) + 1$, esto es un impar.

Observemos que la afirmación es en general, y que esto está considerado al dar a y b cualesquiera, solamente partiendo de la definición de impar. Luego usamos las propiedades de la multiplicación para nuevamente observar el resultado. Normalmente, partir de las definiciones y escribir lo que se está afirmando constituye la pauta de la demostración, si no es que la demostración misma.

Con una demostración logramos asegurar que la afirmación conjeturada adquiera un rango de verdad siempre que se satisfagan las premisas empleadas.

Estaremos enunciando, en lo que sigue, propiedades de los números enteros o reales, y realizaremos algunas demostraciones.

Recordemos que para los números reales se cumple $b^{v+w} = b^v \cdot b^w$. Se define $\log_b x = y$ si y sólo si (sii) $x = b^y$, es igual decir que $\log_b(x)$ es la función inversa de b^y: $y = \log_b x = \log_b b^y$. Demostremos que $\log_b(xy) = \log_b x + \log_b y$:

Usamos la definición en la parte izquierda de lo que deseamos demostrar: $b^{\log_b(xy)} = xy$ y obervamos qué sucede en la parte derecha de la igualdad: $b^{\log_b x + \log_b y}$, a esta última aplicamos la propiedad enunciada de las potencias: $b^{\log_b x} \cdot b^{\log_b y}$ y aplicando nuevamente la definición a esta fórmula, tenemos que $b^{\log_b x + \log_b y} = xy$.

Es decir, se cumple la igualdad. Usaremos simplemente log para indicar $\log_2$ y ln para $\log_e$ (logaritmo natural).

Ejercicio 1 Demuestre que:

1. $\log_b x^y = y \log_b x$
2. $\log_a x = \frac{\log_b x}{\log_b a}$

3. $x^{\log_b y} = y^{\log_b x}$

Veamos otro ejemplo. Se define para $x \in \mathbb{R}$, $\lfloor x \rfloor$, el *suelo* de x, como el mayor entero que no es mayor que x (p.e. $\lfloor 32{,}7 \rfloor = 32$, $\lfloor -0{,}2 \rfloor = -1$), asimismo $\lceil x \rceil$ como el menor entero que no es menor que x, el *techo* de x. Intuitivamente, podemos conjeturar que se cumple: $x - 1 < \lfloor x \rfloor \leq x \leq \lceil x \rceil < x + 1$.

Para asegurar que así es, debe demostrarse cada una de las desigualdades de la expresión anterior. Abordaremos las primeras dos. Observemos que, por definición, el suelo de un real x nunca estará a una distancia de $\lfloor x \rfloor$ mayor que uno (el caso extremo es cuando x es entero), en fórmulas: $x - \lfloor x \rfloor < 1$, luego $x - 1 < \lfloor x \rfloor$. La segunda desigualdad se cumple por definición.

El suelo es útil en las siguientes definiciones: dados dos enteros a, b el cociente de dividir a por b se denota a/b, y el *cociente entero* $a \div b \doteq \lfloor a/b \rfloor$, además a *módulo* b es $a \mod b = a - b \times (a \div b)$.

Ejercicio 2 Demuestre que

1. $x \leq \lceil x \rceil < x + 1$.

2. $a \leq b^{\lceil \log_b a \rceil} < ab$.

3. Para cualesquier $a, b \in \mathbb{Z}$, $b \geq a$, es cierto que $b \mod a < b/2$.

La práctica de demostraciones se ha sistematizado al grado de llegar a establecer un conjunto de reglas para llevar a cabo demostraciones con mayor alcance. La lógica ofrece sistemas para expresar variadas formas de concebir la veracidad de una expresión.

Frecuentemente usamos fórmulas lógicas como $\neg\mathcal{A}$, $\mathcal{A} \to \mathcal{B}$, $\mathcal{A} \vee \mathcal{B}$, $\mathcal{A} \wedge \mathcal{B}$ y $\mathcal{A} \leftrightarrow \mathcal{B}$, donde se tienen los conectivos negación, implicación, disyunción, conjunción, y equivalencia, con $\mathcal{A}$ y $\mathcal{B}$, cualesquier fórmula o, en su caso más simple, una proposición p_i (p.e. “$11 < 3$”, “WTC es el edificio más alto”, etc.) aquellas que no pueden ser descompuestas.

Aunque también conviene referirse a predicados, que generalizan las proposiciones por permitir el uso de variables. Las fórmulas se constituyen ahora con cuantificadores: $\forall x\mathcal{A}$ y $\exists x\mathcal{A}$.

Para las últimas tenemos una relación: si decimos que para todo x se cumple $\mathcal{A}$, estamos afirmando que no existe x que no cumpla con $\mathcal{A}$: $\forall x\mathcal{A}$ equivale a $\neg\exists x\neg\mathcal{A}$. También participan en el lenguaje de la lógica símbolos auxiliares como la coma y los paréntesis.

Deben distinguirse dentro de las expresiones que pueden formarse con este lenguaje aquellas que están bien construidas o fbc. Para la buena formación de expresiones se siguen los patrones que presentaron a los conectivos, unas líneas arriba.

Esto es $\mathcal{A}\neg$ no está bien formada, tampoco $\mathcal{AB}$, pero sí $(\mathcal{B} \vee \mathcal{A}) \leftrightarrow \mathcal{B}$ y $(\mathcal{B} \rightarrow \mathcal{B}) \rightarrow (\neg\mathcal{A} \wedge \mathcal{C})$, etc.

Diremos un poco más sobre las fórmulas compuestas por proposiciones. Como una fbc se forma de acuerdo con los patrones antes vistos, y la veracidad de una fbc depende de la veracidad de sus componentes (composicionalidad), tendremos que conocer la veracidad de los elementos más simples: las proposiciones.

Implícitamente estamos considerando conocimiento de la forma en que se evalúan los conectivos. Más formalmente, una *valuación* es una función v cuyo dominio es el conjunto de las fbcs y cuyo rango es el conjunto $\{V, F\}$, tal que para cualesquier fbcs $\mathcal{A}$ y $\mathcal{B}$:

1. $v(\mathcal{A}) \neq v(\neg\mathcal{A})$,
2. $v(\mathcal{A} \rightarrow \mathcal{B}) = F$ sii $v(\mathcal{A}) = V$ y $v(\mathcal{B}) = F$.
3. $v(\mathcal{A} \vee \mathcal{B}) = F$ sii $v(\mathcal{A}) = F$ y $v(\mathcal{B}) = F$.
4. $v(\mathcal{A} \wedge \mathcal{B}) = V$ sii $v(\mathcal{A}) = V$ y $v(\mathcal{B}) = V$.
5. $v(\mathcal{A} \leftrightarrow \mathcal{B}) = V$ sii $v(\mathcal{A}) = v(\mathcal{B})$.

Lo anterior refiere a la tabla de certeza de una fbc. Por ejemplo: para la fbc $(p \vee q) \rightarrow r$ (cada valuación es un renglón):

p	q	r	$(p \vee q)$	$(p \vee q) \rightarrow r$
V	V	V	V	V
V	V	F	V	F
V	F	V	V	V
V	F	F	V	F
F	V	V	V	V
F	V	V	V	V
F	V	F	V	F
F	F	V	F	V
F	F	F	F	V

Además, se dice que una fbc $\mathcal{A}$ es una *tautología* si para toda valuación v, $v(\mathcal{A}) = V$, y que una fbc $\mathcal{A}$ es una *contradicción* si para toda valuación v, $v(\mathcal{A}) = F$.

Ejercicio 3 Comprobar si las siguientes fbc son tautologías:

1. $\neg\neg p \rightarrow p$

2. $p \rightarrow \neg\neg p$

3. $(p \rightarrow q) \rightarrow (\neg q \rightarrow \neg p)$

4. $(\neg p \rightarrow \neg q) \rightarrow (q \rightarrow p)$

5. $(p \rightarrow q) \rightarrow ((p \rightarrow \neg q) \rightarrow \neg p)$

Observación 1

1. Aunque las fbc anteriores usan solamente proposiciones el resultado obtenido puede generalizarse. Esto es, $p \rightarrow \neg\neg p$ no cambiaría su valor de certeza al sustituir por p cualquier otra fbc, digamos $\mathcal{A}$, $\mathcal{A} \rightarrow \neg\neg\mathcal{A}$. Esto es debido a que en la tabla, finalmente $\mathcal{A}$ sólo podrá, como p, tomar uno de dos valores, V o F, y obtener el resultado final al igual que con p. Así las fbc del ejercicio 3 se cumplen en general: $\neg\neg\mathcal{A} \rightarrow \mathcal{A}$, $\mathcal{A} \rightarrow \neg\neg\mathcal{A}$, etc.

2. Algunas fbc pueden ser reescritas en virtud del *teorema de la deducción*. Por ejemplo: $(p \rightarrow q) \rightarrow (\neg q \rightarrow \neg p)$, como $(p \rightarrow q) \vDash (\neg q \rightarrow \neg p)$: a partir de $(p \rightarrow q)$ podemos inferir $\neg q \rightarrow \neg p$. O bien, $(p \rightarrow q), \neg q \vDash \neg p$: si contamos con las premisas $(p \rightarrow q)$ y $\neg q$, podemos inferir $\neg p$.

La última fbc del ejercicio 3 expresa el esquema conocido como *reducción al absurdo*: Si a partir de p obtenemos q, pero también $\neg q$, entonces concluimos $\neg p$, esto es si suponemos p y llegamos a que q y no q entonces p es falso. Esta es justamente la primera técnica de demostración que vamos a presentar.

Contradicción

Para ilustrar esta técnica de demostración haremos antes algunas definiciones. Un número q se llama *racional* si hay un múltiplo entero de él que es entero: $\exists a \in \mathbb{Z}, a \neq 0$: $aq = b \in \mathbb{Z}$, es decir $q = b/a$ es una fracción. En caso contrario se dice que q es irracional. Claramente tenemos muchos enteros que cumplen con la definición de racional: si $b = qa$ entonces $mb = qma$, $ma, mb \in \mathbb{Z}$.

Convenimos en tomar el múltiplo menor de q que satisfaga la condición antedicha. También se dice que los enteros que definen el racional no tienen factores comunes diferentes a 1 o -1. El siguiente es un ejemplo enunciado por Euclides (hacia 330 a.n.e.).

Teorema 1 $\sqrt{2}$ es irracional.

Demostración. Supongamos que $\sqrt{2}$ es racional. Esto significa $\sqrt{2} = b/a$ con $a, b \in \mathbb{Z}$ sin factores comunes. Entonces, podemos escribir $2 = b^2/a^2$, $2a^2 = b^2$, y b debe ser par. De esta forma, $b = 2c^2$ y, por tanto, $2a^2 = 4c^2$, $a^2 = 2c^2$. Luego, a y b son pares, lo cual es una contradicción. $\square$

En la demostración anterior hemos usado la tautología de reducción al absurdo: $(p \to q), (p \to \neg q) \models \neg p$, tomando como $p =$ "$\sqrt{2}$ es racional", $q =$ "$\sqrt{2} = b/a$ con a y b sin factores comunes", $\neg q =$ "a y b son pares", y $\neg p =$ "$\sqrt{2}$ es irracional".

Antes de ver otro ejemplo recordaremos algunas definiciones. Un número es llamado *primo* si tiene exactamente dos divisores. Para un primo p los divisores aludidos son p y 1, y dicha propiedad se escribe: $p|p$ y $1|p$. Observe que 1 no es primo. Con la relación de *divisibilidad* entre dos enteros a y b, $a|b$ estamos indicando que $b \mod a = 0$. Esta relación tiene las siguientes propiedades: i) si $a|b$ y $a|c$ entonces $a|(b+c)$ y $a|(b-c)$, ii) si $a|b$ y $b|c$ entonces $a|c$.

Teorema 2 Hay una cantidad infinita de números primos.

Demostración. Supongamos que los primos son $P = \{2, 3, 5, \ldots, p_k\}$. Claramente el número $n = (2 \cdot \ldots \cdot p_k) + 1 > p_k$ y, por tanto, debe ser divisible por algún $x > 1$ y $x < n$ (de lo contrario n sería primo y tendríamos una contradicción) . Sea m el menor número tal que $m|n$. m debe ser primo (de no serlo, habría un $p < m$ que divide a m y, por tanto, m no sería el menor divisor de n). Como $m \in P$, $m|(n-1)$, pero esto es imposible puesto que $m|n$. $\square$

La demostración anterior se hace por contradicción y además dentro de ella se hace una construcción que se justifica también por contradicción, las demostraciones referidas aparecen entre paréntesis.

Ejercicio 4 Demostrar las siguientes afirmaciones:

1. Para todo número primo mayor que dos su antecesor es par.
2. Si $c^3 = 5$ entonces c es irracional.
3. El cubo de el mayor de tres enteros consecutivos no puede ser igual a la suma del cubo de los otros dos.
4. La suma de los cuadrados de tres enteros consecutivos no puede tener residuo -1 al ser dividida por 12.

Inducción

El *Principio de Inducción Matemática* (PIM) es una característica de los números naturales. Ésta surge en la construcción intuitiva de tales números, al aprender a contar: "la numeración empieza con 1z "siempre podemos obtener un número mayor sumando 1 al que tenemos". Más formalmente se dice que el conjunto de números naturales cumple la propiedad $\mathcal{A}$ si 1 la cumple, y si cualquier n cumple $\mathcal{A}$ también lo hace $n+1$: $\mathcal{A}(1)$, y $[\mathcal{A}(n) \to \mathcal{A}(n+1)]$.

Cuando deseamos probar que una propiedad es cumplida por todos los elementos de $\mathbb{N}$, basta demostrar dos cosas: (*base*:) 1 satisface dicha propiedad y, (*paso inductivo*:) suponiendo que $n \in \mathbb{N}$ la satisface (*hipótesis de inducción*), $n + 1$ también satisface la propiedad. Aclaremos lo anterior con un ejemplo.

Proposición 1 Para todo natural se cumple $\log n < n$.

Demostración. Por inducción. Base: $\log 1 = 0 < 1$.
Inducción: Debe demostrarse que si $\log n < n$ entonces $\log(n + 1) < n + 1$. Puesto que log es una función creciente al igual que su inversa: $n < 2^n$, por tanto, $(n + 1) < 2^n + 1$, pero es también cierto que $2^n + 1 < 2^n + 2^n$. Resumiendo: $(n + 1) < 2^{n+1}$ lo cual equivale a decir que $\log(n + 1) < n + 1$.
□

Es común establecer la validez de una propiedad $\mathcal{A}$ solamente para una parte de los números naturales, esto es: $\mathcal{A}(n)$ para todo $n \geq n_0$, lo cual

es equivalente a decir que la propiedad $\mathcal{B}(n)$ se cumple para todo $n \in \mathbb{N}$, donde $\mathcal{B}(n) \doteq \mathcal{A}(n + n_0 - 1)$. Así debe demostrarse que $\mathcal{A}(n_0)$ (base) y $\mathcal{A}(n) \rightarrow \mathcal{A}(n+1)$ para $n \geq n_0$ (inducción). El siguiente ejemplo muestra la utilidad de esta forma del PIM.

Proposición 2 Para todo $n \geq 5$, $2^n > n^2$.

Demostración. Base: $32 > 25$. Inducción: La hipótesis de inducción es $\mathcal{A}(n-1) = 2^{n-1} > (n-1)^2$. Parte de $\mathcal{A}(n)$ es: $2^n = 2 \cdot 2^{n-1}$. Por hipótesis de inducción $2^{n-1} > n^2 - 2n + 1$, de aquí $2^n > 2n^2 - 4n + 2 = n^2 + (n-2)^2 - 2 > n^2$, para $n \geq 5$. $\square$

Ejercicio 5

1. Diga para qué valores no se cumple $2^n \leq n^3$.
2. Diga qué valores satisfacen la relación $86400n^2 < n^3$.
3. Determine el valor de c para que se cumpla la desigualdad:
   a) $n^3 - 3n^2 - n - 8 < cn^3$.
   b) $cn^3 < n^3 - 3n^2 - n - 8$.
   c) $n^2 + n^3 + n \log n < cn^3$.

En lo que sigue será empleada notación adicional como sumatorias y productos, conveniente para expresar fórmulas importantes que se apoyan en la inducción. Es común escribir los términos de una suma ”largaçomo $a_1 + a_2 + \ldots + a_n$, donde se supone conocido los términos a_i, y los puntos suspensivos abrevian sólo la escritura.

La suma anterior se representa con $\sum_{i=1}^{n} a_i$, donde identificamos el símbolo de suma ($\sum$), el índice (i), y los límites inferior y superior del índice (en este caso 1 y n, respectivamente). Es claro que si todos los términos de una sumatoria son iguales, entonces se tiene el producto del número de elementos sumados por el término que se suma, por definición de producto: $\sum_{i=1}^{n} a = n \cdot a$.

Igualmente podemos explicar otras operaciones en esta notación, como: $\sum_{i=1}^{n} b \cdot a_i = b \cdot \sum_{i=1}^{n} a_i$ y $\sum_{i=1}^{n} a_i + b_i = \sum_{i=1}^{n} a_i + \sum_{i=1}^{n} b_i$, la primera por distributividad del producto sobre la suma y, la segunda, por la asociatividad de la suma.

Ejercicio 6 Justifique las siguientes igualdades:

1. $$\sum_{i=1}^{n} a_i = \sum_{k=1}^{n} a_k.$$

2. $$\sum_{i=1}^{n} a_i = \sum_{i=0}^{n-1} a_{i+1}.$$

3. $$\sum_{i=0}^{n-1} a_i = \sum_{i=1}^{n} a_{n-i}.$$

4. $$\sum_{i=1}^{n} a_i = \sum_{i=1}^{n-1} a_i + a_n.$$

5. $$(\sum_{i=1}^{n} a_i)(\sum_{j=1}^{m} b_j) = \sum_{i=1}^{n}\sum_{j=1}^{m} a_i b_j.$$

Una operación semejante es el producto de n términos: $a_1 \cdot a_2 \cdot \ldots \cdot a_n = \prod_{i=1}^{n} a_i$. En particular, se define el *factorial* de un natural n como: $n! = \prod_{i=1}^{n} i$. El factorial puede ser entendido como las posibles permutaciones de n objetos. Expliquémoslo más. Debemos tener presente que dado un conjunto A de k objetos, tendríamos k posibles elecciones de un objeto.

Si, en cambio, deseamos formar parejas del conjunto A tendremos k^2 de éstas, cuando permitimos la repetición, pero solamente $k(k-1)$ si no permitimos la repetición de objetos al formar las parejas de A. Es fácil encontrar expresiones para formar ternas, cuartetos u otras tuplas de objetos.

En el razonamiento anterior estamos haciendo uso de una regla del conteo llamada "del producto": multiplicamos el número de posibilidades del conjunto que interviene en cada elección.

Una *permutación* es una manera de disponer en orden n objetos. Para determinar una permutación procedemos eligiendo uno por uno de los objetos: si tenemos n objetos iniciales: para elegir el primero hay n posibilidades, para elegir el segundo sólo habrá $n-1$ etcétera, para el último habrá una

posibilidad. Así, por la regla del producto, la cantidad de posibles permutaciones de n objetos es $n!$. También es útil referirnos a las combinaciones de un conjunto de n objetos.

La diferencia con las permutaciones es que en una combinación no importa el orden de los objetos, lo cual tiene sentido cuando se combinan k objetos a partir de n; ya que combinar n a partir de n sólo tenemos una combinación. Por el argumento expuesto para determinar la cantidad de permutaciones, una *combinación* de k objetos tomada de n, denotado por $\binom{n}{k}$, se forma en primer lugar de $n \cdot (n-1) \cdot \ldots \cdot (n-k+1) = \prod_{i=n}^{n-k+1} i$ formas.

Estos k objetos pueden a su vez ser dispuestos en $k!$ maneras, como se ha visto. Por tanto, si despreciamos el orden de las $\prod_{i=n}^{n-k+1} i$ formas obtenidas tendremos que $\binom{n}{k} = \prod_{i=n}^{n-k+1} i/k!$. La expresión $\binom{n}{k}$ se lee *binomial* k de n.

Observación 2

1. Partiendo de la definición de combinación, veamos algunos valores del binomial:

   a) No tomar ningún elemento de n es una forma de elección: $\binom{n}{0} = 1$.

   b) Al tomar uno, de un total de n, hay n formas de hacerlo: $\binom{n}{1} = n$.

   c) Para tomar todos sólo hay un modo: $\binom{n}{n} = 1$

   d) Tomar $n-1$ de n equivale a quitar uno de ellos, los cuales son n posibles: $\binom{n}{n-1} = n$

   Como será visto adelante, el nombre binomial proviene del empleo de este valor en la potencia de un binomio.

2. En $\binom{n}{k}$ k es menor o igual a n, de manera que si construimos una tabla variando n y k obtendremos un triángulo. Tomemos en cada renglón un valor de n y en cada columna uno de k.

   Lo singular de este triángulo es que: (i) en la diagonal y la primera columna solamente hay unos, ya que $\binom{n}{0} = 1$ y $\binom{n}{n} = 1$, (ii) los demás valores se determinan observando que cada uno es la suma del que está arriba de él con el que está arriba a la izquierda de él: $\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$. Esta tabla es conocida como *Triángulo de Pascal.*

Partiendo de la regularidad observada en las potencias de $(1+x)$ $(1+2x+x^2, 1+3x+3x^2+x^3, \ldots)$ puede conjeturarse la identidad conocida como *Binomio de Newton* demostrada a continuación.

Teorema 3 Para toda $n \geq 0$

$$(1+x)^n = \sum_{k=0}^{n} \binom{n}{k} x^k.$$

Demostración. Por inducción sobre n.
Base: $\sum_{k=0}^{1} \binom{1}{k} x^k = \binom{1}{0} x^0 + \binom{1}{1} x^1 = (1+x)^1$.
Inducción: Suponemos que

$$(1+x)^n = \sum_{k=0}^{n} \binom{n}{k} x^k.$$

Debemos demostrar que

$$(1+x)^{n+1} = \sum_{k=0}^{n+1} \binom{n+1}{k} x^k.$$

Por hipótesis

$$\begin{aligned}(1+x)^{n+1} &= (1+x) \sum_{k=0}^{n} \binom{n}{k} x^k \\ &= \sum_{k=0}^{n} \binom{n}{k} x^k + \sum_{k=0}^{n} \binom{n}{k} x^{k+1}.\end{aligned}$$

Agrupando términos en la última expresión:

$$\begin{aligned}\binom{n}{0} x^0 + \sum_{k=1}^{n} (\binom{n}{k} + \binom{n}{k-1}) x^k + \binom{n}{n} x^{n+1} \\ = \binom{n}{0} x^0 + \sum_{k=1}^{n} \binom{n+1}{k} x^k + \binom{n+1}{n+1} x^{n+1} \\ = \sum_{k=0}^{n+1} \binom{n+1}{k} x^k. \qquad \square\end{aligned}$$

Es importante conocer además algunas fórmulas que relacionan sumas de sucesiones. A menudo usada es la que presenta la siguiente proposición.

Proposición 3 Para $n \in \mathbb{N}$:

$$\sum_{k=1}^{n} k = \frac{1}{2}n(n+1).$$

Demostración. Debe demostrarse que $\sum_{k=1}^{n+1} k = \frac{1}{2}(n+1)(n+2)$.
Base: $1 = \sum_{k=1}^{1} k = \frac{1}{2}1(1+1) = \frac{2}{2}$.
Inducción: Tenemos la hipótesis

$$\sum_{k=1}^{n} k = \frac{1}{2}n(n+1).$$

Así:

$$\begin{aligned}\sum_{k=1}^{n+1} k &= \frac{1}{2}n(n+1) + (n+1) \\ &= \frac{n(n+1)+2(n+1)}{2} = \frac{1}{2}(n^2+3n+2) \\ &= \frac{1}{2}(n+1)(n+2). \qquad \square\end{aligned}$$

Otras sucesiones pueden ser demostradas similarmente a la anterior, como la suma de los cuadrados de los primeros n naturales.

Ejercicio 7

1. Diga por qué
$$\binom{n}{k} = \frac{n!}{(n-k)!k!}.$$

2. Demuestre las identidades:

   a)
   $$\sum_{k=1}^{n} k^2 = \frac{1}{6}n(n+1)(2n+1).$$

   b)
   $$\sum_{k=1}^{n} k \cdot k! = (n+1)! - 1.$$

c)

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

Conviene asimismo utilizar un lenguaje más flexible para las sucesiones. Esto es la *definición por recurrencia* o, más frecuente en el lenguaje computacional, por recursión. Por ejemplo, el factorial puede ser definido en forma recurrente como:

$$n! = \begin{cases} 1 & \text{si } n=0 \\ n\times(n-1)! & \text{si } n>0 \end{cases} .$$

Es decir, el n-ésimo valor está definido en términos de los anteriores. Un caso muy importante es:

$$F_n = \begin{cases} 0 & \text{si } n=0 \\ 1 & \text{si } n=1 \\ F_{n-1}+F_{n-2} & \text{si } n\geq 2 \end{cases} ,$$

la *sucesión de Fibonacci* (S. XIII), una recurrencia de la que pueden derivarse otras expresiones, por ejemplo: $F_n = F_{n+1} - F_{n-1}$ y $F_{n+1} = 2F_{n-1} + F_{n-2}$. Las propiedades de la serie de Fibonacci son realmente sorprendentes, además se tienen diversas aplicaciones, en particular el diseño de algunas estructuras de datos.

A la fecha continua la investigación sobre las propiedades de esta recurrencia. En la siguiente proposición se enuncia una propiedad básica.

Proposición 4 Los términos de la sucesión de Fibonacci satisfacen la siguiente relación:

$$\sum_{i=1}^{n} F_i^2 = F_nF_{n+1}.$$

Demostración. Por inducción sobre n. Base: $\sum_{i=1}^{1} F_i^2 = 1^2$ y $F_1F_2 = 1\cdot 1$. Inducción: Suponemos $\sum_{i=1}^{n} F_i^2 = F_nF_{n+1}$ (HI). Debe demostrarse que $\sum_{i=1}^{n+1} F_i^2 = F_{n+1}F_{n+2}$. Usando HI desarrollamos el lado izquierdo de la última igualdad:

$$\begin{aligned}\sum_{i=1}^{n+1} F_i^2 &= \sum_{i=1}^{n} F_i^2 + F_{n+1}^2 \\ &= F_nF_{n+1} + F_{n+1}^2 \\ &= F_{n+1}(F_{n+1}+F_n) \\ &= F_{n+1}F_{n+2}. \qquad \square\end{aligned}$$

Ya hemos dicho que el PIM puede tener variantes como que el caso base corresponda a un valor diferente a 1. También puede establecerse un predicado $\mathcal{B}(n) = \mathcal{A}(1) \wedge \mathcal{A}(2) \wedge \ldots \wedge \mathcal{A}(n)$, de tal forma que el paso de inducción utilice todos los casos previos y no solamente el anterior, esto es: $\mathcal{A}(i) \rightarrow \mathcal{A}(n+1)$, para $i \leq n$.

En términos de $\mathcal{B}$ queda como originalmente se planteó: $\mathcal{B}(n) \rightarrow \mathcal{B}(n+1)$. Esta forma del PIM se conoce como principio de inducción generalizada del cual el teorema siguiente es un ejemplo.

Uno de los más famosos resultados sobre la sucesión de Fibonacci es la *fórmula de Moivre*. Por medio de esta fórmula es posible calcular el n-ésimo término de la sucesión sin utilizar la definición recurrente.

Aunque podrá observarse que no ahorra de modo alguno el cálculo, debido a las operaciones implicadas y a la necesidad de una precisión tan grande como los valores en juego.

La prueba del teorema siguiente es por inducción generalizada pero esta demostración no puede considerarse dentro de los ejemplos vistos, pues contiene un elemento procedente de los métodos desarrollados para resolver recurrencias, lo cual será presentado en detalle más adelante.

Teorema 4 El n-ésimo término de la sucesión de Fibonacci es:

$$F_n = \frac{\alpha^n - \beta^n}{\sqrt{5}},$$

donde $\alpha = \frac{1}{2}(1+\sqrt{5})$ y $\beta = \frac{1}{2}(1-\sqrt{5})$.

Demostración. Observemos primero que α y β son raíces de la ecuación $1 + x = x^2$. Claramente obtenemos $F_0 = 0$ y $F_1 = 1$. Demostremos que $F_n = \frac{1}{\sqrt{5}}(\alpha^n - \beta^n)$ dado que $F_m = \frac{1}{\sqrt{5}}(\alpha^m - \beta^m)$ para todo $m < n$ (HI). Como siempre, desarrollamos parte de la tesis usando HI:

$$\begin{aligned} F_n &= F_{n-1} + F_{n-2} \\ &= \frac{1}{\sqrt{5}}(\alpha^{n-1} - \beta^{n-1}) + \frac{1}{\sqrt{5}}(\alpha^{n-2} - \beta^{n-2}) \\ &= \frac{1}{\sqrt{5}}(\alpha^{n-1} + \alpha^{n-2} - \beta^{n-1} - \beta^{n-2}) \\ &= \frac{1}{\sqrt{5}}(\alpha^{n-2}(1+\alpha) - \beta^{n-2}(1+\beta)), \end{aligned}$$

puesto que $1 + \alpha = \alpha^2$ y $1 + \beta = \beta^2$, entonces $F_n = \frac{1}{\sqrt{5}}(\alpha^n - \beta^n)$. □

Ejercicio 8

1. Demostrar que la sucesión

$$c_i = \begin{cases} 1 & \text{si } i = 1 \\ 2i + c_{i-1} - 1 & \text{si } i > 1 \end{cases},$$

   satisface que $c_n = n^2$ para $n \geq 1$.

2. Demostrar que los términos de la sucesión de Fibonacci satisfacen las siguientes relaciones:

   a) $\sum_{i=1}^{n} F_i = F_{n+2} - 1$.
   b) $F_1F_2 + F_2F_3 + \ldots + F_{2n-1}F_{2n} = F_{2n}^2$.
   c) $F_{n-1} \cdot F_{n+1} = F_n^2 + (-1)^n$ para $n \geq 2$.

Agregamos, por último, que la inducción puede realizarse con más de una variable, cuando queremos demostrar afirmaciones del estilo $\mathcal{A}(n, m)$, $(n, m \in \mathbb{N})$. Nuevamente, podemos hacer uso de un predicado adicional para convertir lo anterior al planteamiento original del PIM.

La base de la inducción es $\mathcal{A}(1, 1)$. Se consideran ahora dos pasos de inducción, en el primero definimos: $\mathcal{B}(n) \doteq \mathcal{A}(n, 1)$ para probar que $\mathcal{B}(n) \to \mathcal{B}(n+1)$. En el segundo paso el predicado adicional es $\mathcal{C}(m) \doteq (\forall n)\mathcal{A}(n, m)$, y debe demostrarse $\mathcal{C}(m) \to \mathcal{C}(m+1)$.

2.2. Límites

Uno de nuestros objetivos es caracterizar un algoritmo como una función. Es importante, entonces, saber analizar una función y poder compararla con otras, de este modo decidir cuál algoritmo es mejor. La función que representa a un algoritmo dependerá del tamaño de los datos de entrada.

Y aunque sabemos que este tamaño siempre será finito interesa, con fines de clasificación, conocer su *comportamiento asintótico*, es decir su tendencia cuando el tamaño de la entrada de datos crece indefinidamente.

Estos aspectos se expondrán con precisión en la sección 3, por ahora prepararemos las condiciones que permitan llevar a cabo el análisis mencionado.

Para $n > 1$ tenemos que $n < n^2$. Asimismo, $0 < \frac{1}{n} < 1$. Cuando n crece, cada vez $\frac{1}{n}$ es más pequeño pero nunca es cero; éste es llamado su límite.

En general, se espera que siempre que "encerremos" el límite de una función f con una circunferencia de radio ϵ, entonces todos los valores de f estarán dentro de esta circunferencia, excepto quizá algunos cuantos.

Más formalmente, sea $f : \mathbb{N} \to \mathbb{R}$ una función, se dice que f *converge* a ℓ, o que ℓ es el *límite* de f cuando n tiende a infinito sii para toda $\epsilon > 0$, existe $N_\epsilon \in \mathbb{N}$ tal que si $n > N_\epsilon$ entonces $|f(n) - \ell| < \epsilon$. Simbólicamente $\lim_{n\to\infty} f(n) = \ell$.

Alternativamente, f no tiene límite cuando no es posible "encerrar" un valor con una circunferencia de radio arbitrario y que todos los valores de la función estén dentro, excepto algunos.

Esto sucede, por ejemplo, cuando f crece (decrece) indefinidamente, lo que también se simboliza con $\lim_{n\to\infty} f(n) = \infty$ ($\lim_{n\to\infty} f(n) = -\infty$), aunque f más bien diverja. En este caso se define: $f : \mathbb{N} \to \mathbb{R}$ tiende a ∞ sii para todo $M \in \mathbb{N}$ existe $N_M \in \mathbb{N}$ tal que $f(n) > M$ para toda $n > N_M$.

En forma semejante se define que f tiende a $-\infty$. Notemos que la definición de divergencia al infinito puede transformarse en afirmar que: si $f(n) > g(n)$, para $n > n_0$ y $\lim_{n\to\infty} g(n) = \infty$ entonces $\lim_{n\to\infty} f(n) = \infty$. Debe tenerse presente que también se habla de divergencia si f "oscila", ejemplo de ello es $f(n) = (-1)^n$.

Observación 3

1. Para $f(n) = \frac{1}{n}$ intuimos que su límite era 0. Según la definición, debe cumplirse que si $n > N_\epsilon$ entonces $|\frac{1}{n} - 0| = \frac{1}{n} < \epsilon$, para toda ϵ. Debemos entonces elegir N_ϵ dependiendo de ϵ para verificar la implicación. Esto lo hacemos apoyándonos en la desigualdad que deseamos cumplir: $\frac{1}{n} < \epsilon$, equivalentemente $n > \frac{1}{\epsilon}$. Lo último nos conduce a tomar $N_\epsilon = \lfloor \frac{1}{\epsilon} \rfloor$ para lograr nuestro objetivo. Ya que si $n > N_\epsilon$ entonces $n > \frac{1}{\epsilon}$, esto es $\frac{1}{n} < \epsilon$.

2. Conociendo algunos límites básicos es posible calcular otros de manera sencilla. A continuación se enuncian algunas operaciones básicas. Sean $f, g : \mathbb{N} \to \mathbb{R}$ dos funciones tales que $\lim_{n\to\infty} f(n) = a$ y $\lim_{n\to\infty} g(n) = b$, entonces:

   a) $\lim_{n\to\infty} f(n) + g(n) = a + b$.

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

   c) $\lim_{n\to\infty} f(n)/g(n) = a/b$, si $b \neq 0$ y $g(n) \neq 0$ para $n > n_0$.

d) $a \leq b$, si $f(n) \leq g(n)$, para $n > n_0$.

3. Es fácil obtener los límites de muchas funciones, como es el caso de $q(n) = \frac{n^2+2n-3}{3n^2-n+7}$, ya que podemos cambiar la forma de la expresión sin alterarla:
$$q(n) = \frac{1 + \frac{2}{n} - \frac{3}{n^2}}{3 - \frac{1}{n} + \frac{7}{n^2}},$$
luego, como el denominador no es cero, y tanto éste como el numerador están compuestos de funciones que tienen límite, aplicamos las operaciones del inciso anterior para obtener que $\lim_{n\to\infty} q(n) = \frac{1}{3}$.

Ejercicio 9

1. Demuestre con base en la definición que $\lim_{n\to\infty} \frac{n}{n+1} = 1$.

2. Diga por qué se cumplen las siguientes igualdades:

   a) $\lim_{n\to\infty} \frac{1}{n^2} = 0$.

   b) $\lim_{n\to\infty} \frac{1}{n^k} = 0, \quad k \in \mathbb{N}$.

3. Diga cuál es el límite cuando n tiende a infinito de
$$\frac{p_k n^k + p_{k-1} n^{k-1} + \ldots + p_1 n + p_0}{q_j n^j + q_{j-1} n^{j-1} + \ldots + q_1 n + q_0},$$
considerando tres casos: $k > j$, $k = j$ y $k < j$.

4. Demuestre que si una función es creciente entonces diverge.

Nos interesa, en particular, conocer la convergencia de cocientes. Es por ello que analizaremos algunos casos observando la convergencia de las partes de una fracción. Dada $h(n) = f(n)/g(n)$, podemos conjeturar que si $\lim_{n\to\infty} f(n) = a$ y $\lim_{n\to\infty} g(n) = \infty$, entonces h converge a cero.

También, para este caso $\lim_{n\to\infty} g(n)/f(n) = \infty$. Esto se expresa en la siguiente proposición.

Proposición 5 $\lim_{n\to\infty} f(n)/g(n) = 0$ sii $\lim_{n\to\infty} g(n)/f(n) = \infty$.

Obsérvese que puede ser que $\lim_{n\to\infty} f(n) = \infty$ y $\lim_{n\to\infty} g(n) = \infty$, sin embargo $f(n)/g(n)$ converja. Un ejemplo de esto es $\frac{n}{n+1}$. La proposición que sigue es una observación como la mencionada.

Proposición 6 Sea f tal que $\lim_{n\to\infty} f(n+1)/f(n) = a$.

1. Si $|a| < 1$, entonces $\lim_{n\to\infty} f(n) = 0$.

2. Si $a > 1$ y $f(n) > 0$ entonces $\lim_{n\to\infty} f(n) = \infty$.

El teorema de *l'Hôpital*, que se demuestra a continuación, es una herramienta más que ayuda a obtener límites.

Teorema 5 Supongamos que $\lim_{n\to\infty} f(n) = 0$ y $\lim_{n\to\infty} g(n) = 0$, o bien $\lim_{n\to\infty} f(n) = \infty$ y $\lim_{n\to\infty} g(n) = \infty$, y $\lim_{x\to\infty} \bar{f}'(x)/\bar{g}'(x) = \ell$, donde $\bar{f}$ y $\bar{g}$ son las correspondientes extensiones de f y g a los reales ($\bar{f}, \bar{g} : \mathbb{R} \to \mathbb{R}$). Entonces

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

Ejercicio 10

1. Dé dos funciones $f, g : \mathbb{N} \to \mathbb{R}$ tales que f y g no converjan pero $f + g$ si lo haga.

2. Calcule los límites:

   a) $\lim_{n\to\infty} p^n$ con $p \in \mathbb{R}$.

   b) $\lim_{n\to\infty} n^p x^n$ con $p > 0$ y $x \in \mathbb{R}$.

   c) $\lim_{n\to\infty} n^p x^n$ con $p < 0$ y $x \in \mathbb{R}$.

   d) $\lim_{n\to\infty} \frac{\log n}{n}$.

   e) $\lim_{n\to\infty} \frac{\log n}{\sqrt{n}}$.

   f) $\lim_{n\to\infty} \frac{\log n}{n^p}$ con $p > 0$.

   g) $\lim_{n\to\infty} \frac{(\log \log n)^p}{\log n}$, con $p > 0$.

