Estructuras de Datos y Algoritmos usando Java

José Galaviz Casas

Departamento de Matemáticas
Facultad de Ciencias
UNAM

Índice General

Introducción 1

La introducción es divertida y hasta traviesa, el autor se regocija en la presentación de los elementos y recursos de que echará mano en el resto de la obra. [...] Sin duda la audiencia disfrutará de la obra sin importar su edad.

—Presentación del cuento sinfónico *Babar el Elefantito*, INBA, 1998.

1.1 Motivación

A menos que se tengan niños pequeños (cómo es mi caso), los diversos objetos en casa suelen estar acomodados de acuerdo a cierto orden, agrupados con los de su clase: los libros en un librero, la despensa en la alacena, los alimentos perecederos en el refrigerador, la ropa en el ropero o en el "closet", etc. En nuestros lugares de trabajo las cosas son similares, en general los objetos de los que nos servimos suelen estar organizados de acuerdo con un orden preestablecido. La razón para mantener las cosas en un estado por lo menos cercano a la organización perfecta, es la eficiencia. Todos hemos padecido el disgusto, cuando no la desesperación, de buscar algo que hemos dejado en algún lugar ajeno a su colocación habitual. Se pierden miserablemente varias horas buscando, a veces infructuosamente, un objeto perdido en la complejidad de nuestro entorno.

Hay casos extremos en los que la organización de los objetos es fundamental. En

el caso de una biblioteca, por ejemplo, el orden de los libros en la estantería debe ser perfecto. Pensemos en buscar un libro específico en una biblioteca en la que los volúmenes están acomodados aleatoriamente o siguiendo algún criterio extravagante (por tamaño, por grosor o por lo que sugiere el título). ¿Cuanto tiempo tomará encontrar un libro en particular? No podemos saberlo, pero en el peor de los casos será el último luego de haber recorrido todos los títulos en el acervo, así que nos tardaremos más cuanto mayor sea este. Los libros están organizados estrictamente para poder encontrar cualquiera de ellos rápidamente. Lo mismo ocurre con las historias médicas en un hospital, con los archivos de una escuela, los de la compañía de teléfonos o del registro civil. A pequeña escala también ocurre en nuestros hogares. Idealmente organizamos los CD's, los libros, las cuentas por pagar, las facturas, la despensa, el contenido del refrigerador y la ropa por lavar. Toda esta organización es para poder hacer rápidamente ciertas operaciones: poner música, lavar ropa, comer o cepillarse los dientes. Poseer cierta estructura en los objetos que manipulamos permite optimizar el tiempo invertido en las operaciones que hacemos con ellos.

En el contexto de los programas de computadora la situación no es diferente. Los programas operan con datos de entrada, los manipulan a través de una secuencia finita de pasos y luego nos entregan resultados basados en esos datos. Nuestros programas serán más eficientes, aprovecharán mejor los recursos disponibles (tiempo de procesador, memoria y acceso a dispositivos de E/S), dependiendo de la manera en que son organizados los datos de entrada o los resultados parciales del proceso. Igual que en una biblioteca, la manera de organizar los datos establece vínculos entre ellos y se conforma lo que denominaremos una *estructura de datos*, uno de los temas fundamentales del presente texto.

Para poder evaluar la eficiencia de nuestros algoritmos en general, y en particular el impacto que diversas estructuras de datos tienen sobre dicha eficiencia, requeriremos de métodos que nos permitan cuantificar los recursos consumidos. El término adecuado para referirse a esto es: *análisis de algoritmos*, el otro tema central del texto.

Ambas cosas tienen singular relevancia tanto en el ámbito de la computación teórica como en el de la práctica de la programación. No puede haber una propuesta seria de un nuevo algoritmo o un protocolo de comunicación que no esté sustentada por un análisis teórico de los mismos. Asimismo no es posible pensar en que las decisiones de un programador respetable no sean las adecuadas para optimizar el desempeño de sus programas.

Abstracción. El análisis de algoritmos y las estructuras de datos, como el resto de las cosas que se hacen en ciencia, están basados en abstracciones. La abstracción es el camino que los seres humanos utilizamos para comprender la complejidad. La teoría de la gravitación, el álgebra de Boole, los modelos económicos y los ecológicos son ejemplos de ello. Todos son abstracciones hechas en un intento por comprender fenómenos complejos en los que intervienen una multitud de factores con intrincadas relaciones entre ellos. Una abstracción es una descripción simplificada de un sistema y lo más importante en ella es que *enfatiza* las características esenciales y descarta aquellas que se consideran sin

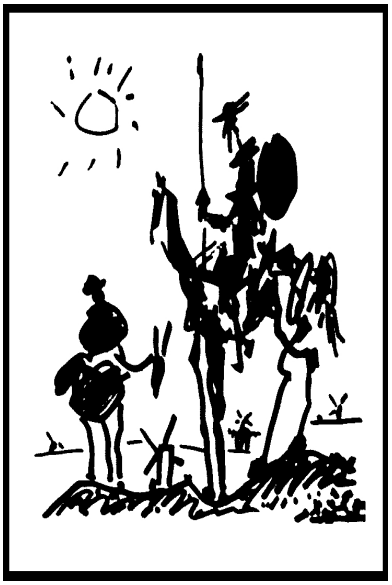

Figura 1.1: El *Quixote* de Pablo Picasso. Un buen ejemplo de abstracción

importancia para explicar el comportamiento del sistema. En la teoría de la gravitación, por ejemplo, no son consideradas las propiedades magnéticas o químicas de los cuerpos, solo interesan cosas como su masa y las distancias entre ellos.

Formular una buena abstracción es como hacer una buena caricatura. Con unos cuantos trazos simples el caricaturista representa un personaje que es posible reconocer sin ambigüedad, nada sobra y nada falta, ha sabido encontrar aquellos rasgos que caracterizan a la persona y ha sabido resaltarlos aislándolos de todos los que no son relevantes. En la figura 1.1 se muestra un buen ejemplo de abstracción. Todos podemos reconocer a Don Quijote y Sancho en sus respectivas monturas y en el episodio de los molinos de viento, no hacen falta los colores ni las texturas ni los detalles de rostros, cuerpos y paisaje.

1.2 TDA's, Estructuras de Datos, Clases y Objetos

Por supuesto la programación de computadoras es un ejercicio de abstracción. Particularmente cuando se definen las propiedades y los atributos de los datos que han de manipular los programas. En el paradigma actual de la programación se pretende modelar con entes abstractos llamados *objetos*, a los actores reales involucrados en el problema que se intenta resolver con el uso del programa. Los atributos y el comportamiento de estos objetos están determinados por la abstracción que el programador

hace de los actores reales con base en el contexto del problema. Si lo que se pretende es modelar personas de una cierta localidad con el propósito de estudiar sus características genéticas, entonces convendrá considerar como abstracción de persona un conjunto de atributos como la estatura, el color de ojos, de cabello, de piel o el tipo sanguíneo; si el problema en cambio, está vinculado con las características socio-económicas entonces los atributos que convendría incluir en la abstracción son los ingresos mensuales, el gasto promedio y el tipo de trabajo de las personas reales.

A fin de cuentas lo que hace el programador es crear un nuevo tipo de dato que modele los objetos involucrados en el problema que pretende resolver. Al igual que los tipos elementales que posee el lenguaje de programación que utilizará, el tipo definido por el programador posee un conjunto de posibles valores, un *dominio* y un conjunto de operaciones básicas definidas sobre los elementos de ese dominio. Un modelo así se conoce como un *Tipo de Dato Abstracto*.

Tipo de Dato Abstracto (TDA).

Un tipo de dato abstracto o *TDA* es un modelo que describe a todos los elementos de un cierto conjunto dominio, especificando su comportamiento bajo ciertas operaciones que trabajan sobre ellos. El uso del término abstracto significa, cómo sabemos, que sólo atiende a aquellas cualidades que interesan de los elementos del dominio y que son manifiestas a través de las operaciones. Esto significa, en términos matemáticos, que un TDA no dice más que aquello que se puede inferir del comportamiento de sus operaciones, es lo que suele llamarse un *sistema formal*, cómo lo son la geometría euclidiana, la lógica de primer orden o los números naturales. Como en todo sistema formal, el comportamiento básico de los elementos del dominio está regido por una serie de premisas elementales que se suponen verdaderas *a priori*, un conjunto de *axiomas*.

La cualidad más relevante en la abstracción de datos es que, al elaborar un nuevo modelo para un tipo de dato lo importante es decidir *qué* es lo que se puede hacer con un dato de ese tipo y no *cómo* es que se hace. La especificación de lo que se puede hacer (las operaciones) debe ser lo suficientemente rigurosa como para explicar completamente su efecto sobre los datos que componen al tipo, pero no debe especificar cómo es que se logra dicho efecto ni cómo estos datos están organizados. Es decir, la definición de un TDA es diferente de su implementación concreta en un lenguaje de programación.