Series

Hemos expresado algunas sumas "largas" haciendo uso del PIM. Consideraremos ahora sumas infinitas. Si tenemos una secuencia dada por los valores de a_i para cada $i \in \mathbb{N}$, se define la n-ésima suma como:

$$s_n = \sum_{i=1}^{n} a_i,$$

entonces la suma de todos los elementos de a_i es $\sum_{i=1}^{\infty} a_i = \lim\limits_{n\to\infty} s_n$.

El resultado que presenta la proposición 7 es muy útil y además fácil de demostrar.

Proposición 7 Si para $i \in \mathbb{N}$, $a_i = (i-1)d$ y $b_i = r^{i-1}$ con $d \in \mathbb{R}$ y $r \neq 1$, entonces

$$\sum_{i=1}^{n} a_i = \frac{d}{2} n(n+1) \quad \text{y} \quad \sum_{i=1}^{n} b_i = \frac{1-r^n}{1-r}.$$

Con base en esta proposición, puede ahora calcularse la suma $\sum_{i=1}^{\infty} b_i$, esto es calcular el límite $\lim\limits_{n\to\infty} \frac{1-r^n}{1-r}$. Aplicando la respuesta de un problema del ejer.10 tenemos que

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

cuando $|r| < 1$. Esto significa que, por ejemplo

$$1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \ldots + \frac{1}{2^i} + \ldots = 2.$$

A pesar de que el resultado encontrado aparenta ser aplicado a la suma $H_n = 1 + \frac{1}{2} + \frac{1}{3} + \ldots \frac{1}{n}$, llamada *serie armónica*, ésta no converge cuando n

tiende a infinito. Consideremos la siguiente agrupación de la suma:

$$\begin{aligned}1 &\geq \frac{1}{2}\\ +\frac{1}{2} &\geq \frac{1}{2}\\ +\frac{1}{3}+\frac{1}{4} &\geq \frac{1}{2}\\ +\frac{1}{5}+\frac{1}{6}+\frac{1}{7}+\frac{1}{8} &\geq \frac{1}{2}\\ &\ldots\\ +\frac{1}{2^i+1}+\ldots+\frac{1}{2^{i+1}} &\geq \frac{1}{2}\\ &\ldots\end{aligned}$$

Lo cual indica que para $n = 2^{i+1}$, $H_n > \sum_{j=0}^{i+1} \frac{1}{2}$, entonces $H_n > \frac{\log n+1}{2}$. Y evidentemente $\lim_{n\to\infty} \frac{\log n+1}{2} = \infty$, luego H_n diverge.

Ejercicio 11

1. Pruebe la prop. 7.
2. Explique por qué $\sum_{i=1}^{\infty} r^{i-1} = \frac{1}{1-r}$.
3. Existe una prueba alternativa de que la serie armónica diverge. Usando integrales se obtiene que $\log(n+1) < H_n \leq 1 + \log(n)$. Proporcione una explicación de este planteamiento.

2.3. Probabilidad

El carácter indeterminístico de los fenómenos naturales no ha impedido el desarrollo de planteamientos que nos lleven a garantizar la obtención de estados sujetos a ciertas premisas.

Por el contrario, la modelación de los fenómenos ha incorporado el indeterminismo como un elemento más de las formulaciones matemáticas. Este componente por tradición se ha basado en la teoría de las probabilidades, de lo cual expondremos sólo lo necesario para nuestro trabajo en el análisis de los algoritmos.

Supongamos que deseamos ejecutar un programa en un sistema con cuatro procesadores, y que es necesario para iniciar la ejecución tener disponibles tres procesadores. En un momento determinado, el sistema tiene una carga específica y es natural preguntarnos en qué momento empezará a ejecutarse el programa.

Podemos simplificar la anterior pregunta diciendo ¿cuál es la probabilidad de que tres procesadores estén libres?

Para dar respuesta a lo anterior debemos precisar algunas ideas como cuáles son los posibles estados del sistema. Se dice que la observación del sistema es un *experimento* que puede tener diferentes *estados* o *resultados*.

Al conjunto de todos los posibles estados se le conoce como *espacio muestral*, $\mathcal{U}$. Denotemos con p_1, p_2, p_3 y p_4 a los procesadores de nuestro ejemplo. Sabemos que cada uno puede estar libre u ocupado. Así, el resultado de observar el sistema puede representarse como una cadena binaria $x_1x_2x_3x_4$, con $x_i = 0$ si p_i está libre y $x_i = 1$ si p_i está ocupado.

Notemos que las cadenas 0100, 1000 y 0001 satisfacen nuestro requisito para ejecutar el programa, pero aún no podemos dar respuesta a nuestra pregunta sobre los procesadores desocupados.

Observemos, en primer lugar, que el requisito no es un estado particular más bien es un conjunto de estados específicos, lo cual es llamado *evento*. Un evento es entonces cualquier subconjunto de $\mathcal{U}$. En nuestro ejemplo, $\mathcal{U} = \{x_1x_2x_3x_4 | x_i = 0, 1\}$, esto es:

$$\begin{array}{cccc} 0000 & 0001 & 0010 & 0011 \\ 0100 & 0101 & 0110 & 0111 \\ 1000 & 1001 & 1010 & 1011 \\ 1100 & 1101 & 1110 & 1111 \end{array}$$

El evento que deseamos describir es “al menos tres procesadores desocupados”. Intuitivamente, necesitamos saber cuál es el tamaño del evento de interés, para referirnos a su probabilidad. Por ejemplo, si nuestro evento es que “un procesador esté ocupado o desocupado”, claramente todos los estados de $\mathcal{U}$ lo satisfacen, en otras palabras, la probabilidad de este evento es segura.

Lo contrario sucede cuando el evento es que “ningún procesador esté libre u ocupado”, su probabilidad es imposible, ningún estado lo satisface. Aunque éstos son dos eventos extremos exhiben la idea de que la probabilidad puede enfocarse desde un punto de vista frecuentista: convenimos que la

probabilidad de un evento sea proporcional al tamaño del subconjunto que lo representa.

Con más generalidad se habla de una medida. Dado un espacio muestral $\mathcal{U}$, se define una *medida de probabilidad* $\Pr : 2^{\mathcal{U}} \rightarrow [0,1]$, que asigna un valor a cada subconjunto de $\mathcal{U}$, y satisface:

1. $\Pr(\mathcal{U}) = 1$.
2. $\Pr(A \cup B) = \Pr(A) + \Pr(B)$, si $A \cap B = \emptyset$.

Observación 4 Con las propiedades de Pr pueden derivarse otras que resultan también útiles para los cálculos como:

1. $\Pr(A^c) = 1 - \Pr(A)$.
2. $\Pr(\emptyset) = 0$.
3. $\Pr(A \cup B) = \Pr(A) + \Pr(B) - \Pr(A \cap B)$.

Cada una de éstas puede ser demostrada con facilidad. La última expresión surge de la identidad de conjuntos $A = A \cap \mathcal{U} = A \cap (B \cup B^c)$.

La medida frecuentista antes aludida se define para un evento, $A \subseteq \mathcal{U}$, como

$$\Pr(A) = \frac{\#A}{\#\mathcal{U}},$$

la fracción de estados favorables respecto de $\mathcal{U}$. Lo cual cumple con las propiedades de medida de probabilidad, por ejemplo $\Pr(\mathcal{U}) = \frac{\#\mathcal{U}}{\#\mathcal{U}} = 1$.

Esto nos conduce como lo habíamos señalado a contar los elementos de los subconjuntos de $\mathcal{U}$. Regresando a nuestro ejemplo, calculemos con esta medida algunos eventos. Sean los eventos $e_1 =$“al menos un procesador libre”, $e_2 =$“tres procesadores libres” y $e_3 =$“al menos tres procesadores libres”. En el primer caso podemos hacer uso del complemento $e_1^c =$“todos ocupados” para el cual sólo hay un estado, por tanto $\Pr(e_1) = 1 - \Pr(e_1^c) = 1 - 1/16$.

El evento e_2 es equivalente a insertar un uno en una cadena de tres ceros para lo cual hay cuatro posibilidaes, luego $\Pr(e_2) = 1/4$. Finalmente, resolvamos el problema originalmente planteado. e_3 puede verse como “tres libres o cuatro libres”, si $e_4 =$“cuatro libres”, tendremos $e_3 = e_2 \cup e_4$ además e_2 y e_4 son excluyentes ($e_2 \cap e_4 = \emptyset$), de este modo $\Pr(e_3) = \Pr(e_2) + \Pr(e_4) = 1/4 + 1/16$.

Supongamos ahora que deseamos conocer la probabilidad de que $e_5 =$"p_1 y p_4 estén libres". Trataremos de descomponer el problema para simplificarlo. Por un lado, que p_1 esté libre tiene probabilidad $1/2$, ya que en las cadenas que representan los estados fijamos una componente a 0 (la primera), las restantes, entonces, tienen ocho posibilidades. Igual podríamos hacer con p_4.

Si nuestros procesadores no siguen una política de maestro-esclavo o algo semejante, podemos decir que el evento dado por $\{0x_2x_3x_4\}$ es independiente del evento $\{x_1x_2x_30\}$: ninguno de ellos de forma alguna implica al otro. Se tiene, también, que A y B son eventos *independientes* sii $\Pr(A \cap B) = \Pr(A)\Pr(B)$. Por tanto $\Pr(e_5) = \Pr(\{0x_2x_3x_4\} \cap \{x_1x_2x_30\}) = \Pr(\{0x_2x_3x_4\})\Pr(\{x_1x_2x_30\}) = \frac{1}{2} \cdot \frac{1}{2}$.

Ejercicio 12

1. Demostrar las igualdades de la obs. 4.
2. Para el problema de sistema de cuatro procesadores, calcular la probabilidad de los siguientes eventos:
   a) Dos procesadores libres.
   b) p_1, p_3 libres y p_4 ocupado.
   c) Dos procesadores libres y uno ocupado.
3. Diga por qué dos eventos excluyentes no son independientes (excepto si alguno tiene probabilidad cero).

Para simplificar la descripción de eventos de interés, en un espacio $\mathcal{U}$, se emplea una función que tome como un valor relacionado con los eventos considerados. Esta función, con dominio $\mathcal{U}$, se llama *variable aleatoria*. Por ejemplo, si nos interesa el número de procesadores libres, X="número de procesadores libres". Recordando los ejemplo anteriores, $e_1 =$"al menos un procesador libre", $\Pr(e_1) = \Pr(X \geq 1)$. Observemos que los posibles valores de X son $\{0, 1, 2, 3, 4\}$, y que $\Pr(X = 0) + \Pr(X = 1) + \Pr(X = 2) + \Pr(X = 3) + \Pr(X = 4) = 1/16 + 4/16 + 6/16 + 4/16 + 1/16 = 1$. Como lo muestra la anterior suma, la probabilidad más alta es para $X = 2$. Esta idea está relacionada con la pregunta ¿Cuál es el valor más probable de X? El valor aludido también se llama *valor esperado* o *esperanza de* X y está definida para una variable aleatoria X como

$$E[X] = \sum_i x_i \Pr(X = x_i).$$

Es importante notar que hemos supuesto, en nuestro ejemplo de los procesadores, que todos los estados son igualmente probables, es decir la probabilidad del estado 0000 es la misma que la de 0101, etc. se dice que los estado tienen una *distribución uniforme*. Esta es una suposición pero no tiene que cumplirse siempre, de hecho nos parecería más adecuado que la probabilidad de que todos los procesadores estuvieran libres fuera menor que la de los demás estados. Será común trabajar con la distribución uniforme. Por ejemplo: sea S, un arreglo de $i-1$ fichas ordenadas por su valor. Dada una nueva ficha C_x, la probabilidad de que C_x ocupe el lugar k en el arreglo S, es $\frac{1}{i}$, esto es puede tenerse $C_x \leq C_1$ o $C_1 < C_x \leq C_2$ o $\ldots$ $C_{i-2} < C_x \leq C_{i-1}$ o $C_{i-1} < C_x$, que son todos los posibles estados y son igualmente probables.

Ejercicio 13

1. Calcule la esperanza de X cuando X representa el número de procesadores ocupados.

2. Para el ejemplo de las fichas antes descrito, sea Y la variable aleatoria que representa el número fichas que deben observarse en orden antes de elegir el lugar que le corresponde a C_x. Calcule $E[Y]$.

Capítulo 3

Introducción a la algoritmia

En este capítulo empezaremos nuestro estudio detallado de algoritmos. Primero definiremos algunos términos: veremos que un *problema*, tal como multiplicar dos enteros positivos, normalmente tendrá infinitos *ejemplares*, tal como multiplicar los enteros 981 y 1234. Un algoritmo debe trabajar correctamente con todo ejemplar del problema que dice resolver.

Después explicamos lo que entendemos por la *eficiencia* de un algoritmo y discutimos formas diferentes para elegir el algoritmo más eficiente en caso de que existan varios algoritmos para solucionar un mismo problema. Veremos que es crucial saber cómo cambia la eficiencia de un algoritmo en la medida que los ejemplares de un problema se hacen más grandes y son (usualmente) por tanto más dificiles de resolver.

También distinguiremos entre la eficiencia promedio de un algoritmo cuando es usado sobre muchos ejemplares de un problema y su eficiencia en el *peor caso* posible. El estimado pesimista del peor caso frecuentemente es apropiado cuando tenemos que estar seguros de resolver un problema en una cantidad limitada de tiempo.

Una vez que hayamos definido lo que entendemos por eficiencia, podemos empezar a investigar los métodos usados para analizar algoritmos. Nuestra línea de ataque es tratar de contar el número de operaciones elementales, tales como las sumas y productos que ejecuta un algoritmo.

No obstante, veremos que los cálculos no son tan directos ya que aún estas operaciones tan comunes se hacen más lentas conforme el tamaño de sus operandos se hace más grande. También tratamos de convenir alguna noción sobre la diferencia práctica entre un algoritmo bueno y uno malo en términos de tiempo de cálculo.

3.1. Problemas y ejemplares

Existen varios algoritmos para resolver el problema de multiplicar dos enteros positivos, un ejemplar de este problema es multiplicar 981 por 1234. No obstante, dichos algoritmos no sólo proveen una forma de multiplicar estos dos números en particular. En efecto, ellos dan una solución general al *problema* de multiplicar dos enteros positivos. Decimos que (981, 1234) es un *ejemplar* de este problema. Multiplicar 789 por 9742 es otro ejemplar del mismo problema y lo podemos expresar como (789, 9742).

Pero multiplicar -12 por 83.7 no es un ejemplar del problema, por dos razones: la primera es que -12 no es positivo y la segunda es que 83.7 no es un entero. (Por supuesto, (-12, 83.7) es un ejemplar de otro problema de multiplicación más general) La mayoría de los problemas interesantes tienen una colección infinita de ejemplares. No obstante, hay excepciones.

Hablando estrictamente, el problema de jugar un juego perfecto de ajedrez tiene sólo un ejemplar, ya que existe solamente una única posición inicial. Más aún existe sólo un número finito de subejemplares (las posiciones intermedias legales). Sin embargo esto no significa que el problema carezca de interés algorítmico.

Un algoritmo debe trabajar correctamente sobre todo ejemplar del problema que resuelve. Para mostrar que un algoritmo es incorrecto, sólo necesitamos encontrar un ejemplar del problema para el cual es incapaz de encontrar una respuesta correcta.

Tal como un teorema propuesto se puede desaprobar con un simple contraejemplo, un algoritmo se puede rechazar sobre la base de un solo resultado incorrecto. Por otro lado, así como puede ser difícil probar un teorema es usualmente difícil probar la corrección de un algoritmo. Para hacer esto posible, cuando se especifica un problema es importante definir su *dominio de definición*, esto es, el conjunto de ejemplares a ser considerados.

Así, los algoritmos mencionados para multiplicar enteros positivos no funcionarán con operandos negativos o fraccionales, al menos no sin alguna modificación. Por supuesto, esto no significa que los algoritmos sean inválidos: porque ejemplares de multiplicación involucrando números negativos o fracciones no están en el dominio de definición que elegimos al enunciar el problema.

Cualquier dispositivo real de cálculo tiene un límite en el tamaño de los ejemplares que puede manejar, ya sea porque los números involucrados son muy grandes o porque desbordamos el espacio de almacenamiento. No

obstante, este límite no se le puede atribuir al algoritmo que elegimos usar.

Máquinas distintas tienen límites distintos y aún programas distintos que implementan el mismo algoritmo sobre la misma máquina pudieran imponer distintas restricciones.

En este curso nos conformaremos con probar que nuestros algoritmos son correctos en abstracto, ignorando las limitaciones prácticas presentes en cualquier programa concreto que los implemente.

3.2. La eficiencia de los algoritmos

Cuando tenemos que resolver un problema, podríamos tener disponibles varios algoritmos. Obviamente nos gustaría elegir el mejor. Esto plantea la pregunta de cómo decidir cuál de esos algoritmos es preferible.

Si sólo tenemos uno o dos ejemplares por resolver, de un problema muy simple, podríamos no preocuparnos demasiado por cuál algoritmo usar: en este caso simplemente podríamos elegir al que sea más fácil de programar, o aquel para el que ya existe un programa, sin preocuparnos sobre sus propiedades teóricas.

Pero si tenemos gran cantidad de ejemplares por resolver o si el problema es muy difícil de resolver, tendriamos que elegir más cuidadosamente.

El *enfoque empírico* (o *a posteriori*) para elegir un algoritmo consiste en programar los algoritmos en competencia y probarlos con ejemplares distintos y la ayuda de una computadora.

El *enfoque teórico* (o *a priori*), el cual favoreceremos en este curso, consiste en determinar matemáticamente la cantidad de recursos que necesita cada algoritmo *como una funcion del tamaño de los ejemplares considerados.*

Los recursos de mayor interés son el tiempo de cálculo y el espacio para almacenamiento, usualmente el primero es el más crítico, en este curso compararemos usualmente algoritmos sobre las bases de su tiempo de ejecución y cuando hablemos de la *eficiencia* de un algoritmo simplemente estaremos diciendo que tan rápido se ejecuta. Sólo ocasionalmente estaremos interesados en los requerimientos de almacenamiento de un algoritmo.

El *tamaño* de un ejemplar formalmente corresponde al número de bits necesarios para representar al ejemplar en una computadora, usando algún esquema de codificación definido precisamente y razonablemente compacto. No obstante, para ser más claro nuestro análisis, usualmente seremos menos formales y cuando usemos la palabra tamaño entenderemos cualquier entero

que de alguna manera mide el número de componentes en un ejemplar.

Por ejemplo, cuando hablamos sobre ordenamiento, usualmente mediremos el tamaño de un ejemplar por el número de artículos que se han de ordenar, ignorando el hecho de que cada uno de estos artículos necesitaría más de un bit para representarse sobre una computadora. Similarmente, cuando hablamos sobre grafos, usualmente mediremos el tamaño de un ejemplar por el número de nodos o vértices (o ambos) involucrados.

Cuando hablemos de problemas que involucren enteros nos apartaremos de esta regla general y algunas veces daremos la eficiencia de nuestros algoritmos en términos del *valor* del ejemplar considerado y no de su tamaño (el cual sería el número de bits necesarios para representar dicho valor en binario).

La ventaja del enfoque teórico es que no depende de la computadora que se use, ni del lenguaje de programación, ni de la habilidad del programador. Ahorra, tanto el tiempo que se gastaría innecesariamente al programar un algoritmo ineficiente, como el tiempo de máquina que se gastaría probándolo.

Más significativo, es que nos permite estudiar la eficiencia de un algoritmo cuando se usa sobre ejemplares de cualquier tamaño. Frecuentemente éste no es el caso con el enfoque empírico, donde consideraciones prácticas podrían forzarnos a probar nuestros algoritmos sólo sobre un número pequeño de ejemplares elegidos arbitrariamente y de tamaño moderado.

Es usual que, un algoritmo descubierto recientemente, empieza a mostrar que tiene un mejor desempeño que sus predecesores sólo cuando es usado sobre ejemplares grandes, éste es un punto muy importante por el cual se prefiere el enfoque a priori.

También es posible analizar algoritmos usando un enfoque *híbrido*, donde la forma de la función que describe la eficiencia del algoritmo se determina teóricamente y luego los parámetros numéricos requeridos se determinan empíricamente para un programa y una máquina particular, al usar este enfoque podemos predecir el tiempo que una implementación real tomará para solucionar un ejemplar mucho mas grande que los usados en las pruebas.

Debemos ser cuidadosos al hacer tales extrapolaciones solamente sobre las bases de un número pequeño de muestras empíricas ignorando todas las consideraciones teóricas. Las predicciones que se hacen sin soporte teórico es probable que sean muy imprecisas, sino es que totalmente equívocas.

Si queremos medir la cantidad de almacenamiento que usa un algoritmo en función del tamaño de los ejemplares, tenemos disponible una unidad natural para nosotros y se llama bit.

Sin importar la máquina que se use la noción de un bit está bien definida. Si por otro lado, como en la mayoría de los casos, queremos medir la eficiencia de un algoritmo en términos del tiempo que toma para producir una respuesta, entonces no hay una elección tan obvia.

Claramente no podríamos expresar esta eficiencia por ejemplo en segundos, ya que no tenemos una computadora estándar a la cual pudieran hacer referencia todas las mediciones.

Una respuesta a este problema esta dada por el *principio de invarianza*, el cual enuncia que dos implementaciones diferentes del mismo algoritmo no diferirán en eficiencia por más de alguna constante multiplicativa.

Si esta constante resulta ser 5, por ejemplo, entonces sabremos que si la primera implementación toma un segundo para resolver ejemplares de cierto tamaño, entonces la segunda implementación (pudiera estar sobre una máquina distinta o escrita en un lenguaje de programación distinto) no tomará más de 5 segundos para resolver los mismos ejemplares.

Más precisamente, si dos implementaciones del mismo algoritmo toman respectivamente $t_1(n)$ y $t_2(n)$ segundos para resolver un ejemplar de tamaño n, entonces siempre existen constantes positivas c y d tales que $t_1(n) \leq ct_2(n)$ y $t_2(n) \leq dt_1(n)$ siempre que n sea suficientemente grande.

En otras palabras, el tiempo de ejecución de cualquiera de las dos implementaciones está acotado por un múltiplo constante del tiempo de ejecución de la otra; por lo tanto es irrelevante a cuál implementación le llamaremos primera y a cuál segunda.

La condición de que n sea suficientemente grande no es realmente necesaria, como veremos posteriormente al analizar la regla del umbral. No obstante, al considerarla podemos encontrar frecuentemente constantes c y d más pequeñas.

Esto es útil si tratamos de calcular buenas cotas sobre el tiempo de ejecución de una implementación cuando conocemos el tiempo de ejecución de la otra.

Este principio no es algo que podamos probar: simplemente enuncia un hecho que puede ser confirmado mediante observación. Más aún tiene una amplia aplicación.

El principio se mantiene cierto sin importar cual computadora se usa para implementar un algoritmo (siempre y cuando la computadora tenga un diseño convencional), sin importar el lenguaje de programación y el compilador empleado, es más, sin importar la habilidad del programador (siempre y cuando él o ella no modifique realmente el algoritmo).

Así un cambio de máquina podría permitirnos resolver un problema 10 o 100 veces más rápido lo cual nos incrementa la rapidez en un factor constante. Por otro lado, un cambio de algoritmo y sólo un cambio de algoritmo podría darnos una mejora que sería más y más marcada conforme aumenta el tamaño de los ejemplares.

Regresando a la pregunta de qué unidad será usada para expresar la eficiencia teórica de un algoritmo, el pricipio de invarianza nos permite decidir que no es necesaria tal unidad. En su lugar, sólo expresamos el tiempo que toma un algoritmo salvo una constante multiplicativa.

Decimos que un algoritmo para algún problema toma un tiempo *en el orden de* $t(n)$, para una función dada t, si existe una constante positiva c y una implementación del algoritmo capaz de resolver todo ejemplar de tamaño n en no más que $ct(n)$ segundos. (Para problemas numéricos, como señalamos anteriormente, n algunas veces puede ser el valor más que el tamaño del ejemplar).

El uso de segundos en esta definición es obviamente arbitrario: sólo necesitamos cambiar la constante para acotar el tiempo por $at(n)$ años o $bt(n)$ microsegundos. Por el principio de invarianza, si cualquiera de las implementaciones del algoritmo tiene la propiedad requerida, entonces también la tendrán todos los demás, no obstante la constante multiplicativa cambiaría de una implementación a la otra.

En la siguiente sección daremos un tratamiento más riguroso de este concepto importante conocido como la *notación asintótica.* A partir de la definición formal será claro porque decimos *en el orden de* en lugar del más usual *del orden de.*

Ciertos ordenes ocurren tan frecuentemente que vale la pena darles un nombre. Por ejemplo, suponga que el tiempo que toma un algoritmo para resolver un ejemplar de tamaño n nunca es mayor a cn segundos donde c es una constante apropiada. Entonces decimos que el algoritmo toma un tiempo en el orden de n, o más simplemente que toma tiempo *lineal.*

En este caso también hablamos de un *algoritmo lineal.* Si un algoritmo nunca toma más de cn^2 segundos para resolver un ejemplar de tamaño n, entonces decimos que toma tiempo en el orden de n^2, o tiempo *cuadrático*, y le llamaremos un *algoritmo cuadrático.*

Similarmente un algoritmo es *cúbico, polinomial o exponencial* si toma un tiempo en el orden de n^3, n^k o c^n, respectivamente, donde k y c son constantes adecuadas.

No caiga en la trampa de olvidar completamente a las *constantes ocultas*,

como son llamadas las constantes multiplicativas usadas en estas definiciones. Comúnmente ignoraremos los valores exactos de estas constantes y asumiremos que todas ellas tienen el mismo orden de magnitud.

Esto nos permitirá decir, por ejemplo, que un algoritmo lineal es más rápido que uno cuadrático sin preocuparnos si nuestra afirmación es verdadera en todos los casos. Sin embargo, algunas veces es necesario ser más cuidadoso.

Considere por ejemplo dos algoritmos cuyas implementaciones sobre una máquina dada toman respectivamente n^2 días y n^3 segundos para resolver un ejemplar de tamaño n, solamente para **¡ejemplares que requieren más de 20 millones de años en ser resueltos!** es que el algoritmo cuadrático llega a ser más rápido que el algoritmo cúbico.

Desde un punto de vista teórico, el primero es *asintóticamente* mejor que el segundo; es decir, su desempeño es mejor sobre todos los ejemplares suficientemente grandes. Desde un punto de vista práctico ciertamente preferiremos al algoritmo cúbico.

No obstante el algoritmo cuadrático puede ser asintóticamente mejor, su constante oculta es tan grande que lo descarta para considerarlo sobre ejemplares de tamaño normal.

3.3. Análisis del peor caso y del caso promedio

El tiempo que consume un algoritmo o el espacio de almacenamiento que usa, puede variar considerablemente entre dos ejemplares distintos del mismo tamaño. Para ilustrar esto, considere dos algoritmos de ordenamiento (ascendente) elementales: ordenamiento por *inserción* y ordenamiento por *selección*.

```
procedure INSERT(T[1...n])
1  for i ← 2 to n
2  do x ← T[i]
3      j ← i − 1
4      while j > 0  and  x < T[j]
5      do T[j + 1] ← T[j]
6          j ← j − 1
7      T[j + 1] ← x
```

```
procedure SELECT(T[1...n])
1  for i ← 1 to n − 1
2  do minj ← i
3      minx ← T[i]
4      for j ← i + 1 to n
5      do if T[j] < minx
6             then minj ← j
7                  minx ← T[j]
8      T[minj] ← T[i]
9      T[i] ← minx
```

Ejercicio 14

1. Simule la operación de los algoritmos de ordenamiento sobre algunos arreglos pequeños, para asegurarse que entiende cómo trabajan.

2. Simule los algoritmos de ordenamiento por inserción y por selección sobre los siguientes dos arreglos: $U = [1, 2, 3, 4, 5, 6]$ y $V = [6, 5, 4, 3, 2, 1]$ ¿Sobre cuál de los arreglos U o V se ejecuta más rápido ordenamiento por inserción?. La misma pregunta pero ahora considerando ordenamiento por selección. Justifique sus respuestas.

3. Suponga que trata de "ordenar" el arreglo $W = [1, 1, 1, 1, 1, 1]$ cuyos elementos son iguales, usando: (a) ordenamiento por inserción y (b) ordenamiento por selección. ¿Cómo se compara esto a ordenar los arreglos U y V del ejercicio 14.2?

El ciclo principal en ordenaminto por inserción, busca sucesivamente cada elemento del arreglo desde el segundo hasta el $n-$ésimo y lo inserta apropiadamente entre sus predecesores en el arreglo. Ordenamiento por selección trabaja tomando al elemento más pequeño en el arreglo y llevándolo al inicio; luego toma al siguiente más pequeño y lo pone en la segunda posición en el arreglo y así sucesivamente.

Sean U y V dos arreglos de n elementos, tales que U está ordenado ascendentemente y V está ordenado descendentemente. Si resolvió bien el ejercicio 14.2 habrá notado que ambos algoritmos consumen más tiempo sobre V que sobre U. En efecto, el arreglo V representa el peor caso posible para estos dos algoritmos: ningún arreglo de n elementos requiere más trabajo.

Sin embargo, el tiempo requerido por el algoritmo de ordenamiento por selección no es muy sensible al orden original del arreglo que ha de ordenarse: la prueba "**if** $T[j] < minx$"se ejecuta exactamente el mismo número de veces en cada caso. La variación en tiempo de ejecución, es debida solamente al número de veces que son ejecutadas las asignaciones en la parte **then** de dicha prueba.

Si se programa este algoritmo y se ejecuta sobre una máquina, encontrará que el tiempo requerido para ordenar un cierto número de elementos no variará más del 15 % cualquiera que sea el orden inicial de los elmentos a ordenar. Como se mostrará en la sección 5.2, el tiempo requerido por $select(T)$ es cuádratico, sin importar el orden inicial de los elementos.

La situación es diferente si comparamos los tiempos que toma el algoritmo de ordenamiento por inserción sobre los dos arreglos. Ya que la condición que controla el ciclo **while** es falsa siempre desde el principio, $insert(U)$ es muy rápido y consume tiempo lineal.

Por otro lado, $insert(V)$ consume tiempo cudrático porque el ciclo **while** se ejecuta $i - 1$ veces para cada valor de i, nuevamente refiérase a la sección 5.2 para más detalle. La variación en tiempo entre estos dos ejemplares es por lo tanto considerable. Más aún, esta variación aumenta conforme aumenta el número de elementos a ordenar.

En alguna implementación del algoritmo de ordenamiento por inserción, se encontró que consume menos de un quinto de segundo para ordenar un arreglo de 5000 elementos que originalmente estaban en orden ascendente, pero consumió tres y medio minutos en ordenar un arreglo con la misma cantidad de elementos, pero inicialmente en orden descendente, es decir, mil veces más.

Si pueden ocurrir variaciones tan grandes, ¿Cómo podemos hablar del tiempo que consume un algoritmo solamente en términos del tamaño del ejemplar a resolver? La respuesta es que usualmente consideraremos el peor caso del algoritmo, esto es, para cada tamaño de ejemplar sólo consideraremos aquellos ejemplares sobre los que el algoritmo requiere más tiempo.

Esto es por lo que dijimos en la sección anterior que un algoritmo debe ser capaz de resolver todo ejemplar de tamaño n en no más de $ct(n)$ segundos, para una constante apropiada c que depende de la implementación, si decimos que un algoritmo se ejecuta en un tiempo en el orden de $t(n)$: implicitamente tenemos en mente el peor caso.

El análisis del peor caso es apropiado para un algoritmo cuyo tiempo de respuesta es crítico. Por ejemplo, si se trata de controlar una planta de energía

nuclear, es crucial conocer un límite superior sobre el tiempo de respuesta del sistema, no importando el ejemplar particular que se resolverá.

Por otro lado, si un algoritmo será usado varias veces sobre muchos ejemplares distintos, pudiera ser más importante conocer el tiempo de ejecución *promedio* sobre instancias de tamaño n. Vimos que el tiempo que consume el algoritmo de ordenamiento por inserción varía entre n y n^2.

Si podemos calcular el tiempo promedio que consume el algoritmo, sobre las $n!$ formas distintas de ordenar inicialmente n elementos distintos, tendremos una idea del tiempo probable que consumirá para ordenar un arreglo en orden aleatorio.

Si $n!$ permutaciones iniciales son igualmente probables, entonces su tiempo promedio está también el orden de n^2. Así, el algoritmo de ordenamiento por inserción consume tiempo cuadrático tanto en el peor caso, como en el caso promedio, no obstante que para algunos ejemplares puede ser mucho más rápido.

Existe otro algoritmo de ordenamiento (quicksort) que también consume tiempo cuadrático en el peor de los casos, pero que sólo requiere un tiempo en el orden de $n \log n$ en el caso promedio.

Aunque este algoritmo tiene un peor caso malo, ya que el desempeño cuadrático es malo para un algoritmo de ordenamiento, en su caso promedio probablemente es el algoritmo conocido más rápido, al menos de entre los algoritmos que usan la técnica de ordenamiento en el lugar, es decir, aquellos que no requieren espacio de almacenamiento adicional.

Usualmente es más difícil analizar el comportamiento promedio de un algoritmo que analizar su comportamiento en el peor caso. Peor aún, el análisis del comportamiento promedio puede ser engañoso, cuando los ejemplares a resolver no son elegidos aleatoriamente, en el momento que el algoritmo es usado en la práctica.

Por ejemplo, hemos enunciado que ordenamiento por inserción toma tiempo cuadrático en promedio, cuando todos los $n!$ ordenamientos iniciales posibles son igualmente probables.

Pero en muchas aplicaciones esta condición puede no ser real. Por ejemplo, si un programa de ordenamiento es usado para actualizar un archivo, podría la mayoría de las veces enfrentarse con datos que están muy cerca del orden deseado, es decir, con sólo unos pocos elementos fuera de su lugar.

En este caso su comportamiento promedio calculado, bajo la hipótesis de que los ejemplares son elegidos aleatoriamente, será una guía pobre con respecto a su desempeño real.

Por lo tanto, para que sea útil el análisis del comportamiento promedio de un algoritmo, se requiere tener algún conocimiento a priori de la distribución de los ejemplares que serán resueltos. Esto normalmente es un requerimiento no realista.

Especialmente cuando un algoritmo se usa como un procedimiento interno en algún algoritmo más complejo, podría ser impráctico estimar cuales ejemplares son más probables de encontrar y cuales sólo ocurrirán rara vez. Afortunadamente existen algoritmos que pueden "desordenar"los datos para que queden con cierta distribución.

En lo que sigue estaremos interesados sólo en el análisis del peor caso al menos que enunciemos lo contrario.

3.4. Operación elemental

Una *operación elemental* es aquella cuyo tiempo de ejecución se puede acotar superiormente por una constante, que sólo depende de la implementación que en particular use una máquina, un lenguaje de programación y así sucesivamente. Así la constante no depende ni del tamaño, ni de los otros parámetros del ejemplar que se está considerando.

Dado que nos interesa el tiempo de ejecución de algoritmos que están por debajo de una constante multiplicativa, lo único que importa en el análisis es el número de operaciones elementales ejecutadas, no el tiempo exacto que requiere cada uno de ellos.

Por ejemplo, suponga que cuando analizamos un algoritmo, encontramos que para resolver un ejemplar de un cierto tamaño se requieren ejecutar a adiciones, m multiplicaciones y s instrucciones de asignamiento.

Suponga que también sabemos que una adición nunca consume más de t_a microsegundos, una multiplicación nunca más de t_m microsegundos y un asignamiento nunca más de t_s microsegundos, donde t_a, t_m y t_s son constantes que dependen de la máquina usada.

Por tanto, adición, multiplicación y asignamiento pueden ser consideradas operaciones elementales. El tiempo total t que requiere nuestro algoritmo puede acotarse mediante

$$\begin{aligned} t &\leq at_a + mt_m + st_s \\ &\leq \text{máx}(t_a, t_m, t_s) \times (a + m + s), \end{aligned}$$

esto es, t está acotado por un múltiplo constante del número de operaciones elementales que son ejecutadas.

Ya que no es importante el tiempo requerido por cada operación elemental, simplificaremos diciendo que las operaciones elementales se pueden ejecutar con *costo unitario*.

En la descripción de un algoritmo, una sola línea de código puede corresponder a un número variable de operaciones elementales. Por ejemplo, si T es un arreglo de n elementos ($n > 0$), el tiempo requerido para calcular

$$x \leftarrow \text{mín}\{T[i] | 1 \leq i \leq n\}$$

incrementa con n ya que es una abreviación para

```
function MIN(T[1...n])
1  x ← T[1]
2  for i ← 2 to n
3  do if T[i] < x
4        then x ← T[i]
5  return x
```

Similarmente, algunas operaciones matemáticas son tan complejas que no se pueden considerar elementales. Si nos permitiéramos contar con costo unitario las operaciones calcular un factorial y prueba por divisibilidad, sin considerar el tamaño de los operandos, entonces por el teorema de Wilson (el cual enuncia que el entero n divide a $(n-1)! + 1$ si y sólo si n es primo para todo $n > 1$) podríamos probar la primalidad de un entero con eficiencia sorprendente.

```
function WILSON(n)
1  if n divide exactamente a (n − 1)! + 1
2    then return true
3    else return false
```

El ejemplo al inicio de esta subsección sugiere que podemos considerar como de costo unitario a las operaciones adición y multiplicación, ya que se asume que el tiempo requerido por estas operaciones se puede acotar mediante una constante. En teoría, no obstante, estas operaciones no son elementales

ya que el tiempo necesario para ejecutarlas incrementa con el tamaño de los operandos.

En la práctica, por otro lado, pudiera ser adecuado considerarlas como operaciones elementales, siempre y cuando los operandos involucrados sean de un tamaño razonable en los ejemplares que esperamos encontrar. Dos ejemplos ilustrarán lo que queremos decir.

```
function SUM(n)
1  sum ← 0
2  for i ← 1 to n
3  do sum ← sum + i
4  return sum
```

```
function FIBONACCI(n)
1  i ← 1
2  j ← 0
3  for k ← 1 to n
4  do j ← i + j
5      i ← j − i
6  return j
```

En el algoritmo llamado SUM el valor de *sum* permanece razonable para todos los ejemplares que en la práctica se esperan. Si estamos usando una máquina con palabras de 32 bits, todas las adiciones pueden ser ejecutadas directamente siempre y cuando n no sea mayor a 65535. No obstante, en teoría, el algoritmo debe trabajar para *todos* los valores posibles de n.