Puede resultar curioso para el lector que se hable de un tipo de dato en un lenguaje de programación como algo concreto. Después de todo no es frecuente encontrarse en la calle con el `float` de Java abordando el metro. Lo que se pretende decir es que un TDA está en un nivel de abstracción mayor que su realización en un lenguaje de programación. Pongamos un par de ejemplos.

EJEMPLO 1.1

Seguramente el lector ya conoce el tipo de dato `boolean` del lenguaje Java. Pues bien, esta es la implementación "concreta" de un TDA en un lenguaje de programación, a saber, el sistema formal determinado por los postulados que elaboró el matemático Edward

Huntington en 1904. En el TDA 1.1 hemos escrito de manera formal los postulados, en adelante usaremos la misma notación para especificar nuestros TDAs.

TDA 1.1 Especificación del TDA Booleano.

TDA **Booleano**
  CONSTANTES
    VERDADERO, FALSO : Booleano
  OPERACIONES
    and : Booleano $\times$ Booleano $\mapsto$ Booleano
    or : Booleano $\times$ Booleano $\mapsto$ Booleano
    not : Booleano $\mapsto$ Booleano
  VARIABLES
    x, y, z : Booleano
  AXIOMAS
    [B1a] $and(\text{VERDADERO}, x) = x$
    [B1b] $or(\text{FALSO}, x) = x$
    [B2a] $and(x, y) = and(y, x)$
    [B2b] $or(x, y) = or(y, x)$
    [B3a] $and(x, or(y, z)) = or(and(x, y), and(x, z))$
    [B3b] $or(x, and(y, z)) = and(or(x, y), or(x, z))$
    [B4a] $and(x, not(x)) = \text{FALSO}$
    [B4b] $or(x, not(x)) = \text{VERDADERO}$
    [B5a] $not(\text{VERDADERO}) = \text{FALSO}$
    [B5b] $not(\text{FALSO}) = \text{VERDADERO}$
FIN **Booleano**

Con base en los axiomas del TDA es posible demostrar, por ejemplo, lo siguiente:

Teorema 1.1 [Idempotencia] Si x es Booleano, entonces:

(a) $or(x, x) = x$

(b) $and(x, x) = x$

Demostración:

$$
\begin{aligned}
x &= or(x, \text{FALSO}) && \text{B1b, B2b} \\
&= or(x, and(x, not(x))) && \text{B4a} \\
&= and(or(x, x), or(x, not(x))) && \text{B3b} \\
&= and(or(x, x), \text{VERDADERO}) && \text{B4b} \\
&= or(x, x) && \text{B1a, B2a}
\end{aligned}
$$

□

Ahora podemos afirmar que, si el tipo `boolean` de Java es una implementación correcta del TDA Booleano, entonces debe cumplir también con la regla de idempotencia y de hecho con cualquier otra regla derivada del conjunto de axiomas del TDA.

Incluso podríamos ir un poco más allá, podríamos pensar en implementar una clase de objetos `Conjunto`, con dos constantes definidas: el conjunto `universal` y el conjunto `vacio` y tres operaciones: `complemento`, `interseccion` y `union` de tal forma que cumplan con los postulados de Huntington haciendo el mapeo:

- `universal` $\mapsto$ VERDADERO
- `vacio` $\mapsto$ FALSO
- `complemento` $\mapsto$ *not*
- `union` $\mapsto$ *or*
- `interseccion` $\mapsto$ *and*

De ser así, este tipo de dato también cumpliría con todos los teoremas derivados en el TDA Booleano. Ahora es claro que un TDA es un modelo formal que admite múltiples implementaciones diferentes. Como en todo sistema formal, lo que se puede deducir es válido por las reglas de escritura especificadas en los axiomas, sin importar el significado. Lo que se pueda deducir del TDA Booleano es verdadero independientemente de la implementación, no importa si se está hablando de conjuntos, de cálculo de predicados o de funciones de conmutación (otra "implementación" de Booleano), no importa la *semántica*, sólo la *forma*.

◊

EJEMPLO 1.2

Probablemente el lector ya entró en contacto, en alguno de sus cursos previos, con los axiomas que el matemático italiano Guiseppe Peano formuló para construir los números

TDA 1.2 Especificación del TDA Naturales.

TDA **Natural**
REQUIERE
Booleano
CONSTANTES
0 : Natural
OPERACIONES
$sucesor$: Natural $\mapsto$ Natural
$igual$: Natural $\times$ Natural $\mapsto$ Booleano
VARIABLES
x, y : Natural
AXIOMAS
[N1] $igual(0, 0) =$ VERDADERO
[N2] $igual(x, y) = igual(y, x)$
[N3] $igual(sucesor(x), 0) =$ FALSO
[N4] $igual(sucesor(x), sucesor(y)) = igual(x, y)$
FIN **Natural**

naturales[1]. Podemos pensar en esto como un tipo de dato abstracto, tal como se muestra en el TDA 1.2.

Basta con renombrar, como estamos acostumbrados:

$$sucesor(0) = 1, sucesor(sucesor(0)) = 2, \ldots$$

y tenemos especificado todo el conjunto que solemos denotar con $\mathbb{N}$, el dominio de nuestro TDA.

Ahora bien, con base en el TDA y usando el principio de inducción es posible definir más operaciones en el conjunto. Por ejemplo la suma y el producto.

Definición 1.1 Sean x y y dos elementos del tipo Natural. Se definen:

suma

- $suma(x, 0) = x$
- $suma(x, sucesor(y)) = sucesor(suma(x, y))$
- $suma(x, y) = suma(y, x)$

producto

- $producto(x, 0) = 0$

[1] En este texto los números naturales incluyen al cero, es decir $\mathbb{N} = \{0, 1, \ldots\}$.

- $producto(x, sucesor(y)) = suma(x, producto(x, y))$
- $producto(x, y) = producto(y, x)$

En lenguajes de programación como C y C++, existen tipos de datos enteros sin signo (`unsigned int`, por ejemplo) que constituyen una implementación del TDA que acabamos de especificar, pero no la única. También podríamos decir que nuestro dominio no es todo $\mathbb{N}$, sino sólo el conjunto $\{0, 1\}$ y decir que $sucesor(0) = 1$ y $sucesor(sucesor(0)) = 0$ y habremos definido a los enteros módulo 2.

$\Diamond$

Precondiciones y postcondiciones.

En los ejemplos de TDA revisados en este capítulo, las operaciones no terminan con una condición de excepción, es decir, en nuestros axiomas nunca aparece, del lado derecho, la palabra *error*, lo que significaría que la operación no puede llevarse a cabo normalmente debido a que, al menos uno de sus argumentos de entrada, no satisface alguna condición necesaria para la aplicación de la operación. Si, por ejemplo, tuviéramos definida una operación *cociente(x,y)* que hiciera la división entera de su primer argumento entre el segundo, tendríamos que decir que $y = 0$ es un error. A las condiciones que deben satisfacer los argumentos de una operación para poder efectuarse se les denomina *precondiciones* y están implícitas en los axiomas del TDA. De la misma forma en que una operación necesita ciertas garantías para poder llevarse a cabo, también ofrece ciertas garantías en sus resultados. La operación *producto* definida en nuestro último ejemplo, garantiza que regresa cero si y sólo si alguno de sus argumentos es cero y garantiza también que regresa x veces y como resultado. A las condiciones que son satisfechas por los resultados de una operación se les denomina *postcondiciones*. En futuros TDAs las pre y post condiciones serán más claras y útiles.

Los axiomas de un TDA en general, así como los que establecen las precondiciones en particular, son de gran utilidad para demostrar la corrección[2] de los algoritmos que usan el TDA, porque encadenando los resultados de una operación y sus postcondiciones con los argumentos y precondiciones de otra, se forma una secuencia de deducciones que finalmente llevan a garantizar que los resultados son correctos. Desde este punto de vista, la corrección de un algoritmo se establece como un teorema dentro del sistema formal definido por el TDA.

El concepto de *clase* en los lenguajes de programación orientados a objetos es muy similar a nuestro concepto de TDA y esto podría dar lugar a una confusión. Conviene entonces hacer una distinción clara entre ambos y mencionar de paso otros términos que estaremos utilizando a lo largo del texto.

TDA y el concepto de clase.

En un lenguaje de programación, una clase es la definición de los atributos (datos característicos) y las operaciones que son comunes a un conjunto de objetos, a los que

[2] A veces se prefiere usar el término "correctez" para referirse a la cualidad de correcto de un algoritmo. En el diccionario de la Real Académia Española de la Lengua, sin embargo, se especifica que el término *corrección* se aplica tanto a la acción de corregir, como a la cualidad de ser correcto, por lo que acuñar un nuevo término es, en el mejor de los casos, innecesario.

se les denomina *instancias de la clase*. Hasta aquí el parecido con un TDA. La clase incluye además la implementación de las operaciones (que en este contexto se llaman métodos), es decir no sólo especifica qué se puede hacer sino también cómo se hace, lo que contradice lo mencionado en el caso de un TDA. Lo cierto es que una clase puede verse como la implementación de un TDA en un lenguaje de programación concreto.

Estructura de datos.

Como parte de la definición de una clase se incluyen los datos (atributos) que poseen todas y cada una de las instancias de dicha clase. Además se especifica también cómo es que estos datos están organizados en la memoria, cuáles son los vínculos entre ellos y cómo pueden ser accedidos por los métodos propios de la clase. Esto es la *estructura de datos* subyacente a la clase.

Ahora tenemos el panorama completo: Un TDA es la definición formal de un conjunto de cosas en un cierto dominio y ciertas operaciones, cuyas propiedades son especificadas por un conjunto de axiomas. Una clase es la realización de un TDA usando la sintaxis de un lenguaje de programación particular y en ella se especifican como se llevan a cabo internamente las operaciones y que estructura poseerán los datos especificados en el TDA.

Precisemos con un ejemplo. Todo mundo sabe lo que es un automóvil, sabe que significa acelerarlo, frenarlo, prender las luces o los limpiadores de parabrisas, sabemos que posee un tanque de gasolina, un tacómetro y un velocimetro, podríamos poner esto en una especificación de lo que debe hacer un automóvil y obtendríamos un TDA que lo describe genéricamente y en el que los datos son el combustible disponible, la velocidad y el resto de las variables del tablero. Las operaciones serían, por ejemplo: acelerar, frenar y cambiar de velocidad. Cada una de estas operaciones altera los valores de los datos mencionados. Podemos también pensar en el diseño de un modelo en particular, un automóvil compacto de Ford, por ejemplo. Este diseño equivale a una definición de clase y en él están contempladas las cosas que hacen que el automóvil funcione, se hacen explícitas las relaciones entre las partes y se dice cómo funciona cada una. En este diseño se usan partes preexistentes: unos frenos de disco, un cigüeñal, un tipo particular de suspensión y en general, cosas que sirven para implementar el automóvil. Esto es el equivalente a las estructuras de datos. Por último podemos fijar nuestra atención en uno de esos automóviles circulando por la calle, una instancia de la clase, un objeto funcional único identificado unívocamente por su placa de circulación, el equivalente de un objeto en un programa en ejecución.

Encapsulación.

El paradigma de la programación orientada a la representación de objetos, o simplemente programación orientada a objetos (POO), enfatiza las características de la abstracción de datos mencionadas anteriormente. El hecho de que al modelar mediante TDA's deba pensarse primero en *qué* es lo que se puede hacer con cierto tipo de dato antes que en *cómo* hacerlo, nos lleva, en el contexto de objetos, a separar la estructura interna (el estado) del comportamiento del objeto. Antes que pensar en cuál debe ser la estructura de datos que se utiliza para implantar el objeto, debe pensarse en cual debe ser su comportamiento, es este, a fin de cuentas, el que le interesará a cualquier usuario potencial del objeto: qué servicios ofrece sin importar como son implementados

dichos servicios. En programación esto se conoce como *encapsulación*, ocultar los detalles de implantación de los objetos, impedir el acceso directo a la estructura de datos que constituye el estado del objeto y ofrecer, a cambio, una interfaz que permita, mediante ciertas operaciones, conocer y modificar dicho estado.

Bajo acoplamiento.

Si la estructura interna de un objeto que se encuentra definido en una clase es privada, es decir, inaccesible directamente para otros objetos o módulos del programa, entonces uno puede cambiar la estructura interna y/o el código que ejecuta el objeto, sin que se vea alterada su funcionalidad y por tanto la del sistema en el que se encuentra inmerso. Es decir, la funcionalidad del sistema depende de los servicios que los objetos proporcionan y no de la manera en que dichos servicios son implementados. Esta cualidad deseable se denomina *bajo acoplamiento*. El acoplamiento es, pues, una medida de la iterdependencia de diversos módulos de programación en términos de datos. Lo correcto es que un módulo dependa de los servicios que otro proporciona, pero no de la manera en que esos servicios están implementados.

Alta cohesión.

Es también deseable que todos los datos requeridos para que un objeto funcione formen parte de los atributos propios del objeto, de tal forma que no dependa del acceso directo a datos contenidos en otros objetos o módulos del sistema. A esta cualidad se le denomina *alta cohesión*. Por supuesto está vinculada con el acoplamiento, módulos con baja cohesión tienden a estar acoplados con otros. Lo deseable es que todas las variables correlacionadas con la operación de un módulo pertenezcan a él. Cuanto más complejo sea un módulo, cuanto más heterogéneas sean las labores que debe realizar, la cantidad de datos inconexos que necesita es mayor. Tanto la cohesión como el acoplamiento son cualidades que se obtienen del diseño, no de la implementación: una correcta descomposición de las tareas a realizar a través del análisis del problema, procurando minimizar las labores de cada módulo es lo que garantiza la alta cohesión y el bajo acoplamiento.

Tanto la cohesión como el acoplamiento son conceptos acuñados antes de la programación con objetos, son herencia del paradigma de la programación estructurada, por eso se ha utilizado el término "módulo" en los párrafos anteriores, en vez de "objeto". Probablemente por ser conceptos anteriores a la programación con objetos suelen ser ignorados en los textos más modernos, prefiero pensar que ya no se mencionan porque se dan por obvios, siguen siendo y serán, cómo todas las premisas de buen diseño aprendidas de la programación estructurada, prioritarios.

1.3 Análisis de Algoritmos.

La otra abstracción fundamental que necesitamos es la que nos permitirá determinar la eficiencia de nuestros algoritmos.

Decimos que algo es eficiente, una máquina, una persona, un proceso, etcétera, si logra hacer una tarea que le ha sido encomendada con el mínimo consumo de recursos.

La eficiencia es una medida de qué tan bien son aprovechados los recursos para lograr el objetivo. En el contexto de los algoritmos los recursos a considerar son generalmente dos: el tiempo que toma su ejecución y la cantidad de memoria que se utiliza durante el proceso. Existen entornos particulares en los que es útil considerar otros recursos: la cantidad de mensajes enviados por un protocolo de comunicación, por ejemplo. Pero el tiempo y la memoria son recursos presentes en todo algoritmo.

Medir significa comparar dos cosas, cuando medimos la longitud de una mesa usamos un *patrón* estándar que comparamos con la mesa, el metro o el centímetro por ejemplo y luego podemos establecer la longitud de la mesa con base en ese patrón: 1.23 metros, por decir algo. Una medida así, es útil porque es significativa para todo el mundo, todo aquel que conoce el metro tendrá una idea clara de la longitud de la mesa. Si lo que queremos es medir el tiempo que toma a un algoritmo resolver un problema, entonces podríamos pensar en programarlo y contar cuantos segundos o minutos le toma a la computadora ejecutarlo.

Complejidad de entrada.

Pero hay otro factor a tomar en cuenta. Recordemos que un algoritmo es un conjunto ordenado y finito de pasos, que operan sobre ciertos datos de entrada para producir los datos de salida. Así que en general, el tiempo invertido por un algoritmo en su ejecución depende de ciertas características de sus datos de entrada. Un algoritmo de ordenamiento, por ejemplo, tardará más cuanto mayor sea el número de elementos a ordenar y/o de que tan desordenados estén originalmente. Un algoritmo que busca el mínimo elemento de una lista se tarda siempre lo mismo si la lista ha sido previamente ordenada y de no ser así, será tanto más tardado cuanto mayor sea la lista. A las consideraciones hechas sobre las características de los datos de entrada que influyen en el desempeño de un algoritmo les llamaremos *complejidad de entrada*.

Lo que necesitamos es, entonces, un mecanismo que nos permita medir el tiempo[3] que tarda la ejecución de un algoritmo en función de la complejidad de su entrada. Esto es similar a lo que se suele hacer en el caso de las competencias atléticas de pista: se pone a los diferentes competidores en un mismo lugar, sometidos a las mismas condiciones ambientales, a recorrer una cierta distancia fijada de antemano y el propósito es determinar un ganador comparando los tiempos de cada competidor. La diferencia es que nosotros quisiéramos obtener esa medida para toda posible complejidad de entrada, lo que equivale a poner a correr a los competidores todas las posibles distancias. Esto es, por supuesto, imposible tanto para los algoritmos como para los corredores (para los que además resulta poco saludable). Así que necesitamos un medio que nos permita predecir cuanto tiempo sería consumido ante una cierta complejidad de entrada sin tener que medirlo explícitamente.