Ninguna máquina real en efecto puede ejecutar estas adiciones con costo unitario si n es elegido suficientemente grande. El análisis del algoritmo por tanto debe depender del dominio previsto de la aplicación (no del dominio del problema).

La situación es diferente en el caso de FIBONACCI. Aquí es suficiente tomar $n = 47$ para que la última adición "$j \leftarrow i + j$" cause desbordamiento aritmético sobre una máquina de 32 bits. Para guardar el resultado que corresponde a $n = 65535$ necesitaríamos 45496 bits, o más de 1420 palabras de computadora. Por la tanto, como un aspecto práctico no es realista considerar que estas operaciones pueden ejecutarse con costo unitario.

Más bien, debemos atribuirles un costo proporcional a la longitud de los operandos involucrados. En la subsección 5.1.2 se mostrará que este algoritmo consume tiempo cuadrático, aunque a primera vista su tiempo de ejecución parece ser lineal.

En el caso de la multiplicación aún podría ser razonable considerarla una operación elemental para operandos suficientemente pequeños. Desafortunadamente, es más fácil producir operandos grandes por multiplicaciones

repetidas que por adiciones, por lo tanto es muy importante asegurar que las operaciones aritméticas no se desborden.

Aún más, el tiempo requerido para efectuar una adición crece linealmente con respecto al tamaño de los operandos, pero el tiempo requerido para efectuar una multiplicación se cree que crece más rápido que eso.

Un problema similar puede ocurrir cuando analizamos algoritmos que involucran números reales si la precisión requerida aumenta con el tamaño de los ejemplares a resolver. Un ejemplo típico de este fenómeno es cuando se usa la fórmula de de Moivre para calcular los valores de la secuencia de fibonacci.

Esta fórmula nos dice que f_n el $n-$ésimo término en la secuencia, es aproximadamente igual a $\phi^n/\sqrt{5}$ donde $\phi = (1+\sqrt{5})/2$ es la *proporción dorada*. La aproximación es tan suficientemente buena, que en principio podemos obtener el valor exacto de f_n tomando simplemente el entero más

cercano. No obstante, vimos que se necesitan alrededor de 45496 bits para representar exactamente a f_{65535}. Esto significa que tendríamos que calcular la aproximación con el mismo grado de exactitud que el requerido para obtener al número exacto. La aritmética de punto flotante de precisión simple o doble, usando una o dos palabras de computadora, ciertamente no sería suficientemente exacta.

En la mayoría de las situaciones prácticas, no obstante, el uso de aritmética de punto flotante de precisión simple o doble ha sido satisfactoria, a pesar de la inevitable pérdida de precisión. Cuando esto es así, es razonable considerar dichas operaciones como de costo unitario.

Sumarizando, decidir cuando una instrucción tan aparentemente inofensiva como "$j \leftarrow i + j$" puede ser considerada como elemental o no, es algo que siempre requiere de nuestro juicio.

En lo que sigue, consideraremos como operaciones elementales a las adiciones, sustracciones, multiplicaciones, divisiones, operaciones módulo, operaciones Booleanas, comparaciones y asignamientos; y por lo tanto que pueden ejecutarse con costo unitario, al menos que explícitamente enunciemos lo contrario.

Ejercicio 15

1. Use la fórmula de de Moivre para f_n y demuestre que f_n es el entero más cercano a $\phi^n/\sqrt{5}$ para todo $n \geq 1$.

2. Sea $g(n)$ el número de maneras de escribir una cadena de n ceros y unos tal que nunca hay dos ceros consecutivos. Por ejemplo, cuando $n = 1$ las cadenas posibles son 0 y 1, así $g(1) = 2$; cuando $n = 2$ las posibles cadenas son $01, 11$ y 10, así $g(2) = 3$; cuando $n = 3$ las cadenas posibles son $010, 011, 101, 110$ y 111, así $g(3) = 5$. Demuestre que $g(n) = f_{n+2}$.

3. ¿Es razonable, como asunto práctico, considerar a la división como una operación elemental: (a) siempre, (b) algunas veces, (c) nunca? Justifique su respuesta. Si lo considera necesario, puede tratar separadamente la división de enteros y la división de números reales.

4. En la subsección 3.4, vimos que el teorema de Wilson podría ser usado para probar la primalidad de cualquier número en tiempo constante, si los factoriales y las pruebas de divisibilidad de enteros se contarán de costo unitario, sin importar el tamaño de los números involucrados. Claramente esto no sería razonable. Use el teorema de Wilson junto con el teorema binomial de Newton para diseñar un algoritmo capaz de decidir en un tiempo en el orden de $\log n$ cuando o no un entero n es primo, considerando que son contadas con costo unitario las adiciones, multiplicaciones, y pruebas por divisibilidad de enteros, pero no así los factoriales y exponenciales. Lo más importante de este ejercicio no es proporcionar un algoritmo útil, sino demostrar que no es razonable, en general, considerar a las multiplicaciones como operaciones elementales.

3.5. La importancia de la eficiencia

Ya que las computadoras se hacen cada vez más y más rápidas, pudiera parecer que no vale la pena gastar nuestro tiempo tratando de diseñar algoritmos más eficientes. ¿No sería más fácil esperar a la siguiente generación de computadoras? Los principios establecidos en las subsecciones anteriores muestra que esto no es verdad.

Suponga, para ilustrar el argumento, que para solucionar un problema particular tiene disponible un algoritmo exponencial y una computadora que puede correr este algoritmo sobre ejemplares de tamaño n en $10^{-4} \times 2^n$ segundos. Así el programa puede resolver ejemplares de tamaño 10 en $10^{-4} \times 2^{10}$ segundos o alrededor de un décimo de segundo.

Resolver un ejemplar de tamaño 20 consumiría alrededor de mil veces más o aproximadamente dos minutos. Para resolver un ejemplar de tamaño 30

tomaría mil veces más, por lo cual no serían suficientes los cálculos de un día entero. Suponiendo que se pudiera ejecutar una computadora sin interrupción y sin errores por una año completo, sólo se podría resolver un ejemplar de tamaño 38.

Suponga que necesita resolver ejemplares más grandes que esto y que tiene el dinero suficiente para comprar una computadora nueva un ciento de veces más rápida que la primera. Con el mismo algoritmo ahora se puede resolver un ejemplar de tamaño n en sólo $10^{-6} \times 2^n$ segundos.

Podría sentir que ha desperdiciado su dinero cuando se dé cuenta que, si corre la nueva máquina por un año entero, ni siquiera podrá resolver un ejemplar de tamaño 45. En general, si antes se podía resolver ejemplares de tamaño n en un tiempo dado, con la nueva computadora en el mejor de los casos resolverá ejemplares de tamaño $n + \log 100$ o alrededor de $n + 7$, en el mismo tiempo.

Suponga que mejor decide invertir en algoritmia y que habiendo gastado la misma cantidad de dinero, ha conseguido un algoritmo cúbico para solucionar el problema. Imagine por ejemplo, que usando la primera máquina con el nuevo algoritmo se pude resolver un ejemplar de tamaño n en $10^{-2} \times n^3$ segundos.

Así, para resolver un ejemplar de tamaño 10 consumirá 10 segundos y un ejemplar de tamaño 20 requerirá entre uno y dos minutos. Pero ahora un ejemplar de tamaño 30 se puede resolver en cuatro y medio minutos y en un día se pueden resolver ejemplares de tamaño mayor a 200; con los cálculos de un año se pueden resolver ejemplares de tamaño cercano a 1500.

El nuevo algoritmo no sólo ofrece una mejora mucho mayor que la compra del nuevo hardware, suponiendo que puede afrontar ambos gastos, haría mucho más productiva la compra del nuevo hardware. Si puede hacer uso del nuevo algoritmo y de la máquina 100 veces más rápida, entonces podrá resolver ejemplares cuatro o cinco veces más grandes que con sólo el nuevo algoritmo, en la misma cantidad de tiempo, el factor exacto es $\sqrt[3]{100}$.

Compare esto a la situación con el viejo algoritmo, donde se le podía *sumar* 7 al tamaño del ejemplar, aqui se puede *multiplicar* el tamaño de los ejemplares por cuatro o cinco. No obstante, el nuevo algoritmo no debe ser usado descuidadamente sobre todos los ejemplares del problema, en particular sobre los muy pequeños.

Vimos que sobre la primera máquina el algoritmo nuevo consume 10 segundos para resolver un ejemplar de tamaño 10, lo cual es un ciento de veces más lento que el algoritmo exponencial. El nuevo algoritmo es más rápido

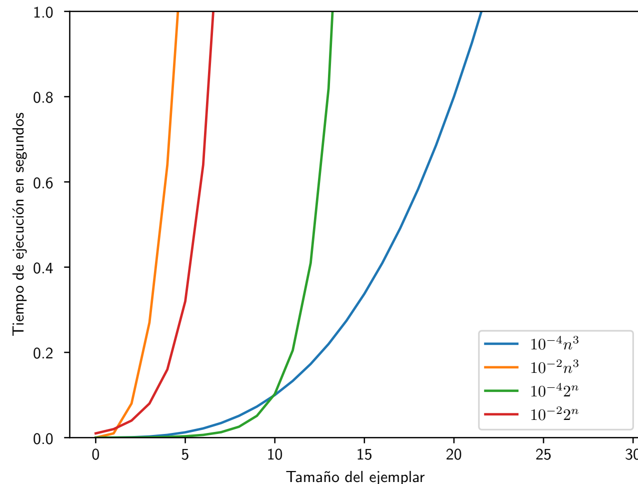

Figura 3.1: Algoritmia contra hardware.

sólo para ejemplares de tamaño 20 o mayores, ver Figura 3.1.

Naturalmente, es posible combinar ambos algoritmos en un tercero que verifique el tamaño del ejemplar que hay que resolver, antes de decidir que método usar.

Ejercicio 16

1. Algunas veces se afirma que el hardware de hoy en día es tan barato y la mano de obra tan cara, que no vale la pena gastar el tiempo de un programador para *"rasurarle"* unos pocos segundos al tiempo de ejecución de un programa. ¿Esto significa que la algoritmia está destinada a ser únicamente una ocupación teórica con fines de formalidad, sin aplicación práctica? Justifique su respuesta.

2. Usando la técnica llamada "memoría virtual", es posible liberar al programador de la mayoría de las consideraciones sobre el tamaño real de almacenamiento disponible en la máquina. ¿Significa esto que la cantidad de almacenamiento usado por un algoritmo nunca es de interés en la práctica? Justifique su respuesta.

3. Suponga que mide el desempeño de un programa, quizas usando alguna clase de traza en tiempo de ejecución, luego optimiza las partes muy usadas del código. No obstante se asegura de no cambiar el algoritmo subyacente. ¿Qué esperaría obtener: (a) una ganancia en eficiencia mediante un factor constante, o (b) una ganancia en eficiencia que es proporcionalmente mayor conforme el tamaño del ejemplar aumenta? Justifique su respuesta.

4. Un algoritmo de ordenamiento consume 1 segundo para ordenar 1000 artículos en su computadora local.¿Cuánto consumiría para ordenar 10000 artículos si (a) cree que el algoritmo consume tiempo aproximadamente proporcional a n^2 y (b) cree que el algoritmo consume tiempo aproximadamente proporcional a $n \log n$?

5. Dos algoritmos consumen respectivamente n^2 días y n^3 segundos para resolver un ejemplar de tamaño n. Muestre que sólo sobre ejemplares que requieren más de 20 millones de años en ser resueltos, el algoritmo cuadrático supera al algoritmo cúbico.

6. Dos algoritmos consumen respectivamente n^2 días y 2^n segundos para resolver un ejemplar de tamaño n. ¿Cuál es el tamaño del ejemplar más pequeño para el cual el primer algoritmo supera al segundo? ¿Aproximadamente cuánto tiempo consumiría dicho ejemplar en ser resuelto?

7. Cierto algoritmo consume $10^{-4} \times 2^n$ segundos para resolver un ejemplar de tamaño n. Muestre que en un año sólo podría resolver un ejemplar de tamaño 38. ¿Qué tamaño de ejemplar podría ser resulto en un año con una máquina un ciento de veces más rápida? Un segundo algoritmo consume $10^{-2} \times n^3$ segundos para resolver ejemplares de tamaño n. ¿Qué tamaño de ejemplar puede resolver en un año? ¿Qué tamaño de ejemplar puede resolver en un año con una máquina un ciento de veces más rápida? Muestre que el segundo algoritmo es más lento que el primero para ejemplares de tamaño menor que 20.

3.6. Un algoritmo lineal para ordenamiento

Se puede demostrar que ningún algoritmo de ordenamiento, que trabaje comparando los elementos a ser ordenados puede ser más rápido que el

orden de $n \log n$. No obstante, se pueden encontrar otros algoritmos de ordenamiento más eficientes para casos *muy especiales*. Suponga por ejemplo que los elementos a ordenar son enteros que se sabe están en el rango de 1 a 10000. Entonces el siguiente algoritmo puede usarse.

```
procedure CASILLAS(T[1...n])
 1  array U[1...10000]
 2  for k ← 1 to 10000
 3  do U[k] ← 0
 4  for i ← 1 to n
 5  do k ← T[i]
 6      U[k] ← U[k] + 1
 7  i ← 0
 8  for k ← 1 to 10000
 9  do while U[k] ≠ 0
10      do i ← i + 1
11          T[i] ← k
12          U[k] ← U[k] − 1
```

Aquí U es un arreglo de "casillas" para los elementos que se van a ordenar. Debe haber una casilla para cada elemento posible que se pueda encontrar en T. El primer ciclo limpia las casillas, el segundo pone cada elemento de T en el lugar adecuado y el tercero los saca de U para volverlos a poner en T en orden ascendente.

Es fácil mostrar que este algoritmo y sus variantes consumen un tiempo en el orden de n. (La constante multiplicativa oculta depende del número que acota los valores de los elementos a ordenar, en este caso 10000.)

Cuando este algoritmo se aplica vence a cualquier otro algoritmo que trabaje con comparaciones, por otro lado el requerimiento de que debe poder usar una casilla por cada elemento posible, implica que es aplicable mucho menos veces en comparación a los otros algortimos de ordenamiento.

Ejercicio 17

1. Se le pide ordenar un archivo que contiene enteros entre 0 y 999999. Pero no puede usar un millón de casillas, así que decide usar casillas numeradas desde 0 hasta 999. Empieza a ordenar poniendo cada entero en la casilla que le corresponde de acuerdo a sus primeros tres dígitos.

Después usa ordenamiento por inserción mil veces, para ordenar separadamente los contenidos de cada casilla y finalmente vacía las casillas para obtener una secuencia totalmente ordenada. ¿Esperaría que esta técnica sea más rápida, más lenta o igual, con respecto a usar ordenamiento por inserción sobre toda la secuencia (a) en promedio y (b) en el caso peor? Justifique sus respuestas.

2. Muestre que ordenamiento por casillas consume un tiempo en el orden de n para ordenar n elementos que están dentro de las cotas.

3.7. Especificación completa de un algoritmo

Al inicio dijimos que la ejecución de un algoritmo normalmente no debe involucrar decisiones subjetivas, ni debe requerir el uso de la intuición o creatividad; posteriormente dijimos que casi siempre debemos estar contentos si demostramos que nuestros algoritmos son correctos en abstracto, ignorando las limitaciones prácticas; después, propusimos considerar que la mayoría de las operaciones aritméticas son elementales al menos que explícitamente aclaremos lo contrario.

Todo esto está muy bien, pero ¿qué debemos hacer si consideraciones prácticas nos obligan a abandonar esta posición conveniente y tenemos que tomar en cuenta las limitaciones de las máquinas disponibles?

Por ejemplo, cualquier algoritmo que calcule el valor exacto de f_{100} será forzado a considerar que ciertas operaciones aritméticas, ciertamente la adición y también posiblemente la multiplicación no son operaciones elementales (recuerde que f_{100} es un número con 21 dígitos decimales).

Más probablemente esto se tomará en cuenta para usar un paquete de programas que permitan operaciones aritméticas sobre enteros muy largos. Si no especificamos exactamente de qué manera el paquete debe implementar aritmética de precisión multiple, entonces la elección de qué método usar podría ser considerada una decisión subjetiva y el algoritmo propuesto estaría especificado incompletamente. ¿Esto importa?

La respuesta es que en ciertos casos sí importa. Posteriormente veremos algoritmos cuyo desempeño en verdad depende del método que se use para multiplicar enteros largos.

Para tales algoritmos (y formalmente hablando para *cualquier* algoritmo) no es suficiente escribir simplemente una instrucción como $x \leftarrow y \times z$,

dejándole al lector elegir cualquier técnica que tenga a la mano para implementar esta multiplicación. Para especificar completamente al algoritmo, también debemos especificar como serán implementadas las operaciones aritméticas necesarias.

No obstante, para hacernos la vida más fácil, continuaremos usando la palabra algoritmo para ciertas descripciones incompletas de este tipo. Los detalles serán completados cuando el análisis así lo requiera.

Capítulo 4

Clases de funciones

Una concluisión que hemos obtenido en la sección anterior es que el mejor algoritmo es aquél que resuelve el problema planteado de acuerdo con la situación. La respuesta para el caso general, para todas las situaciones, la hemos dado generalizando, también, el tamaño del ejemplar.

Es por ello que hablamos del comportamiento asintótico de un algoritmo. Vamos a ver en esta sección que es posible manipular los comportamientos de los algoritmos para llevar a cabo análisis más detallados de ellos. Justificaremos, por ejemplo, el uso de la "escala de funciones" respecto de su comportamiento (contante, logarítmica, lineal, cuadrática, cúbica etc.).

Particularmente, nos interesa saber qué tanto conviene mejorar un algoritmo si cambiamos una parte él. Aunque esta pregunta está relacionada además con el diseño, requeriremos de una serie de reglas para determinar con rapidez cuestiones como la anterior.

4.1. Cota superior

Formalizaremos en primer lugar la noción usada con anterioridad para referirnos a que un algoritmo es mejor que otro o bien que un algortitmo a lo más se comporta tan lento como otro. Por supuesto, haremos estas convenciones retomando la idea de que un algoritmo se caracteriza por una *función de comportamiento* dependiente del tamaño del ejemplar cuyo valor es el tiempo que requiere el algoritmo para realizar su tarea.

Conviene, asimismo, señalar que todas estas funciones son no negativas pues siempre un algoritmo tarda al menos cero unidades, lo cual supondremos

sin indicar.

Desde que se planteó la necesidad de comparar funciones de comportamiento, digamos f y g, fue mencionado que para afirmar que el algoritmo de g no podía ser más lento que el de f, debía cumplirse que g no rebasara a un múltiplo de f para ejemplares de tamaño mayores o iguales que N.

Esto significa, que un algoritmo, con función g, será mejor o a lo más igual que otro, con función f, si $g(n) \leq cf(n)$ para toda $n > N$ y cierta constante $c > 0$ (recordemos que el principio de invarianza refería a dos desigualdades para indicar el mismo comportamiento). Precisando: Se dice que una función $g : \mathbb{N} \to \mathbb{R}$ está en *el orden de* $f : \mathbb{N} \to \mathbb{R}$ sii existe $c > 0$ tal que $g(n) \leq cf(n)$ para toda $n \geq N$, $N \in \mathbb{N}$.

Destacamos que para afirmar que g está en el orden de f debe determinarse c, el múltiplo, y N, el umbral, de tal forma que $g(n) \leq cf(n)$ para toda $n \geq N$. Notemos que dada una función f, habrá muchas funciones g que satisfagan esta relación, por ejemplo si $f(n) = n^2$, entonces están en el orden de f: $g_1(n) = n$, $g_2(n) = \log(n)$, $g_3(n) = n\log(n)$, $g_4(n) = 2n^2$, etc.

Llamaremos a esta clase de funciones $O(f(n))$. Así:

$$O(f(n)) = \{g : \mathbb{N} \to \mathbb{R} | (\exists c > 0, N \in \mathbb{N})(\forall n \geq N)(g(n) \leq cf(n))\}.$$

$O(f(n))$ contiene a todas las funciones que están *acotadas superiormente* por un múltiplo de f.

La definición de g "está en el orden de" f contiene una redundancia al exigir la determinación tanto del múltiplo como del umbral. La constante c permite que el múltiplo de f supere a una función g siempre que ésta no crezca más rápido que f.

Pero también es posible que en un intervalo finito la función f tome valores negativos, lo cual se evita "saltando" este intervalo al dar un valor de umbral mayor que el límite superior de ese intervalo. Lo mismo podemos hacer cuando f toma valor cero.

Ya que las funciones de comportamiento no toman valores negativos, aunque sí el valor cero, podemos llevar nuestra observación hacia una formulación más simple de la clase de funciones con cota superior f. Lo anterior es una justificación de la proposición 8.

Proposición 8 (Regla del Umbral) Sea f estrictamente positiva. $g(n) \in O(f(n))$ si y sólo si existe $c > 0$ tal que $g(n) \leq cf(n)$, para toda $n \in \mathbb{N}$.

Al afirmar que una función g está en la clase $O(f(n))$ no sólo queremos decir que el algoritmo de g no puede ser más lento que el de f, además la

función f puede ser empleada como un "punto en la escala de funciones" y de este modo ubicar a g en relación con los demás algoritmos en la escala.

Las funciones de comportamiento son expresiones obtenidas del análisis de los algoritmos y será común encontrar que un algoritmo esté compuesto de más de un paso, lo cual se traducirá en que su función de comportamiento sea la suma correspondiente de las funciones de sendos pasos.

Con el fin de determinar la ubicación de una función en la escala, y de manera semejante al razonamiento que se hace con los límites, cuando f es la suma de funciones $\{f_i\}_i$ podremos despreciar algunas funciones para quedarnos con la función representativa de f.

El hecho implícito en la idea expuesta es que habrá en está suma alguna función f_j que no pueda ser superada por las demás, entonces $O(\sum_i f_i(n)) = O(f_j)$.

Por ejemplo, $O(n^2 + n) = O(n^2)$, esto es, $\max\{n^2, n\} = n^2$. Usamos el operador $\max\{f_j(n), f_i(n)\} = f_j(n)$ para indicar que f_j siempre supera a f_i excepto en un número finito de puntos: $f_i(n) < f_j(n)$, para toda $n > N$. Este resultado es útil y por ello lo enunciamos como la *Regla del Máximo.*

Proposición 9 (Regla del Máximo) Si f_1 y f_2 son funciones de comportamiento, entonces $O(f_1(n) + f_2(n)) = O(\max\{f_1(n), f_2(n)\})$.

Debe cuidarse la premisa implícita establecida al inicio: las funciones de comportamiento son no negativas, y, en muchos casos, estrictamente positivas. Un ejemplo que muestra la obtención de errores al violar la premisa antedicha es: $O(n) = O(n + n^2 - n^2) = O(\max\{n, n^2, -n^2\}) = O(n^2)$. El error es que aparece una función que no es de comportamiento (positiva).

Observación 5

1. En el comentario del primer párrafo tenemos claramente que $g_i(n) \in O(f(n))$, $i = 1, 2, 3, 4$. Pero además, si $h(n) = n^3$, entonces $f(n) \in O(h(n))$, puesto que con $c = 1$ y $N = 1$ se cumple que $n^2 \leq cn^3$ para toda $n \geq N$. Por otro lado, es fácil explicar por qué también $g_i(n) \in O(h(n))$: si sabemos que $g_i(n) \leq c_i f(n)$ y que $f(n) \leq ch(n)$, entonces $g_i(n) \leq cc_i h(n)$, esto es $g_i(n) \in O(h(n))$. Lo anterior puede sostenerse sin que haya referencia alguna a los ejemplos particulares que hemos dado, bastan las definiciones e hipótesis.

2. Un hecho importante pero aparentemente sin utilidad es que para toda función de comportamiento f se cumple que $f(n) \in O(f(n))$.

3. Es posible aún imaginar que un par de funciones f, g cumpla $f(n) \in O(g(n))$ y $g(n) \in O(f(n))$. Aquí también es fácil ver que $O(f(n)) = O(g(n))$, pues para cualquier f_1 tal que $f_1 \in O(f(n))$ se tiene $f_1(n) \leq cf(n)$ $(n > N_1)$, pero además por hipóteis $f(n) \leq dg(n)$ $(n > N_2)$ luego para toda $f_1 \in O(f(n))$: $f_1(n) \leq cdg(n)$ $(n > \max N_1, N_2)$. Por tanto $O(f(n)) \subset O(g(n))$. Similarmente puede verse que $O(g(n)) \subset O(f(n))$.

4. Si convenimos definir la relación "$\in O$", ésta es como vimos reflexiva, transitiva y antisimétrica, y por tanto es una relación de orden.

Ejercicio 18

1. Demuestre o refute las siguientes afirmaciones:

   a) $n^3 \in O(n^2)$.

   b) $2^{n+1} \in O(2^n)$.

   c) $n! \in O((n+1)!)$.

2. Sea $k \in \mathbb{N}$. Demuestre que si una función de comportamiento f expresada por $f(n) = f_1(n) + f_2(n) + \cdots + f_k(n)$ donde $f_1, f_2, \ldots, f_k$ son también funciones de comportamiento, y $g(n) = \max\{f_1(n), f_2(n), \ldots, f_k(n)\}$ entonces $O(f(n)) = O(g(n))$.

3. Usando el resultado del problema anterior, por qué no es posible afirmar que

$$O(n) = O(\max\{\overbrace{n, \ldots, n}^{n \text{ veces}}\}) = O(\overbrace{n + \cdots + n}^{n \text{ veces}}) = O(n^2).$$

Como seguramente ha sido observado, la definición de la clase de funciones acotadas superiormente por otra guarda cierta semejanza con la definición de límite. En efecto, la regla que a continuación se enuncia establece esta relación.

Proposición 10 (Regla del Límite) Sean f_1 y f_2 dos funciones de comportamiento.

1. Si $\lim_{n\to\infty} f_1(n)/f_2(n) = 0$ entonces $f_1(n) \in O(f_2(n))$ y $f_2(n) \notin O(f_1(n))$.

2. Si $\lim_{n\to\infty} f_1(n)/f_2(n) = \infty$ entonces $f_2(n) \in O(f_1(n))$ y $f_1(n) \notin O(f_2(n))$.

3. Si $\lim_{n\to\infty} f_1(n)/f_2(n) > 0$ entonces $f_1(n) \in O(f_2(n))$ y $f_2(n) \in O(f_1(n))$.

La demostración de esta proposición es directa a partir de las definiciones. Por ejemplo: $|\frac{f_1(n)}{f_2(n)}| < \delta$ para toda $n > N_\delta$ conduce a $f_1(n) < \delta f_2(n)$, i.e. $f_1(n) \in O(f_2(n))$. Además suponiendo que existieran $c > 0$ y $N \in \mathbb{N}$ tales que $f_2(n) \leq cf_1(n)$ entonces $\frac{f_1(n)}{f_2(n)} \geq \frac{1}{c}$, para toda $n \geq N$, lo cual contradiría que $\lim_{n\to\infty} f_1(n)/f_2(n) = 0$.

Observación 6

1. De acuerdo con el ejercicio 10.2, $\lim_{n\to\infty} \frac{\log n}{\sqrt{n}} = 0$, podemos concluir que $\log n \in O(\sqrt{n})$ y $\sqrt{n} \notin O(\log n)$.

2. Partiendo de que $n \in O(2^{\lfloor \log n \rfloor})$ y $2^{\lfloor \log n \rfloor} \in O(n)$ podría suponerse que $\lim_{n\to\infty} 2^{\lfloor \log n \rfloor}/n$ converge, pero no es así: cada vez que n es una potencia de dos se cumple $2^{\lfloor \log n \rfloor}/n = 1$ pero cuando n crece $2^{\lfloor \log n \rfloor}/n$ oscila. En conclusión, el recíproco de la regla del límite no es cierto en general.

Ejercicio 19

1. Use la proposición 5 para demostrar el segundo inciso de la regla del límite.

2. Demuestre que $n \in O(2^{\lfloor \log n \rfloor})$ y $2^{\lfloor \log n \rfloor} \in O(n)$.

3. ¿Cuál algoritmo sería más rápido, uno con comportamiento $f(n) = (\log \log n)^2$ u otro con comportamiento $g(n) = \log n$? Justifique su respuesta.

4.2. Cota inferior

Así como se estableció que un algoritmo no puede ser más tardado que otro, podemos hacer algo para decidir que un algoritmo no puede ser más veloz que otro, lo cual parece sencillo pues esta idea maneja la negación de lo que antes hemos visto. La clase de funciones que están *acotadas inferiormente* por f es:

$$\Omega(f(n)) = \{g : \mathbb{N} \to \mathbb{R} | (\exists d > 0, N \in \mathbb{N})(\forall n \geq N)(g(n) \geq df(n))\}.$$

La relación entre O y Ω está dada por la proposición que sigue.

Proposición 11 (Regla de la Dualidad) Sean f_1 y f_2 funciones de comportamiento, entonces $f_2(n) \in \Omega(f_1(n))$ sii $f_1(n) \in O(f_2(n))$.

Ejercicio 20

1. Demuestre la regla de la dualidad.
2. Demuestre o refute las siguientes afirmaciones:
   a) $n^3 \in \Omega(n^2)$.
   b) $2^{n+1} \in \Omega(2^n)$.
   c) $n! \in \Omega((n+1)!)$.
3. Tome en cuenta la observación 5 para demostrar que la relación de pertenencia a $\Omega(f(n))$ es una relación de orden parcial.

La interpretación de Ω contrasta con la de O: Cuando f_2 mide el peor comportamiento y $f_2 \in O(f_1(n))$, entonces en cualquier ejemplar f_2 será a lo más tan lento como f_1. En tanto, afirmar $f_2 \in \Omega(f_1(n))$ significa que f_2 no puede mejorar a f_1 (en el caso peor), pero es posible que haya ejemplares en los que f_2 mejore a f_1 (casos mejores).

4.3. Clase de equivalencia

Recordemos que el principio de invarianza establece que cualesquier implantaciones de un mismo algoritmo con comportamientos $f_1(n)$ y $f_2(n)$ satisfacen que $f_1(n) \leq cf_2(n)$ y $df_2(n) \leq f_1(n)$ para ciertas constantes $c, d > 0$ y para toda $n \geq N$. Desde entonces hemos considerado que los comportamientos "equivalentes" satisfacen las desigualdades mencionadas.

Este hecho aclara el uso de $g(n) \in O(f(n))$ y $g(n) \in \Omega(f(n))$, al afirmar que $g(n)$ no puede ser más lenta que $f(n)$ y $g(n)$ no puede ser más rápida que $f(n)$, la equivalencia de comportamientos es entonces cuando $g(n)$ no puede ser más lenta ni más rápida que $f(n)$. Tenemos así que la clase de funciones *equivalentes* a f es:

$$\Theta(f(n)) = O(f(n)) \cap \Omega(f(n)).$$

O bien $\Theta(f(n)) = \{g(n) | (\exists c, d > 0, N \in \mathbb{N})(\forall n \geq N)(df(n) \leq g(n) \leq cf(n))\}$

Observación 7

1. Análogas a las reglas del umbral y el máximo para O, son válidas para Θ:

   Regla del Umbral para Θ: Sea f estrictamente positiva. $g(n) \in \Theta(f(n))$ si y sólo si existen $c, d > 0$ tal que $df(n) \leq g(n) \leq cf(n)$, para toda $n \in \mathbb{N}$.

   Regla del Máximo para Θ: $\Theta(f_1(n)+f_2(n)) = \Theta(\max\{f_1(n), f_2(n)\})$.

2. Con ligeros cambios respecto de O tenemos la regla del límite para Θ:

   Regla del Límite para Θ: Sea $\lim_{n\to\infty} f_1(n)/f_2(n) = a$,

   a) si $a > 0$ entonces $f_1(n) \in \Theta(f_2(n))$.
   b) si $a = 0$ entonces $f_1(n) \in O(f_2(n))$ y $f_1(n) \notin \Theta(f_2(n))$.
   c) Si $a = \infty$ entonces $f_1(n) \in \Omega(f_2(n))$ y $f_1(n) \notin \Theta(f_2(n))$.

Como esperaríamos Θ divide el conjunto de funciones en clases las cuales corresponden a los puntos de la "escala de funciones". Específicamente tenemos la proposición que sigue.

Proposición 12 La relación entre funciones de comportamiento $f(n) \sim g(n)$ sii $f(n) \in \Theta(g(n))$ es una relación de equivalencia.

Tal y como hemos planteado intuitivamente en una sección anterior podemos concebir una escala que permita dar respuesta a una pregunta fundamental del análisis de algoritmos ¿cuál algoritmo es mejor? Se ha dicho que la respuesta es relativa a la situación en la que será aplicado. Dentro del enfoque asintótico la respuesta está dada por comparar las funciones.

En virtud de la observación 5 y la proposición 12 distinguimos a las funciones constantes $\Theta(1)$, funciones logarítmicas $\Theta(\log n)$, funciones lineales $\Theta(n)$, funciones cuadráticas $\Theta(n^2)$, etc. como los puntos de la escala.

Por otro lado, la relación $f(n) \lessdot g(n)$ sii $f(n) \in O(g(n))$ es una relación de orden y podríamos decir que un algoritmo con función de comportamiento $f(n)$ sería mejor que otro con función $g(n)$ si $\hat{f}(n) \lessdot \hat{g}(n))$ donde $f(n) \in \Theta(\hat{f}(n))$ y $g(n) \in \Theta(\hat{g}(n))$.

Finalmente, hemos de precisar que la escala no es tal pues la relación $\lessdot$ es parcial. Esto es, no es cierto que cualesquier par de funciones sea comparable. Tenemos, más bien, una jerarquía de clases de funciones, y la escala es solamente una aproximación útil a esta estructura.

Ejercicio 21

1. Demuestre la regla del límite para Θ.

2. Demuestre la prop. 12.

3. Considere un algoritmo que requiera un tiempo en la clase $\Theta(n^{\log 3})$ ¿Es correcto decir que requiere un tiempo en la clase $O(n^{1{,}59})$?, o ¿en la clase $\Omega(n^{1{,}59})$? o ¿en la clase $\Theta(n^{1{,}59})$? (*Nota:* $\log 3 = 1{,}58496\ldots$)

4. Dadas dos funciones de comportamiento f, g demostrar que:

   a) $O(f(n)) = O(g(n))$ sii $f(n) \in \Theta(g(n))$ sii $\Theta(f(n)) = \Theta(g(n))$.

   b) $O(f(n)) \subsetneq O(g(n))$ sii $f(n) \in O(g(n))$ pero $f(n)) \notin \Omega(g(n))$.

5. Ubique en la escala de funciones las siguientes funciones: $n \log n$, n^8, $n^{1+\epsilon}$, $(1+\epsilon)^n$, $n^2/\log n$ y $(n^2 - n + 1)^4$.

4.4. Cotas condicionales

Una clase más y algunas definiciones facilitarán el trabajo con los algoritmos conocidos como "divide y vencerás". Para que una función de comportamiento pertenezca a esta clase es necesario que la variable independiente (tamaño de ejemplar) cumpla una propiedad. El conjunto de funciones acotadas superiormente condicionadas por P es:

$$O(f(n)|P(n)) = \\ \{g(n)|(\exists c > 0, N \in \mathbb{N})(\forall n \geq N)(P(n) \rightarrow g(n) \leq cf(n))\}$$

Similarmente definimos $\Omega(f(n)|P(n))$ y $\Theta(f(n)|P(n))$. Se dice que una función de comportamiento es *asintóticamente no decreciente* (andec) si $\exists N \in \mathbb{N}$ tal que $(\forall n \geq N)(f(n) \leq f(n+1)$. Dada una función andec, f, se dice que es *b-uniforme* si $f(bn) \in O(f(n))$, $b \in \mathbb{R}$. Cuando para toda $b \geq 2$, f es b-uniforme, f se llama *suave*. Las funciones n^k y $n \log n$ son suaves, pero ni $n^{\log n}$, ni 2^n lo son, pues crecen muy rápido. Por ejemplo, si suponemos que $n^{\log n}$ fuera suave entonces $(2n)^{\log 2n} \in O(n^{\log n})$ y por tanto $(2n)^{\log 2n} \leq cn^{\log n}$, pero $(2n)^{\log 2n}/n^{\log n} = 2n^2$ no puede acotarse por una constante. Concluimos la presente sección con el enunciado que generaliza la pertenencia a una clase condicionada para funciones suaves.

Proposición 13 (Regla de la Uniformidad) Si $f(n)$ es suave, $g(n)$ andec y $g(n) \in \Theta(f(n)|n = b^k)$, entonces $g(n) \in \Theta(f(n))$.

Demostración. Supongamos que $g(n) \in \Theta(f(n)|n = b^k)$. Sea $\underline{n} = b^{\lfloor \log_b n \rfloor}$ y $\bar{n} = b^{\lceil \log_b n \rceil + 1}$, así $\underline{n} \leq n \leq \bar{n}$. Demostraremos que $g(n) \in O(f(n))$ y $g(n) \in \Omega(f(n))$, partiendo de que $g(n) \in O(f(n)|n = b^k)$ y $g(n) \in \Omega(f(n)|n = b^k)$. Con base en las hipótesis: $g(n) \leq g(\bar{n}) = g(\underline{n}b) \leq cf(\underline{n}b) \leq acf(\underline{n}) \leq acf(n)$ para $n \geq \text{máx}\{1, b^N\}$. Similarmente: $g(n) \geq g(\underline{n}) \geq cf(\underline{n}) \geq \frac{c}{a}f(\underline{n}b) \geq \frac{c}{a}f(\bar{n}) \geq \frac{c}{a}f(n)$. $\square$

Ejercicio 22

1. Diga la razón del uso de cada una de las desigualdades que aparecen en la demostración de la prop. 13.

2. Demuestre que 2^n no es suave.

3. ¿Cómo aplicaría la regla de la uniformidad a un algoritmo que busca un nodo en un árbol binario?

Capítulo 5

Análisis de Algoritmos

Cuando encuentra que varios algoritmos distintos solucionan el mismo problema, tiene que decidir cual es el más apropiado para la aplicación que desea desarrollar. Una herramienta esencial para este propósito es el *análisis de algoritmos*. Sólo después de que ha determinado la eficiencia de los distintos algoritmos será capaz de hacer una decisión bien informada.

El problema es que no hay una fórmula mágica para analizar la eficiencia de los algoritmos. Principalmente el análisis es un asunto de juicio, intuición y experiencia. No obstante, existen algunas técnicas básicas que son útiles con frecuencia, tales como saber de que forma tratar con estructuras de control y ecuaciones de recurrencia. Esta sección cubre las técnicas más comúnmente usadas y las ilustra con ejemplos.