Esto es lo que se hace todo el tiempo en las ciencias experimentales. A partir de lo datos recopilados haciendo experimentos en un rango limitado de posibles valores de las variables de control, se deduce una regla de correspondencia que determina el valor de las

[3]Estaremos usando el tiempo como el recurso que nos interesa cuantificar, pero todo lo que se dirá puede aplicarse igualmente a algún otro recurso, como el consumo de memoria, por ejemplo.

variables que dependen de ellas. Se generaliza el comportamiento a partir de un conjunto pequeño de observaciones. Esto equivaldría a tratar de determinar la ecuación que calcula el tiempo de un corredor, a partir de las observaciones hechas poniéndolo a correr diversas distancias en un intervalo razonable (sin que muera por insuficiencia cardiaca), o la ecuación que determina el tiempo que tarda un algoritmo, a partir de observaciones hechas ejecutándolo sobre datos de entrada con complejidades controladas en cierto intervalo. Los estudiantes de física están familiarizados con este tipo de procedimientos, los utilizan para deducir las ecuaciones de cinemática del tiro parabólico o de la caída de los cuerpos o del movimiento pendular, pero hay más abstracción de por medio.

En efecto, cuando en el laboratorio de física se toman mediciones tendientes a calcular las leyes del movimiento, se desprecian variables que están allí, influyen determinantemente en el comportamiento observado, pero complicarían innecesariamente las cosas haciéndonos perder de vista lo esencial. Es por eso que se hacen abstracciones que desprecian características insoslayables, pero que harían imposible deducir lo fundamental: la resistencia del viento, la masa de los resortes, la fricción. En nuestro caso hay también variables que siempre estarán allí, que influirán determinantemente en el tiempo de ejecución de un algoritmo y que debemos despreciar en nuestra abstracción. Todo algoritmo en ejecución estará implementado en algún lenguaje de programación, será puesto en el lenguaje binario de alguna computadora particular, por un compilador particular y se ejecutará en un ambiente administrado por un sistema operativo, todo ello constituye el entorno de ejecución del algoritmo y deberemos ignorarlo si se pretende obtener una regla que nos permita predecir el comportamiento del algoritmo como ente abstracto.

Supongamos que programamos el algoritmo que queremos evaluar en una computadora M usando el lenguaje de programación P, con el compilador C y sistema operativo S y medimos cuantos segundos se tarda en ejecutarlo para distintas complejidades de entrada $\{n_1, \ldots n_k\}$. Quisiéramos deducir, a partir de los tiempos medidos: $t_1, \ldots t_k$, la regla de correspondencia f que nos entregue, dada una cierta complejidad de entrada n, el tiempo $t = f(n)$ que se estima que tardará en algoritmo procesando una entrada con esa complejidad. Pero queremos también ignorar el efecto que tienen M, P, C y S sobre $t_1, \ldots, t_k$. Esto significa que no nos interesa realmente una expresión detallada de f, sino sólo la forma general de su dependencia de n. Perderemos necesariamente mucha información, pero que supondremos es introducida por el entorno y no proviene de la manera en que opera el algoritmo.

EJEMPLO 1.3

En el libro de Jon Bentley ([1], columna 8), se plantea un problema para el que se poseen diversos algoritmos que lo resuelven. Dado un vector de tamaño arbitrario, con entradas reales arbitrarias, el problema consiste en encontrar un segmento de él, tal que la suma de las entradas en el segmento sea máxima. Por segmento se entiende aquí un

conjunto de entradas consecutivas, es decir un subvector del vector dado como entrada. La complejidad de la entrada está dada, en este caso, por el tamaño del vector completo.

En una primera aproximación, el algoritmo planteado para resolver el problema es el siguiente:

```
MaxSum1(v, n)
 1  sumamax ← 0
 2  for i ∈ {0, ..., n − 1} do
 3      for j ∈ {i, ..., n − 1} do
 4          sumactual ← 0
 5          for k ∈ {i, ..., j} do
 6              sumactual ← sumactual + v[k]
 7          endfor
 8          if sumactual > sumamax then
 9              sumamax ← sumactual
10          endif
11      endfor
12  endfor
13  return sumamax
14  end
```

Donde $\mathbf{v}$ denota al vector y n a su tamaño. El algoritmo considera todos los subvectores posibles y calcula la suma de las entradas de cada uno de ellos. Los índices i y j sirven para especificar el subvector a considerar y luego, usando el índice k se suman las entradas en ese intervalo. En la figura 1.2 aparecen graficados los tiempos observados experimentalmente en la ejecución del algoritmo en función de la complejidad de entrada n (tamaño del vector), cada punto graficado es la media tomada sobre 20 observaciones. En la gráfica se observa que los puntos están sobre una curva que crece rápidamente conforme crece el tamaño del vector de entrada, sin embargo este crecimiento no es tan acelerado como podríamos esperar si se tratase de una relación exponencial, de hecho los valores mostrados se parecen a los de n^3. Así que la curva parece corresponder a una relación del tipo $t = k\,n^3$. Esta hipótesis puede comprobarse, dado que, de ser cierta: $t^{1/3} = k'\,n$ sería una recta, lo que en efecto podemos observar en la gráfica de la derecha.

Hasta aquí nuestro nivel de detalle. Hemos deducido que, en el caso del algoritmo 1, el tiempo t de ejecución depende de un polinomio de grado tres del tamaño del vector n. No nos interesa qué polinomio en específico, sólo su grado. Mayores detalles serian superfluos.

Una mejor estrategia para resolver el problema, consiste en, dado el índice de una entrada del vector, calcular la suma hasta cualquier otra entrada manteniendo fijo el índice inicial e iterar este proceso sobre dicho índice inicial. Esto da lugar al algoritmo 2:

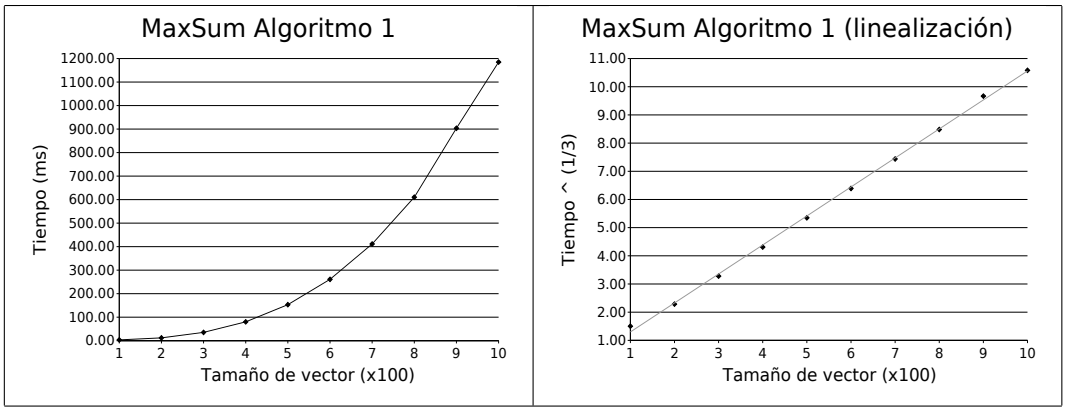

Figura 1.2: Gráfico de los datos observador para el algoritmo MaxSum1 (20 experimentos por punto).

```
MaxSum2(v, n)
 1  sumamax ← 0
 2  for i ∈ {0, ..., n − 1} do
 3      sumactual ← 0
 4      for j ∈ {i, ..., n − 1} do
 5          sumactual ← sumactual + v[j]
 6          if sumactual > sumamax then
 7              sumamax ← sumactual
 8          endif
 9      endfor
10  endfor
11  return sumamax
12  end
```

El la figura 1.3 se muestran los valores experimentales obtenidos para los tiempos de ejecución de este algoritmo. Nuevamente observamos una relación potencial, en este caso parece corresponder a un polinomio de grado 2. En la parte derecha de a figura se comprueba dicha hipótesis.

Una nueva mejora en el desempeño la obtenemos partiendo el problema original en dos subproblemas más simples. Calcular el subvector máximo entre las entradas de índice inf y sup consiste en: calcular el subvector máximo en la primera mitad, comprendida entre las entradas inf e $(inf + sup)/2$, calcular el subvector máximo entre las entradas $(inf + sup)/2 + 1$ y sup, calcular el subvector máximo en la intersección de las dos mitades y regresar el máximo de los tres.

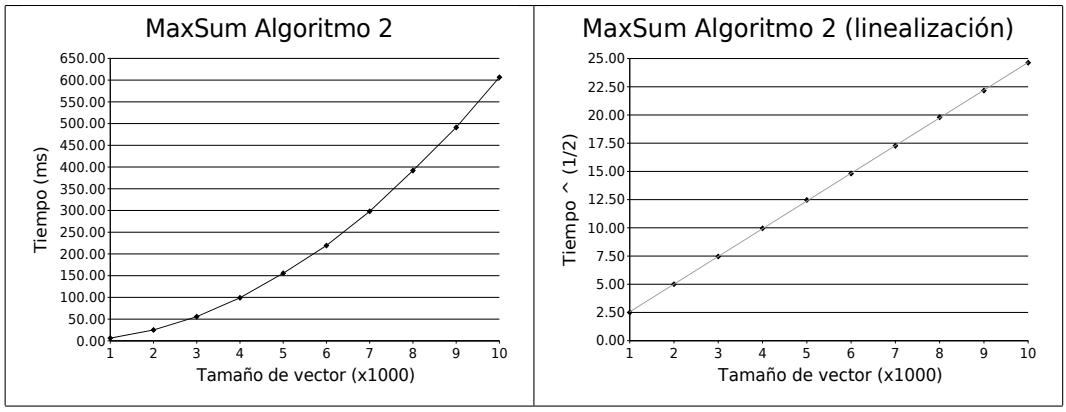

Figura 1.3: Gráfico de los datos observador para el algoritmo MaxSum2 (50 experimentos por punto).

```
MaxSum3(v, n)
 1  return SumParticion(v, 0, n − 1)
 2  end

SumParticion(v, inf, sup)
 1  if inf > sup then
 2      return 0
 3  endif
 4  if inf = sup then
 5      return Maximo(0, v[inf])
 6  endif
 7  m ← (inf + sup)/2
 8  sumactual ← 0
 9  izqmax ← 0
10  for i ∈ {m, ..., inf} do
11      sumactual ← sumactual + v[i]
12      if sumactual > izqmax then
13          izqmax ← sumactual
14      endif
15  endfor
16  sumactual ← 0
17  dermax ← 0
18  for i ∈ {m + 1, ..., sup} do
19      sumactual ← sumactual + v[i]
20      if sumactual > dermax then
21          dermax ← sumactual
22      endif
23  endfor
24  sumainterseccion ← izqmax + dermax
25  maxpartizq ← SumParticion(v, inf, m)
26  maxpartder ← SumParticion(v, m + 1, sup)
27  return Maximo(sumainterseccion, Maximo(maxpartizq, maxpartder))
28  end
```

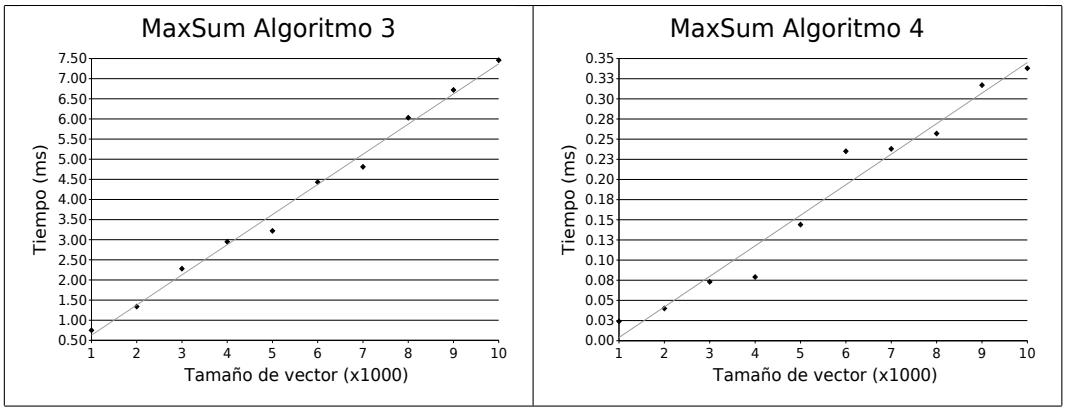

Figura 1.4: Gráfico de los datos observador para los algoritmos MaxSum3 y MaxSum4 (100 experimentos por punto).

Los datos experimentales para el algoritmo 3 se encuentran graficados el la parte izquierda de la figura 1.4. Ahora la relación entre t y la variable independiente n parece ser lineal.

La mejor estrategia posible para resolver el problema del subvector máximo consiste en recorrer el vector acumulando la suma de las entradas mientras se gane más de lo que se pierde con entradas negativas. En cuanto la suma sea negativa, lo que significa que se ha perdido todo lo ganado, se reinicia la suma acumulada. Si la suma acumulada hasta el momento excede la mayor conocida hasta el momento entonces se actualiza este valor.

```
MaxSum4(v, n)
 1  sumamax ← 0
 2  sumaactual ← 0
 3  for i ∈ {0, ..., n − 1} do
 4      sumactual ← sumactual + v[i]
 5      if sumactual < 0 then
 6          sumactual ← 0
 7      endif
 8      if sumactual > sumamax then
 9          sumamax ← sumactual
10      endif
11  endfor
12  return sumamax
13  end
```

Los datos experimentales del algoritmo 4 aparecen también en la figura 1.4, esta

vez a la derecha. La relación entre t y n parece ser también lineal como en el caso del algoritmo 3, aunque con una recta de pendiente mucho menor.

$\Diamond$

Aparentemente lo hecho en ejemplo anterior nos proporciona el método adecuado para analizar la complejidad de nuestros algoritmos. Pero subsisten algunos inconvenientes:

- Cómo es claro en las gráficas de la figura 1.4, hay puntos que no se ajustan tan bien como otros a la hipótesis de dependencia formulada y no podemos saber, *a priori* si esto se debe a que nuestra hipótesis es incorrecta, a alguna característica peculiar del entorno de ejecución (por ejemplo, algo vinculado con el sistema operativo de la máquina donde se hicieron los experimentos), o a la precisión de las medidas entregadas por el sistema.
- Nuestra hipótesis es formulada con base en unas cuantas mediciones tomadas sobre un pequeño subconjunto de el total de posibles entradas.
- Cada punto experimental es resultado de promediar un conjunto de mediciones observadas, esto se hace en un intento por minimizar los efectos de eventos impredecibles que pueden tener lugar en el sistema donde se ejecuta el programa. El simple hecho de mover el apuntador del ratón mientras se ejecuta el programa puede dar lugar a mediciones de tiempo extrañamente grandes. Se está a expensas del azar en buena medida.

Para eliminar estos inconvenientes deberíamos tener, cómo se hace en las ciencias experimentales, una teoría que sólo comprobamos haciendo experimentos. Las mediciones experimentales de desempeño quedarían ahora al final y no al principio de nuestro análisis. Necesitamos entonces aún más abstracción.

Modelo para análisis de algoritmos.

El primer paso es hacer un modelo. Un marco conceptual que nos permita definir qué es lo que se puede hacer, cuales son las operaciones válidas y qué queremos contabilizar, qué significa "gastar tiempo" o "gastar memoria" según el caso. Cuando lavamos ropa a mano nos comportamos cómo autómatas, vistos desde fuera de nosotros mismos somo unas máquinas bastante simples que sólo saben hacer unas cuantas cosas elementales: tomar una prenda, mojarla, enjabonarla, tallar, enjuagar, exprimir, sacudir y tender. Si hiciéramos un modelo teórico de algoritmos de lavado estas serían nuestras operaciones básicas. En general podemos suponer que ninguna operación ocupa más tiempo que tallar, esta es, por decirlo de algún modo, nuestra operación esencial al lavar a mano, nuestra medida de la complejidad deberá entonces estar referida al número de operaciones de tallado que se llevan a cabo y no a los enjuages o las sacudidas. Las cosas cambian si se cambia de modelo, en otro modelo podríamos considerar tener una lavadora automática, con lo que cambian nuestras operaciones básicas y con ello el criterio de complejidad.

Definición 1.2 Un algoritmo para un dispositivo de cómputo M es un conjunto de pasos que

cumple con las siguientes características:

- Es finito.
- Es ordenado.
- Cada paso es una operación elemental válida para M o un algoritmo para M.
- Cada paso es ejecutable por M en un tiempo finito.

Un modelo completo debe entonces contemplar lo que sabe hacer quien ejecuta el algoritmo para tener claro cuales son las operaciones válidas y debe decidir cuáles de estas operaciones son las relevantes para cuantificar la complejidad.

En el modelo de la geometría euclidiana sintética las operaciones elementales básicas son aquellas que pueden hacerse con una regla no graduada y un compás. En ese modelo hay algoritmo para trazar un hexágono, un pentágono o un triángulo equilátero, pero no para trisecar un ángulo o para trazar un cuadrado con la misma área que un circulo dado como entrada. En geometría analítica estas operaciones son triviales: un cambio de modelo representa, en general, un cambio en lo que puede hacerse y que tan complejo resulta hacerlo.

EJEMPLO 1.4

En los algoritmos usados en el ejemplo estamos suponiendo implícitamente que nuestro dispositivo de cómputo que los ejecuta sabe sumar, comparar dos números y asignar el valor de una variable. Debemos decidir ahora cuál de estas operaciones es la esencial. Sin duda la operación más importante en este caso es la suma. Es la que nos permite conocer la suma de las entradas en el subvector. Ahora podemos proceder a analizar los algoritmos cuantificando el número de sumas que hacen, suponiendo que es esa operación la que consume la mayor parte del tiempo del algoritmo en vez de contabilizar el tiempo de ejecución experimentalmete.