5.1. Análisis de estructuras de control

El análisis de algoritmos usualmente se lleva a cabo desde adentro hacia afuera. Primero determinamos el tiempo requerido por las instrucciones individuales (este tiempo frecuentemente está acotado por una constante); luego combinamos estos tiempos de acuerdo a las estructuras de control que componen las instrucciones en el programa.

Algunas estructuras de control tales como la secuencia (poner una instrucción después de otra) son fáciles de analizar, mientras que otras, como los ciclos **while**, son más difíciles.

En esta subsección daremos principios generales que son útiles en el análisis involucrado con las estructuras de control encontradas más frecuentemen-

te, así como ejemplos de la aplicación de estos principios.

5.1.1. Secuencia

Sean P_1 y P_2 dos fragmentos de un algoritmo. Estos pudieran ser instrucciones simples o subalgoritmos complejos. Sean t_1 y t_2 los tiempos que consumen P_1 y P_2, respectivamente. Los tiempos pudieran depender de varios parámetros, tales como el tamaño del ejemplar. La **regla para secuenciar** dice que el tiempo requerido para calcular "$P_1; P_2$", esto es, primero P_1 y después P_2, simplemente es $t_1 + t_2$. Por la regla del máximo, este tiempo está en $\Theta(\text{máx}(t_1, t_2))$.

A pesar de su simplicidad, aplicar esta regla algunas veces es menos obvio de lo que pudiera parecer. Por ejemplo, pudiera suceder que uno de los parámetros que controla a t_2 depende del resultado del cálculo efectuado por P_1. Así, el análisis de la secuencia "$P_1; P_2$" no simpre puede ser realizado considerando que P_1 y P_2 son independientes.

5.1.2. Ciclos for

Los ciclos **for** son los más fáciles de analizar. Considere el siguiente ciclo.

$$\textbf{for } i \leftarrow 1 \textbf{ to } m \textbf{ do } P(i)$$

Aquí y en el resto de nuestra discusión, adoptaremos la convención de que $m = 0$ no es un error; simplemente significa que la instrucción controlada $P(i)$ nunca es ejecutada. Suponga que este ciclo es parte de un algoritmo más extenso, el cual trabaja sobre un ejemplar de tamaño n. (Tenga cuidado en no confundir m y n) El caso más fácil es cuando el tiempo consumido por $P(i)$ realmente no depende de i, no obstante pudiera depender del tamaño del ejemplar, más generalmente, del ejemplar en sí.

Sea t el tiempo requerido para calcular $P(i)$. En este caso, el análisis obvio del ciclo es que $P(i)$ es ejecutado m veces, cada vez con costo t, así el tiempo total requerido por el ciclo simplemente es $l = mt$. A pesar de que este enfoque usualmente es adecuado, existe un peligro potencial: no tomamos en consideración el tiempo necesario para *controlar el ciclo*. Después de todo, nuestro ciclo **for** es una abreviación para algo como el siguiente ciclo **while**.

$i \leftarrow 1$
while $i \leq m$ **do**
  $P(i)$
  $i \leftarrow i + 1$

En la mayoría de las situaciones es razonable contar como de costo unitario la prueba $i \leq m$, la instrucción $i \leftarrow 1$ y $i \leftarrow i + 1$ y las operaciones secuencia (**go to**) implicitas en el ciclo **while**. Sea c una cota superior sobre el tiempo requerido por cada una de estas operaciones. El tiempo l consumido por el ciclo está acotado superiormente por

$$
\begin{array}{llll}
l & \leq & c & \text{para } i \leftarrow 1 \\
 & + & (m+1)c & \text{para las pruebas } i \leq m \\
 & + & mt & \text{para las ejecuciones de } P(i) \\
 & + & mc & \text{para las ejecuciones de } i \leftarrow i+1 \\
4 & + & mc & \text{para las operaciones secuencia} \\
 & \leq & (t+3c)m + 2c &
\end{array}
$$

Más aún, este tiempo está claramente acotado inferiormente por mt. Si c es despreciable comparado con t, nuestro aproximado anterior de que l era aproximadamente igual a mt fue por tanto justificado, excepto por un caso crucial: $l \approx mt$ es completamente erroneo cuando $m = 0$ (¡aún es peor cuando m es negativo!).

Cuando estudiemos la técnica del barómetro, veremos que despreciar el tiempo requerido para el control del ciclo, puede guiarnos a errores serios en tales circunstancias.

Resista la tentación de decir que el tiempo consumido por el ciclo está en $\Theta(mt)$ con el pretexto de que a la notación Θ se le exige ser efectiva a partir de un umbral tal como $m \geq 1$. El problema con este argumento, es que, si en efecto estamos analizando el algoritmo entero más que simplemente el ciclo **for**, el umbral implicado por la notación Θ involucra a n, el tamaño del ejemplar, en vez de m, el número de veces que pasamos por el ciclo, $m = 0$ pudiera ocurrir para valores de n arbitrariamente grandes.

Por otro lado, se puede demostrar (le sugerimos que lo haga) que si t está acotado inferiormente por una constante (lo cual simpre es el caso en

la práctica) y si existe un umbral n_0 tal que $m \geq 1$ siempre que $n \geq n_0$, entonces l en verdad está en $\Theta(mt)$ cuando l, m y t son considerados como funciones de n.

El análisis del ciclo **for** es más interesante cuando el tiempo $t(i)$ requerido para $P(i)$ varía como una función de i. (En general, el tiempo requerido para $P(i)$ pudiera depender no sólo de i sino también del tamaño n del ejemplar, incluso del ejemplar mismo.) Si despreciamos el tiempo consumido por el control del ciclo, lo cual usualmente es adecuado cuando $m \geq 1$, el mismo ciclo **for**

$$\textbf{for } i \leftarrow 1 \textbf{ to } m \textbf{ do } P(i)$$

consume un tiempo dado no por una multiplicación sino por una suma: esto es $\sum_{i=1}^{m} t(i)$.

Ilustraremos el análisis de ciclos **for** con un algoritmo iterativo simple para calcular la secuencia de Fibonacci. Recuerde que el algoritmo es

```
function FIBITER(n)
1  i ← 1
2  j ← 0
3  for k ← 1 to n
4  do j ← i + j
5     i ← j − i
6  return j
```

Si contamos todas las operaciones aritméticas con costo unitario, las instrucciones dentro del ciclo **for** consumen tiempo constante. Suponga que el tiempo consumido por estas instrucciones está acotado superiormente por alguna constante c. Sin tomar en cuenta el control del ciclo, el tiempo consumido por el ciclo **for** está acotado superiormente por n veces esta constante: nc.

Ya que las instrucciones antes y después del ciclo consumen un tiempo despreciable, concluimos que el algoritmo consume un tiempo en $O(n)$. Un razonamiento similar nos lleva a que este tiempo también está en $\Omega(n)$, así está en $\Theta(n)$.

No obstante, vimos en la subsección 3.4 que no es razonable contar como de costo unitario, las adiciones involucradas en el cálculo de la secuencia de Fibonacci, al menos que n sea muy pequeño. Por lo tanto, debemos tomar

en cuenta el hecho de que una instrucción tan simple como "$j \leftarrow i + j$" se encarece incrementalmente cada vez que pasamos por el ciclo.

Es fácil programar adiciones y sustracciones de enteros largos, tal que el tiempo necesario para sumar y sustraer dos enteros, esté en el orden exacto del número de dígitos del operando más grande. Para determinar el tiempo consumido por la $k-$ésima pasada por el ciclo, necesitamos conocer la longitud de los enteros involucrados.

Se puede demostrar por inducción matemática que los valores de i y j al final de la $k-$ésima iteración son respectivamente f_{k-1} y f_k. Esto es precisamente por lo que el algoritmo funciona: regresa el valor de j al final de la $n-$ésima iteración, el cual es por lo tanto f_n, como se requiere. Más aún, a partir de la fórmula de de Moivre se puede demostrar que el tamaño de f_k está en $\Theta(k)$.

Por lo tanto la $k-$ésima iteración consume un tiempo $\Theta(k-1) + \Theta(k)$, lo cual es lo mismo que $\Theta(k)$. Sea c una constante tal que este tiempo está acotado superiormente por ck para todo $k \geq 1$. Si despreciamos el tiempo requerido por el control del ciclo y por las instrucciones antes y después del ciclo, podemos concluir que el tiempo requerido por el algoritmo está acotado superiormente por

$$\sum_{k=1}^{n} ck = c \sum_{k=1}^{n} k = c\frac{n(n+1)}{2} \in O(n^2).$$

Un razonamiento similar nos conduce a que este tiempo está en $\Omega(n^2)$ y por lo tanto en $\Theta(n^2)$. Así, en el análisis de FIBITER, contar como de costo unitario a las operaciones aritméticas, constituye una diferencia crucial con respecto a no contarlas como de costo unitario.

El análisis de ciclos **for** que inician en valores distintos a 1, o que se ejecutan a pasos más grandes, debe ser obvio en este punto. Por ejemplo considere el siguiente ciclo.

for $i \leftarrow 5$ **to** m **step** 2 **do** $P(i)$

Aquí, $P(i)$ se ejecuta $((m-5) \div 2) + 1$ veces siempre que $m \geq 3$. Para que un ciclo **for** tenga sentido (se ejecute cero o más veces), el punto final debe ser siempre al menos tan grande como el punto inicial *menos* el paso.

5.1.3. Llamados recursivos

El análisis de algoritmos recursivos usualmente es directo, al menos hasta cierto punto. Una simple inspección al algoritmo frecuentemente da lugar a una **ecuación de recurrencia** que imita el flujo de control en el algoritmo. Una vez que ha sido obtenida la ecuación de recurrencia, se pueden aplicar técnicas generales para resolver dichas ecuaciones y transformarlas a la notación asintótica no recursiva, la cual es más simple.

Como un ejemplo, considere de nuevo el problema de calcular la secuencia de Fibonacci, pero esta vez con el algoritmo recursivo FIBREC

```
function FIBREC(n)
1  if n < 2
2     then return n
3     else  return FIBREC(n − 1) + FIBREC(n − 2)
```

Sea $T(n)$ el tiempo consumido por un llamado a $Fibrec(n)$. Si $n < 2$, el algoritmo simplemente regresa n, lo cual consume tiempo constante a. De otra manera, la mayor parte del trabajo se gasta en los dos llamados recursivos, los cuales consumen respectivamente tiempo $T(n-1)$ y $T(n-2)$. Más aún, debe ser ejecutada una adición que involucra a f_{n-1} y a f_{n-2} (los cuales son los valores regresados por los llamados recursivos), así como, el control de la recursión y la prueba “**if** $n < 2$”.

Sea $h(n)$ que denota el trabajo involucrado en esta adición y control, esto es, el tiempo requerido por un llamado a $Fibrec(n)$ ignorando el tiempo gastado dentro de los dos llamados recursivos. Por definición de $T(n)$ y $h(n)$ obtenemos la siguiente recurrencia.

$$T(n) = \begin{cases} a & \text{si } n = 0 \text{ o } n = 1, \\ T(n-1) + T(n-2) + h(n) & \text{en otro caso} \end{cases} \tag{5.1}$$

Si contamos las adiciones de costo unitario, $h(n)$ está acotada por una constante. Al aplicarle a (5.1) técnicas para solución de recurrencias encontramos que $T(n) \in O(f_n)$. Un razonamiento similar muestra que $T(n) \in \Omega(f_n)$ y por tanto $T(n) \in \Theta(f_n)$. Usando la fórmula de de Moivre, concluimos que $Fibrec(n)$ consume tiempo exponencial en n. Éste es *doble* exponencial en el tamaño del ejemplar ya que el *valor* de n es exponencial en el *tamaño* de n.

Si no contamos la adición de costo unitario, $h(n)$ ya no estará acotada por una constante. En su lugar $h(n)$ está dominada por el tiempo requerido para la adición de f_{n-1} y f_{n-2} para n suficientemente grande. Ya hemos visto que esta adición consume un tiempo en el orden exacto de n. Por tanto $h(n) \in \Theta(n)$.

Sorprendentemente, al aplicarle a (5.1) técnicas para solución de recurrencias, el resultado es el mismo sin importar cuando $h(n)$ es constante o lineal: es decir se sigue cumpliendo que $T(n) \in \Theta(f_n)$. En conclusión, ¡$Fibrec(n)$ consume tiempo exponencial en n sin importar que consideremos o no a las adiciones como de costo unitario! La única diferencia recae en la constante multiplicativa oculta en la notación Θ.

5.1.4. Ciclos while y repeat

Los ciclos **while** y **repeat** usualmente son más difíciles de analizar que los ciclos **for** ya que no existe una manera a priori obvia para saber cuántas veces se entrará al ciclo. La técnica estándar para analizar estos ciclos es encontrar una función de las variables involucradas cuyo valor decrezca conforme se entre al ciclo.

Para concluir que el ciclo eventualmente termina es suficiente con demostrar que este valor debe ser un entero positivo, ya que no se puede decrementar indefinidamente un entero positivo. Para determinar cuántas veces se repite el ciclo, necesitamos entender bien cómo decrece el valor de dicha función.

Un enfoque alternativo para el análisis de los ciclos **while** consiste en tratarlos como algoritmos recursivos. Ilustraremos ambas técnicas con el mismo ejemplo. El análisis de los ciclos **repeat** se realiza de la misma manera y no se darán ejemplos de ellos.

Usaremos el algoritmo de BinarySearch para ilustrar el análisis de ciclos **while**. El propósito de la búsqueda binaria es encontrar un elemento x en un arreglo $T[1..n]$ que está en orden no decreciente. Asuma por simplicidad que se garantiza que x aparezca al menos una vez en T. Se requiere encontrar un entero i tal que $1 \leq i \leq n$ y $T[i] = x$. La idea básica detrás de la búsqueda binaria es comparar x con el elmento y que está en la mitad de T. La búsqueda termina si $x = y$, si $x > y$ se busca en la mitad superior del arreglo, si no es el caso se busca en la mitad inferior del arreglo, tenemos así el siguiente algoritmo.

```
function BinarySearch(T[1...n], x)
1  i ← 1
2  j ← n
3  while i < j
4  do k ← (i + j) ÷ 2
5      switch
6          case x < T[k] : j ← k − 1
7          case x = T[k] : i, j ← k
8          case x > T[k] : i ← k + 1
9  return i
```

Recuerde que para analizar el tiempo de ejecución de un ciclo **while** debemos encontrar una función de las variables involucradas cuyo valor decrezca cada vez que se entra al ciclo. En este caso, es natural considerar a $j - i + 1$ al que llamaremos d. Así, d representa el número de elementos que todavía se tienen que considerar, inicialmente $d = n$. El ciclo termina cuando $i \geq j$, lo cual es equivalente a que $d \leq 1$. En cada iteración existen tres posibilidades: j toma el valor $k - 1$, i toma el valor $k + 1$ o tanto i como j toman el valor k. Sean d y d' respectivamente el valor de $j - i + 1$ antes y después de la iteración bajo consideración, usaremos i, j, i' y j' de manera similar.

Si $x < T[k]$, la instrucción "$j \leftarrow k - 1$" se ejecuta y así

$$i' = i \text{ y } j' = [(i + j) \div 2] - 1.$$

Por lo tanto,

$$\begin{aligned} d' = j' - i' + 1 = [(i + j) \div 2] - 1 - i + 1 = \\ (i + j) \div 2 - i \leq (i + j)/2 - i = \\ (i + j - 2i)/2 = (j - i)/2 < \\ (j - i)/2 + 1/2 = (j - i + 1)/2 = d/2. \end{aligned}$$

Similarmente, si $x > T[k]$, la instrucción "$i \leftarrow k + 1$" se ejecuta y así

$$i' = [(i + j) \div 2] + 1 \text{ y } j' = j.$$

Por lo tanto,

$$\begin{aligned} d' = j' - i' + 1 = j - [((i + j) \div 2) + 1] + 1 = \\ j - (i + j) \div 2 - 1 + 1 \leq j - (i + j)/2 = \\ (2j - (i + j))/2 = (j - i)/2 < \\ (j - i)/2 + 1/2 = (j - i + 1)/2 = d/2. \end{aligned}$$

Finalmente, is $x = T[k]$ entonces i y j toman el mismo valor por tanto $d' = 1$; pero d fue al menos 2 ya que de otro modo no se habría entrado al ciclo (la condición del ciclo es que $i < j$ y $d = j - i + 1$). Concluimos que $d' = d/2$ en cualquier caso, lo que significa que el valor de d se reduce al menos a la mitad en cada iteración. Ya que paramos cuando $d \leq 1$, el proceso eventualmente termina, pero ¿cuánto tiempo toma?

Para determinar una cota superior sobre el tiempo de ejecución de la búsqueda binaria, sea d_l que denota el valor $j - i + 1$ al final de la $l-$ésima iteración para $l \geq 1$ y sea $d_0 = n$. Ya que d_{l-1} es el valor de $j - i + 1$ **antes** de iniciar la $l-$ésima iteración y como hemos demostrado que $d_l \leq d_{l-1}/2$ para todo $l \leq 1$, se sigue inmediatamente por inducción matemática que $d_l \leq n/2^l$. Pero el ciclo termina cuando $d \leq 1$, lo cual sucede a lo más cuando $l = \lceil \lg n \rceil$[1].

Concluimos que se entra al ciclo a lo más $\lceil \lg n \rceil$ veces. Ya que cada iteración tarda un tiempo constante, la búsqueda binaria tarda un tiempo en $O(\log n)$. Un razonamiento similar nos da como resultado una cota inferior en $\Omega(\log n)$, por tanto la búsqueda binaria tarda un tiempo en $\Theta(\log n)$.

El enfoque alternativo para analizar el tiempo de ejecución de la búsqueda binaria empieza muy parecido. La idea es pensar en el ciclo **while** como si estuviera pensado recursivamente en lugar de iterativamente. En cada iteración, reducimos el rango de posibles posiciones en el arreglo para x.

Sea $t(d)$ que denota el tiempo máximo necesario para terminar el ciclo **while** cuando $j - i + 1 \leq d$, esto es cuando existen a lo más d elementos por considerar. Hemos visto que el valor de $j - i + 1$ al menos decrece la mitad en cada iteración.

En términos recursivos esto significa que $t(d)$ es a lo más el tiempo constante b necesario para realizar una vez la iteración, más el tiempo $t(d \div 2)$ suficiente para que la iteración se termine. Ya que determinar si el ciclo ha terminado, cuando $d = 1$, tarda un tiempo constante c, obtenemos la siguiente recurrencia.

$$t(d) = \begin{cases} c & \text{si } d = 1 \\ b + t(d \div 2) & \text{en otro caso} \end{cases}$$

Las técnicas que veremos en las Subsección 5.3 se aplican para que fácilmente concluyamos que $t(n) \in O(\log n)$.

[1] Recuerde que en los preliminares matemáticos dijimos que $\lg x$ es una abreviación para $\log_2 x$.

5.2. Uso de un barómetro

El análisis de muchos algoritmos se simplifica significativamente cuando una instrucción, o una prueba, se puede identificar como *barómetro*. Una instrucción barómetro es una que se ejecuta al menos tan frecuentemente como cualquier otra instrucción en el algoritmo.

No hay problema si algunas instrucciones se ejecutan hasta un número constante de veces más a menudo que el barómetro, ya que su contribución se absorbe en la notación asintótica. Siempre que el tiempo que tarda cada instrucción está acotado por una constante, el tiempo que tarda todo el algoritmo está en el orden exacto del número de veces que la instrucción barómetro se ejecuta.

Esto es útil porque nos permite despreciar los tiempos exactos que tarda cada instrucción. En particular, evita la necesidad de introducir constantes tales como aquellas que acotan el tiempo que tardan varias operaciones elementales, las cuales no son importantes ya que dependen de la implementación y que son descartadas cuando el resultado final se expresa en términos de la notación asintótica.

Por ejemplo, considere el análisis de FIBITER de la subsección 5.1.2 cuando contamos todas las operaciones aritméticas como de costo unitario. Vimos que el algoritmo tarda un tiempo acotado superiormente por cn para alguna constante c despreciable, y por lo tanto tarda un tiempo en $\Theta(n)$.

Hubiera sido más simple decir que la instrucción $j \leftarrow i+j$ se puede tomar como barómetro, que obviamente esta instrucción se ejecuta n veces y por tanto el algoritmo tarda un tiempo en $\Theta(n)$.

Cuando un algoritmo involucra varios ciclos anidados, cualquier instrucción del ciclo más interno usualmente puede tomarse como barómetro. No obstante, esto se debe hacer con cuidado porque hay casos donde es necesario tomar en cuenta el control implícito del ciclo.

Esto sucede tipicamente cuando algunos de los ciclos se ejecutan cero veces, ya que tales ciclos consumen tiempo aunque no se llegue a ejecutar la instrucción barómetro. Si esto sucede con frecuencia, el número de veces que la instrucción barómetro se ejecuta puede ser minimizado por el número de veces que se ejecuta el control de los ciclos vacíos y por tanto sería un error considerar tal instrucción más interna como barómetro.

Considere por ejemplo ordenamiento por casillas (Sección 3.6). Aquí generalizamos el algoritmo para manejar el caso donde los elementos a ordenar son enteros que se sabe están entre 1 y s en lugar de entre 1 y 10000. Recuer-

de que $T[1 \dots n]$ es el arreglo a ordenar y $U[1 \dots s]$ es un arreglo construido tal que $U[k]$ contiene el número de veces que el entero k aparece en T. La fase final del algoritmo reconstruye T en order no decreciente a partir de la información que hay en U.

```
procedure ExtractoDeCasillas()
1   i ← 0
2   for k ← 1 to s
3   do while U[k] ≠ 0
4       do i ← i + 1
5           T[i] ← k
6           U[k] ← U[k] − 1
```

Para analizar el tiempo requerido para este procedimiento, usaremos "$U[k]$" para denotar el valor almacenado *originalmente* en $U[k]$ ya que todos estos valores son definidos a 0 durante el proceso. Es tentador elegir a cualquiera de las instrucciones en el ciclo más interno como barómetro. Para cada valor de k, estas instrucciones son ejecutadas $U[k]$ veces.

Por tanto el número total de veces que son ejecutadas es $\sum_{k=1}^{s} U[k]$. Pero esta suma es igual a n, el número de enteros a ordenar, ya que la suma del número de veces que cada elemento aparece da el número total de elementos. Si en verdad estas instrucciones sirvieran como barómetro, concluiríamos que este procedimiento tarda un tiempo en el orden exacto de n.

Un ejemplo simple es suficiente para convencernos que éste no es necesariamente el caso. Suponga que $U[k] = 1$ cuando k es un cuadrado perfecto y $U[k] = 0$ en otro caso. Esto correspondería a ordenar un arreglo T que contiene exactamente una vez cada cuadrado perfecto entre 1 y n^2, usando $s = n^2$ casillas. En este caso, el procedimiento tarda claramente un tiempo en $\Omega(n^2)$ ya que el ciclo más externo se ejecuta s veces.

Por lo tanto no puede ser que el tiempo que tarda esté en $\Theta(n)$. Esto demuestra que haber elegido una de las instrucciones en el ciclo más interno como barómetro fue incorrecto. El problema surge ya que sólo podemos despreciar el tiempo utilizado en iniciar y controlar los ciclos siempre y cuando estemos seguros de haber incluido algo que tome en cuenta si el ciclo se ejecuta cero veces.

El análisis correcto y detallado del procedimiento es como sigue. Sea a el tiempo necesariio para la prueba $U[k] \neq 0$ cada vez que se pretende ejecutar el ciclo más interno y sea b el tiempo que tarda en una ejecución de las

instrucciones en el ciclo más interno, incluyendo la operación de secuenciación para regresar al inicio del ciclo para volver a probar su condición.

Ejecutar completamente el ciclo más interno para un valor dado de k tarda un tiempo $t_k = (1 + U[k])a + U[k]b$, donde sumamos 1 a $U[k]$ antes de multiplicarlo por a para tomar en cuenta el hecho de que la prueba se realiza cada vez que va a empezar otra iteración y una más para determinar que el ciclo se ha completado.

El asunto crucial es que este tiempo no es cero aún cuando $U[k] = 0$. El procedimiento completo tarda un teimpo $c + \sum_{k=1}^{s}(d + t_k)$, donde c y d son constantes nuevas que toman en cuenta el tiempo necesario para inciar y controlar el ciclo más externo, respectivamente.

Cuando se simplifica, esta expresión nos da $c + (a + d)s + (a + b)n$.. Concluimos que el procedimiento tarda un tiempo en $\Theta(n+s)$. Así, el tiempo depende de dos parámetros independientes n y s; no se puede expresar como una función de sólo uno de ellos.

Es fácil ver que la fase de iniciación de ordenamiento por casillas también tarda un tiempo en $\Theta(n + s)$. Esta técnica de ordenamiento tarda un tiempo total en $\Theta(n + s)$ para ordenar n enteros entre 1 y s. Si prefiere, se puede invocar a la regla del máximo para establecer que este tiempo está en $\Theta(\text{máx}\{n, s\})$.

Así, ordenamiento por casillas vale la pena si demostramos que s es suficientemente pequeño comparado con n. Por ejemplo, si estamos interesados en el tiempo requerido como una función que dependa solamente del número de elementos a ordenar, esta técnica tiene éxito en un tiempo lineal asombroso si $s \in O(n)$ pero alcanza tiempo cuadrático si $s \in \Theta(n^2)$.

A pesar de lo anterior, el uso de un barómetro es apropiado para analizar ordenamiento por casillas. El problema es que no se eligió el barómetro adecuado. En vez de las instrucciones *dentro* del ciclo interno, se debe elegir como barómetro la *prueba* $U[k] \neq 0$ del ciclo más interno. En verdad, ninguna instrucción en el procedimiento se ejecuta más veces que dicha prueba, lo cual es la definición de un barómetro.

Es fácil mostrar que $U[k] \neq 0$ se ejecuta exactamente $n + s$ veces, por lo tanto la conclusión correcta del sobre el tiempo de ejecución del procedimiento se sigue inmediatamente sin necesidad de introducir constantes sin sentido. En conclusión, el uso de un barómetro es una herramienta útil para simplificar el análisis de muchos algoritmos, pero esta técnica debe usarse con cuidado.

Como otro ejemplo del uso de la técnica del barómetro y del análisis de

ciclos anidados, veamos el algoritmo de ordenamiento por selección visto en la Sección 3.3.

```
procedure SELECT(T[1...n])
1  for i ← 1 to n − 1
2  do minj ← i
3      minx ← T[i]
4      for j ← i + 1 to n
5      do if T[j] < minx
6            then minj ← j
7                 minx ← T[j]
8      T[minj] ← T[i]
9      T[i] ← minx
```

No obstante el tiempo que tarda cada iteración del ciclo más interno no es constante, toma más tiempo cuando $T[j] < minx$, este tiempo está acotado superiormente por alguna constante c (que toma en cuenta el control del ciclo más interno).

Para cada valor de i, las instrucciones en el ciclo más interno se ejecutan $n - (i + 1) + 1 = n - i$ veces, por lo tanto el tiempo que tarda el ciclo más interno es $t(i) \leq (n - i)c$.

El tiempo que tarda la $i-$ésima iteración del ciclo más externo está acotado superiormente por $b + t(i)$ para una constante apropiada b que toma en cuenta las operaciones elementales antes y después del ciclo más interno y el control del ciclo más externo.

Por lo tanto, el tiempo total que tarda el algoritmo está acotado superiormente por:

$$\begin{aligned} \sum_{i=1}^{n-1} b + (n-i)c &= \sum_{i=1}^{n-1} (b + cn) - c \sum_{i=1}^{n-1} i \\ &= (n-1)(b+cn) - cn(n-1)/2 \\ &= \frac{1}{2}cn^2 + \left(b - \frac{1}{2}c\right) n - b, \end{aligned}$$

el cual está en $O(n^2)$. Un razonamiento similar muestra que este tiempo esta también en $\Omega(n^2)$ en todos los casos, y por tanto ordenamiento por selección tarda un tiempo en $\Theta(n^2)$ para ordenar n elementos.

El argumento anterior se puede simplificar, obviando la necesidad de introducir constantes explícitas tales como b y c, cuando nos sentimos cómodos con la noción de una instrucción barómetro. Aquí, es natural tomar la prueba del ciclo más interno $T[j] < minx$ como barómetro y contar el número exacto de veces que se ejecuta.

Esto es una buena medida del tiempo total del algoritmo ya que ninguno de los ciclos se puede ejecutar cero veces (en cuyo caso el control del ciclo podría consumir más tiempo que el barómetro). Se ve fácilmente que el número de veces que la prueba es ejecutada es:

$$\begin{aligned}\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} 1 &= \sum_{i=1}^{n-1}(n-i) \\ &= \sum_{k=1}^{n-1} k \\ &= n(n-1)/2.\end{aligned}$$

Así el número de veces que la instrucción barómetro se ejecuta está en $\Theta(n^2)$, lo cual da automáticamente el tiempo de ejecución de todo el algoritmo.

5.3. Solución de recurrencias

El último paso indispensable en el análisis de algoritmos es saber resolver **ecuaciones de recurrencia** (como se ha visto en secciones anteriores). Con un poco de experiencia e intuición la mayoría de las recurrencias se pueden resolver haciendo *conjeturas inteligentes*.

No obstante, existe una técnica poderosa que se puede usar para resolver ciertas clases de recurrencia casi automáticamente. Este es el tópico principal de esta sección: la **técnica de la ecuación caraterística**.

5.3.1. Recurrencias homogéneas

Empezaremos nuestro estudio de la **técnica de la ecuación caraterística** con la resolución de recurrencias lineales homogéneas con coeficientes constantes, es decir recurrencias de la forma

$$a_0 t_n + a_1 t_{n-1} + \cdots + a_k t_{n-k} = 0 \tag{5.2}$$

donde los t_i son los valores que estamos buscando.

Además de la Ecuación (5.2), los valores de t_i sobre k valores de i (generalmente $0 \leq i \leq k-1$ o $1 \leq i \leq k$) son necesarios para determinar la secuencia. Estas **condiciones iniciales** se considerarán más adelante. Hasta entonces, la Ecuación (5.2) normalmente tiene infinitas soluciones. Esta recurrencia es

- lineal porque no contiene términos de la forma $t_{n-i}t_{n-j}, t_{n-i}^2$, etc;
- homogénea porque las combinaciones lineales de las t_{n-i} son iguales a cero; y
- con coeficientes constantes porque las a_i son constantes.

Considere nuestra ahora familiar recurrencia de la secuencia de Fibonacci.

$$f_n = f_{n-1} + f_{n-2}$$

Esta recurrencia fácilmente se puede poner en la forma de la Ecuación (5.2) después de hacer la reescritua obvia.

$$f_n - f_{n-1} - f_{n-2} = 0$$

Por lo tanto, la secuencia de Fibonacci corresponde a una recurrencia lineal homogénea con coeficientes constantes, donde $k = 2, a_0 = 1$ y $a_1 = a_2 = -1$.

Incluso antes de comenzar a buscar soluciones a la Ecuación (5.2), es interesante notar que cualquier combinación lineal de soluciones es en sí misma una solución.

En otras palabras, si f_n y g_n satisfacen la Ecuación (5.2), entonces $\sum_{i=0}^{k} a_i f_{n-i} = 0$ y similarmente para g_n, si definimos $t_n = cf_n + dg_n$ para las constantes arbitrarias c y d, entonces t_n también es una solución de la Ecuación (5.2). Esto es cierto porque

$$\begin{aligned}
&a_0 t_n + a_1 t_{n-1} + \cdots + a_k t_{n-k} = \\
&a_0(cf_n + dg_n) + a_1(cf_{n-1} + dg_{n-1}) + \cdots + a_k(cf_{n-k} + dg_{n-k}) = \\
&c(a_0 f_n + a_1 f_{n-1} + \cdots + a_k f_{n-k}) + d(a_0 g_n + a_1 g_{n-1} + \cdots + a_k g_{n-k}) = \\
&c \times 0 + d \times 0 = 0.
\end{aligned}$$

Esta regla se generaliza a combinaciones lineales de cualquier número de soluciones.

Buscamos soluciones de la forma

$$t_n = x^n$$

donde x es una constante aún desconocida. Si sustituimos esta solución en la Ecuación (5.2), obtenemos

$$a_0 x^n + a_1 x^{n-1} + \cdots + a_k x^{n-k} = 0$$

Esta ecuación se satisface si $x = 0$, una solución trivial que no es de interés. De otra manera la ecuación se satisface si y sólo si

$$a_0 x^k + a_1 x^{k-1} + \cdots + a_k = 0$$

Esta ecuación de grado k en x se llama la **ecuación característica** de la recurrencia (5.2) y

$$p(x) = a_0 x^k + a_1 x^{k-1} + \cdots + a_k$$

se llama su **polinomio característico**.

Recuerde que el teorema fundamental del álgebra establece que cualquier polinomio $p(x)$ de grado k tiene exactamente k raíces (no necesariamente distintas), lo que significa que puede factorizarse como un producto de k monomios.

$$p(x) = \prod_{i=1}^{k} (x - r_i),$$

donde las r_i pueden ser números complejos. Aún más, estas r_i son las únicas soluciones de la ecuación $p(x) = 0$.

Considere cualquier raíz r_i del polinomio característico. Como $p(r_i) = 0$ se deduce que $x = r_i$ es una solución a la ecuación característica y, por lo tanto, r_i^n es una solución de la recurrencia. Dado que cualquier combinación lineal de soluciones también es una solución, concluimos que

$$t_n = \sum_{i=1}^{k} c_i r_i^n \tag{5.3}$$

satisface la recurrencia para cualquier elección de constantes $c_1, c_2, \ldots, c_k$. El hecho notable, que no probamos aquí, es que la Ecuación (5.2) sólo tiene soluciones de esta forma siempre que todas las r_i sean distintas. En este caso, las k constantes se pueden determinar a partir de k condiciones iniciales resolviendo un sistema de k ecuaciones lineales con k incógnitas.

Ejemplo 1 (Fibonacci) Considere la recurrencia

$$f_n = \begin{cases} n & \text{si } n = 0 \text{ o } n = 1 \\ f_{n-1} + f_{n-2} & \text{en otro caso} \end{cases}$$

Primero reescribimos esta recurrencia en la forma de la Ecuación 5.2.

$$f_n - f_{n-1} - f_{n-2} = 0$$

Por lo tanto su polinimio característico es

$$x^2 - x - 1$$

cuyas raíces son

$$r_1 = \frac{1+\sqrt{5}}{2} \text{ y } r_2 = \frac{1-\sqrt{5}}{2}.$$

Por lo tanto la solución general es de la forma

$$f_n = c_1 r_1^n + c_2 r_2^n. \tag{5.4}$$

Falta usar las condiciones iniciales para determinar las constantes c_1 y c_2. Cuando $n = 0$, la Ecuación (5.4) nos da $f_0 = c_1 + c_2$. Pero sabemos que $f_0 = 0$. Por lo tanto, $c_1 + c_2 = 0$. Similarmente cuando $n = 1$, la Ecuación (5.4) junto con la segunda condición inicial nos dice que $f_1 = c_1 r_1 + c_2 r_2 = 1$. Recordando que los valores de r_1 y r_2 son conocidos, esto nos da dos ecuaciones lineales con las incognitas c_1 y c_2.

$$\begin{aligned} c_1 + c_2 &= 0 \\ r_1 c_1 + r_2 c_2 &= 1 \end{aligned}$$

Resolviendo estas ecuaciones obtenemos

$$c_1 = \frac{1}{\sqrt{5}} \text{ y } c_2 = -\frac{1}{\sqrt{5}}.$$

Así

$$f_n = \frac{1}{\sqrt{5}} \left[\left(\frac{1+\sqrt{5}}{2} \right)^n - \left(\frac{1-\sqrt{5}}{2} \right)^n \right],$$

note que hemos obtenido la famosa fórmula de de Moivre para la secuencia de Fibonacci.

Ejemplo 2 Considere la recurrencia

$$t_n = \begin{cases} 0 & \text{si } n = 0 \\ 5 & \text{si } n = 1 \\ 3t_{n-1} + 4t_{n-2} & \text{en otro caso} \end{cases}$$

Primero reescribimos la recurrencia

$$t_n - 3t_{n-1} - 4t_{n-2} = 0$$

Su polinomio característico es

$$x^2 - 3x - 4 = (x+1)(x-4)$$

cuyas raíces son $r_1 = -1$ y $r_2 = 4$. La solución por tanto es de la forma

$$t_n = c_1(-1)^n + c_2 4^n$$

Las condiciones iniciales dan

$$\begin{aligned} c_1 + c_2 &= 0 \qquad n = 0 \\ -c_1 + 4c_2 &= 5 \qquad n = 1 \end{aligned}$$

Resolviendo estas ecuaciones obtenemos que $c_1 = -1$ y $c_2 = 1$. Por lo tanto

$$t_n = 4^n - (-1)^n.$$

La situación se vuelve un poco más complicada cuando el polinomio característico tiene múltiples raíces, es decir, cuando las k raíces no son todas distintas. Sigue siendo cierto que la Ecuación (5.3) satisface la recurrencia de cualquier valor de las constantes c_i, pero ésta ya no es la solución más general.

Para encontrar otras soluciones, sea

$$p(x) = a_0 x^k + a_1 x^{k-1} + \cdots + a_k$$

el polinomio característico de nuestra recurrencia, y sea r una raíz múltiple. Por definición de raíces múltiples, existe un polinomio $q(x)$ de grado $k-2$ tal que $p(x) = (x-r)^2 q(x)$.

Para cada $n \geq k$ considere los polinomios de n-ésimo grado

$$u_n(x) = a_0x^n + a_1x^{n-1} + \cdots + a_kx^{n-k} \text{ y}$$
$$v_n(x) = a_0nx^n + a_1(n-1)x^{n-1} + \cdots + a_k(n-k)x^{n-k}.$$

Observe que $v_n(x) = x \times u_n'(x)$, donde $u_n'(x)$ denota la derivada de $u_n(x)$ con respecto de x. Pero $u_n(x)$ se puede reescribir como

$$u_n(x) = x^{n-k}p(x) = x^{n-k}(x-r)^2q(x) = (x-r)^2 \times [x^{n-k}q(x)].$$

Usando la regla para calcular la derivada de un producto de funciones, obtenemos que la derivada de $u_n(x)$ con respecto de x es

$$u_n'(x) = 2(x-r)x^{n-k}q(x) + (x-r)^2[x^{n-k}q(x)]'.$$

Por lo tanto $u_n'(r) = 0$, lo cual implica que $v_n(r) = r \times u_n'(r) = 0$ para todo $n \geq k$. En otras palabras,

$$a_0nr^n + a_1(n-1)r^{n-1} + \cdots + a_k(n-k)r^{n-k} = 0.$$

Concluimos que $t_n = nr^n$ también es una solución a la recurrencia. Ésta es una solución genuinamente nueva en el sentido de que no puede obtenerse eligiendo las constantes apropiadas c_i en la Ecuación (5.3).