En el algoritmo 1 se efectúa una suma en le línea 6. En número de sumas está entonces determinado por el número de repeticiones del ciclo de las líneas 5 –7, que se lleva a cabo desde $k = i$ hasta $k = j$. Este ciclo a su vez está dentro del ciclo de las líneas 3 –11, que se repite desde $j = i$ hasta $j = n-1$. Por ultimo, este ciclo está dentro del ciclo más exterior, de las líneas 2–12, que se repite desde $i = 0$ hasta $i = n-1$. Así

que el número total de sumas hechas por el algoritmo 1 es:

$$\begin{aligned}
S(MaxSum1) &= \sum_{i=0}^{n-1} \sum_{j=i}^{n-1} \sum_{k=i}^{j} 1 \\
&= \sum_{i=0}^{n-1} \sum_{j=i}^{n-1} j - i + 1 \\
&= \sum_{i=0}^{n-1} \sum_{m=1}^{n-i} m \\
&= \sum_{i=0}^{n-1} \left[\frac{(n-i)(n-i+1)}{2} \right] \\
&= \sum_{i=1}^{n} \left[\frac{i(i+1)}{2} \right] \\
&= \frac{1}{2} \sum_{i=1}^{n} i^2 + i \\
&= \frac{1}{2} \left[\sum_{i=1}^{n} i^2 + \sum_{i=1}^{n} i \right] \\
&= \frac{1}{6}n^3 + \frac{1}{2}n^2 + \frac{1}{3}n
\end{aligned} \tag{1.1}$$

Ahora queda claro el por qué la gráfica de la figura 1.2 tiene la forma que observamos y por qué se linealiza cuando obtenemos la raíz cúbica del tiempo. Resulta que el número de sumas es un polinomio de grado tres, lo que predice perfectamente el tiempo que tarda el algoritmo para un tamaño de entrada dado.

$\Diamond$

Parece que al fin tenemos lo que necesitábamos, una abstracción que logra captar lo esencial despreciando detalles superfluos como aquellos vinculados con el entorno de ejecución particular de un algoritmo ya programado. Podemos ahora analizar la complejidad de un algoritmo dado con sólo tener su pseudocódigo, sin tener que implementarlo y ejecutarlo.

Pero aún falta un paso de abstracción más. Este es de carácter opcional podríamos decir, y consiste en despreciar la precisión de complejidad obtenida para un algoritmo dado. El polinomio de la expresión 1.1 dice *exactamente* cuantas sumas lleva a cabo el algoritmo para un vector de tamaño n, en muchas ocasiones no interesa tanto saber exactamente la complejidad de un algoritmo sino saber que no es peor que otro, o lo que es lo mismo, saber que la función de complejidad del algoritmo, no crece más rápido que otra usada como patrón de comparación, tal como utilizamos el patrón del metro para medir longitudes.

Esto es particularmente conveniente cuando la función de complejidad de un algoritmo es complicada, tiene muchos altibajos (lo que se conoce como *no monótona*) y es difícil intuir que tan rápido crece en todo momento. Lo que suele hacerse entonces es acotarla por arriba con alguna función conocida, o mejor dicho, con el múltiplo de alguna función conocida y decir entonces que la función de complejidad no crece más rápido que la usada como cota y que hace las veces de nuestro patrón de medida. Esto también resulta conveniente en el caso de que la función de complejidad exhibe un comportamiento no muy bueno para valores de complejidad de entrada (n) menores a cierto umbral (N). Podría ocurrir que un algoritmo tenga una complejidad relativamente grande para valores de n menores que cierta N y después mejore notablemente porque el crecimiento de su función de complejidad no es tan acelerado, en este caso convendría fijar la cota ya que ha pasado la región de mal comportamiento. A fin de cuentas nos interesa evaluar el algoritmo *para todo* posible valor de n y hay más valores posteriores a N que anteriores a ella, establecer la cota antes podría ser injusto, podríamos descartar algoritmos creyéndolos malos cuando en realidad son malos sólo en un rango inicial limitado o a la inversa, podríamos considerar algoritmos como buenos ignorando que, a la larga, resultan muy malos porque su función de complejidad crece rápidamente.

Cota superior.

A fin de cuentas, establecer una cota superior a partir de cierto valor umbral es comparar la rapidez con la que la función de complejidad crece y no los valores intrínsecos de dicha función. Es, en general, mucho más útil tener un algoritmo que se tarda 3 segundos con una complejidad de entrada de 10 y 8 segundos con una de 100 que uno que tarda 8 milisegundos con la 10 y 3 años con la de 100.

En realidad lo que pretendemos hacer es clasificar las funciones de crecimiento del tiempo en la ejecución de los algoritmos y además definir un orden en el conjunto de clases. Queremos tener algo como $f(n) \leq g(n)$ donde f y g son funciones que determinan el tiempo que tarda el algoritmo y n es el tamaño de la entrada del algoritmo, pero debemos tener en cuenta los resultados de nuestro análisis previo: despreciar constantes que se suman o se multiplican y considerar todos los posibles tamaños de entradas. De hecho no está bien utilizar el símbolo "$\leq$" como lo hemos hecho porque uno está tentado a decir que f debe ser menor o igual a g en todo su dominio y eso no es lo que queremos porque no estamos tomando en cuenta que f puede estar por arriba de g durante un rato antes de ser rebasada. La notación estándar para expresar lo que queremos es $f = O(g)$.

La O-sota.

Multiplicar por una constante positiva de tamaño arbitrario una función acelera el crecimiento de está mucho más que sumarle una constante, así que de hecho podemos quedarnos sólo con la multiplicación. Ahora podemos concluir que $f = O(g)$ si, a partir de cierta complejidad de entrada, g misma o g multiplicada por alguna constante, queda por arriba de f y así se queda *ad infinitum*. La definición siguiente lo pone en términos más precisos.

Definición 1.3 Diremos que f es *del orden de* g, o en notación $f = O(g)$ si y solo si existen

un número natural N y una constante real positiva c tales que para cualquier natural $n > N$:

$$f(n) \leq c\, g(n)$$

Este es un concepto fundamental en la ciencia de la computación, así que conviene parafrasearlo con el fin de hacerlo lo más intuitivo posible. Recordar la definición debe ser fácil si se recuerda la idea esencial.

$f = O(g)$ significa que:

- f no crece más rápido que g.
- f crece más lento, o a lo más tan rápido, como g.
- f crece a lo más tan rápido como g.
- g crece al menos tan rápido como f.
- g no crece más lento que f.

Ahora bien, como f y g son funciones que miden "la tardanza", que para hablar propiamente llamaremos la *complejidad de los algoritmos*, entonces podemos traducir las aseveraciones anteriores como sigue:

- El algoritmo con complejidad f no es menos eficiente que el de complejidad g.
- El algoritmo con complejidad f es más eficiente o al menos tan eficiente como el de complejidad g.
- El algoritmo con complejidad f no es más ineficiente que el de complejidad g.
- El algoritmo con complejidad g es al menos tan ineficiente como el de complejidad f.
- El algoritmo con complejidad g no es más eficiente que el de complejidad f.

Esta es la medida que utilizaremos para calcular las complejidades en tiempo de ejecución y en uso de memoria de nuestros algoritmos. Además posee ciertas propiedades que nos serán muy útiles en dichos cálculos. La primera que demostraremos se refiere a la complejidad de un algoritmo que se obtiene aplicando secuencialmente dos sub-algoritmos.

Teorema 1.2 Sean $t_1(n)$ y $t_2(n)$ las respectivas complejidades de dos algoritmos A_1 y A_2 tales que $t_1(n) = O(f(n))$ y $t_2(n) = O(g(n))$. Si se construye un algoritmo A que consiste de A_1 seguido de A_2 y cuya complejidad es $t(n) = t_1(n) + t_2(n)$ entonces $t(n) = O(\max\{f(n), g(n)\})$.

Demostración:

Sabemos que $t_1(n) = O(f(n))$, así que existen c_1 y N_1 tales que, para toda $n > N_1$, $t_1(n) \leq c_1\, f(n)$.

Sabemos también que $t_2(n) = O(g(n))$, así que existen c_2 y N_2 tales que, para toda $n > N_2$, $t_2(n) \leq c_2\, g(n)$.