De manera más general, si la raíz r tiene multiplicidad m, entonces $t_n = r^n, t_n = nr^n, t_n = n^2r^n, \ldots, t_n = n^{m-1}r^n$ son todas las soluciones distintas de la recurrencia. La solución general es una combinación lineal de estos términos y de los términos aportados por las otras raíces del polinomio característico.

En resumen, si $r_1, r_2, \ldots, r_l$ son las l raíces distintas del polinomio característico y si sus multiplicidades son $m_1, m_2, \ldots, m_l$, respectivamente, entonces

$$t_n = \sum_{i=1}^{l} \sum_{j=0}^{m_i-1} c_{ij}n^jr_i^n$$

es la solución general de la Ecuación (5.2). Nuevamente, las constantes $c_{ij}, 1 \leq i \leq l$ y $0 \leq j \leq m_i-1$, deben ser determinadas por las k condiciones iniciales.

Existen esas k constantes porque $\sum_{i=1}^{l} m_i = k$ (la suma de las multiplicidades de las distintas raíces es igual al número total de raíces). Por simplicidad, normalmente etiquetaremos las constantes mediante $c_1, c_2, \ldots, c_k$ en lugar de utilizar dos índices.

Ejemplo 3 Considere la recurrencia

$$t_n = \begin{cases} n & \text{si } n = 0, 1 \text{ o } 2 \\ 5t_{n-1} - 8t_{n-2} + 4t_{n-3} & \text{en otro caso} \end{cases}$$

Primero reescribimos la recurrencia.

$$t_n - 5t_{n-1} + 8t_{n-2} - 4t_{n-3} = 0.$$

Su polinomio característico es

$$x^3 - 5x^2 + 8x - 4 = (x-1)(x-2)^2.$$

Por lo tanto las raíces son $r_1 = 1$ de multiplicidad $m_1 = 1$ y $r_2 = 2$ de multiplicidad $m_2 = 2$ y la solución general es

$$t_n = c_1 1^n + c_2 2^n + c_3 n 2^n.$$

Las condiciones iniciales dan

$$\begin{aligned} c_1 + c_2 &= 0 \quad & n = 0 \\ c_1 + 2c_2 + 2c_3 &= 1 \quad & n = 1 \\ c_1 + 4c_2 + 8c_3 &= 2 \quad & n = 2 \end{aligned}$$

Solucionando estas ecuaciones obtenemos $c_1 = -2, c_2 = 2$ y $c_3 = -\frac{1}{2}$. Por lo tanto la solución de la recurrencia es

$$t_n = 2^{n+1} - n2^{n-1} - 2.$$

Ejercicio 23 Resuelva las siguientes recurrencias:

1. $$t_n = \begin{cases} 1 & \text{si } n = 0 \\ 0 & \text{si } n = 1 \\ 5t_{n-1} - 6t_{n-2} & \text{en otro caso} \end{cases}$$

2. $$t_n = \begin{cases} 6 & \text{si } n = 0 \\ 8 & \text{si } n = 1 \\ 4t_{n-1} - 4t_{n-2} & \text{en otro caso} \end{cases}$$

3.
$$t_n = \begin{cases} 0 & \text{si } n = 0 \\ 1 & \text{si } n = 1 \\ -4t_{n-1} - 4t_{n-2} & \text{en otro caso} \end{cases}$$

4.
$$t_n = \begin{cases} 0 & \text{si } n = 0 \\ 4 & \text{si } n = 1 \\ 4t_{n-2} & \text{en otro caso} \end{cases}$$

5.
$$t_n = \begin{cases} 3 & \text{si } n = 0 \\ 6 & \text{si } n = 1 \\ t_{n-1} + 6t_{n-2} & \text{en otro caso} \end{cases}$$

6.
$$t_n = \begin{cases} 3 & \text{si } n = 0 \\ -3 & \text{si } n = 1 \\ -6t_{n-1} - 9t_{n-2} & \text{en otro caso} \end{cases}$$

7.
$$t_n = \begin{cases} 9n^2 - 15n + 106 & \text{si } n = 0, 1 \text{ o } 2 \\ t_{n-1} + 2t_{n-2} - 2t_{n-3} & \text{en otro caso} \end{cases}$$

8.
$$t_n = \begin{cases} n & \text{si } n = 0, 1 \text{ o } 2 \\ t_{n-1} + t_{n-3} - t_{n-4} & \text{en otro caso} \end{cases}$$

5.3.2. Recurrencias no homogéneas

La solución de una recurrencia lineal con coeficientes constantes se vuelve más difícil cuando la recurrencia no es homogénea, es decir, cuando la combinación lineal no es igual a cero. En particular, ya no es cierto que cualquier combinación lineal de soluciones sea una solución. Empezamos con un caso sencillo. Considere la siguiente recurrencia.

$$a_0t_n + a_1t_{n-1} + \cdots + a_kt_{n-k} = b^np(n) \tag{5.5}$$

El lado izquierdo es el mismo que antes, pero en el lado derecho tenemos $b^np(n)$, donde

- b es una constante y
- $p(n)$ es un polinomio en n de grado d.

Ejemplo 4 Considere la recurrencia

$$t_n - 2t_{n-1} = 3^n. \tag{5.6}$$

En este caso, $b = 3$ y $p(n) = 1$, un polinomio de grado 0. Un poco de astucia nos permite reducir este ejemplo al caso homogéneo que conocemos. Para ver esto, primero multiplique la recurrencia por -3, obteniendo

$$-3t_n + 6t_{n-1} = -3^{n+1}.$$

Ahora reemplace n por $n-1$ en esta recurrencia para obtener

$$-3t_{n-1} + 6t_{n-2} = -3^n. \tag{5.7}$$

Finalmente, si sumamos (5.6) y (5.7) tenemos

$$t_n - 5t_{n-1} + 6t_{n-2} = 0. \tag{5.8}$$

la cual se puede resolver por el método de la subsección 5.3.1. El polinomio característico es

$$x^2 - 5x + 6 = (x-2)(x-3)$$

y por lo tanto todas las soluciones son de la forma

$$t_n = c_1 2^n + c_2 3^n. \tag{5.9}$$

Sin embargo, ya no es cierto que una elección arbitraria de las constantes c_1 y c_2 en la ecuación (5.9) produzca una solución a la recurrencia incluso cuando no se tengan en cuenta las condiciones iniciales.

Peor: incluso las soluciones básicas $t_n = 2^n$ y $t_n = 3^n$, que son por supuesto soluciones a la ecuación (5.8), no son soluciones a la recurrencia original dada por la ecuación (5.6). ¿Qué está pasando?

La explicación es que las ecuaciones (5.6) y (5.8) no son equivalentes: la ecuación (5.8) se puede resolver dando valores arbitrarios para t_0 y t_1 (las condiciones iniciales), mientras que nuestra ecuación (5.6) original implica que $t_1 = 2t_0 + 3$.

La solución general de la la recurrencia original se puede determinar como una función de t_0 resolviendo dos ecuaciones lineales con incógnitas c_1 y c_2.

$$\begin{aligned} c_1 + c_2 &= t_0 & n &= 0 \\ 2c_1 + 3c_2 &= 2t_0 + 3 & n &= 1 \end{aligned} \tag{5.10}$$

Solucionándolas obtenemos que $c_1 = t_0 - 3$ y $c_2 = 3$. Por lo tanto, la solución general es

$$t_n = (t_0 - 3)2^n + 3^{n+1}$$

y así $t_n \in \Theta(3^n)$ independientemente de la condición inicial.

Siempre que $t_0 \geq 0$, una demostración fácil por inducción matemática basada en la Ecuación (5.6) muestra que $t_n \geq 0$ para todo $n \geq 0$. Por lo tanto, es inmediato de la Ecuación (5.9) que $t_n \in O(3^n)$: no hay necesidad de resolver las constantes c_1 y c_2 para llegar a esta conclusión.

Sin embargo, esta ecuación por sí sola no es suficiente para concluir que $t_n \in \Theta(3^n)$ porque podría haber sido a priori que $c_2 = 0$. No obstante, resulta que el valor de c_2 se puede obtener directamente, sin necesidad de establecer el sistema de ecuaciones lineales (5.10).

Esto es suficiente para concluir que $t_n \in \Theta(3^n)$ cualquiera que sea el valor de t_0 (incluso si es negativo). Para ver esto, sustituya la Ecuación (5.9) en la recurrencia original.

$$\begin{aligned} 3^n &= t_n - 2t_{n-1} \\ &= (c_1 2^n + c_2 3^n) - 2(c_1 2^{n-1} + c_2 3^{n-1}) \\ &= c_2 3^{n-1}. \end{aligned}$$

Independientemente de la condición inicial, concluimos que $c_2 = 3$.

En los ejemplos que siguen, a veces establecemos un sistema de ecuaciones lineales para determinar todas las constantes que aparecen en la solución general, mientras que en otras ocasiones determinamos solo las constantes necesarias sustituyendo la solución general en la recurrencia original.

Ejemplo 5 Deseamos encontrar la solución general de la siguiente recurrencia.

$$t_n - 2t_{n-1} = (n + 5)3^n, n \geq 1. \tag{5.11}$$

La manipulación necesaria para transformarla en una recurrencia homogénea es ligeramente más complicada que con el Ejemplo 4. Debemos

1. escribir la recurrencia (5.11),
2. reemplazar n en la recurrencia (5.11) por $n-1$ y multiplicar por -6 y
3. reemplazar n en la recurrencia (5.11) por $n-2$ y multiplicar por 9.

Así obtenemos:

$$\begin{aligned} t_n - 2t_{n-1} \qquad\qquad & &=& (n+5)3^n \\ -\,6t_{n-1} & +12t_{n-2} &=& -\,6(n+4)3^{n-1} \\ & 9t_{n-2} - 18t_{n-3} &=& 9(n+3)3^{n-2} \end{aligned}$$

Sumando estas tres ecuaciones obtenemos una recurrencia homogénea

$$t_n - 8t_{n-1} + 21t_{n-2} - 18t_{n-3} = 0.$$

Su polinomio característico es

$$x^3 - 8x^2 + 21x - 18 = (x-2)(x-3)^2$$

y por lo tanto todas las soluciones son de la forma

$$t_n = c_1 2^n + c_2 3^n + c_3 n 3^n. \tag{5.12}$$

Una vez más, cualquier elección de valores para las constantes c_1, c_2 y c_3 en la Ecuación (5.12) proporciona una solución a la recurrencia homogénea, pero la recurrencia original impone restricciones a estas constantes porque requiere que $t_1 = 2t_0 + 18$ y $t_2 = 2t_1 + 63 = 4t_0 + 99$.

Por lo tanto, la solución general se encuentra resolviendo el siguiente sistema de ecuaciones lineales.

$$\begin{aligned} c_1 + c_2 &= t_0 & n&=0 \\ 2c_1 + 3c_2 + 3c_3 &= 2t_0 + 18 & n&=1 \\ 4c_1 + 9c_2 + 18c_3 &= 4t_0 + 99 & n&=2 \end{aligned}$$

Esto implica que $c_1 = t_0 - 9, c_2 = 9$ y $c_3 = 3$. Por lo tanto, la solución general a la Ecuación 5.11 es $t_n = (t_0 - 9)2^n + (n+3)3^{n+1}$ y así $t_n \in \Theta(n3^n)$ independientemente de la condición inicial.

Alternativamente, podemos sustituir la Ecuación 5.12 en la recurrencia original. Después de una simple manipulación, obtenemos

$$(n+5)3^n = c_3 n 3^{n-1} + (2c_3 + c_2)3^{n-1}.$$

Igualando los coeficientes de $n3^n$ obtenemos que $c_3 = 3$. El hecho de que c_3 es estrictamente positivo es suficiente para establecer el orden exacto de t_n, sin necesidad de resolver las otras constantes. No obstante, ya que se conoce c_3, el valor de c_2 es fácil de obtener igualando los coeficientes de 3^n, así $c_2 = 9$.

Mirando hacia atrás en los Ejemplos 4 y 5, vemos que parte del polinomio característico proviene del lado izquierdo de la Ecuación 4.10 y el resto del lado derecho. La parte que viene del lado izquierdo es exactamente como si la ecuación hubiera sido homogénea: $(x - 2)$ para ambos ejemplos. La parte que viene del lado derecho es el resultado de nuestra manipulación.

Generalizando, podemos demostrar que para resolver la Ecuación 5.5 es suficiente usar el siguiente polinomio característico.

$$(a_0 x^k + a_1 x^{k-1} + \cdots + a_k)(x - b)^{d+1}$$

Recuerde que d es el grado del polinomio $p(n)$. Una vez obtenido este polinomio, proceda como en el caso homogéneo, excepto que algunas de las ecuaciones necesarias para determinar las constantes no se obtienen de las condiciones iniciales sino de la recurrencia en sí.

Ejemplo 6 El número de movimientos requeridos por un anillo en el problema de las Torres de Hanoi es

$$t(m) = \begin{cases} 0 & \text{si } m = 0 \\ 2t(m-1) + 1 & \text{en otro caso} \end{cases}$$

Esto se puede reescribir como

$$t(m) - 2t(m-1) = 1, \tag{5.13}$$

la cual es de la forma de la Ecuación (5.5) con $b = 1$ y $p(n) = 1$, un polinomio de grado 0. Por lo tanto su polinomio característico es

$$(x - 2)(x - 1)$$

donde el factor $(x-2)$ proviene del lado izquierdo de la Ecuación (5.13) y el factor $(x-1)$ del lado derecho. Las raíces de este polinomio son 1 y 2, ambos de multiplicidad 1, por lo que todas las soluciones de esta recurrencia son de la forma

$$t(m) = c_1 1^m + c_2 2^m. \tag{5.14}$$

Necesitamos dos condiciones iniciales para determinar las constantes c_1 y c_2. Sabemos que $t(0) = 0$; para encontrar la segunda condición inicial usamos la recurrencia misma para calcular

$$t(1) = 2t(0) + 1 = 1.$$

Esto nos da dos ecuaciones lineales con las constantes desconocidas.

$$\begin{aligned} c_1 + c_2 &= 0 \quad m = 0 \\ c_1 + 2c_2 &= 1 \quad m = 1 \end{aligned}$$

De donde obtenemos las soluciones $c_1 = -1$ y $c_2 = 1$ y por lo tanto

$$t(m) = 2^m - 1$$

Si todo lo que queremos es determinar el orden exacto de $t(m)$, no es necesario calcular las constantes de la Ecuación (5.14). Esta vez ni siquiera necesitamos sustituir la Ecuación (5.14) en la recurrencia original. Saber que $t(m) = c_1 + c_2 2^m$ es suficiente para concluir que $c_2 > 0$ y, por tanto, $t(m) \in \Theta(2^m)$. Para esto, tenga en cuenta que $t(m)$, el número de movimientos requeridos por un anillo, ciertamente no es negativo ni constante, ya que claramente $t(m) \geq m$.

Ejemplo 7 Considere la recurrencia

$$t_n = 2t_{n-1} + n$$

Se puede reescribir como

$$t_n - 2t_{n-1} = n$$

la cual es de la forma de la Ecuación (5.5) con $b = 1$ y $p(n) = n$, un polinomio de grado 1. Así, su polinomio característico es

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

con raíces 2 (de multiplicidad 1) y 1 (de multiplicidad 2). Por lo tanto todas las soluciones son de la forma

$$t_n = c_1 2^n + c_2 1^n + c_3 n 1^n. \tag{5.15}$$

Siempre que $t_0 \geq 0$, y por lo tanto $t_n \geq 0$ para todo n, concluimos inmediatamente que $t_n \in O(2^n)$. Pero se requiere un análisis más detallado para afirmar que $t_n \in \Theta(2^n)$.

Si sustituimos la Ecuación (5.15) en la recurrencia original, obtenemos

$$\begin{aligned} n &= t_n - 2t_{n-1} \\ &= (c_1 2^n + c_2 + c_3 n) - 2(c_1 2^{n-1} + c_2 + c_3(n-1)) \\ &= (2c_3 - c_2) - c_3 n \end{aligned}$$

de donde leemos directamente que $2c_3 - c_2 = 0$ y $-c_3 = 1$, independientemente de la condición inicial. Esto implica que $c_3 = -1$ y $c_2 = -2$. Al principio estamos decepcionados porque es c_1 lo que es relevante para determinar el orden exacto de t_n como lo indica la ecuación (5.15), y en su lugar obtuvimos las otras dos constantes. Sin embargo, esas constantes convierten la Ecuación (5.15) en

$$t_n = c_1 2^n - n - 2. \tag{5.16}$$

Siempre que $t_0 \geq 0$, y por lo tanto $t_n \geq 0$ para todo n, la Ecuación (5.16) implica que c_1 debe ser estrictamente positivo. Por lo tanto, tenemos derecho a concluir que $t_n \in \Theta(2^n)$ sin necesidad de resolver explícitamente para c_1. Por supuesto, c_1 ahora se puede obtener fácilmente a partir de la condición inicial si así se desea.

Alternativamente, las tres constantes se pueden determinar como funciones de t_0 estableciendo y resolviendo el sistema apropiado de ecuaciones lineales obtenido de la Ecuación (5.15) y los valores de t_1 y t_2 calculados a partir de la recurrencia original.

A estas alturas, puede estar convencido de que, para todos los propósitos prácticos, no hay necesidad de preocuparse por las constantes: el orden exacto de t_n siempre se puede leer directamente de la solución general. ¡Incorrecto!

O tal vez piense que las constantes obtenidas por la técnica más simple de sustituir la solución general en la recurrencia original son siempre suficientes para determinar su orden exacto. ¡Nuevamente incorrecto! Considere el siguiente ejemplo.

Ejemplo 8 Considere la recurrencia

$$t_n = \begin{cases} 1 & \text{si } n = 0 \\ 4t_{n-1} - 2^n & \text{en otro caso} \end{cases}$$

Primero reescribimos la recurrencia.

$$t_n - 4t_{n-1} = -2^n$$

la cual es de la forma de la Ecuación (5.5) con $b = 2$ y $p(n) = -1$, un polinomio de grado 0. Su polinomio característico es

$$(x-4)(x-2)$$

con raíces 4 y 2, ambas de multiplicidad 1. Por lo tanto todas las soluciones son de la forma

$$t_n = c_1 4^n + c_2 2^n. \tag{5.17}$$

Puede sentirse tentado a afirmar sin más preámbulos que $t_n \in \Theta(4^n)$ ya que ese es claramente el término dominante en la Ecuación (5.17).

Si no tiene prisa, es posible que desee sustituir la Ecuación (5.17) en la recurrencia original para ver qué sale.

$$\begin{aligned} -2^n &= t_n - 4t_{n-1} \\ &= c_1 4^n + c_2 2^n - 4(c_1 4^{n-1} + c_2 2^{n-1}) \\ &= -c_2 2^n \end{aligned}$$

Por tanto, $c_2 = 1$ independientemente de la condición inicial. El conocimiento de c_2 no es directamente relevante para determinar el orden exacto de t_n, como lo indica la Ecuación (5.17). Sin embargo, a diferencia del ejemplo anterior, no se puede afirmar nada concluyente sobre c_1 por el mero hecho de que c_2 sea positivo.

Incluso si hubiéramos encontrado que c_2 es negativo, todavía no podríamos concluir inmediatamente nada con respecto a c_1 porque esta vez no hay ninguna razón obvia para creer que t_n deba ser positivo.

Habiendo fallado todo lo demás, nos vemos obligados a determinar todas las constantes. Podríamos establecer el sistema habitual de dos ecuaciones lineales con dos incógnitas obtenidas de la Ecuación (5.17) y los valores de t_0 y t_1, pero ¿por qué desechar el conocimiento que ya hemos adquirido sobre el valor de c_2? Sabemos que $t_n = c_1 4^n + 2^n$. Sustituyendo la condición inicial $t_0 = 1$ da como resultado $1 = c_1 + 1$ y por lo tanto $c_1 = 0$.

La conclusión es que la solución exacta para la recurrencia es simplemente $t_n = 2^n$, y que la afirmación anterior de que $t_n \in \Theta(4^n)$ ¡era incorrecta! Sin embargo, tenga en cuenta que t_n estaría en $\Theta(4^n)$ si se especificara cualquier valor mayor de t_0 como condición inicial, ya que en general $c_1 = t_0 - 1$.

Por otro lado, con una condición inicial $t_0 < 1$, t_n toma valores negativos que crecen exponencialmente rápido. Este ejemplo ilustra la importancia de la condición inicial para algunas recurrencias, mientras que los ejemplos

anteriores habían demostrado que el comportamiento asintótico de muchas recurrencias no se ve afectado por la condición inicial, al menos cuando $t_0 \geq 0$.

Una generalización adicional del mismo tipo de argumento nos permite finalmente resolver recurrencias de la forma

$$a_0 t_n + a_1 t_{n-1} + \cdots + a_k t_{n-k} = b_1^n p_1(n) + b_2^n p_2(n) + \cdots \tag{5.18}$$

donde las b_i son constantes distintas y $p_i(n)$ son polinomios en n respectivamente de grado d_i. Dichas recurrencias se resuelven utilizando el siguiente polinomio característico:

$$(a_0 x^k + a_1 x^{k-1} + \cdots + a_k)(x - b_1)^{d_1+1}(x - b_2)^{d_2+1} \cdots ,$$

que contiene un factor correspondiente al lado izquierdo y un factor correspondiente a cada término del lado derecho. Una vez obtenido el polinomio característico, la recurrencia se resuelve como antes.

Ejemplo 9 Considere la recurrencia

$$t_n = \begin{cases} 0 & \text{si } n = 0 \\ 2t_{n-1} + n + 2^n & \text{en otro caso} \end{cases}$$

Primero reescriba la recurrencia como

$$t_n - 2t_{n-1} = n + 2^n,$$

que tiene la forma de la Ecuación (5.18) con $b_1 = 1, p_1(n) = n, b_2 = 2$ y $p_2(n) = 1$. El grado de $p_1(n)$ es $d_1 = 1$ y el grado de $p_2(n)$ es $d_2 = 0$. Su polinomio característico es

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

que tiene raíces 1 y 2, ambas de multiplicidad 2. Todas las soluciones de la recurrencia tienen, por tanto, la forma

$$t_n = c_1 1^n + c_2 n 1^n + c_3 2^n + c_4 n 2^n. \tag{5.19}$$

Concluimos de esta ecuación que $t_n \in O(n2^n)$ sin calcular las constantes, pero necesitamos saber si $C_4 > 0$ para determinar el orden exacto de t_n. Para esto, sustituya la Ecuación 5.19 en la recurrencia original, que da

$$n + 2^n = (2c_2 - c_1) - c_2 n + c_4 2^n.$$

Al igualar los coeficientes de 2^n, obtenemos inmediatamente que $c_4 = 1$ y, por lo tanto, $t_n \in \Theta(n2^n)$. Las constantes c_1 y c_2 son igualmente fáciles de leer si se desea. La constante c_3 puede obtenerse de la ecuación (5.19), el valor de las otras constantes, y la condición inicial $t_0 = 0$.

Alternativamente, las cuatro constantes se pueden determinar resolviendo cuatro ecuaciones lineales con cuatro incógnitas. Como necesitamos cuatro ecuaciones y solo tenemos una condición inicial, usamos la recurrencia para calcular el valor de otros tres puntos: $t_1 = 3, t_2 = 12$ y $t_3 = 35$. Esto da lugar al siguiente sistema.

$$\begin{aligned} c_1 + c_3 &= 0 & n &= 0 \\ c_1 + c_2 + 2c_3 + 2c_4 &= 3 & n &= 1 \\ c_2 + 2c_2 + 4c_3 + 8c_4 &= 12 & n &= 2 \\ c_1 + 3c_2 + 8c_3 + 24c_4 &= 35 & n &= 3 \end{aligned}$$

Resolviendo este sistema se obtiene $c_1 = -2, c_2 = -1, c_3 = 2$ y $c_4 = 1$. Por lo tanto, finalmente obtenemos

$$t_n = n2^n + 2^{n+1} - n - 2.$$

Ejercicio 24 Resuelva las siguientes recurrencias exactamente con y sin manipulación algebraica.

1. $$t_n = \begin{cases} n+1 & \text{si } n = 0 \text{ o } n = 1 \\ 3t_{n-1} - 2t_{n-2} + 3 \cdot 2^{n-2} & \text{en otro caso} \end{cases}$$

2. $$T(n) = \begin{cases} a & \text{si } n = 0 \text{ o } n = 1 \\ T(n-1) + T(n-2) + c & \text{en otro caso} \end{cases}$$

3. $$T(n) = \begin{cases} a & \text{si } n = 0 \text{ o } n = 1 \\ T(n-1) + T(n-2) + cn & \text{en otro caso} \end{cases}$$

4. $a_n = a_{n-1} + n + 2, a_0 = 0$

5. $a_n = a_{n-1} + 7n, a_0 = 0$

5.3.3. Cambio de variable

A veces es posible resolver recurrencias más complicadas haciendo un cambio de variable. En los siguientes ejemplos, escribimos $T(n)$ para el término de una recurrencia general y t_i para el término de una nueva recurrencia obtenida de la primera por un cambio de variable. Asegúrese de estudiar el ejemplo 13 que se encuentra entre las recurrencias más importantes para propósitos algorítmicos.

Ejemplo 10 Considere la siguiente recurrencia donde n es una potencia de 2.

$$T(n) = \begin{cases} 1 & \text{si } n = 1 \\ 3T(n/2) + n & \text{si } n \text{ es una potencia de } 2 \end{cases}$$

Para transformar esto en una forma que sepamos resolver, reemplazamos n por 2^i. Esto se logra introduciendo una nueva recurrencia t_i, definida por $t_i = T(2^i)$. Esta transformación es útil porque $n/2$ se convierte en $2^i/2 = 2^{i-1}$. En otras palabras, nuestra recurrencia original en la que $T(n)$ se define como una función de $T(n/2)$ da paso a una en la que t_i se define como una función de t_{i-1}, precisamente el tipo de recurrencias que hemos aprendido a resolver.

$$\begin{aligned} t_i = T(2^i) &= 3T(2^{i-1}) + 2^i \\ &= 3t_{i-1} + 2^i. \end{aligned}$$

Una vez que se reescribe como

$$t_i - 3t_{i-1} = 2^i$$

esta recurrencia es de la forma de la Ecuación 5.5. Su polinomio caracrerístico es

$$(x-3)(x-2)$$

y así todas las soluciones para t_i son de la forma

$$t_i = c_1 3^i + c_2 2^i.$$

Usamos el hecho de que $T(2^i) = t_i$ y así que $T(n) = t_{\lg n}$ cuando $n = 2^i$ para obtener

$$\begin{aligned} T(n) &= c_1 3^{\lg n} + c_2 2^{\lg n} \\ &= c_1 n^{\lg 3} + c_2 n \end{aligned} \tag{5.20}$$

cuando n es una potencia de 2, lo cual es suficiente para concluir que

$$T(n) \in O(n^{\lg 3} | n \text{ es potencia de 2}).$$

Sin embargo, debemos demostrar que c_1 es estrictamente positivo antes de poder afirmar algo sobre el orden exacto de $T(n)$.

Ahora estamos familiarizados con dos técnicas para determinar las constantes. En aras de la didáctica, apliquemos cada una de ellas a esta situación. El enfoque más directo, que no siempre proporciona la información deseada, es sustituir la solución proporcionada por la Ecuación (5.20) en la recurrencia original. Teniendo en cuenta que $(1/2)^{lg3} = 1/3$, esto produce

$$\begin{aligned} n &= T(n) - 3T(n/2) \\ &= (c_1 n^{\lg 3} + c_2 n) - 3(c_1 (n/2)^{\lg 3} + c_2 (n/2)) \\ &= -c_2 (n/2) \end{aligned}$$

y por lo tanto $c_2 = -2$. Aunque no obtuvimos el valor de c_1, que es la constante más relevante, estamos en condiciones de afirmar que debe ser estrictamente positiva, porque de lo contrario la Ecuación (5.20) implicaría falsamente que $T(n)$ es negativa. El hecho de que

$$T(n) \in \Theta(n^{\lg 3} | n \text{ es potencia de 2}) \tag{5.21}$$

queda así establecido. Por supuesto, el valor de c_1 ahora sería fácil de obtener de la Ecuación (5.20), el hecho de que $c_2 = -2$ y la condición inicial $T(1) = 1$, pero esto no es necesario si estamos satisfechos con resolver la recurrencia en notación asintótica. Además, hemos aprendido que la Ecuación (5.21) se cumple independientemente de la condición inicial, siempre que $T(n)$ sea positivo.

El enfoque alternativo consiste en establecer dos ecuaciones lineales en las dos incógnitas c_1 y c_2. Se garantiza que producirá el valor de ambas constantes. Para esto, necesitamos el valor de $T(n)$ en dos puntos. Ya sabemos que $T(1) = 1$. Para obtener otro punto, usamos la recurrencia en sí: $T(2) = 3T(1) + 2 = 5$. Sustituyendo $n = 1$ y $n = 2$ en la Ecuación 5.20 se obtiene el siguiente sistema.

$$\begin{aligned} c_1 + c_2 &= 1 \quad n = 1 \\ 3c_1 + 2c_2 &= 5 \quad n = 2 \end{aligned}$$

Resolviendo estas ecuaciones obtenemos $c_1 = 3$ y $c_2 = -2$. Por lo tanto

$$T(n) = 3n^{\lg 3} - 2n$$

cuando n es una potencia de 2.

Ejemplo 11 Considere la recurrencia

$$T(n) = 4T(n/2) + n^2$$

cuando n es una potencia de 2, $n \geq 2$.

Procedemos como en el ejemplo previo.

$$\begin{aligned} t_i = T(2^i) &= 4T(2^{i-1}) + (2^i)^2 \\ &= 4t_{i-1} + 4^i. \end{aligned}$$

Reescribimos la recurrencia en la forma de la Ecuación (5.5).

$$t_i - 4t_{i-1} = 4^i.$$

Su polinomio característico es $(x-4)^2$ y así todas las soluciones son de la forma

$$t_i = c_1 4^i + c_2 i 4^i.$$

En términos de $T(n)$, esto es

$$T(n) = c_1 n^2 + c_2 n^2 \lg n. \tag{5.22}$$

Sustituyendo la Ecuación (5.22) en la recurrencia original da

$$n^2 = T(n) - 4T(n/2) = c_2 n^2$$

y así $c_2 = 1$. Por lo tanto

$$T(n) \in \Theta(n^2 \lg n | n \text{ es potencia de } 2),$$

sin importar la condición inicial (aún si $T(1)$ es negativo).

Ejemplo 12 Considere la recurrencia

$$T(n) = 2T(n/2) + n \lg n$$

cuando n es una potencia de 2, $n \geq 2$.

Como antes obtenemos

$$\begin{aligned} t_i = T(2^i) &= 2T(2^{i-1}) + i2^i \\ &= 2t_{i-1} + i2^i. \end{aligned}$$

Escribimos esto en forma de la Ecuación (5.5).

$$t_i - 2t_{i-1} = i2^i.$$

Su polinomio característico es $(x-2)(x-2)^2 = (x-2)^3$ y así todas las soluciones son de la forma

$$t_i = c_1 2^i + c_2 i 2^i + c_3 i^2 2^i.$$

En términos de $T(n)$, esto es

$$T(n) = c_1 n + c_2 n \lg n + c_3 n \lg^2 n. \tag{5.23}$$

Sustituyendo la Ecuación (5.23) en la recurrencia original tenemos

$$n \lg n = T(n) - 2T(n/2) = (c_2 - c_3)n + 2c_3 n \lg n,$$

lo cual implica que $c_2 = c_3$ y $2c_3 = 1$, así $c_2 = c_3 = \frac{1}{2}$. Por lo tanto

$$T(n) \in \Theta(n \lg^2 n | n \text{ es potencia de } 2),$$

sin importar las condiciones iniciales.

Ejemplo 13 Ahora estamos listos para resolver una de las recurrencias más importantes para efectos algorítmicos. Esta recurrencia es particularmente útil para el análisis de algoritmos divide y vencerás, como veremos en el Capítulo 7. Las constantes $n_0 \geq 1, \ell \geq 1, b \geq 2$ y $k \geq 0$ son números enteros, mientras que c es un número real estrictamente positivo. Sea $T : \mathbb{N} \to \mathbb{R}^+$ una función eventualmente no decreciente tal que

$$T(n) = \ell T(n/b) + cn^k \quad n > n_0 \tag{5.24}$$

cuando n/n_0 es una potencia exacta de b, esto es cuando $n \in \{bn_0, b^2 n_0, b^3 n_0, \ldots\}$.

En esta ocasión, el cambio de variable apropiado es $n = b^i n_0$.

$$\begin{aligned} t_i = T(b^i n_0) &= \ell T(b^{i-1} n_0) + c(b^i n_0)^k \\ &= \ell t_{i-1} + c n_0^k b^{ik} \end{aligned}$$

La escribimos en la forma de la Ecuación (5.5).

$$t_i - \ell t_{i-1} = (c n_0^k)(b^k)^i.$$

El lado derecho tiene la forma requerida $a^i p(i)$ donde $p(i) = c n_0^k$ es un polinomio constante (de grado 0) y $a = b^k$. Por tanto, el polinomio característico es $(x - \ell)(x - b^k)$ cuyas raíces son ℓ y b^k. A partir de esto, es tentador (¡pero falso en general!) Concluir que todas las soluciones son de la forma

$$t_i = c_1 \ell^i + c_2 (b^k)^i. \tag{5.25}$$

Para escribir esto en términos de $T(n)$, observe que $i = \log_b(n/n_0)$ cuando n tiene la forma adecuada y, por lo tanto, $d^i = (n/n_0)^{\log_b d}$ para valores positivos arbitrarios de d. Por lo tanto,

$$\begin{aligned} T(n) &= (c_1 / n_0^{\log_b \ell}) n^{\log_b \ell} + (c_2 / n_0^k) n^k \\ &= c_3 n^{\log_b \ell} + c_4 n^k, \end{aligned} \tag{5.26}$$

para las nuevas constantes apropiadas c_3 y c_4. Para conocer estas constantes, sustituimos la Ecuación 5.26 en la recurrencia original.

$$\begin{aligned} c n^k &= T(n) - \ell T(n/b) \\ &= c_3 n^{\log_b \ell} + c_4 n^k - \ell (c_3 (n/b)^{\log_b \ell} + c_4 (n/b)^k) \\ &= \left(1 - \frac{\ell}{b^k}\right) c_4 n^k. \end{aligned}$$

Por lo tanto $c_4 = c/(1 - \ell/b^k)$. Para expresar $T(n)$ en notación asintótica, necesitamos mantener sólo el término dominante en la Ecuación (5.26). Hay tres casos a considerar, dependiendo de si ℓ es menor, mayor o igual que b^k.

- Si $\ell < b^k$ entonces $c_4 > 0$ y $k > \log_b \ell$. Por lo tanto el término $c_4 n^k$ domina la Ecuación (5.26). Así, $T(n) \in \Theta(n^k | (n/n_0)$ es una potencia de $b)$.

Pero n^k es una función suave[2] y $T(n)$ por suposición es una función eventualmente no decreciente. Por lo tanto $T(n) \in \Theta(n^k)$.

- Si $\ell > b^k$ entonces $c_4 < 0$ y $\log_b \ell > k$. El hecho de que c_4 sea negativa implica que c_3 es positiva, porque de otra manera la Ecuación (5.26) implicaría que $T(n)$ es negativa, contrario a la especificación de que $T : \mathbb{N} \to \mathbb{R}^+$. Así, el término $c_3 n^{\log_b \ell}$ domina la Ecuación (5.26). Aún más, $n^{\log_b \ell}$ es una función suave y $T(n)$ eventualmente no es decreciente. Por lo tanto $T(n) \in \Theta(n^{\log_b \ell})$.

- Si $\ell = b^k$ estamos en problemas ¡ya que la fórmula para c_4 involucra una división por cero! Lo que está mal es que en este caso el polinomio característico tiene una sola raíz de multiplicidad 2 en lugar de dos raíces distintas. Por lo tanto la Ecuación (5.25) no proporciona la solución general a la recurrencia. Por lo que la solución general es este caso es

$$t_i = c_5 (b^k)^i + c_6 i (b^k)^i.$$

En términos de $T(n)$, esto es

$$T(n) = c_7 n^k + c_8 n^k \log_b (n/n_0), \tag{5.27}$$

para constantes apropiadas c_7 y c_8. Sustituyendo en la recurrencia original, nuestra manipulación usual nos da un valor sorprendentemente simple de $c_8 = c$. Por lo tanto, $cn^k \log_b n$ es el término dominante en la Ecuación (5.27) ya que desde el inicio del ejemplo se asumió que c era estrictamente positiva. Ya que $n^k \log n$ es suave y $T(n)$ eventualmente es no decreciente concluimos que $T(n) \in \Theta(n^k \log n)$.

Poniendo todo junto,

$$T(n) \in \begin{cases} \Theta(n^k) & \text{si } \ell < b^k \\ \Theta(n^k \log n) & \text{si } \ell = b^k \\ \Theta(n^{\log_b \ell}) & \text{si } \ell > b^k \end{cases} \tag{5.28}$$

[2] Una función $f : \mathbb{N} \to \mathbb{R}^{\geq 0}$ es eventualmente no decreciente si existe un umbral entero n_0 tal que $f(n) \leq f(n+1)$ para todo $n \geq n_0$. Esto implica por inducción matemática que $f(n) \leq f(n+1)$ siempre que $m \geq n \geq n_0$. Sea $b \geq 2$ cualquier entero. La función f es **suave**$-b$ si además de ser eventualmente no decreciente, satisface la condición $f(bn) \in O(f(n))$. En otras palabras, existe una constante c (que depende de b) tal que $f(bn) \leq cf(n)$ para toda $n \geq n_0$ (no hay pérdida de generalidad si se usa el umbral n_0 para ambos propósitos). Una función es **suave** si es suave$-b$ para todo entero $b \geq 2$.

Ejercicio 25 Resuelva las siguientes recurrencias exactamente, primero mediante cambio de variable, luego verifique sus respuestas empleando el resultado del Ejemplo 13.