Sea $N = \max\{N_1, N_2\}$, como N es mayor o igual que N_1 y N_2 simultáneamente, entonces para $n > N$ se cumplen también simultáneamente:

$$\begin{aligned} t_1(n) &\leq c_1\, f(n) \\ t_2(n) &\leq c_2\, g(n) \end{aligned}$$

sumándolas obtenemos, que para toda $n > N = \max\{N_1, N_2\}$:

$$\begin{aligned} t_1(n) + t_2(n) &\leq c_1\, f(n) + c_2\, g(n) \\ &\leq c_1\, \max\{f(n), g(n)\} + c_2\, \max\{f(n), g(n)\} \\ &= (c_1 + c_2)\, \max\{f(n), g(n)\} \end{aligned}$$

Así que hemos encontrado N y $c = c_1 + c_2$ tales que, para toda $n > N$:

$$t(n) = t_1(n) + t_2(n) \leq c\, \max\{f(n), g(n)\}$$

de acuerdo con la definición, esto significa que la complejidad de A es:

$$t(n) = O(\max\{f(n), g(n)\})$$

□

Otra propiedad interesante tiene que ver con la complejidad de un algoritmo que se construye aplicando un sub-algoritmo fijo A_2 cada vez que otro sub-algoritmo A_1 ejecuta una de sus *operaciones esenciales*, es decir aquellas en las que resulta relevante el consumo del recurso que se está contabilizando. Más adelante aclararemos un poco más esta noción.

Teorema 1.3 Sean $t_1(n)$ y $t_2(n)$ las respectivas complejidades de dos algoritmos A_1 y A_2 tales que $t_1(n) = O(f(n))$ y $t_2(n) = O(g(n))$. Si se construye un algoritmo A que consiste en aplicar A_2 por cada operación esencial de A_1 y cuya complejidad es $t(n) = t_1(n)t_2(n)$ entonces $t(n) = O(f(n)\, g(n))$.

Demostración:

Sabemos que $t_1(n) = O(f(n))$, así que existen c_1 y N_1 tales que, para toda $n > N_1$, $t_1(n) \leq c_1\, f(n)$.

Sabemos también que $t_2(n) = O(g(n))$, así que existen c_2 y N_2 tales que, para toda $n > N_2$, $t_2(n) \leq c_2\, g(n)$.

Sea $N = \max\{N_1, N_2\}$, como N es mayor o igual que N_1 y N_2 simultáneamente, entonces para $n > N$ se cumplen también simultáneamente:

$$\begin{aligned} t_1(n) &\leq c_1\, f(n) \\ t_2(n) &\leq c_2\, g(n) \end{aligned}$$

multiplicándolas obtenemos, que para toda $n > N = \max\{N_1, N_2\}$:

$$t_1(n)\, t_2(n) \leq c_1\, c_2\, f(n) g(n)$$

Así que hemos encontrado N y $c = c_1\, c_2$ tales que, para toda $n > N$:

$$t(n) = t_1(n)\, t_2(n) \leq c\, f(n), g(n)$$

de acuerdo con la definición, esto significa que la complejidad de A es:

$$t(n) = O(f(n)\, g(n))$$

□

A diferencia del teorema 1.2, el 1.3 no es tán fácil de comprender. Para aclararlo echaremos mano, a la manera de Donald Knuth, de algo de nuestro repertorio lírico. A continuación hemos reproducido la letra de una canción navideña infantil.

El primer día de navidad
mi amado me obsequió
un gorrioncillo volador.
El segundo día de navidad
mi amado me obsequió
dos tortolitas,
y un gorrioncillo volador.
El tercer día de navidad
mi amado me obsequió
tres gallinitas,
dos tortolitas,
y un gorrioncillo volador.
El cuarto día de navidad
mi amado me obsequió
cuatro aves canoras,
tres gallinitas,
dos tortolitas,
y un gorrioncillo volador.
El quinto día de navidad
mi amado me obsequió
cinco anillos de oro,
cuatro aves canoras,
tres gallinitas,
dos tortolitas,
y un gorrioncillo volador.
El sexto día de navidad
mi amado me obsequió
seis gansos dormidos,
cinco anillos de oro,
cuatro aves canoras,
tres gallinitas,
dos tortolitas,
y un gorrioncillo volador.
El séptimo día de navidad
mi amado me obsequió
siete lindos patos,
seis gansos dormidos,
cinco anillos de oro,
cuatro aves canoras,
tres gallinitas,
dos tortolitas,
y un gorrioncillo volador.
El octavo día de navidad
mi amado me obsequió
ocho hermosas vacas,
siete lindos patos,
seis gansos dormidos,
cinco anillos de oro,
cuatro aves canoras,
tres gallinitas,
dos tortolitas,
y un gorrioncillo volador.
El noveno día de navidad
mi amado me obsequió
nueve bailarinas,
ocho hermosas vacas,
siete lindos patos,
seis gansos dormidos,
cinco anillos de oro,
cuatro aves canoras,
tres gallinitas,
dos tortolitas,
y un gorrioncillo volador.
El décimo día de navidad
mi amado me obsequió
diez caballitos,
nueve bailarinas,
ocho hermosas vacas,
siete lindos patos,
seis gansos dormidos,
cinco anillos de oro,
cuatro aves canoras,
tres gallinitas,
dos tortolitas,
y un gorrioncillo volador.
El onceavo día de navidad
mi amado me obsequió
once flautistas,
diez caballitos,
nueve bailarinas,
ocho hermosas vacas,
siete lindos patos,
seis gansos dormidos,
cinco anillos de oro,
cuatro aves canoras,
tres gallinitas,
dos tortolitas,
y un gorrioncillo volador.
El doceavo día de navidad
mi amado me obsequió
doce tamborileros,
once flautistas,
diez caballitos,
nueve bailarinas,
ocho hermosas vacas,
siete lindos patos,
seis gansos dormidos,
cinco anillos de oro,
cuatro aves canoras,
tres gallinitas,
dos tortolitas,
y un gorrioncillo volador.

Pensando como computólogos que somos de inmediato se nos ocurre abreviar la canción. Sin llamamos $verso[1]$ al enunciado *un gorrioncillo volador*, $verso[2]$ a *dos tortolitas* y así sucesivamente, entonces podemos expresar la canción como sigue:

```
LosDoceDiasDeNavidad()
1  for j ∈ {1, ..., 12} do
2      Cantar( El  i -ésimo día de navidad mi amado me obsequió )
3      for i ∈ {j, ... 1} do
4          Cantar(verso[i])
5      endfor
6  endfor
7  end
```

La operación que podríamos llamar esencial del primer ciclo (líneas 2 a la 7) es la de la línea 3, *Cantar*. Pero cada vez que se lleva a cabo una repetición de ese ciclo se

llevan a cabo también las repeticiones del ciclo en las líneas 4 a 6. Así que hay dos ciclos anidados, cada repetición del externo conlleva varias, al menos una, del interno. ¿Cuantos versos se cantan en total? La primera vez que se ejecuta el ciclo externo sólo se canta un verso, la segunda vez se cantan dos, la tercera tres, y así sucesivamente, así que el número total de versos cantados hasta la n-ésima estrofa es:

$$1 + 2 + 3 + \cdots + n = \sum_{i=1}^{n} i = \frac{n(n+1)}{2} = O(n^2)$$

Esto resulta de tener anidados dos ciclos, cada uno repitiéndose $O(n)$ veces. Ahora es más claro lo que dice el teorema 1.3.

EJEMPLO 1.5

Ya determinamos la complejidad del algoritmo 1 para el problema de el subvector máximo. Llegamos al polinomio de tercer grado:

$$S(MaxSum1) = \frac{1}{6}n^3 + \frac{1}{2}n^2 + \frac{1}{3}n$$

Claramente, para toda $n > 1$, si $c = 1$, se tiene que:

$$S(MaxSum1) \leq c\, n^3$$

así que podemos decir que: $S(MaxSum1) = O(n^3)$.

Para el algoritmo 2 de MaxSum, se hace una suma en la línea 5, que está contenida en el ciclo de las líneas 4 –9, que se repite desde $j = i$ hasta $j = n-1$. Este está contenido a su vez en el ciclo de las líneas 2 –10, que se repite desde $i = 0$ hasta $i = n - 1$, así que el número total de sumas es:

$$S(MaxSum2) = \sum_{i=0}^{n-1}\sum_{j=i}^{n-1} 1 = \sum_{i=0}^{n-1} n - i = \sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

En este caso, para toda $n > 0$, si $c = 1$, se tiene que:

$$S(MaxSum2) \leq c\, n^2$$

así que: $S(MaxSum2) = O(n^2)$

Para el algoritmo 3 la situación es un poco diferente. Supongamos que si el subalgoritmo SumParticion recibe valores para los índices inferior y superior tales que $inf = sup$, entonces se tarda un tiempo

$$T(1) = 1 \tag{1.2}$$

Para procesar un vector de tamaño n el tiempo será $T(n)$ que, como podemos ver en el algoritmo, será:

$$T(n) = 2\,T\,(n/2) + c\,n \tag{1.3}$$

dado que el algoritmo se llama a sí mismo dos veces y además hace el cálculo de el subvector máximo en la intersección, lo que suponemos se lleva a cabo en tiempo proporcional a $c\,n$.