1. Cuando n es una potencia de 2.

   a)
   $$T(n) = \begin{cases} 1 & \text{si } n = 1 \\ 2T(n/2) + 1 & \text{en otro caso} \end{cases}$$

   b)
   $$T(n) = \begin{cases} 1 & \text{si } n = 1 \\ 4T(n/2) + n & \text{en otro caso} \end{cases}$$

   c)
   $$T(n) = \begin{cases} 1 & \text{si } n = 1 \\ 2T(n/2) + \lg n & \text{en otro caso} \end{cases}$$

   d)
   $$T(n) = \begin{cases} 1 & \text{si } n = 1 \\ 5T(n/2) + (n \lg n)^2 & \text{en otro caso} \end{cases}$$

2. Cuando n es una potencia de 3.

   a)
   $$T(n) = \begin{cases} 1 & \text{si } n = 1 \\ 2T(n/3) + 4 & \text{en otro caso} \end{cases}$$

   b)
   $$T(n) = \begin{cases} 1 & \text{si } n = 1 \\ T(n/3) + 2 & \text{en otro caso} \end{cases}$$

3. Cuando n es de la forma 2^{2^k}.
   $$T(n) = \begin{cases} 1 & \text{si } n = 2 \\ 2T(\sqrt{n}) + \lg n & \text{en otro caso} \end{cases}$$

4. Investigue qué es el teorema maestro y haga un análisis de éste comparándolo con lo que hemos visto.

Capítulo 6

Algoritmos voraces

Los algoritmos voraces son la primera familia de algoritmos que examinamos en detalle, la razón es simple: generalmente son los más sencillos. Como sugiere su nombre, su enfoque es miope y toman decisiones sobre la base de la información que tienen a mano sin preocuparse por el efecto que estas decisiones puedan tener en el futuro.

Por lo tanto, son fáciles de inventar, fáciles de implementar y, cuando funcionan, son eficientes. Sin embargo, dado que el mundo rara vez es tan simple, muchos problemas no pueden resolverse correctamente con un enfoque tan crudo.

Los algoritmos voraces se utilizan normalmente para resolver problemas de optimización. Los ejemplos que aparecen más adelante en este capítulo incluyen encontrar la ruta más corta de un nodo a otro a través de una red, o encontrar el mejor orden para ejecutar un conjunto de trabajos en una computadora.

En tal contexto, un algoritmo voraz funciona eligiendo el arco, o el trabajo, que parece más prometedor en cualquier momento; nunca reconsidera esta decisión, cualquiera que sea la situación que pueda surgir posteriormente. No es necesario evaluar alternativas ni emplear procedimientos de contabilidad elaborados que permitan deshacer decisiones anteriores. Comenzamos el capítulo con un ejemplo cotidiano en el que esta táctica funciona bien.

6.1. Dar cambio

Supongamos que vivimos en un país donde están disponibles las siguientes monedas: dólares (100 centavos), cuartos (25 centavos), dieces (10 centavos), cincos (5 centavos) y centavos (1 centavo).

Nuestro problema es diseñar un algoritmo para pagar una cantidad determinada a un cliente utilizando la menor cantidad posible de monedas. Por ejemplo, si debemos pagar \$2.89 (289 centavos), la mejor solución es darle al cliente 10 monedas: 2 dólares, 3 cuartos, 1 diez y 4 centavos.

La mayoría de nosotros resolvemos este tipo de problemas todos los días sin pensarlo dos veces, inconscientemente usando un algoritmo voraz obvio: comenzando con nada, en cada etapa agregamos a las monedas ya elegidas una moneda del mayor valor disponible que no sobrepase la cantidad a pagar.

El algoritmo se puede formalizar como sigue.

```
function MAKE-CHANGE(n)
 1  const C = {100, 25, 10, 5, 1}
 2  S ← ∅
 3  s ← 0
 4  while s ≠ n
 5  do x ← el elemento más grande en C tal que s + x ≤ n
 6      if no existe tal elemento
 7         then return "no se encontró solución"
 8      S ← S ∪ { una moneda de valor x }
 9      s ← s + x
10  return S
```

Es fácil convencerse a uno mismo (pero sorprendentemente difícil de probar formalmente) de que con los valores dados para las monedas, y siempre que se disponga de un suministro adecuado de cada denominación, este algoritmo siempre produce una solución óptima a nuestro problema.

Sin embargo, con una serie de valores diferentes, o si el suministro de algunas de las monedas es limitado, es posible que el algoritmo voraz no funcione; consulte los problemas 6.2 y 6.4.

En algunos casos, puede elegir un conjunto de monedas que no es óptimo (es decir, el conjunto contiene más monedas de las necesarias), mientras que en otros puede no encontrar una solución aunque exista (si bien esto no puede suceder si se tiene una cantidad ilimitada de monedas de 1 unidad).

El algoritmo es "voraz"porque en cada paso elige la moneda más grande que puede, sin preocuparse de si esta será una decisión acertada a largo plazo. Además, nunca cambia de opinión: una vez que se ha incluido una moneda en la solución, ésta seguirá ahí para siempre. Como explicaremos en la siguiente sección, estas son las características de esta familia de algoritmos.

Para el problema particular de dar cambio, en el capítulo 8 se describe un algoritmo completamente diferente. Este algoritmo alternativo usa programación dinámica.

El algoritmo de programación dinámica siempre funciona, mientras que el algoritmo voraz puede fallar; sin embargo, es menos sencillo que el algoritmo voraz y (cuando ambos algoritmos funcionan) menos eficiente.

6.2. Características generales de los algoritmos voraces

Por lo general, los algoritmos voraces y los problemas que pueden resolver se caracterizan por la mayoría o todas las características siguientes.

- Tenemos algún problema que solucionar de forma óptima. Para construir la solución de nuestro problema, tenemos un conjunto (o lista) de candidatos: las monedas que están disponibles, las aristas de un grafo que pueden usarse para construir una ruta, el conjunto de trabajos a programar, o lo que sea.

- A medida que avanza el algoritmo, acumulamos otros dos conjuntos. Uno contiene candidatos que ya han sido considerados y elegidos, mientras que el otro contiene candidatos que han sido considerados y rechazados.

- Existe una función que comprueba si un conjunto particular de candidatos proporciona una solución a nuestro problema, ignorando las cuestiones de optimización por el momento. Por ejemplo, ¿las monedas que hemos elegido se suman a la cantidad a pagar? ¿Las aristas seleccionadas proporcionan una ruta al nodo que deseamos alcanzar? ¿Se han programado todos los trabajos?

- Una segunda función comprueba si un conjunto de candidatos es factible, es decir, si es posible o no completar el conjunto añadiendo más

candidatos para obtener al menos una solución a nuestro problema. Aquí tampoco nos preocupamos por el momento de la optimización. Por lo general, esperamos que el problema tenga al menos una solución que se pueda obtener utilizando candidatos del conjunto inicialmente disponible.

- Otra función más, la función de selección, indica en cualquier momento cuál de los candidatos restantes, que no han sido elegidos ni rechazados, es el más prometedor.

- Finalmente, una función objetivo da el valor de la solución que hemos encontrado: la cantidad de monedas que usamos para dar el cambio, la longitud del camino que construimos, el tiempo necesario para procesar todos los trabajos en el programa, o cualquier otro valor que estemos tratando de optimizar. A diferencia de las tres funciones mencionadas anteriormente, la función objetivo no aparece explícitamente en el algoritmo voraz.

Para resolver nuestro problema buscamos un conjunto de candidatos que constituya una solución, y que optimice (minimice o maximice, según sea el caso) el valor de la función objetivo. Un algoritmo voraz avanza paso a paso. Inicialmente, el conjunto de candidatos elegidos está vacío.

Luego, en cada paso, consideramos agregar a este conjunto el mejor candidato no probado restante, y nuestra elección se guía por la función de selección. Si el conjunto ampliado de candidatos elegidos ya no fuera factible, rechazamos al candidato que estamos considerando actualmente. En este caso, el candidato que ha sido juzgado y rechazado nunca se vuelve a considerar.

Sin embargo, si el conjunto ampliado aún es factible, agregamos el candidato actual al conjunto de candidatos elegidos, donde permanecerá de ahora en adelante. Cada vez que ampliamos el conjunto de candidatos elegidos, comprobamos si ahora constituye una solución a nuestro problema. Cuando un algoritmo voraz funciona correctamente, la primera solución que se encuentra de esta manera siempre es óptima.

```
function GREEDY(C : set)
1  S ← ∅
2  while C ≠ ∅ and not SOLUTION(S)
3  do x ← SELECT(x)
4       C ← C \ {x}
5       if FEASIBLE(S ∪ {x})
6          then S ← S ∪ {x}
7  if SOLUTION(S)
8     then return S
9     else  return  "no hay soluciones"
```

Está claro por qué tales algoritmos se denominan "voraces": en cada paso, el procedimiento elige el mejor bocado que puede tragar, sin preocuparse por el futuro. Nunca cambia de opinión: una vez que se incluye a un candidato en la solución, está ahí para siempre; una vez que se excluye a un candidato de la solución, nunca se reconsidera.

La función de selección suele estar relacionada con la función objetivo. Por ejemplo, si estamos tratando de maximizar nuestras ganancias, es probable que elijamos al candidato restante que tenga el valor individual más alto.

Si estamos tratando de minimizar el costo, entonces podemos seleccionar el candidato restante más barato, y así sucesivamente. Sin embargo, veremos que en ocasiones puede haber varias funciones de selección plausibles, por lo que tenemos que elegir la correcta si queremos que nuestro algoritmo funcione correctamente.

Volviendo por un momento al ejemplo de dar cambio, aquí hay una forma en que las características generales de los algoritmos voraces pueden equipararse con las características particulares de este problema.

- Los candidatos son un conjunto de monedas, representando en nuestro ejemplo 100, 25, 10, 5 y 1 unidades, con suficientes monedas de cada valor que nunca se nos acaban (sin embargo, el conjunto de candidatos debe ser finito).

- La función de solución comprueba si el valor de las monedas elegidas hasta ahora es exactamente el importe a pagar.

- Un juego de monedas es factible si su valor total no excede la cantidad a pagar.

- La función de selección elige la moneda de mayor valor que queda en el conjunto de candidatos.
- La función objetivo cuenta el número de monedas utilizadas en la solución.

Obviamente, es más eficiente rechazar todas las monedas de 100 unidades restantes (digamos) de una vez cuando la cantidad restante a representar cae por debajo de este valor.

Usar la división de enteros para calcular cuántas monedas de un valor particular se deben elegir, también es más eficiente que proceder mediante sustracciones sucesivas. Si se adopta alguna de estas tácticas, podemos relajar la condición de que el conjunto de monedas disponible debe ser finito.

6.3. Grafos: árboles de expansión mínimos

Sea $G = \langle N, A \rangle$ un grafo no dirigido y conectado donde N es el conjunto de nodos y A es el conjunto de aristas. Cada arista tiene una longitud no negativa dada.

El problema es encontrar un subconjunto T de las aristas de G tal que todos los nodos permanezcan conectados cuando sólo se usan las aristas en T y la suma de las longitudes de las aristas en T es lo más pequeña posible.

Dado que G está conectado, debe existir al menos una solución. Si G tiene aristas de longitud 0, entonces pueden existir varias soluciones cuya longitud total sea la misma pero que involucren diferentes números de aristas. En este caso, dadas dos soluciones con la misma longitud total, preferimos la que tiene menos aristas. Incluso con esta salvedad, el problema puede tener varias soluciones diferentes de igual valor.

En lugar de hablar de longitud, podemos asociar un costo a cada arista. El problema es entonces encontrar un subconjunto T de las aristas cuyo costo total sea lo más pequeño posible. Evidentemente, este cambio de terminología no afecta la forma en que resolvemos el problema.

Sea $G' = \langle N, T \rangle$ el grafo parcial formado por los nodos de G y las aristas en T y suponga que hay n nodos en N. Un grafo conectado con n nodos debe tener al menos $n - 1$ aristas, por lo que este es el número mínimo de aristas que puede haber en T.

Por otro lado, un grafo con n nodos y más de $n - 1$ aristas contiene al menos un ciclo; vea el problema 6.7 del libro de texto. Por tanto, si G' está

conectado y T tiene más de $n - 1$ aristas, podemos eliminar al menos una de estas sin desconectar G', siempre que elijamos una arista que sea parte de un ciclo.

Esto disminuirá la longitud total de las aristas en T o dejará la longitud total igual (si hemos eliminado una arista con longitud 0) mientras que disminuye el número de aristas en T. En cualquier caso, la nueva solución es preferible a la anterior.

Por tanto, un conjunto T con n o más aristas no puede ser óptimo. De ello se deduce que T debe tener exactamente $n-1$ aristas y dado que G' está conectado, debe ser un árbol.

El grafo G' se llama árbol de expansión mínimo para el grafo G. Este problema tiene muchas aplicaciones. Por ejemplo, suponga que los nodos de G representan ciudades y que el costo de una arista $\{a, b\}$ sea el costo de colocar una línea telefónica de a a b.

Entonces, un árbol de expansión mínimo de G corresponde a la red más barata posible que sirve a todas las ciudades en cuestión, siempre que sólo se puedan usar enlaces directos entre ciudades (en otras palabras, siempre que no se nos permita construir centrales telefónicas en el campo entre las ciudades).

Relajar esta condición equivale a permitir la adición de nodos auxiliares adicionales a G. Esto puede permitir que se obtengan soluciones más baratas: consulte el problema 6.8 del libro de texto.

A primera vista, al menos dos líneas de ataque parecen posibles si esperamos encontrar un algoritmo voraz para este problema. Claramente, nuestro conjunto de candidatos debe ser el conjunto A de aristas en G. Una posible táctica es comenzar con un conjunto T vacío y seleccionar en cada etapa la arista más corta que aún no se ha elegido o rechazado, independientemente de dónde se encuentre situada esta arista en G.

Otra línea de ataque implica elegir un nodo y construir un árbol desde allí, seleccionando en cada etapa la arista más corta disponible que pueda extender el árbol a un nodo adicional. ¡Inusualmente, para este problema particular ambos enfoques funcionan! Antes de presentar los algoritmos, mostramos cómo se aplica el esquema general de un algoritmo voraz en este caso y presentamos un lema para su uso posterior.

- Los candidatos, como ya se señaló, son las aristas en G.
- Un conjunto de aristas es una solución si constituye un árbol de expansión para los nodos en N.

- Un conjunto de aristas es factible si no incluye un ciclo.
- La función de selección que usamos varía con el algoritmo.
- La función objetivo a minimizar es la longitud total de las aristas en la solución.

También necesitamos más terminología. Decimos que un conjunto factible de aristas es prometedor si puede extenderse para producir no solo una solución, sino una solución óptima a nuestro problema. En particular, el conjunto vacío siempre es prometedor (ya que siempre existe una solución óptima).

Además, si un conjunto de aristas prometedor ya es una solución, entonces la extensión requerida es vacía y esta solución debe ser óptima. A continuación, decimos que una arista abandona un conjunto dado de nodos si exactamente un extremo de esta arista está en el conjunto.

Por lo tanto, una arista no puede dejar un conjunto dado de nodos porque ninguno de sus extremos está en el conjunto o, menos evidentemente, porque ambos lo están. El siguiente lema es crucial para demostrar la corrección de los próximos algoritmos.

Lema 1 Sea $\langle N, A\rangle$ un grafo conectado no dirigido donde cada arista tiene asociada una longitud. Sea $B \subset N$ un subconjunto propio de los nodos de G. Sea $T \subseteq A$ un conjunto de aristas prometedor tal que ninguna arista en T deja a B. Sea v la arista más corta que deja B (o una de las más cortas si hay empate). Entonces $T \cup \{v\}$ es prometedor.

Demostración: Sea U un árbol de expansión mínimo de G tal que $T \subseteq U$. Tal U debe existir ya que T es prometedor por suposición. Si $v \in U$, no hay nada que probar. De lo contrario, cuando agregamos la arista v a U, creamos exactamente un ciclo (ésta es una de las propiedades de un árbol).

En este ciclo, dado que v sale de B, necesariamente existe al menos otra arista, digamos u, que también sale de B, o el ciclo no podría cerrarse. Si ahora quitamos u, el ciclo desaparece y obtenemos un nuevo árbol V que expande a G. Sin embargo, la longitud de v por definición no es mayor que la longitud de u, y por lo tanto la longitud total de las aristas en V no excede la longitud total de las aristas en U.

Por lo tanto, V es también un árbol de expansión mínimo de G, e incluye v. Para completar la demostración, queda señalar que $T \subseteq V$ porque la arista u que se eliminó deja B, y por lo tanto no podría haber sido una arista de T.

6.3.1. Algoritmo de Kruskal

El conjunto T de aristas está inicialmente vacío. A medida que avanza el algoritmo, se agregan aristas a T. Siempre que no haya encontrado una solución, el grafo parcial formado por los nodos de G y las aristas en T consta de varios componentes conectados (inicialmente, cuando T está vacío, cada nodo de G forma un componente conectado trivial distinto).

Los elementos de T incluidos en un componente conectado dado forman un árbol de expansión mínimo para los nodos de este componente. Al final del algoritmo, solo queda un componente conectado, por lo que T es un árbol de expansión mínimo para todos los nodos de G.

Para construir componentes conectados cada vez más grandes, examinamos las aristas de G en orden de longitud creciente. Si una arista une dos nodos en diferentes componentes conectados, lo agregamos a T. En consecuencia, los dos componentes conectados ahora forman un solo componente.

De lo contrario, la arista se rechaza: ya que une dos nodos en el mismo componente conectado y, por lo tanto, no se puede agregar a T sin formar un ciclo (porque las aristas en T forman un árbol para cada componente). El algoritmo se detiene cuando sólo queda un componente conectado.

Para ilustrar cómo funciona este algoritmo, considere el grafo de la Figura 6.1. En orden creciente de longitud, las aristas son: $\{1,2\},\{2,3\},\{4,5\},\{6,7\},\{1,4\},\{2,5\},\{4,7\},\{3,5\},\{2,4\},\{3,6\},\{5,7\}$ y $\{5,6\}$. El algoritmo procede como sigue.

Paso	Arista considerada	Componentes conectadas
Inicio		{1}{2}{3}{4}{5}{6}{7}
1	{1, 2}	{1, 2} {3} {4} {5} {6} {7}
2	{2, 3}	{1, 2, 3} {4} {5} {6} {7}
3	{4, 5}	{1, 2, 3} {4, 5} {6} {7}
4	{6, 7}	{1, 2, 3} {4, 5} {6, 7}
5	{1, 4}	{1, 2, 3 ,4 , 5} {6, 7}
6	{2, 5}	rechazado
7	{4, 7}	{1, 2, 3, 4, 5, 6, 7}

Cuando el algoritmo se detiene, T contiene las aristas elegidas $\{1,2\},\{2,3\},\{4,5\},\{6,7\},\{1,4\}$ y $\{4,7\}$. Este árbol de expansión mínimo se muestra con líneas gruesas en la Figura 6.1; su longitud total es 17.

Teorema 6 El algoritmo de Kruskal encuentra un árbol de expansión mínimo.

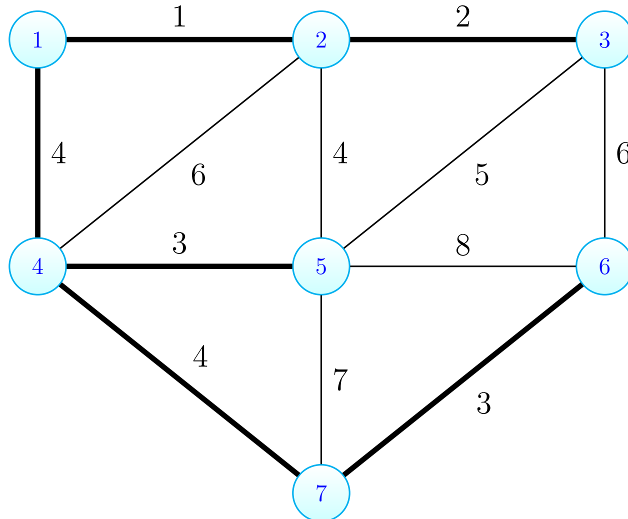

Figura 6.1: Un grafo y su árbol de expansión mínimo

Demostración: La prueba es por inducción matemática sobre el número de aristas en el conjunto T. Demostraremos que si T es prometedor en cualquier etapa del algoritmo, entonces sigue siendo prometedor cuando se ha agregado una arista extra. Cuando el algoritmo se detiene, T da una solución a nuestro problema; como también es prometedora, esta solución es óptima.

Base: El conjunto vacío es prometedor porque G está conectado y, por lo tanto, debe existir una solución.

Paso inductivo: Suponga que T es prometedor justo antes de que el algoritmo agregue una nueva arista $e = \{u, v\}$. Las aristas en T dividen los nodos de G en dos o más componentes conectados; el nodo u está en uno de estos componentes y v está en un componente diferente. Sea B el conjunto de nodos del componente que incluye a u. Ahora

- el conjunto B es un subconjunto propio de los nodos de G (ya que no incluye v, por ejemplo);
- T es un conjunto prometedor de aristas de manera que ninguna arista en T sale de B (ya que una arista en T tiene ambos extremos en B o no tiene ningún extremo en B, por lo que por definición no sale de B); y

- e es una de las aristas más cortas que sale de B (ya que todas las aristas estrictamente más cortas ya se han examinado, y se han incorporado a T o se han rechazado porque tenían ambos extremos en el mismo componente conectado).

Por tanto, se cumplen las condiciones del Lema 1 y concluimos que el conjunto $T \cup \{e\}$ también es prometedor.

Esto completa la demostración por inducción matemática de que el conjunto T es prometedor en cada etapa del algoritmo y, por lo tanto, cuando el algoritmo se detiene, T no sólo da una solución a nuestro problema, sino una solución óptima.

Para implementar el algoritmo, tenemos que manejar un cierto número de conjuntos, es decir, los nodos en cada componente conectado. Hay que realizar dos operaciones rápidamente: FIND(x), que nos dice en qué componente conectado se encuentra el nodo x, y MERGE(A, B), para fusionar dos componentes conectados.

Por lo tanto, utilizamos estructuras de conjuntos disjuntos; consulte la Sección 5.9 del libro de texto. Para este algoritmo es preferible representar el grafo como un vector de aristas con sus longitudes asociadas en lugar de como una matriz de distancias; vea el problema 6.9 del libro de texto. Aquí está el algoritmo.

```
function KRUSKAL(G = ⟨N, A⟩ : graph, lenght : A → ℝ⁺)
 1  A ← SORTBYINCREASINGLENGTH(A)
 2  n ← NUMBEROFNODESIN(N)
 3  T ← ∅
 4  Inicializar n conjuntos, cada uno con un elemento diferente de N
 5  repeat
 6           e ← {u, v}  la arista más corta aún no considerada
 7           ucomp ← FIND(u)
 8           vcomp ← FIND(v)
 9           if ucomp ≠ vcomp
10              then MERGE(ucomp, vcomp)
11                   T ← T ∪ {e}
12    until T contenga n − 1 aristas
13  return T
```

Podemos evaluar el tiempo de ejecución del algoritmo de la siguiente manera. En un grafo con n nodos y a aristas, el número de operaciones está en

- $\Theta(a \log a)$ para ordenar las aristas, lo cual es equivalente a $\Theta(a \log n)$ ya que $n - 1 \leq a \leq n(n-1)/2$;
- $\Theta(n)$ para inicializar los n conjuntos disjuntos;
- $\Theta(2a\alpha(2a, n))$ para todas las operaciones FIND y MERGE, donde α es la función de crecimiento lento definida en la Sección 5.9 del libro de texto (esto se deduce de los resultados de la Sección 5.9 del libro de texto, ya que hay como máximo $2a$ operaciones de búsqueda y $n - 1$ operaciones de fusión en un universo que contiene n elementos); y
- en el peor de los casos $O(a)$ para las operaciones restantes.

Concluimos que el tiempo total del algoritmo está en $\Theta(a \log n)$ porque $\Theta(\alpha(2a, n)) \subset \Theta(\log n)$. Aunque esto no cambia el análisis del peor caso, es preferible mantener las aristas en un heap invertido (consulte la Sección 5.7 del libro de texto): por lo tanto, la arista más corta está en la raíz del heap.

Esto permite que la inicialización se realice en un tiempo en $\Theta(a)$, aunque cada búsqueda de un mínimo en el ciclo de repetición ahora lleva un tiempo en $\Theta(\log a) = \Theta(\log n)$. Esto es particularmente ventajoso si el árbol de expansión mínimo se encuentra en un momento en el que quedan por probar un número considerable de aristas. En tales casos, el algoritmo original pierde tiempo ordenando estas aristas inútiles.

Capítulo 7

Divide y vencerás

Divide y vencerás es una técnica de diseño de algoritmos que consiste en descomponer la instancia a resolver en varias subinstancias más pequeñas del mismo problema, resolviendo sucesiva e independientemente cada una de estas subinstancias, y luego combinar las subsoluciones así obtenidas para obtener la solución de la instancia original.

Dos preguntas que naturalmente vienen a la mente son "¿Por qué alguien querría hacer esto?" y "¿Cómo deberíamos resolver las subinstancias?" La eficacia de la técnica divide y vencerás radica en la respuesta a esta última pregunta.

7.1. Introducción: multiplicando enteros grandes

Considere el problema de multiplicar números enteros grandes. El algoritmo clásico que la mayoría de nosotros aprendemos en la escuela requiere un tiempo en $\Theta(n^2)$ para multiplicar dos números de n cifras.

Estamos tan acostumbrados a este algoritmo que quizás ni siquiera cuestionaste su optimalidad. ¿Podemos hacerlo mejor? La multiplicación *à la russe* no ofrece ninguna mejora en el tiempo de ejecución.

Otro algoritmo que usa la técnica "divide y vencerás" consiste en reducir la multiplicación de dos números de n cifras a cuatro multiplicaciones de números de $\frac{n}{2}$ cifras. Desafortunadamente, el algoritmo resultante tampoco produce ninguna mejora sobre el algoritmo de multiplicación clásico a menos que seamos más inteligentes.

Para superar el algoritmo clásico y, por tanto, apreciar plenamente las virtudes divide y vencerás, debemos encontrar una manera de reducir la multiplicación original no a cuatro, sino a *tres* multiplicaciones de la mitad del tamaño.

Ilustramos el proceso con el ejemplo usado en la Sección 1.2 del libro de texto: la multiplicación de 981 por 1234. Primero rellenamos el operando más corto con un cero no significativo para que tenga la misma longitud que el más largo; así 981 se convierte en 0981.

Luego dividimos cada operando en dos mitades: 0981 da lugar a $w = 09$ y $x = 81$, y 1234 a $y = 12$ y $z = 34$. Observe que $981 = 10^2 w + x$ y $1234 = 10^2 y + z$. Por lo tanto, el producto requerido se puede calcular como

$$\begin{aligned} 981 \times 1234 &= (10^2 w + x) \times (10^2 y + z) \\ &= 10^4 wy + 10^2 (wz + xy) + xz \\ &= 1080000 + 127800 + 2754 = 1021554. \end{aligned}$$

Si cree que simplemente hemos reformulado el algoritmo de la Sección 1.2 del libro de texto con más símbolos, está perfectamente en lo cierto. El procedimiento anterior todavía necesita cuatro multiplicaciones de la mitad del tamaño: wy, wz, xy y xz.

La observación clave es que no es necesario calcular wz ni xy; todo lo que realmente necesitamos es la *suma* de estos dos términos. ¿Es posible obtener $wz + xy$ al costo de una sola multiplicación? Esto parece imposible hasta que recordemos que también necesitamos los valores de wy y xz para aplicar la fórmula anterior. Con esto en mente, considere el producto

$$r = (w + x) \times (y + z) = wy + (wz + xy) + xz.$$

Después de una sola multiplicación, obtenemos la suma de los tres términos necesarios para calcular el producto deseado. Esto sugiere proceder de la siguiente manera.

$$\begin{aligned} p &= wy = 09 \times 12 = 108 \\ q &= xz = 81 \times 34 = 2754 \\ r &= (w + x) \times (y + z) = 90 \times 46 = 4140 \end{aligned}$$

y finalmente

$$\begin{aligned} 981 \times 1234 &= 10^4 p + 10^2 (r - p - q) + q \\ &= 1080000 + 127800 + 2754 = 1210554. \end{aligned}$$

Así, el producto de 981 y 1234 se puede reducir a tres multiplicaciones de números de dos cifras ($09 \times 12, 81 \times 34$ y 90×46) junto con un cierto número de corrimientos (multiplicaciones por potencias de 10), adiciones y restas.

Sin duda, el número de sumas -contar las restas como si fueran sumas- es mayor que con el algoritmo original divide y vencerás de la Sección 1.2 del libro de texto. ¿Vale la pena realizar cuatro sumas más para ahorrar una multiplicación?

La respuesta es no cuando multiplicamos números pequeños como los de nuestro ejemplo. Sin embargo, vale la pena cuando los números que se van a multiplicar son grandes, y lo es cada vez más cuando los números aumentan.

Cuando los operandos son grandes, el tiempo requerido para las adiciones y corrimientos se vuelve insignificante en comparación con el tiempo que toma una sola multiplicación. Por tanto, parece razonable esperar que reducir cuatro multiplicaciones a tres nos permitirá reducir el 25 % del tiempo de cálculo necesario para las multiplicaciones grandes. Como veremos, nuestro ahorro será significativamente mejor.

Para ayudar a comprender lo que hemos logrado, suponga que una implementación dada del algoritmo de multiplicación clásico requiere un tiempo $h(n) = cn^2$ para multiplicar dos números de n cifras, por alguna constante c que depende de la implementación (esta es una simplificación ya que en realidad el tiempo requerido tendría una forma más complicada, como $cn^2 + bn + a$).

De manera similar, sea $g(n)$ el tiempo que tarda el algoritmo divide y venceras para multiplicar dos números de n cifras, sin contar el tiempo necesario para realizar las tres multiplicaciones de la mitad del tamaño.

En otras palabras, $g(n)$ es el tiempo necesario para las adiciones, corrimientos y varias operaciones generales. Es fácil implementar estas operaciones de modo que $g(n) \in \Theta(n)$. Ignore por el momento lo que sucede si n es impar o si los números no tienen la misma longitud.

Si cada una de las tres multiplicaciones de la mitad del tamaño se realiza mediante el algoritmo clásico, el tiempo necesario para multiplicar dos números de n cifras es

$$3h(n/2) + g(n) = 3c(n/2)^2 + g(n) = \frac{3}{4}cn^2 + g(n) = \frac{3}{4}h(n) + g(n).$$

Como $h(n) \in \Theta(n^2)$ y $g(n) \in \Theta(n)$, el término $g(n)$ es insignificante en comparación con $\frac{3}{4}h(n)$ cuando n es lo suficientemente grande, lo que significa que hemos ganado aproximadamente un 25 % en velocidad en comparación

con el algoritmo clásico, como anticipamos. Aunque esta mejora no debe ser despreciada, no hemos logrado cambiar el orden del tiempo requerido: el nuevo algoritmo aún toma tiempo cuadrático.

Para hacerlo mejor, volvemos a la pregunta planteada en el párrafo inicial: ¿cómo se deben resolver las subinstancias? Si son pequeños, el algoritmo clásico puede ser la mejor manera de proceder.

Sin embargo, cuando las subinstancias son lo suficientemente grandes, ¿no sería mejor utilizar nuestro nuevo algoritmo de forma recursiva? ¡La idea es análoga a beneficiarse de una cuenta bancaria que capitaliza los pagos de intereses! Cuando hacemos esto, obtenemos un algoritmo que puede multiplicar dos números de n cifras en un tiempo $t(n) = 3t(n/2) + g(n)$ cuando n es par y suficientemente grande.

Esta recurrencia es similar a la que estudiamos en el Ejemplo 10, resolviéndola obtuvimos $t(n) \in \Theta(n^{\log 3} | n$ es potencia de 2). Tenemos que contentarnos con la notación asintótica condicional porque todavía no hemos abordado la cuestión de cómo multiplicar números de longitud impar; vea el problema 7.1 del libro de texto.

Dado que $\lg 3 \approx 1{,}585$ es menor que 2, este algoritmo puede multiplicar dos enteros grandes mucho más rápido que el algoritmo de multiplicación clásico, y cuanto mayor sea n, más vale la pena tener esta mejora. Una buena implementación probablemente no utilizará la base 10, sino la base más grande para la que el hardware permita que se multipliquen directamente dos “dígitos”.

Un factor importante en la eficacia práctica de este enfoque de la multiplicación, y de hecho de cualquier algoritmo divide y vencerás, es saber cuándo dejar de dividir las instancias y usar el algoritmo clásico en su lugar.

Aunque el enfoque divide y vencerás se vuelve más valioso a medida que la instancia a resolver se hace más grande, de hecho puede ser más lento que el algoritmo clásico en instancias que son demasiado pequeñas.

Por lo tanto, un algoritmo divide y vencerás debe evitar proceder de manera recursiva cuando el tamaño de las subinstancias ya no lo justifique. Volveremos a este tema en la siguiente sección.

En aras de la simplicidad, hasta ahora se han ocultado varios temas importantes. ¿Cómo nos ocupamos de los números de longitud impar? Aunque ambas mitades del multiplicador y el multiplicando sean de tamaño $n/2$, puede suceder que su suma se desborde y sea de tamaño 1 mayor.

Por lo tanto, fue un poco incorrecto afirmar que $r = (w + x) \times (y + z)$ implica una multiplicación de la mitad del tamaño. ¿Cómo afecta esto al

análisis del tiempo de ejecución? ¿Cómo multiplicamos dos números de diferentes tamaños? ¿Hay operaciones aritméticas distintas de la multiplicación que podamos manejar de manera más eficiente que usando algoritmos clásicos?

Los números de longitud impar se multiplican fácilmente dividiéndolos tan cerca de la mitad como sea posible: un número de n cifras se divide en un número de $\lfloor n/2 \rfloor$ cifras y un número de $\lceil n/2 \rceil$ cifras. La segunda pregunta es más complicada. Considere la posibilidad de multiplicar 5678 por 6789. Nuestro algoritmo divide los operandos en $w = 56, x = 78, y = 67$ y $z = 89$. Las tres multiplicaciones, de la mitad del tamaño, involucradas son

$$\begin{aligned} p &= wy = 56 \times 67 \\ q &= xz = 78 \times 89, \text{ y} \\ r &= (w + x) \times (y + z) = 134 \times 156. \end{aligned}$$

La tercera multiplicación implica números de tres cifras y, por lo tanto, no es realmente la mitad del tamaño en comparación con la multiplicación original de números de cuatro cifras. Sin embargo, el tamaño de $w + x$ y de $y + z$ no puede exceder $1 + \lceil n/2 \rceil$.

Para simplificar el análisis, sea $t(n)$ el tiempo que tarda este algoritmo en el peor de los casos para multiplicar dos números de tamaño como máximo n (en lugar de exactamente n). Por definición, $t(n)$ es una función no decreciente.

Cuando n es lo suficientemente grande, nuestro algoritmo reduce la multiplicación de dos números de tamaño como máximo n a tres multiplicaciones más pequeñas $p = wy, q = xz$ y $r = (w + x) \times (y + z)$ de tamaños como máximo $\lfloor n/2 \rfloor, \lceil n/2 \rceil$ y $1 + \lceil n/2 \rceil$, respectivamente, además de manipulaciones fáciles que toman un tiempo en $O(n)$. Por tanto, existe una constante positiva c tal que

$$t(n) \leq t(\lfloor n/2 \rfloor) + t(\lceil n/2 \rceil) + t(1 + \lceil n/2 \rceil) + cn$$

para todo n suficientemente grande. Ésta es precisamente la recurrencia que estudiamos en el ejemplo 4.7.14 del libro de texto, que produce el ahora familiar $t(n) \in O(n^{\lg 3})$. Por tanto, siempre es posible multiplicar números de n cifras en un tiempo en $O(n^{\lg 3})$.

Un análisis del peor caso de este algoritmo muestra que, de hecho, $t(n) \in \Theta(n^{\lg 3})$, pero esto es de interés limitado porque existen algoritmos de mul-

tiplicación aún más rápidos; consulte los problemas 7.2 y 7.3 del libro de texto.

Regresando a la cuestión de multiplicar números de diferente tamaño, sean u y v números enteros de tamaño m y n, respectivamente. Si m y n están dentro de un factor de dos entre sí, es mejor rellenar el operando más pequeño con ceros no significativos para que tenga la misma longitud que el otro operando, como hicimos cuando multiplicamos 981 por 1234.

Sin embargo, este enfoque debe desalentarse cuando un operando es mucho más grande que el otro. ¡Incluso podría ser peor que usar el algoritmo de multiplicación clásico! Sin pérdida de generalidad, suponga que $m \leq n$. El algoritmo divide y vencerás que se usa con el relleno y el algoritmo clásico toman tiempo en $\Theta(n^{\lg 3})$ y $\Theta(mn)$, respectivamente, para calcular el producto de u y v.

Teniendo en cuenta que es probable que la constante oculta del primero sea mayor que la del segundo, vemos que divide y vencerás con relleno es más lento que el algoritmo clásico cuando $m < n^{\lg(3/2)}$, y por lo tanto en particular cuando $m < \sqrt{n}$.

Sin embargo, es sencillo combinar ambos algoritmos para obtener un algoritmo realmente mejor. La idea es dividir el operando v más largo en bloques de tamaño m y usar el algoritmo divide y vencerás para multiplicar u por cada bloque de v, de modo que el algoritmo divide y vencerás se use para multiplicar pares de operandos del mismo tamaño.

El producto final de u y v se obtiene entonces fácilmente mediante simples adiciones y corrimientos. El tiempo de ejecución total está dominado por la necesidad de realizar $\lceil n/m \rceil$ multiplicaciones de números de m cifras.

Dado que cada una de estas multiplicaciones más pequeñas toma un tiempo en $\Theta(m^{\lg 3})$ y dado que $\lceil n/m \rceil \in \Theta(n/m)$, el tiempo total de ejecución para multiplicar un número de n cifras por un número de m cifras está en $\Theta(nm^{\lg(3/2)})$ cuando $m \leq n$.

La multiplicación no es la única operación interesante que involucra números enteros grandes. La exponenciación modular es crucial para la criptografía moderna; consulte la Sección 7.8 del libro de texto.

La división de enteros, las operaciones de módulo y el cálculo de la parte entera de una raíz cuadrada se pueden realizar en un tiempo cuyo orden es el mismo que el requerido para la multiplicación; consulte la Sección 12.4 del libro de texto. Algunas otras operaciones importantes, como calcular el máximo común divisor, pueden ser intrínsecamente más difíciles de calcular; no se tratan aquí.

7.2. La plantilla general

La multiplicación de números enteros grandes no es un ejemplo aislado del beneficio que se obtiene del enfoque de divide y vencerás. Considere un problema arbitrario y suponga que ADHOC es un algoritmo simple capaz de resolver el problema.

Pedimos a ADHOC que sea eficiente en instancias pequeñas, pero su rendimiento en instancias grandes no es motivo de preocupación. Lo llamamos el subalgoritmo básico. El algoritmo de multiplicación clásico es un ejemplo de un subalgoritmo básico.

La plantilla general para los algoritmos divide y vencerás es la siguiente.

function DC(x)
1 **if** x es suficientemente pequeño o simple
2 **then** ADHOC(x)
3 descomponga x en instancias más pequeñas $x_1, x_2, \ldots, x_\ell$
4 **for** $i \leftarrow 1$ **to** ℓ
5 **do** $y_i \leftarrow \mathrm{DC}(x_i)$
6 recombine las y_i para obtener una solución y para x
7 **return** y

Algunos algoritmos divide y vencerás no siguen este esquema exactamente: por ejemplo, podrían requerir que la primera subinstancia se resuelva antes de formular la segunda subinstancia; consulte la Sección 7.5 del libro de texto.

El número de subinstancias, ℓ, suele ser pequeño e independiente de la instancia particular a resolver. Cuando $\ell = 1$, no tiene mucho sentido "descomponer x en una instancia más pequeña x_1" y es difícil justificar llamar a la técnica divide y vencerás.

Sin embargo, tiene sentido reducir la solución de una instancia grande a la de una más pequeña. Divide y vencerás se conoce con el nombre de *simplificación* en este caso; consulte las Secciones 7.3 y 7.7 del libro de texto. Cuando se usa la simplificación, a veces es posible reemplazar la recursividad inherente a divide y vencerás por un ciclo iterativo.

Implementado en un lenguaje convencional como Pascal en una máquina convencional que ejecuta un compilador poco sofisticado, es probable que un algoritmo iterativo sea algo más rápido que la versión recursiva, aunque solo por un factor multiplicativo constante.

Por otro lado, es posible ahorrar una cantidad sustancial de almacenamiento de esta manera: para una instancia de tamaño n, el algoritmo recursivo usa una pila cuya profundidad a menudo está en $\Omega(\lg n)$ y en casos malos incluso en $\Omega(n)$.

Para que valga la pena divide y vencerás, se deben cumplir tres condiciones. La decisión de cuándo utilizar el subalgoritmo básico en lugar de hacer llamadas recursivas debe tomarse con prudencia, debe ser posible descomponer una instancia en subinstancias y recombinar las subsoluciones de manera bastante eficiente, y las subinstancias deben ser, en la medida de lo posible, aproximadamente del mismo tamaño.

La mayoría de los algoritmos divide y vencerás son tales que el tamaño de las l subinstancias es aproximadamente n/b para alguna constante b, donde n es el tamaño de la instancia original.

Por ejemplo, nuestro algoritmo divide y vencerás para multiplicar números enteros grandes necesita un tiempo en $\Theta(n)$ para descomponer la instancia original en tres subinstancias de aproximadamente la mitad del tamaño y recombinar las subsoluciones: $\ell = 3$ y $b = 2$.

El análisis del tiempo de ejecución de tales algoritmos de divide y vencerás es casi automático, gracias a los ejemplos 13 y 4.7.16 del libro de texto. Sea $g(n)$ el tiempo requerido por DC en instancias de tamaño n, sin contar el tiempo necesario para las llamadas recursivas. El tiempo total $t(n)$ que toma este algoritmo de divide y vencerás es algo así como

$$t(n) = \ell t(n \div b) + g(n)$$

siempre que n sea lo suficientemente grande. Si existe un entero k tal que $g(n) \in \Theta(n^k)$, entonces se aplica el ejemplo 4.7.16 del libro de texto para concluir que

$$T(n) \in \begin{cases} \Theta(n^k) & \text{si } \ell < b^k \\ \Theta(n^k \log n) & \text{si } \ell = b^k \\ \Theta(n^{\log_b l}) & \text{si } \ell > b^k \end{cases} \tag{7.1}$$

Las técnicas utilizadas en la Sección 4.7.6 y el Ejemplo 4.7.14 del libro de texto generalmente se aplican para producir la misma conclusión, incluso si algunas de las subinstancias son de un tamaño que difiere de $\lfloor n/b \rfloor$ como máximo en una constante aditiva, y en particular si alguna de las las subinstancias son de tamaño $\lceil n/b \rceil$.

Como ejemplo, nuestro algoritmo divide y vencerás para la multiplicación de enteros grandes se caracteriza por $\ell = 3, b = 2$ y $k = 1$. Como $\ell > b^k$, se

aplica el tercer caso y obtenemos inmediatamente que el algoritmo toma un tiempo en $\Theta(n^{\lg 3})$ sin necesidad de preocuparse por el hecho de que dos de las subinstancias son de tamaño $\lceil n/2 \rceil$ y $1 + \lceil n/2 \rceil$ en lugar de $\lfloor n/2 \rfloor$.

En casos más complicados cuando $g(n)$ no está en el orden exacto de un polinomio, puede aplicarse el problema 4.44 del libro de texto.

Queda por ver cómo determinar si dividir la instancia y hacer llamadas recursivas, o si la instancia es tan simple que es mejor invocar el subalgoritmo básico directamente. Aunque esta elección no afecta el orden del tiempo de ejecución del algoritmo, también nos preocupa que la constante multiplicativa oculta en la notación Θ sea lo más pequeña posible.

Con la mayoría de los algoritmos divide y vencerás, esta decisión se basa en un umbral simple, generalmente denotado n_0. El subalgoritmo básico se utiliza para resolver cualquier instancia cuyo tamaño no supere a n_0.

Volvamos al problema de multiplicar números enteros grandes para ver por qué es importante la elección del umbral y cómo elegirlo. Para evitar nublar los aspectos esenciales, usamos una fórmula de recurrencia simplificada para el tiempo de ejecución del algoritmo divide y vencerás para multiplicar números enteros grandes:

$$T(n) = \begin{cases} h(n) & \text{si } n \leq n_0 \\ 3T(\lceil n/2 \rceil) + g(n) & \text{en otro caso,} \end{cases}$$

donde $h(n) \in \Theta(n^2)$ y $g(n) \in \Theta(n)$.

En aras de la argumentación, considere una implementación donde $h(n) = n^2$ microsegundos y $g(n) = 16n$ microsegundos. Suponga que nos dan dos números de 5000 cifras para multiplicar. Si el algoritmo divide y vencerás procede de forma recursiva hasta obtener subinstancias de tamaño 1, es decir, si $n_0 = 1$, se necesitan más de 41 segundos para calcular el producto.

Esto es ridículo, ya que los mismos números se pueden multiplicar en 25 segundos usando el algoritmo clásico. El algoritmo clásico supera ligeramente al algoritmo divide y vencerás incluso para multiplicar números con hasta 32 789 cifras, cuando ambos algoritmos requieren más de un cuarto de hora de tiempo de cálculo ¡para una sola multiplicación!

¿Debemos concluir que divide y vencerás nos permite pasar de un algoritmo cuadrático a un algoritmo cuyo tiempo de ejecución está en $\Theta(n^{\lg 3})$, pero solo a costa de un aumento de la constante oculta tan enorme que el nuevo algoritmo nunca es económico en instancias de tamaño razonable?

Afortunadamente no: para continuar con nuestro ejemplo, los números de 5000 cifras se pueden multiplicar en poco más de 6 segundos, siempre que elijamos el umbral n_0 inteligentemente; en este caso, $n_0 = 64$ es una buena opción. Con el mismo umbral, se necesitan poco más de dos minutos para multiplicar dos números de 32 789 cifras.

La elección del mejor umbral es complicada por el hecho de que el valor óptimo generalmente depende no solo del algoritmo en cuestión, sino también de la implementación particular. Además, en general, no existe un valor óptimo uniforme del umbral.

En nuestro ejemplo, es mejor utilizar el algoritmo clásico para multiplicar números de 67 cifras, mientras que es mejor repetir una vez para multiplicar números de 66 cifras. Así, 67 es mejor que 64 como umbral en el primer caso, mientras que en el segundo caso ocurre lo contrario. En el futuro abusaremos del término "umbral óptimo"para significar casi óptimo.

Entonces, ¿cómo elegiremos el umbral? Dada una implementación particular, el umbral óptimo se puede determinar empíricamente. Variamos el valor del umbral y el tamaño de las instancias utilizadas para nuestras pruebas y cronometramos la implementación en varios casos.

A menudo es posible estimar un umbral óptimo tabulando los resultados de estas pruebas o dibujando algunos diagramas. Sin embargo, los cambios en el valor del umbral en un cierto rango pueden no tener ningún efecto sobre la eficiencia del algoritmo cuando sólo se consideran instancias de algún tamaño específico.

Por ejemplo, se necesita exactamente el mismo tiempo para multiplicar dos números de 5000 cifras cuando el umbral se establece entre 40 y 78, ya que cualquier valor de ese tipo para el umbral hace que la recursividad se detenga cuando las subinstancias alcanzan el tamaño 40, por debajo del tamaño 79, en el séptimo nivel de recursividad.

Sin embargo, estos umbrales no son equivalentes en general, ya que los números de 41 cifras tardan un 17 % más en multiplicarse con el umbral establecido en 40 en lugar de en 64. Por lo tanto, generalmente no es suficiente simplemente variar el umbral para una instancia cuyo tamaño permanece fijo.

Este enfoque empírico puede requerir una cantidad considerable de tiempo informático (¡y humano!). Una vez les pedimos a los estudiantes de un curso de algoritmos que implementaran el algoritmo divide y vencerás para multiplicar números enteros grandes y que lo comparen con el algoritmo clásico.

Varios grupos trataron de estimar empíricamente el umbral óptimo, ¡cada

grupo utilizó en el intento más de 5000 dólares en tiempo de máquina! Por otro lado, rara vez es posible un cálculo puramente teórico del umbral óptimo, dado que varía de una implementación a otra.

El enfoque híbrido, que recomendamos, consiste en determinar teóricamente la forma de las ecuaciones de recurrencia y luego encontrar empíricamente los valores de las constantes utilizadas en estas ecuaciones para la implementación en cuestión.

El umbral óptimo se puede estimar entonces encontrando el tamaño n de la instancia para la que no importa si aplicamos el algoritmo clásico directamente o si continuamos con un nivel más de recursividad; vea el problema 7.8 del libro de texto.

Es por eso que elegimos $n_0 = 64$: el algoritmo de multiplicación clásico requiere $h(64) = 64^2 = 4096$ microsegundos para multiplicar dos números de 64 cifras, mientras que si usamos un nivel más de recursividad en el enfoque divide y vencerás, la misma multiplicación requiere $g(64) = 16 \times 64 = 1024$ microsegundos además de tres multiplicaciones de números de 32 cifras por el algoritmo clásico, a un costo de $h(32) = 32^2 = 1024$ microsegundos cada uno, para el mismo total de $3h(32) + g(64) = 4096$ microsegundos.

Surge una dificultad práctica con esta técnica híbrida. Aunque el algoritmo de multiplicación clásico requiere tiempo cuadrático, fue una simplificación excesiva afirmar que $h(n) = cn^2$ para alguna constante c que depende de la implementación.

Es más probable que existan tres constantes a, b y c tales que $h(n) = cn^2 + bn + a$. Aunque $bn + a$ se vuelve insignificante en comparación con cn^2 cuando n es grande, el algoritmo clásico se usa de hecho precisamente en instancias de tamaño moderado.

Por lo tanto, generalmente es insuficiente estimar simplemente la constante de orden superior c. En cambio, es necesario medir $h(n)$ varias veces para varios valores diferentes de n para estimar todas las constantes necesarias. La misma observación se aplica a $g(n)$.

Capítulo 8

Programación dinámica

En el capítulo anterior vimos que a menudo es posible dividir una instancia en subinstancias, resolver las subinstancias (quizás dividiéndolas más) y luego combinar las soluciones de las subinstancias para resolver la instancia original.

A veces sucede que la forma natural de dividir una instancia sugerida por la estructura del problema nos lleva a considerar varias subinstancias superpuestas. Si resolvemos cada uno de estos de forma independiente, a su vez crearán una gran cantidad de subinstancias idénticas.

Si no prestamos atención a esta duplicación, es probable que terminemos con un algoritmo ineficaz; si, por otro lado, aprovechamos la duplicación y nos disponemos a resolver cada subinstancia sólo una vez, guardando la solución para su uso posterior, resultará un algoritmo más eficiente.

La idea subyacente de la programación dinámica es, por lo tanto, bastante simple: evite calcular lo mismo dos veces, generalmente manteniendo una tabla de resultados conocidos que se llena a medida que se resuelven las subinstancias.

Divide y vencerás es un método de arriba hacia abajo. Cuando un problema se resuelve mediante el método divide y vencerás, atacamos inmediatamente la instancia completa, que luego dividimos en subinstancias cada vez más pequeñas a medida que avanza el algoritmo.

La programación dinámica, por otro lado, es una técnica de abajo hacia arriba. Por lo general, comenzamos con las subinstancias más pequeñas y, por lo tanto, las más simples. Combinando sus soluciones, obtenemos las respuestas a subinstancias de tamaño creciente, hasta que finalmente llegamos a la solución de la instancia original.

Comenzamos el capítulo con dos ejemplos simples de programación dinámica que ilustran la técnica general en un entorno sencillo. Las siguientes secciones retoman los problemas de dar cambio, que encontramos en la Sección 6.1, y de llenar una mochila, que encontramos en la Sección 6.5 del libro de texto.

8.1. Dos ejemplos simples

8.1.1. Cálculo del coeficiente binomial

Considere el problema de calcular el coeficiente binomial

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

Suponga que $0 \leq k \leq n$. Si calculamos $\binom{n}{k}$ directamente con

```
function C(n, k)
1  if k = 0 or k = n
2     then return 1
3     else  return C(n − 1, k − 1) + C(n − 1, k)
```

muchos de los valores $C(i, j), i < n, j < k$, se calculan una y otra vez. Por ejemplo, el algoritmo calcula $C(5, 3)$ como la suma de $C(4, 2)$ y $C(4, 3)$. Ambos resultados intermedios requieren que calculemos $C(3, 2)$. De manera similar, el valor de $C(2, 2)$ se usa varias veces.

Dado que el resultado final se obtiene sumando un número de 1s, el tiempo de ejecución de este algoritmo seguramente estará en $\Omega(\binom{n}{k})$. Encontramos un fenómeno similar antes en el algoritmo Fibrec para calcular la secuencia de Fibonacci; consulte la Sección 5.1.3.

Si, por el contrario, utilizamos una tabla de resultados intermedios, este es, por supuesto, el triángulo de Pascal, obtenemos un algoritmo más eficiente; vea la Tabla 8.1.

La tabla debe llenarse línea por línea. De hecho, ni siquiera es necesario almacenar la tabla completa: basta con mantener un vector de longitud k, que represente la línea actual, y actualizar este vector de izquierda a derecha.

	0	1	2	$\cdots$	$k-1$	k
0	1					
1	1	1				
2	1	2	1			
$\vdots$						
$n-1$	1				C$(n-1,k-1)$	C$(n-1,k)$
n	1					C(n,k)

Tabla 8.1: El triángulo de Pascal

Así, para calcular $\binom{n}{k}$ el algoritmo toma un tiempo en $\Theta(nk)$, si asumimos que la suma es una operación elemental, y un espacio en $\Theta(k)$.

8.1.2. La serie mundial

Imagine una competencia en la que dos equipos A y B juegan no más de $2n-1$ juegos, siendo el ganador el primer equipo en lograr n victorias. Suponemos que no hay juegos empatados, que los resultados de cada partido son independientes y que para cualquier partido dado hay una probabilidad constante p de que el equipo A sea el ganador y, por lo tanto, una probabilidad constante $q=1-p$ de que el equipo B ganará.

Sea $P(i,j)$ la probabilidad de que el equipo A gane la serie dado que todavía necesita i victorias más para lograrlo, mientras que el equipo B aún necesita j victorias más si quiere ganar.

Por ejemplo, antes del primer juego de la serie, la probabilidad de que el equipo A sea el ganador de la serie es $P(n,n)$: ambos equipos aún necesitan n victorias para ganar la serie. Si el equipo A requiere 0 victorias más, entonces de hecho ya ganó la serie, por lo que $P(0,i)=1, 1\leq i\leq n$.

De manera similar, si el equipo B requiere 0 victorias más, entonces ya ha ganado la serie y la probabilidad de que el equipo A sea el ganador de la serie es cero: entonces $P(i,0)=0, 1\leq i\leq n$.

Dado que no puede haber una situación en la que ambos equipos hayan ganado todos los partidos que necesitan, $P(0,0)$ no tiene sentido. Finalmente, dado que el equipo A gana un partido dado con probabilidad p y lo pierde con probabilidad q,

$$P(i,j)=pP(i-1,j)+qP(i,j-1), i\geq 1, j\geq i.$$

Así podemos calcular $P(i, j)$ como sigue.

function P(i, j)
1 **if** $i = 0$
2 **then return** 1
3 **else** **if** $j = 0$
4 **then return** 0
5 **else** **return** $p\mathrm{P}(i-1, j) + q\mathrm{P}(i, j-1)$

Sea $T(k)$ el tiempo necesario en el peor de los casos para calcular $P(i, j)$, donde $k = i + j$. Con este método, vemos que

$$T(1) = c$$
$$T(k) \leq 2T(k-1) + d, k > 1$$

donde c y d son constantes. Reescribiendo $T(k-1)$ en términos de $T(k-2)$, y así sucesivamente, encontramos

$$\begin{aligned} T(k) &\leq 4T(k-2) + 2d + d, k > 2 \\ &\vdots \\ &\leq 2^{k-1}T(1) + (2^{k-2} + 2^{k-3} + \cdots + 2 + 1)d \\ &= 2^{k-1}c + (2^{k-1} - 1)d \\ &= 2^k(c/2 + d/2) - d. \end{aligned}$$

Por lo tanto $T(k)$ está en $O(2^k)$, que está en $O(4^n)$ si $i = j = n$. De hecho, si observamos la forma en que se generan las llamadas recursivas, encontramos el patrón que se muestra en la Figura 8.1.2, que es idéntico al obtenido en el cálculo ingenuo del coeficiente binomial. Para ver esto, imagine que cualquier llamada $P(m, n)$ en la figura se reemplaza por $C(m + n, n)$.

Así, $P(i, j)$ se reemplaza por $C(i + j, j)$, $P(i-1, j)$ por $C(i + j - 1, j)$ y $P(i, j-1)$ por $C(i + j - 1, j - 1)$. Ahora el patrón de llamadas que muestran las flechas corresponde al cálculo

$$C(i + j, j) = C(i + j - 1, j) + C(i + j - 1, j - 1)$$

de un coeficiente binomial.

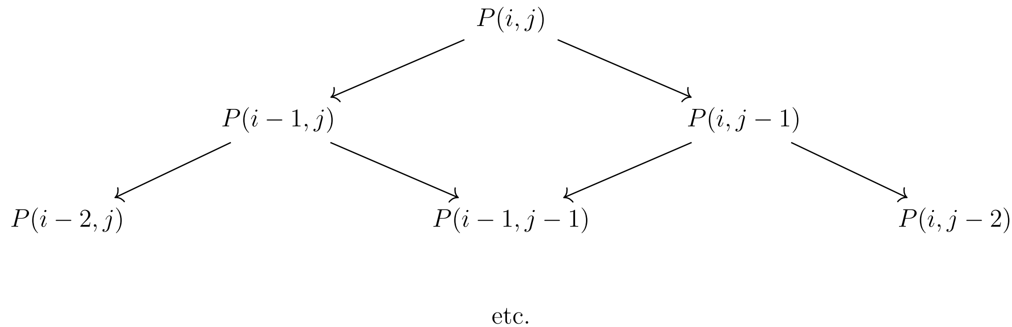

Figura 8.1: Llamados recursivos que se hacen por un llamado a $P(i,j)$.

Por tanto, el número total de llamadas recursivas es exactamente $2\binom{i+j}{j}-2$; vea el problema 8.1 del libro de texto. Para calcular la probabilidad $P(n,n)$ de que gane el equipo A dado que la serie aún no ha comenzado, el tiempo requerido está entonces en $\Omega(\binom{2n}{n})$.

El problema 1.42 del libro de texto pide al lector que muestre que $\binom{2n}{n} \geq 4^n/(2n+1)$. Combinando estos resultados, vemos que el tiempo requerido para calcular $P(n,n)$ está en $O(4^n)$ y en $\Omega(4^n/n)$. Por lo tanto, el método no es práctico para valores grandes de n (aunque las competiciones deportivas con $n > 4$ son la excepción, ¡este problema tiene otras aplicaciones!).

Para acelerar el algoritmo, procedemos más o menos como con el triángulo de Pascal: declaramos una matriz del tamaño apropiado y luego completamos las entradas. Esta vez, sin embargo, en lugar de llenar la matriz línea por línea, trabajamos diagonal por diagonal. Aquí está el algoritmo para calcular $P(n,n)$.

function SERIES(n, p)
1 **array** $P[0 \ldots n, 0 \ldots n]$
2 $q \leftarrow 1 - p$
3 **for** $s \leftarrow 1$ **to** n
4 **do** $P[0, s] \leftarrow 1;\ P[s, 0] \leftarrow 0$
5   **for** $k \leftarrow 1$ **to** $s - 1$
6   **do** $P[k, s-k] \leftarrow pP[k-1, s-k] + qP[k, s-k-1]$
7 **for** $s \leftarrow 1$ **to** n
8 **do for** $k \leftarrow 0$ **to** $n - s$
9   **do** $P[s+k, n-k] \leftarrow pP[s+k-1, n-k] + qP[s+k, n-k-1]$
10 **return** $P[n, n]$

Dado que el algoritmo tiene que llenar una matriz $n \times n$ y dado que se requiere un tiempo constante para calcular cada entrada, su tiempo de ejecución está en $\Theta(n^2)$. Al igual que con el triángulo de Pascal, es fácil implementar este algoritmo para que un espacio de almacenamiento en $\Theta(n)$ sea suficiente.

Capítulo 9

Complejidad computacional

Este capítulo es una traducción de partes del libro *Introduction to the Theory of Computation, Michael Sipser, Third Edition*, las alusiones en este capítulo a "libro de texto" se refieren a este libro.

9.1. Preliminares

Empezamos dando algunas definiciones tomadas de la **Teoría Formal de Lenguajes y Autómatas**.

Definición 1 Un **alfabeto** Σ es un conjunto finito y no vacío de símbolos.

Note que el término **símbolo** se debe entender de la manera más abstracta posible, así un símbolo puede ser tan simple como $a, b, c, \ldots$; 0, 1; **for, while, if, ...**; un elemento grafo atómico en una imagen, etc.

Podemos concatenar o yuxtaponer dos símbolos de una alfabeto mediante la siguiente definición.

Definición 2 Sean $a_1, a_2 \in \Sigma$, la **concatenación** (**yuxtaposición**) de estos símbolos se logra poniendo a_1 seguido de a_2, es decir, a_1a_2.

Ahora podemos generalizar esta definición a la concatenación (yuxtaposición) de dos conjuntos.

Definición 3 Sean A y B dos conjuntos, definimos la **concatenación** de A y B como $AB = \{ab | a \in A \text{ y } b \in B\}$.

Queremos definir el concepto de palabra (cadena), para empezar vamos a denotar la **palabra vacía** mediante el símbolo ϵ (algunos autores usan el símbolo λ), pero antes de definir qué es una cadena, definimos la potencia n-ésima de un conjunto (denotado A^n) de la siguiente manera.

Definición 4 **La potencia n-ésima** de un conjunto A se define recursivamente mediante las siguientes reglas:

1. $A^0 = \{\epsilon\}$,
2. $A^n = A^{n-1}A$.

Ya que hemos definido la potencia n-ésima de un conjunto, podemos definir la cerradura de Kleene (cerradura estrella) a través de la siguiente definición.

Definición 5 La **cerradura de Kleene** del conjunto A se difine como

$$A^* = \bigcup_{i \geq 0} A^i.$$

Como caso particular tenemos la **cerradura positiva** definida como

$$A^+ = \bigcup_{i \geq 1} A^i.$$

Teniendo las definiciones de cerradura, ahora podemos definir el concepto de palabra (cadena) de una manera muy directa.

Definición 6 A un elemento $w \in \Sigma^*$ o Σ^+ le llamaremos **palabra** (**cadena**).

En este punto tenemos que hacer las siguientes observaciones:

1. Si Σ no contiene ϵ, entonces Σ^+ tampoco la contendrá,
2. Σ^* siempre contiene a ϵ,
3. ϵ está "sobrecargado" actua como el símbolo vacío cuando pertenece a Σ y en los demás casos como la palabra vacía.

Dada la definición de Σ^*, podemos definir formalmente de manera directa el concepto de lenguaje.

Definición 7 Decimos que un conjunto L es un **lenguaje** si y sólo si $L \subseteq \Sigma^*$.

El siguiente concepto importante, tanto para el *Análisis y Diseño de Algoritmos* como para la *Complejidad Computacional*, es el de lenguaje decidible.

Definición 8 Un lenguaje $L \subseteq \Sigma^*$ es **decidible** si y sólo si existe una máquina de Turing T que lo reconoce, es decir, para toda $w \in \Sigma^*$ la máquina de Turing T regresa **sí** cuando $w \in L$ y regresa **no** si $w \notin L$.

Si bien el concepto de Máquina de Turing es la formalización de lo que es computable, de igual manera puede considerar un programa en cualquier lenguaje de programación que reconozca el lenguaje L.

Para terminar los preliminares damos la siguiente definición de la clase de complejidad *time*.

Definición 9 Sea $t : \mathbb{N} \to \mathbb{R}^+$ una función. Defina la **clase de complejidad time**, en símbolos $\text{TIME}(t(n))$, como la colección de todos los lenguajes que son decidibles por una máquina de Turing en tiempo $O(t(n))$.

9.2. La clase P

Para nuestros propósitos, las diferencias polinomiales en el tiempo de ejecución se consideran pequeñas, mientras que las diferencias exponenciales se consideran grandes. Veamos por qué elegimos hacer esta separación entre polinomios y exponenciales en lugar de entre algunas otras clases de funciones.

Primero, observe la gran diferencia entre la tasa de crecimiento de polinomios típicos como n^3 y exponenciales que ocurren típicamente como 2^n. Por ejemplo, sea $n = 1000$, el tamaño de una entrada razonable para un algoritmo.

En ese caso, n^3 es mil millones, un número grande pero manejable, mientras que 2^n es un número mucho mayor que el número de átomos en el universo. Los algoritmos de tiempo polinomial son lo suficientemente rápidos para muchos propósitos, pero los algoritmos de tiempo exponencial rara vez son útiles.

Los algoritmos de tiempo exponencial surgen típicamente cuando resolvemos problemas buscando exhaustivamente en un espacio de soluciones, llamado **búsqueda de fuerza bruta**. Por ejemplo, una forma de factorizar un número en sus primos constituyentes es buscar en todos los divisores potenciales.

El tamaño del espacio de búsqueda es exponencial, por lo que esta búsqueda utiliza tiempo exponencial. A veces, la búsqueda por fuerza bruta puede evitarse mediante una comprensión más profunda de un problema, lo que puede revelar un algoritmo de tiempo polinomial de mayor utilidad.

Todos los modelos computacionales deterministas razonables son **polinomialmente equivalentes**. Es decir, cualquiera de ellos puede simular otro con sólo un aumento polinomial en el tiempo de ejecución.

Cuando decimos que todos los modelos deterministas razonables son polinomialmente equivalentes, no intentamos definir lo razonable. Sin embargo, tenemos en mente una noción lo suficientemente amplia como para incluir modelos que se aproximan mucho a los tiempos de ejecución en computadoras reales.

A partir de aquí, nos centraremos en aspectos de la teoría de la complejidad del tiempo que no se ven afectados por las diferencias polinomiales en el tiempo de ejecución. Ignorar estas diferencias nos permite desarrollar una teoría que no depende de la selección de un modelo particular de computación.

Recuerde, nuestro objetivo es presentar las propiedades fundamentales de la *computación*, más que las propiedades de las máquinas de Turing o cualquier otro modelo especial de cómputo, como puede ser algún otro lenguaje de programación.

Puede sentir que ignorar las diferencias polinomiales en el tiempo de ejecución es absurdo. Los programadores reales ciertamente se preocupan por estas diferencias y trabajan duro sólo para que sus programas se ejecuten el doble de rápido.

Sin embargo, ignoramos los factores constantes hace un tiempo cuando introdujimos la notación asintótica. Ahora proponemos ignorar las diferencias polinomiales mucho mayores, como la que existe entre el tiempo n y n^3.

Nuestra decisión de ignorar las diferencias polinomiales no implica que consideremos tales diferencias sin importancia. Por el contrario, ciertamente consideramos que la diferencia entre el tiempo n y el tiempo n^3 es importante.

Pero algunas preguntas, como la polinomialidad o no polinomialidad del problema de factorización, también son importantes y no dependen de diferencias polinomiales. Simplemente elegimos enfocarnos en este tipo de pregunta aquí. Ignorar los árboles para ver el bosque no significa que uno sea más importante que el otro, solo da una perspectiva diferente.

Ahora llegamos a una definición importante en la teoría de la complejidad.

Definición 10 P es la clase de lenguajes que son decidibles en tiempo polinomial por una Máquina de Turing determinista con una sola cinta. En otras palabras,

$$P = \bigcup_k \text{TIME}(n^k).$$

La clase P juega un papel central en nuestra teoría y es importante porque:

1. P es invariante para todos los modelos de cálculo que son polinomialmente equivalentes a la máquina de Turing determinista de una sola cinta, y

2. P corresponde aproximadamente a la clase de problemas que se pueden resolver de manera realista en una computadora.

El ítem 1 indica que P es una clase matemáticamente robusta. No se ve afectado por los detalles del modelo de cálculo que estamos usando.

El ítem 2 indica que P es relevante desde un punto de vista práctico. Cuando un problema está en P, tenemos un método para resolverlo que se ejecuta en el tiempo n^k para alguna constante k. Si este tiempo de ejecución es práctico depende de k y de la aplicación.

Por supuesto, es poco probable que un tiempo de ejecución de n^{100} sea de utilidad práctica. No obstante, se ha demostrado que es útil llamar al tiempo polinomial el umbral de la capacidad de solución práctica.

Una vez que se ha encontrado un algoritmo de tiempo polinomial para un problema que anteriormente parecía requerir un tiempo exponencial, se ha obtenido una visión clave de él y, por lo general, se siguen reducciones adicionales en su complejidad, a menudo hasta el punto de una utilidad práctica real.

9.2.1. Ejemplos de problemas en P

Cuando presentamos un algoritmo de tiempo polinomial, damos una descripción de alto nivel sin hacer referencia a las características de un modelo computacional en particular. Al hacerlo, se evitan los tediosos detalles de las cintas y los movimientos de la cabeza de una máquina de Turing. Seguimos ciertas convenciones cuando describimos un algoritmo para poder analizar su polinomialidad.

Seguimos describiendo algoritmos con pasos numerados. Ahora debemos ser sensibles al número de pasos de la máquina de Turing requeridos para

implementar cada paso, así como al número total de pasos que usa el algoritmo.

Cuando analizamos un algoritmo para mostrar que se ejecuta en tiempo polinomial, debemos hacer dos cosas. Primero, tenemos que dar un límite superior polinomial (generalmente en notación O) en el número de pasos que usa el algoritmo cuando se ejecuta en una entrada de longitud n.

Luego, tenemos que examinar los pasos individuales en la descripción del algoritmo para estar seguros de que cada uno puede implementarse en tiempo polinomial en un modelo determinista razonable. Elegimos los pasos cuando describimos el algoritmo para que esta segunda parte del análisis sea fácil de hacer.

Cuando se han completado ambas tareas, podemos concluir que el algoritmo se ejecuta en tiempo polinomial porque hemos demostrado que se ejecuta para un número polinomial de pasos, cada uno de los cuales se puede realizar en tiempo polinomial, y la composición de polinomios es una polinomio.

Un punto que requiere atención es el método de codificación utilizado para los problemas. Continuamos usando la notación de corchetes angulares $\langle \cdot \rangle$ para indicar una codificación razonable de uno o más objetos en una cadena, sin especificar ningún método de codificación en particular.

Ahora bien, un método razonable es el que permite la codificación y decodificación de objetos en tiempo polinomial en representaciones internas naturales o en otras codificaciones razonables. Los métodos de codificación familiares para grafos, autómatas y similares son razonables.

Pero tenga en cuenta que la notación unaria para codificar números (como en el número 17 codificado por la cadena unaria 11111111111111111) no es razonable porque es exponencialmente más grande que las codificaciones verdaderamente razonables, como la notación base k para cualquier $k \geq 2$.

Muchos problemas computacionales que encontrará en este capítulo contienen codificaciones de grafos. Una codificación razonable de un grafo es una lista de sus nodos y aristas. Otra es la matriz de adyacencia, donde la entrada (i, j)-ésima es 1 si hay un arista del nodo i al nodo j y 0 si no la hay. Cuando analizamos algoritmos de grafos, el tiempo de ejecución se puede calcular en términos del número de nodos en lugar del tamaño de la representación del grafo.

En representaciones razonables de grafos, el tamaño de la representación es un polinomio en el número de nodos. Así, si analizamos un algoritmo y mostramos que su tiempo de ejecución es polinomial (o exponencial) en el número de nodos, sabemos que es polinomial (o exponencial) en el tamaño

de la entrada.

El primer problema se refiere a los grafos dirigidos. Un grafo dirigido G contiene los nodos s y t. El problema $PATH$ consiste en determinar si existe una ruta dirigida de s a t. Sea

$$PATH = \{\langle G, s, t\rangle | G \text{ es un grafo dirigido que tiene un camino dirigido de } s \text{ a } t\}$$

Teorema 7 $PATH \in$ P.

IDEA DE LA PRUEBA Demostramos este teorema presentando un algoritmo de tiempo polinomial que decide $PATH$. Antes de describir ese algoritmo, observemos que un algoritmo de fuerza bruta para este problema no es lo suficientemente rápido.

Un algoritmo de fuerza bruta para $PATH$ procede al examinar todas las rutas potenciales en G y determinar si alguna es una ruta dirigida de s a t.

Una ruta potencial es una secuencia de nodos en G que tiene una longitud de como máximo m, donde m es el número de nodos en G (si existe alguna ruta dirigida de s a t, existe una que tenga una longitud de como máximo m porque nunca es necesario repitir un nodo).

Pero el número de tales caminos potenciales es aproximadamente m^m, que es exponencial en el número de nodos en G. Por lo tanto, este algoritmo de fuerza bruta usa tiempo exponencial.

Para obtener un algoritmo de tiempo polinomial para $PATH$, debemos hacer algo que evite la fuerza bruta. Una forma es utilizar un método de búsqueda de grafos como la búsqueda primero en amplitud.

Aquí, marcamos sucesivamente todos los nodos en G que son accesibles desde s por caminos dirigidos de longitud 1, luego 2, luego 3, hasta m. Es fácil limitar el tiempo de ejecución de esta estrategia por un polinomio.

PRUEBA Un algoritmo polinomial M para $PATH$ opera como sigue.

M = "Como entrada recibe $\langle G, s, t\rangle$, donde G es un grafo dirigido con nodos s y t como origen y destino.

1. Coloque una marca en el nodo s.
2. Repita lo siguiente hasta que no se marquen nodos adicionales:
3. Escanee todos las aristas de G. Si se encuentra una arista (a, b) de un nodo marcado a a un nodo no marcado b, marque el nodo b.

4. Si t está marcado *acepta*. De lo contrario *rechaza*".

Ahora analizamos este algoritmo para demostrar que se ejecuta en tiempo polinomial. Obviamente, los pasos 1 y 4 se ejecutan solo una vez. El paso 3 se ejecuta como máximo m veces porque cada vez, excepto la última, marca un nodo adicional en G. Por lo tanto, el número total de pasos utilizadas es como máximo $1 + 1 + m$, lo que da un polinomio del tamaño de G.

Los pasos 1 y 4 de M se implementan fácilmente en tiempo polinomial en cualquier modelo determinista razonable. El paso 3 implica un escaneo de la entrada y una prueba de si ciertos nodos están marcados, que también se implementa fácilmente en tiempo polinomial. Por tanto, M es un algoritmo en tiempo polinomial para $PATH$.

Pasemos a otro ejemplo de un algoritmo de tiempo polinomial. Supongamos que dos números son primos relativos si 1 es el entero más grande que los divide a ambos. Por ejemplo, 10 y 21 son relativamente primos, aunque ninguno de ellos es un número primo en sí mismo, mientras que 10 y 22 no son relativamente primos porque ambos son divisibles por 2. Sea $RELPRIME$ el problema de probar si dos números son relativamente primos. Así

$$RELPRIME = \{\langle x, y\rangle | x \text{ y } y \text{ son primos relativos}\}.$$

Teorema 8 $RELPRIME \in$ P.

IDEA DE LA PRUEBA Un algoritmo que resuelve este problema busca en todos los divisores posibles de ambos números y acepta si ninguno es mayor que 1.

Sin embargo, la magnitud de un número representado en binario, o en cualquier otra notación base k para $k \geq 2$, es exponencial en la longitud de su representación. Por lo tanto, este algoritmo de fuerza bruta busca a través de un número exponencial de divisores potenciales y tiene un tiempo de ejecución exponencial.

En su lugar, resolvemos este problema con un antiguo procedimiento numérico, llamado algoritmo euclidiano, para calcular el máximo común divisor. El máximo común divisor de números naturales x y y, que se escribe como $mcd(x, y)$, es el entero más grande que divide tanto a x como a y. Por ejemplo, $mcd(18, 24) = 6$.

Obviamente, x y y son primos relativos si y sólo si $mcd(x, y) = 1$. Describimos el algoritmo euclidiano como el algoritmo E en la demostración. Utiliza la función mod, donde $x \ mod \ y$ es el resto después de la división entera de x entre y.

PRUEBA El algoritmo euclidiano E es como sigue.

$E =$ "Recibe como entrada $\langle x, y\rangle$, donde x y y son números naturales en binario:

1. Repite hasta que $y = 0$:
2. Asigna a $x \leftarrow x \ mod \ y$.
3. Intercambia x y y.
4. Regresa x."

El algoritmo R resuelve $RELPRIME$ usando E como subrutina.

$R =$ "Recibe como entrada $\langle x, y\rangle$, donde x y y son números naturales en binario:

1. Ejecuta E con $\langle x, y\rangle$.
2. Si el resultado es 1 *acepta*, en otro caso *rechaza*."

Claramente, si E se ejecuta correctamente en tiempo polinomial, también lo hace R y, por lo tanto, solo necesitamos analizar E para determinar el tiempo y su corrección. La corrección de este algoritmo es bien conocida, por lo que no lo discutiremos más aquí.

Para analizar la complejidad temporal de E, primero mostramos que cada ejecución del paso 2 (excepto posiblemente el primero) reduce el valor de x por lo menos a la mitad. Después de ejecutar el paso 2, $x < y$ debido a la naturaleza de la función *mod*.

Después del paso 3, $x > y$ porque los dos se han intercambiado. Por tanto, cuando se ejecuta posteriormente el paso 2, $x > y$. Si $x/2 \geq y$, entonces $x \ mod \ y < y \leq x/2$ y x se reduce al menos a la mitad. Si $x/2 < y$, entonces $x \ mod \ y = x - y < x/2$ y x se reduce al menos a la mitad.

Los valores de x y y se intercambian cada vez que se ejecuta el paso 3, por lo que cada uno de los valores originales de x y y se reduce al menos a la mitad cada dos veces a lo largo del ciclo. Por lo tanto, el número máximo de veces que se ejecutan los pasos 2 y 3 es el menor de $2\log_2 x$ y $2\log_2 y$.

Estos logaritmos son proporcionales a las longitudes de las representaciones, dando el número de pasos ejecutadas en $O(n)$. Cada paso de E usa solo tiempo polinomial, por lo que el tiempo total de ejecución es polinomial.

El último ejemplo de un algoritmo en tiempo polinomial muestra que cada lenguaje libre de contexto es decidible en tiempo polinomial.

Teorema 9 Todo lenguaje libre de contexto es miembro de P.

IDEA DE LA PRUEBA En el Teorema 4.9 del libro de texto, probamos que cada CFL es decidible. Para ello, propusimos un algoritmo que decide todo CFL. Si ese algoritmo se ejecuta en tiempo polinomial, el teorema actual se sigue como corolario. Recordemos ese algoritmo y averigüemos si se ejecuta lo suficientemente rápido.

Sea L un CFL generad por CFG G que está en forma normal de Chomsky. Del problema 2.26 del libro de texto, cualquier derivación de una cadena w tiene $2n - 1$ pasos, donde n es la longitud de w porque G está en la forma normal de Chomsky.

El algoritmo que decide L funciona probando todas las derivaciones posibles con $2n - 1$ pasos cuando su entrada es una cadena de longitud n. Si alguno de estos es una derivación de w, el algoritmo que decide acepta; si no, lo rechaza.

Un análisis rápido de este algoritmo muestra que no se ejecuta en tiempo polinomial. El número de derivaciones con k pasos puede ser exponencial en k, por lo que este algoritmo puede requerir un tiempo exponencial.

Para obtener un algoritmo de tiempo polinomial, utilizamos programación dinámica (ver Capítulo 8). Esta técnica utiliza la acumulación de información sobre subproblemas más pequeños para resolver problemas más grandes.

Registramos la solución a cualquier subproblema para que tengamos que resolverlo sólo una vez. Lo hacemos haciendo una tabla de todos los subproblemas y registrando sus soluciones sistemáticamente a medida que las encontramos.

En este caso, consideramos los subproblemas de determinar si cada variable en G genera cada subcadena de w. El algoritmo registra la solución a este subproblema en una tabla $n \times n$. Para $i \leq j$, la entrada $(i, j)-$ésima de la tabla contiene la colección de variables que generan la subcadena $w_i w_{i+1} \cdots w_j$. Para $i > j$, las entradas de la tabla no se utilizan.

El algoritmo completa las entradas de la tabla para cada subcadena de w. Primero rellena las entradas de las subcadenas de longitud 1, luego las de longitud 2, y así sucesivamente. Utiliza las entradas para las longitudes más cortas para ayudar a determinar las entradas para las longitudes más largas.

Por ejemplo, suponga que el algoritmo ya ha determinado qué variables generan todas las subcadenas de longitud hasta k. Para determinar si una variable A genera una subcadena particular de longitud $k + 1$, el algoritmo divide esa subcadena en dos partes no vacías de las k formas posibles.

Para cada división, el algoritmo examina cada regla $A \rightarrow BC$ para determinar si B genera la primera pieza y C genera la segunda pieza, utilizando las entradas de la tabla calculadas previamente.

Si tanto B como C generan las piezas respectivas, A genera la subcadena y así se agrega a la entrada de la tabla asociada. El algoritmo inicia el proceso con las cadenas de longitud 1 examinando la tabla para las reglas $A \rightarrow b$.

PRUEBA

El siguiente algoritmo D implementa la idea de la prueba. Sea G una CFG en la forma normal de Chomsky que genera el CFL L. Suponga que S es la variable inicial (recuerde que la cadena vacía se maneja especialmente en una gramática en forma normal de Chomsky, el algoritmo maneja el caso especial en el que $w = \epsilon$ en el paso 1). Los comentarios aparecen entre corchetes dobles.

$D =$ "Recibe como entrada $w = w_1 \cdots w_n$:

1. Si $w = \epsilon$ y $S \rightarrow \epsilon$ es una regla, *acepta*; sino, *rechaza*. "caso $w = \epsilon$"
2. Para $i = 1$ a n: "examina cada subcadena de longitud 1"
3. Para cada variable A:
4. Si $A \rightarrow w_i$ es una regla:
5. $table(i, i) \leftarrow A$.
6. Para $l = 2$ a n: "l es la longitud de la subcadena"
7. Para $i = 1$ a $n - l + 1$: "i es la posición inicial de la subcadena"
8. $j \leftarrow i + l - 1$. "j es la posición final de la subcadena"
9. Para $k = i$ a $j - 1$: "k es donde se divide la subcadena"
10. Para cada regla $A \rightarrow BC$:
11. Si $B \in table(i, k)$ y $C \in table(k + 1, j)$: $table(i, j) \leftarrow A$.
12. Si S está en $table(1, n)$, *acepta*; sino, *rechaza*."

Ahora analizamos D. Cada etapa se implementa fácilmente para ejecutarse en tiempo polinomial. Los pasos 4 y 5 se ejecutan a lo más nv veces, donde v es el número de variables en G y es una constante fija independiente

de n; por lo tanto, estos pasos se ejecutan $O(n)$ veces. El paso 6 se ejecuta como máximo n veces. Cada vez que se ejecuta el paso 6, el paso 7 se ejecuta como máximo n veces.

Cada vez que se ejecuta el paso 7, los pasos 8 y 9 se ejecutan como máximo n veces. Cada vez que se ejecuta el paso 9, el paso 10 se ejecuta r veces, donde r es el número de reglas de G y es otra constante fija. Por lo tanto, el paso 11, el ciclo interno del algoritmo, se ejecuta $O(n^3)$ veces. La suma del total muestra que D ejecuta $O(n^3)$ pasos.

9.3. La clase NP

Como observamos en la sección 9.2, podemos evitar la búsqueda por fuerza bruta en muchos problemas y obtener soluciones de tiempo polinomial. Sin embargo, los intentos de evitar la fuerza bruta en algunos otros problemas, incluidos muchos interesantes y útiles, no han tenido éxito y no se sabe que existan algoritmos de tiempo polinomial que los resuelvan.

¿Por qué no hemos logrado encontrar algoritmos de tiempo polinomial para estos problemas? No sabemos la respuesta a esta importante pregunta. Quizás estos problemas tengan algoritmos de tiempo polinomial aún sin descubrir que se basan en principios desconocidos.

O posiblemente algunos de estos problemas simplemente **no se pueden resolver** en tiempo polinomial. Pueden ser intrínsecamente difíciles.

Un descubrimiento notable relacionado con esta cuestión muestra que las complejidades de muchos problemas están vinculadas. Se puede utilizar un algoritmo de tiempo polinomial para uno de esos problemas para resolver toda una clase de problemas. Para comprender este fenómeno, comencemos con un ejemplo.

Un camino hamiltoniano en un grafo dirigido G es una ruta dirigida que atraviesa cada nodo exactamente una vez. Consideramos el problema de probar si un grafo dirigido contiene un camino hamiltoniano que conecta dos nodos específicos, como se muestra en la Figura 9.1. Sea

$$HAMPATH = \{\langle G, s, t\rangle | G \text{ es un grafo dirigido con un camino Hamiltoniano de } s \text{ a } t\}.$$

Podemos obtener fácilmente un algoritmo de tiempo exponencial para el problema $HAMPATH$ modificando el algoritmo de fuerza bruta para $PATH$ dado en el Teorema 7. Sólo necesitamos agregar una prueba para ve-

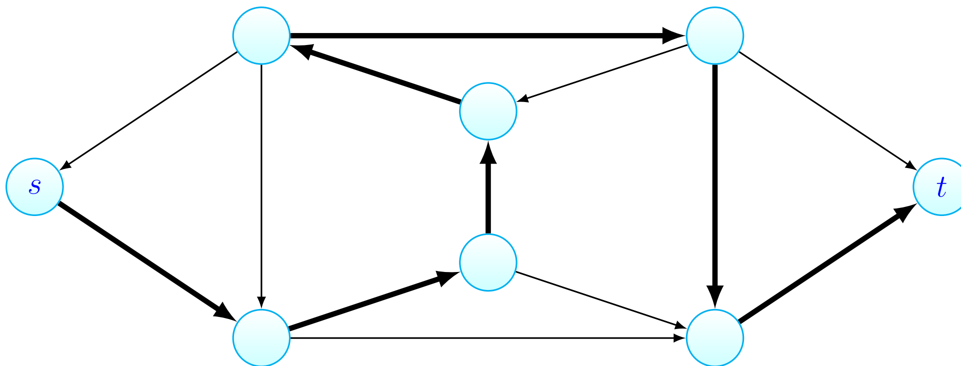

Figura 9.1: Un grafo con un camino Hamiltoniano representado con aristas gruesas (el camino recorre todos los nodos exactamente una vez, empezando en s y terminando en t).

rificar que el camino potencial sea hamiltoniano. Nadie sabe si $HAMPATH$ se puede resolver en tiempo polinomial.

El problema $HAMPATH$ tiene una característica llamada **verificabilidad polinomial** que es importante para comprender su complejidad.

Aunque no conocemos una forma rápida (es decir, en tiempo polinomial) de determinar si un grafo contiene un camino hamiltoniano, si dicho camino se descubriera de alguna manera (tal vez usando el algoritmo en tiempo exponencial), podríamos convencer fácilmente a alguien más de su existencia simplemente presentándolo.

En otras palabras, verificar la existencia de un camino hamiltoniano puede ser mucho más fácil que determinar su existencia.

Otro problema polinomialmente verificable es la composición. Recuerde que un número natural es compuesto si es el producto de dos números enteros mayores que 1 (es decir, un número compuesto es uno que no es un número primo). Sea

$$COMPOSITES = \{x|x = pq, \text{ para enteros } p, q > 1\}.$$

Podemos verificar fácilmente que un número es compuesto; todo lo que se necesita es un divisor de ese número. Recientemente, se descubrió un algoritmo de tiempo polinomial para probar si un número es primo o compuesto, pero es considerablemente más complicado que el método anterior para verificar la composición.

Algunos problemas pueden no ser verificables polinomialmente. Por ejemplo, tome $\overline{HAMPATH}$, el complemento del problema $HAMPATH$. Incluso si pudiéramos determinar (de alguna manera) que un grafo no tiene un camino hamiltoniano, no conocemos una forma para que alguien más verifique su inexistencia sin usar el mismo algoritmo de tiempo exponencial para tomar la determinación en primer lugar. Sigue una definición formal.

Definición 11 Un verificador para un lenguaje A es un algoritmo V, donde

$$A = \{w|V \text{ acepta } \langle w, c\rangle \text{ para alguna cadena } c\}.$$

Medimos el tiempo de un verificador solo en términos de la longitud de w, por lo que un **verificador de tiempo polinomial** se ejecuta en tiempo polinomial en la longitud de w. Un lenguaje A es **verificable polinomialmente** si tiene un verificador de tiempo polinomial.

Un verificador usa información adicional, representada por el símbolo c en la Definición 11, para verificar que una cadena w es miembro de A. Esta información se llama **certificado** o **prueba** de pertenencia a A.

Observe que para los verificadores polinomiales, el certificado tiene una longitud polinomial (en la longitud de w) porque eso es todo lo que el verificador puede acceder en su límite de tiempo. Apliquemos esta definición a los lenguajes $HAMPATH$ y $COMPOSITES$.

Para el problema $HAMPATH$, un certificado para una cadena $\langle G, s, t\rangle \in HAMPATH$ simplemente es una ruta hamiltoniana de s a t. Para el problema $COMPOSITES$, un certificado para el número compuesto x simplemente es uno de sus divisores.

En ambos casos, el verificador puede comprobar en tiempo polinomial que la entrada está en el lenguaje partiendo del certificado que se ha dado.

Definición 12 NP es la clase de lenguajes que tienen verificadores de tiempo polinomial.

La clase NP es importante porque contiene muchos problemas de interés práctico. De la discusión anterior, tanto $HAMPATH$ como $COMPOSITES$ son miembros de NP. Como mencionamos, $COMPOSITES$ también es miembro de P, que es un subconjunto de NP; pero demostrar este resultado más fuerte es mucho más difícil.

El término NP proviene de **tiempo polinomial no determinista** y se deriva de una caracterización alternativa mediante el uso de máquinas de

Turing de tiempo polinomial no deterministas. Los problemas en NP a veces se denominan problemas NP.

La siguiente es una máquina de Turing no determinista (NTM) que decide el problema $HAMPATH$ en tiempo polinomial no determinista. Recuerde que en la Definición 7.9 del libro de texto, definimos el tiempo de una máquina no determinista como el tiempo utilizado por la rama de cálculo más larga.

N_1 = "Como entrada recibe $\langle G, s, t \rangle$, donde G es un grafo dirigido con nodos s y t.

1. Escribe una lista de m números, $p_1, \ldots, p_m$, donde m es el número de nodos en G. Cada número en la lista se selecciona de manera no determinista para que esté entre 1 y m.

2. Revisa repeticiones en la lista. Si se encuentra alguna, *rechaza*.

3. Revisa si $s = p_1$ y $t = p_m$. Si no es así, *rechaza*.

4. Para cada i entre 1 y $m - 1$, revisa si (p_i, p_{i+1}) es una arista en G. Si alguna no lo es *rechaza*. En otro caso, todas las pruebas se han pasado, por lo tanto *acepta*."

Para analizar este algoritmo y verificar que se ejecuta en tiempo polinomial no determinista, examinamos cada uno de sus pasos.

En el paso 1, la selección no determinista se ejecuta claramente en tiempo polinomial.

En los pasos 2 y 3, cada parte es una verificación simple, por lo que juntas se ejecutan en tiempo polinomial.

Finalmente, el paso 4 también se ejecuta claramente en tiempo polinomial. Por lo tanto, este algoritmo se ejecuta en un tiempo polinomial no determinista.

Teorema 10 Un lenguaje está en NP si y sólo si es decidido en tiempo polinomial por alguna máquina de Turing no determinista.

IDEA DE LA PRUEBA Mostramos cómo convertir un verificador de tiempo polinomial en una NTM de tiempo polinomial equivalente y viceversa. La NTM simula al verificador adivinando el certificado. El verificador simula la NTM utilizando como certificado la rama que acepta.

PRUEBA Para la dirección hacia adelante de este teorema, sea $A \in$ NP y demuestre que A es decidido en tiempo polinomial por una NTM N. Sea V

el verificador de tiempo polinomial para A que existe según la definición de NP. Suponga que V es una TM que se ejecuta en el tiempo n^k y construya N de la siguiente manera.

$N =$ "Como entrada recibe w de longitud n:

1. Selecciona de manera no determinista una cadena c de longitud a lo más n^k.

2. Ejecuta V con entrada $\langle w, c \rangle$.

3. Si V acepta entonces *acepta*, de lo contrario *rechaza*."

Para probar la otra dirección del teorema, suponga que A se decide en tiempo polinomial por una NTM N y construya un verificador de tiempo polinomial V como sigue.

$V =$ "Como entrada recibe $\langle w, c \rangle$, donde w y c son cadenas:

1. Simula N con la entrada w, tratando cada símbolo de c como una descripción de la elección no determinista a realizar en cada paso.

2. Si acepta esta rama del cálculo de N entonces *acepta*, de lo contrario *rechaza*."

Definimos la clase de complejidad temporal no determinista NTIME$(t(n))$ de manera análoga a la clase de complejidad temporal determinista TIME$(t(n))$.

Definición 13 Sea $t : \mathbb{N} \to \mathbb{R}^+$ una función. Defina la **clase de complejidad ntime**, en símbolos NTIME$(t(n))$, como la colección de todos los lenguajes que son decidibles por una máquina de Turing no determinista en tiempo $O(t(n))$.

Corolario 1

$$\mathrm{NP} = \bigcup_k \mathrm{NTIME}(n^k).$$

La clase NP es insensible a la elección de un modelo computacional no determinista razonable porque todos esos modelos son polinomialmente equivalentes.

Al describir y analizar algoritmos de tiempo polinomial no determinista, seguimos las convenciones anteriores para algoritmos de tiempo polinomial determinista.

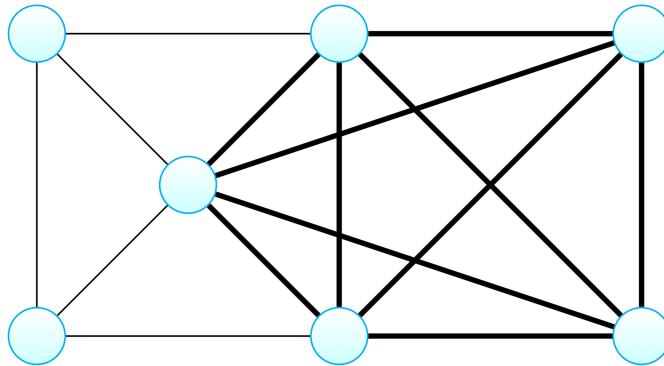

Figura 9.2: Un grafo con un clique−5.

Cada etapa de un algoritmo de tiempo polinomial no determinista debe tener una implementación obvia en tiempo polinomial no determinista en un modelo computacional no determinista razonable.

Analizamos el algoritmo para mostrar que cada rama usa, como mucho, polinomialmente muchas etapas.

9.3.1. Ejemplos de problemas en NP

Un **clique** en un grafo no dirigido es un subgrafo en el que cada dos nodos están conectados por una arista. Una clique$-k$ es un clique que contiene k nodos. La Figura 9.2 ilustra un grafo con un clique-5.

El problema $CLIQUE$ es determinar si un grafo contiene un clique de un tamaño específico. Sea

$$CLIQUE = \{\langle G, k\rangle | G \text{ es un grafo no dirigido con un clique}-k\}.$$

Teorema 11 $CLIQUE$ está en NP.

IDEA DE LA PRUEBA El clique es el certificado.
PRUEBA El siguiente es un verificador V para $CLIQUE$.
V = "Como entrada recibe $\langle\langle G, k\rangle, c\rangle$:

1. Prueba si c es un subgrafo con k nodos en G.

2. Prueba si G contiene todas las aristas que conectan los nodos en c.

3. Si ambas pruebas tienen éxito, entonces *acepta*; de lo contrario *rechaza*."

PRUEBA ALTERNATIVA Si prefiere pensar NP en términos de máquinas de Turing de tiempo polinomial no determinista, puede probar este teorema dando una que decida $CLIQUE$. Observe la similitud entre las dos pruebas.

$N =$ "Como entrada recibe $\langle G, k\rangle$, donde G es un grafo:

1. De manera no determinista selecciona un subconjunto c de k nodos de G.

2. Prueba si G contiene todas las aristas que conectan nodos en c.

3. Si es así, *acepta*; de lo contrario, *rechaza*."

A continuación, consideramos el problema $SUBSET\text{--}SUM$ relativo a la aritmética de enteros. Se nos da una colección de números $x_1, \ldots, x_k$ y un número objetivo t. Queremos determinar si la colección contiene una subcolección que suma t.

Así,

$$SUBSET\text{--}SUM = \left\{ \langle S, t\rangle | S = \{x_1, \ldots, x_k\} \text{ y para algún} \right.$$
$$\left. \{y_1, \ldots, y_l\} \subseteq \{x_1, \ldots, x_k\}, \text{ tenemos que } \sum_i y_i = t \right\}.$$

Por ejemplo, $\langle\{4, 11, 16, 21, 27\}, 25\rangle \in SUBSET\text{--}SUM$ ya que $4 + 21 = 25$. Note que se considera que $\{x_1, \ldots, x_k\}$ y $\{y_1, \ldots, y_l\}$ son **multiconjuntos** por lo que se permiten repeticiones.

Teorema 12 $SUBSET\text{--}SUM \in$ NP.

IDEA DE LA PRUEBA El subconjunto es el certificado.

PRUEBA El siguiente es un verificador V para $SUBSET\text{--}SUM$.

$V =$ "Como entrada recibe $\langle\langle S, t\rangle, c\rangle$:

1. Prueba si c es una colección de números que suman t.

2. Prueba si S continen todos los números en c.

3. Si ambas pruebas tienen éxito, entonces *acepta*; de lo contrario *rechaza*."

PRUEBA ALTERNATIVA También podemos probar este teorema dando una máquina de Turing de tiempo polinomial no determinista para $SUBSET\text{–}SUM$ como sigue.

$N =$ "Como entrada recibe $\langle S, t\rangle$:

1. De manera no determinista selecciona un subconjunto c de los números en S.

2. Prueba si c es una colección de números que suman t.

3. Si es así, *acepta*; de lo contrario, *rechaza*."

Observación 8 Observe que los complementos de estos conjuntos, $\overline{CLIQUE}$ y $\overline{SUBSET\text{–}SUM}$, no son evidentemente miembros de NP. Verificar que algo **no** está presente parece ser más difícil que verificar que **sí** está presente. Creamos una clase de complejidad separada, llamada coNP, que contiene los lenguajes que son complementos de los lenguajes en NP. No sabemos si coNP es diferente de NP.

9.3.2. ¿P = NP?

Como hemos dicho, NP es la clase de lenguajes que se pueden resolver en tiempo polinomial en una máquina de Turing no determinista; o, de forma equivalente, es la clase de lenguajes mediante los cuales se puede verificar la pertenencia al lenguaje en tiempo polinomial.

P es la clase de lenguajes donde se puede decidir la pertenencia en tiempo polinomial. Resumimos esta información de la siguiente manera, donde nos referimos vagamente al tiempo polinomial resoluble como resoluble rápidamente".

P = la clase de lenguajes para los que se puede **decidir** rápidamente la membresía.

NP = la clase de lenguajes para los que se puede **verificar** rápidamente la membresía.

Hemos presentado ejemplos de lenguajes, como $HAMPATH$ y $CLIQUE$, que son miembros de NP pero que no se sabe si están en P.

El poder de la verificabilidad polinomial parece ser mucho mayor que el de la decidibilidad polinomial. Pero, por difícil que sea de imaginar, P y NP

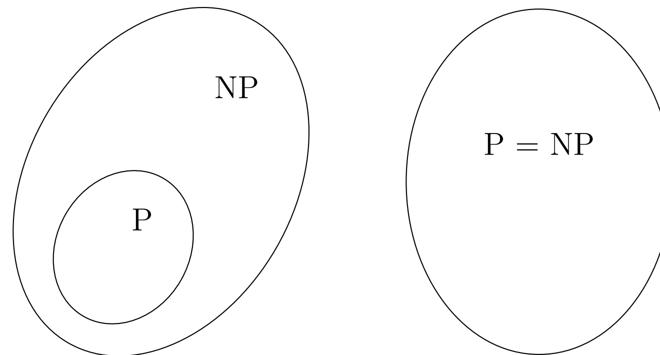

Figura 9.3: Una de las dos posibilidades es correcta.

podrían ser iguales. No se ha podido **demostrar** la existencia de un solo lenguaje en NP que esté en P.

La pregunata si P = NP es uno de los mayores problemas sin resolver en la computación teórica y las matemáticas contemporáneas. Si estas clases fueran iguales, cualquier problema polinomialmente verificable sería polinomialmente decidible.

La mayoría de los investigadores creen que las dos clases no son iguales porque la gente ha invertido un esfuerzo enorme para encontrar algoritmos de tiempo polinomial para ciertos problemas en NP, sin éxito.

Los investigadores también han intentado demostrar que las clases son desiguales, pero eso implicaría demostrar que no existe un algoritmo rápido para reemplazar la búsqueda por fuerza bruta. Hacerlo está actualmente más allá del alcance científico. La Figura 9.3 muestra las dos posibilidades.

El mejor método determinista actualmente conocido para decidir lenguajes en NP utiliza tiempo exponencial. En otras palabras, podemos demostrar que

$$\mathrm{NP} \subseteq \mathrm{EXPTIME} = \bigcup_k \mathrm{TIME}(2^{n^k}),$$

pero no sabemos si NP está contenido en una clase de complejidad temporal determinista más pequeña.

9.4. Completitud-NP

Un avance importante sobre la pregunta P versus NP se produjo a principios de la década de 1970 con el trabajo de Stephen Cook y Leonid Levin. Descubrieron ciertos problemas en NP cuya complejidad individual está relacionada con la de toda la clase.

Si existe un algoritmo de tiempo polinomial para cualquiera de estos problemas, todos los problemas en NP podrían resolverse en tiempo polinomial. Estos problemas se denominan NP-completos. El fenómeno de la completitud-NP es importante tanto por razones teóricas como prácticas

En el aspecto teórico, un investigador que intente demostrar que P no es igual a NP puede centrarse en un problema NP completo. Si algún problema en NP requiere más que tiempo polinomial, uno NP-completo también lo requerirá. Además, un investigador que intente demostrar que P es igual a NP sólo necesita encontrar un algoritmo de tiempo polinomial para un problema NP-completo para lograr este objetivo.

Desde el punto de vista práctico, el fenómeno de la completitud-NP puede evitar perder tiempo buscando un algoritmo de tiempo polinomial inexistente para resolver un problema particular. Aunque no tengamos las matemáticas necesarias para demostrar que el problema no tiene solución en tiempo polinomial, creemos que P no es igual a NP. Entonces, demostrar que un problema es NP-completo es una fuerte evidencia de su **no polinomialidad**.

El primer problema NP-completo que presentamos se llama **problema de satisfacibilidad**. Recuerde que las variables que pueden tomar los valores TRUE y FALSE se denominan **variables booleanas**.

Usualmente, representamos TRUE por 1 y FALSE por 0. Las **operaciones booleanas** AND, OR y NOT, representadas por los símbolos $\wedge, \vee$, y $\neg$ respectivamente, se describen en la siguiente lista. Usamos la barra superior como forma abreviada del símbolo $\neg$, por lo que $\overline{x}$ significa $\neg x$.

$$
\begin{array}{lll}
0 \wedge 0 = 0 & 0 \vee 0 = 0 & \overline{0} = 1 \\
0 \wedge 1 = 0 & 0 \vee 1 = 1 & \overline{1} = 0 \\
1 \wedge 0 = 0 & 1 \vee 0 = 1 & \\
1 \wedge 1 = 1 & 1 \vee 1 = 1 &
\end{array}
$$

Una **fórmula booleana** es una expresión que involucra variables y ope-

raciones booleanas. Por ejemplo,

$$\phi = (\overline{x} \wedge y) \vee (x \wedge \overline{z})$$

es una fórmula booleana. Una fórmula booleana es **satisfactoria** si alguna asignación de 0s y 1s a sus variables hace que la fórmula se evalúe a 1. La fórmula anterior es satisfactoria porque la asignación $x = 0$, $y = 1$ y $z = 0$ hace que ϕ se evalúe a 1. Decimos la asignción satisface ϕ. El problema de satisfacibilidad es probar si una fórmula booleana es satisfactoria. Sea

$$SAT = \{\langle\phi\rangle | \phi \text{ es una fórmula booleana satisfactoria}\}.$$

Ahora planteamos un teorema que vincula la complejidad del problema SAT con las complejidades de todos los problemas en NP.

Teorema 13 $SAT \in$ P si y sólo si P = NP.

A continuación, desarrollamos el método que es fundamental para la demostración de este teorema.

9.4.1. Reducibilidad en tiempo polinomial

Cuando el problema A se reduce al problema B, se puede usar una solución de B para resolver A. Ahora definimos una versión de reducibilidad que toma en cuenta la eficiencia del cálculo. Cuando el problema A se puede reducir eficientemente al problema B, se puede usar una solución eficiente para B para resolver A de manera eficiente.

Definición 14 Una función $f : \Sigma^* \to \Sigma^*$ es una **función computable en tiempo polinomial** si existe alguna máquina de Turing M de tiempo polinomial que se detiene dejando $f(w)$ en su cinta, cuando se inicia con cualquier entrada w.

Definición 15 El lenguaje A es **reducible mediante un mapeo en tiempo polinomial**, o simplemente **reducible en tiempo polinomial**, al lenguaje B, en símbolos $A \leq_P B$, si existe una función computable en tiempo polinomial $f : \Sigma^* \to \Sigma^*$, donde para cada w,

$$w \in A \Leftrightarrow f(w) \in B.$$

A la función f le llamamos **reducción en tiempo polinomial** de A a B.

Hay otras formas eficientes de reducibilidad disponibles, pero la reducibilidad en tiempo polinomial es una forma simple que es adecuada para nuestros propósitos, por lo que no discutiremos las otras aquí.

Al igual que con un mapeo de reducción ordinario, una reducción en tiempo polinomial de A a B proporciona una forma de convertir las pruebas de pertenencia en A a pruebas de pertenencia en B, pero ahora la conversión se realiza de manera eficiente. Para probar si $w \in A$, usamos la reducción f para mapear w a $f(w)$ y probar si $f(w) \in B$.

Si un lenguaje es reducible en tiempo polinomial a un lenguaje que ya se sabe que tiene una solución en tiempo polinomial, obtenemos una solución en tiempo polinomial para el lenguaje original, como en el siguiente teorema.

Teorema 14 Si $A \leq_P B$ y $B \in$ P entonces $A \in P$.

PRUEBA Sea M el algoritmo en tiempo polinomial que decide B y f la reducción en tiempo polinomial de A a B. Describimos un algoritmo de tiempo polinomial N que decide A de la siguiente manera.

$N =$ "Como entrada recibe w:

1. Calcula $f(w)$.

2. Ejecuta M con $f(w)$ sobre la cinta y responde lo que M responda.

Tenemos que $w \in A$ siempre que $f(w) \in B$ porque f es una reducción de A a B. Por lo tanto, M acepta $f(w)$ siempre que $w \in A$. Además, N se ejecuta en tiempo polinomial porque cada un de sus dos pasos se ejecuta en tiempo polinomial. Tenga en cuenta que el paso 2 se ejecuta en tiempo polinomial porque la composición de dos polinomios es un polinomio.

Antes de demostrar una reducción en tiempo polinomial, presentamos $3SAT$, un caso especial del problema de satisfacibilidad en el que todas las fórmulas están en una forma especial. Una **literal** es una variable booleana o una variable booleana negada, como en x o $\overline{x}$.

Una **cláusula** son varias literales conectados con $\vee$s, como en $(x_1 \vee x_2 \vee x_3 \vee x_4)$. Una fórmula booleana está en **forma normal conjuntiva**, llamada **fórmula-cnf**, si comprende varias cláusulas conectadas con $\wedge$s, como en

$$(x_1 \vee \overline{x_2} \vee \overline{x_3} \vee x_4) \wedge (x_3 \vee \overline{x_5} \vee x_6) \vee (x_3 \vee \overline{x_6}).$$

Es una **fórmula-3cnf** si todas las cláusulas tienen tres literales, como en

$$(x_1 \vee \overline{x_2} \vee \overline{x_3}) \wedge (x_3 \vee \overline{x_5} \vee x_6) \wedge (x_3 \vee \overline{x_6} \vee x_4) \wedge (x_4 \vee x_5 \vee x_6).$$

Sea

$$3SAT = \{\langle\phi\rangle|\phi \text{ es una fórmula-3cnf satisfactoria}\}.$$

Si una asignación satisface una fórmula-cnf, cada cláusula debe contener al menos una literal que se evalúe a 1.

El siguiente teorema presenta una reducción en tiempo polinomial del problema $3SAT$ al problema $CLIQUE$.

Teorema 15 $3SAT$ es reducible en tiempo polinomial a $CLIQUE$.

IDEA DE LA PRUEBA La reducción en tiempo polinomial f que mostramos de $3SAT$ a $CLIQUE$ convierte fórmulas en grafos. En los grafos construidos, los cliques de un tamaño específico corresponden a asignaciones satisfactorias de la fórmula. Las estructuras dentro del grafo están diseñadas para imitar el comportamiento de las variables y cláusulas.

PRUEBA Sea ϕ una fórmula con k cláusulas como

$$\phi = (a_1 \vee b_1 \vee c_1) \wedge (a_2 \vee b_2 \vee c_2) \wedge \cdots \wedge (a_k \vee b_k \vee c_k)$$

La reducción f genera la cadena $\langle G, k\rangle$, donde G es un grafo no dirigido definido como sigue.

Los nodos de G están organizados en k grupos de tres nodos, los grupos se llama triples, $t_1, \ldots, t_k$. Cada triple corresponde a una de las cláusulas en ϕ, y cada nodo en un triple corresponde a una literal en la cláusula asociada. Etiquete cada nodo de G con su literal correspondiente en ϕ.

Las aristas de G conectan todos menos dos tipos de pares de nodos en G. No hay una arista presente entre nodos en el mismo triple, y no hay una arista presente entre dos nodos con etiquetas contradictorias, como en x_2 y $\overline{x_2}$. La Figura 9.4 ilustra esta construcción cuando $\phi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2)$.

Ahora demostramos por qué funciona esta construcción. Demostramos que ϕ es satisfactoria si G tiene un clique$-k$.

Suponga que ϕ tiene una asignación satisfactoria. En esa asignación satisfactoria, al menos una literal es verdadera en cada cláusula. En cada triple de G, seleccionamos un nodo correspondiente a una literal verdadera en la asignación satisfactoria.

Si más de una literal es verdadera en una cláusula en particular, elegimos arbitrariamente una de las literales verdaderas. Los nodos recién seleccionados forman un clique$-k$.

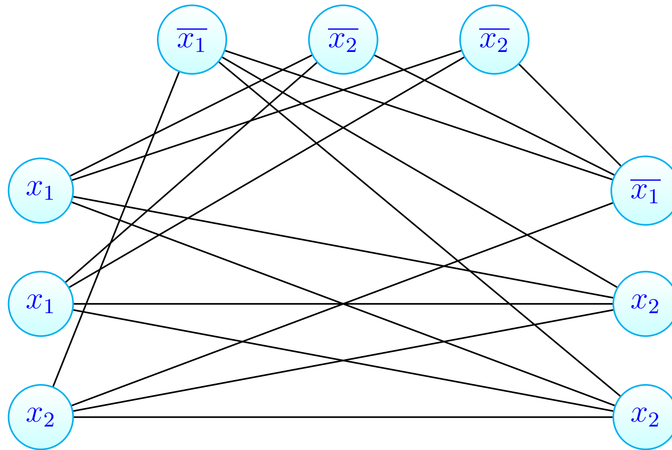

Figura 9.4: El grafo que produce la reducción para $\phi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2)$.

El número de nodos seleccionados es k porque elegimos uno para cada uno de los k triples. Cada par de nodos seleccionados está unido por una arista porque ningún par se ajusta a una de las excepciones descritas anteriormente.

No podían ser del mismo triple porque seleccionamos sólo un nodo por triple. No podían tener etiquetas contradictorias porque las literales asociados eran ambas verdaderas en la asignación satisfactoria. Por tanto, G contiene un clique$-k$.

Suponga que G tiene un clique$-k$. No hay dos nodos del clique en el mismo triple porque los nodos del mismo triple no están conectados por aristas. Por lo tanto, cada uno de los k triples contiene exactamente uno de los k nodos clique.

Asignamos valores de verdad a las variables de ϕ para que cada literal que etiquete un nodo clique se convierta en verdadero. Hacerlo siempre es posible porque dos nodos etiquetados de manera contradictoria no están conectados por una arista y, por lo tanto, ambos no pueden estar en el clique.

Esta asignación a las variables satisface ϕ porque cada triple contiene un nodo clique y, por lo tanto, cada cláusula contiene una literal a la que se le asigna TRUE. Por tanto, ϕ es satisfactoria.

Los teoremas 14 y 15 nos dicen que si $CLIQUE$ se puede resolver en tiempo polinomial, también $3SAT$ se puede resolver en tiempo polinomial. A primera vista, esta conexión entre estos dos problemas parece bastante

notable porque, superficialmente, son bastante diferentes.

Pero la reducibilidad de tiempo polinomial nos permite vincular sus complejidades. Ahora pasamos a una definición que nos permitirá vincular de manera similar las complejidades de toda una clase de problemas.

9.4.2. Definición de Completitud-NP

Definición 16 Un lenguaje B es **NP-completo** si satisface dos condiciones:

1. B está en NP y
2. todo A en NP es reducible en tiempo polinomial a B.

Teorema 16 Si B es NP-completo y $B \in \mathrm{P}$, entonces $\mathrm{P} = \mathrm{NP}$.

PRUEBA Este teorema se deriva directamente de la definición de reducibilidad en tiempo polinomial.

Teorema 17 Si B es NP-completo y $B \leq_P C$ para C in NP, entonces C es NP-completo.

PRUEBA Ya sabemos que C está en NP, por lo que debemos demostrar que cada A en NP es reducible en tiempo polinomial a C. Como B es NP-completo, cada lenguaje en NP es reducible en tiempo polinomial a B, y B a su vez es reducible en tiempo polinomial a C.

Componga las reducciones de tiempo de polinomial; es decir, si A es reducible en tiempo polinomial a B y B es reducible en tiempo polinomial a C, entonces A es reducible en tiempo polinomial a C. Por tanto, cada lenguaje en NP es reducible en tiempo polinomial a C.

9.4.3. El teorema de Cook-Levin

Una vez que tenemos un problema NP-completo, podemos obtener otros mediante reducciones de tiempo polinomial. Sin embargo, establecer el primer problema NP-completo es más difícil. Ahora lo hacemos probando que SAT es NP-completo.

Teorema 18 SAT es NP-completo.

Este teorema implica el Teorema 13.

IDEA DE LA PRUEBA Demostrar que SAT está en NP es fácil y lo haremos en breve. La parte difícil de la demostración es mostrar que cualquier lenguaje en NP es reducible en tiempo polinomial a SAT.

Para hacerlo, construimos una reducción en tiempo polinomial para cada lenguaje A en NP a SAT. La reducción para A toma una cadena w y produce una fórmula booleana ϕ que simula la máquina NP para A con la entrada w.

Si la máquina acepta, ϕ tiene una asignación satisfactoria que corresponde al cálculo de aceptación. Si la máquina no acepta, ninguna asignación satisface ϕ. Por lo tanto, w está en A si y solo si ϕ es satisfactoria.

En realidad, construir la reducción para que funcione de esta manera es una tarea conceptualmente simple, aunque debemos lidiar con muchos detalles. Una fórmula booleana puede contener las operaciones booleanas AND, OR y NOT, y estas operaciones forman la base de los circuitos utilizados en las computadoras electrónicas.

Por tanto, no sorprende el hecho de que podamos diseñar una fórmula booleana para simular una máquina de Turing. Los detalles están en la implementación de esta idea.