A una expresión del tipo de 1.3 se le denomina *recurrencia*, la evaluación de una función cuando su argumento adquiere un valor determinado, se especifica en términos de su evaluación cuando su argumento tiene un valor menor. A lo largo del texto encontraremos recurrencias frecuentemente.

Usando 1.3 y 1.2 podemos escribir:

$$\begin{aligned}
&T(1) = 1 \\
&T(2) = 2 + 2c \\
&\quad\vdots \qquad\qquad (1.4)\\
&T(4) = 4 + 4 \cdot 2c \\
&\quad\vdots \qquad\qquad (1.5)\\
&T(8) = 8 + 8 \cdot 3c \\
&\quad\vdots \qquad\qquad (1.6)\\
&T(2^k) = 2^k(1 + kc) \qquad\qquad (1.7)
\end{aligned}$$

si $n = 2^k$ entonces $k = \log_2 n$, así que:

$$T(n) = n + c\,n\,\log_2 n$$

Usando el teorema 1.2 tenemos que: $T(n) = O(n \log n)$

Resulta curioso que la complejidad del algoritmo 3 sea $O(n \log n)$ y en los resultados experimentales mostrados en la figura 1.4 aparezca como lineal. Lo que ocurre es que tuvimos una impresión equivocada, ya habíamos dicho que uno de los problemas de la aproximación experimental a la complejidad es que, cómo sólo es posible medir en un rango bastante pequeño de posibles valores de complejidad de entrada, podemos tener una idea errónea de la complejidad en todo el dominio. Observando la figura 1.5 nos damos cuenta del motivo de nuestra apreciación incorrecta. La complejidad del algoritmo 4 se parece mucho, localmente, a una relación lineal, sólo una vista mucho más amplia del dominio podría darnos pistas correctas.

En el algoritmo 4 para MaxSum se hace una suma en la línea 4, que se repite tanto como dice el ciclo de las líneas 3 –1, es decir desde $i = 0$ hasta $i = n - 1$, es decir n veces y es el único ciclo en el que se encuentra la línea 4, así que la complejidad es

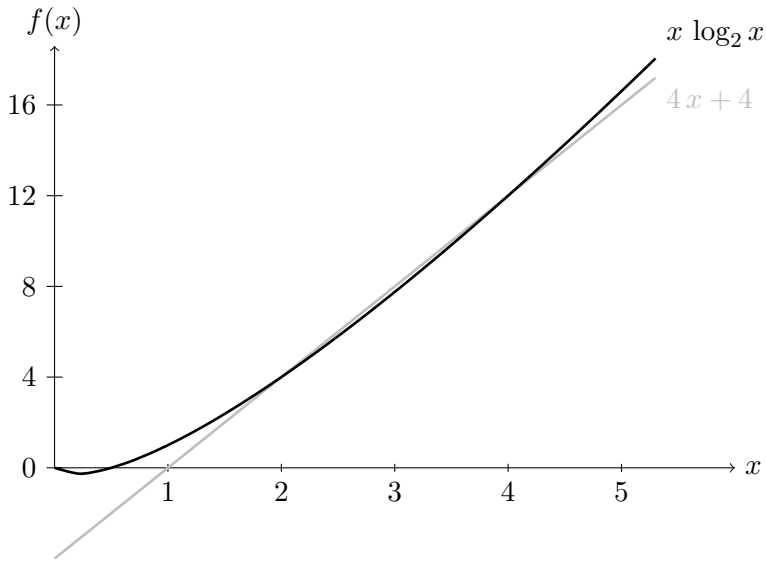

Figura 1.5: Gráfico que compara los valores de la función de complejidad (versión continua) del algoritmo 3 de MaxSum con una relación lineal, de hecho $f(x) = 4\,x - 4$. La complejidad (en negro), vista en un intervalo suficientemente pequeño del dominio, es muy similar a la recta (en gris).

simplemente:

$$S(MaxSum4) = \sum_{i=0}^{n-1} 1 = n$$

de donde: $S(MaxSum4) = O(n)$

¡Excelente! Pasamos de tener un algoritmo de complejidad $O(n^3)$ a uno de complejidad lineal para el mismo problema.

$\Diamond$

EJEMPLO 1.6

Cálculo de potencias

Supongamos que se nos dan un par de números naturales cualesquiera, digamos b y n y que se nos pide que elaboremos un algoritmo que obtenga la n-ésima potencia de b, es decir $p = b^n$. Por supuesto el algoritmo que se nos ocurre en primera instancia es el mostrado en la figura 1.1. Procederemos a analizar su complejidad.

Lo primero que hay que decidir es cuál recurso vamos a cuantificar, como sabemos hay dos opciones tiempo y memoria. En este caso elegiremos el primero, analizar el consumo de memoria no es interesante en este caso. Ahora bien, ¿cuáles de las opera-

Algoritmo 1.1 Algoritmo para obtener b^n.

```
POTENCIA(b, n)
1  p ← 1
2  if n > 0 then
3      for j ∈ {1, ..., n} do
4          p ← p * b
5      endfor
6  endif
7  return p
8  end
```

ciones hechas en cada uno de los pasos del algoritmo es la que consideramos esencial en el sentido mencionado anteriormente?, ¿en qué ocupamos más tiempo?, evidentemente en las multiplicaciones que se llevan a cabo cada vez que se ejecuta la línea 6. Así que vamos a medir nuestra complejidad en términos de número de multiplicaciones hechas. Dadas una n y una b como entrada del algoritmo, ¿cuantas multiplicaciones se hacen? de ver el pseudo-código mostrado nos percatamos de que esta pregunta se traduce en ¿cuantas veces se repite el ciclo de las líneas 3 a 5? la respuesta evidente es n, el recorrido de la variable j. Así que nuestro algoritmo hace $O(n)$ multiplicaciones, donde n es el valor del exponente.

Pero hay otra manera de elevar b a la n. Pongamos un ejemplo. Digamos que queremos obtener 3^{14}, el 14 en binario se escribe $1110_2 = 1\times 2^3 + 1\times 2^2 + 1\times 2^1 + 0\times 2^0$. Así que podríamos escribir:

$$\begin{aligned} 3^{1110_2} &= 3^{1\times 2^3 + 1\times 2^2 + 1\times 2^1 + 0\times 2^0} \\ &= 3^8 \times 3^4 \times 3^2 \\ &= ((3^2)^2)^2 \times (3^2)^2 \times 3^2 \end{aligned}$$

Si leemos esto de derecha a izquierda vemos cómo simplificar el proceso: el i-ésimo término, contando desde la derecha, es el anterior elevado al cuadrado y este término se toma en cuenta si el bit asociado con él es 1. El algoritmo aparece en la figura 1.2.

La complejidad del algoritmo POTENCIASQM es menor que la de POTENCIA. El número de multiplicaciones que hay que hacer en POTENCIASQM es, a lo más el número de bits con valor 1 en la expresión binaria de n, esto no puede ser mayor a $\lceil \log_2(n) \rceil$ lo que, por supuesto, es menor que n. Así que la complejidad de POTENCIASQM es $O(\log(n))$. Alguien podría decir que no estamos considerando las divisiones que hay que hacer para obtener la expresión binaria de n en la línea 2 del algoritmo, pero eso significaría también $O(\log(n))$ divisiones, una por cada bit, así que por el teorema 1.2 la

Algoritmo 1.2 Otro algoritmo para obtener b^n. La función Binario(n) regresa una cadena con la expresión binaria del argumento n. Con longitud($binexp$) se denota la longitud, en bits, de la cadena binaria $binexp$ y con $binexp[i]$ se denota el bit asociado con el factor 2^i de la expresión binaria de n ($i \in \{0, \ldots, \text{longitud}(binexp) - 1\}$).

```
POTENCIASQM(b, n)
 1  binexp ← Binario(n)
 2  p ← 1
 3  s ← b
 4  if n > 0 then
 5      if binexp[0] = 1 then
 6          p ← b
 7      endif
 8      for j ∈ {1, ..., longitud(binexp) − 1} do
 9          s ← s^2
10          if binexp[j] = 1 then
11              p ← p * s
12          endif
13      endfor
14  endif
15  return p
16  end
```

```java
/**
 * Obtiene la potencia deseada.
 * @param base es el número usado como base.
 * @param expo es el usado como exponente.
 * @return <code>base</code> elevado a la potencia
 * indicada por <code>expo</code>.
 */
public static int potencia(int base, int expo) {
    // se usará el método "Square and Multiply", descrito
    // por Neal Koblitz
    int i;
    int p = 1;
    int s = base;
    int bitcount = Integer.toBinaryString(expo).length();
    if (expo > 0) {
        if ((expo % 2) == 1) {
            p = base;
        }
        for (i = 1; i < bitcount; i++) {
            s *= s;
            if (((expo >>> i) % 2) == 1) {
                p *= s;
            }
        }
    }
    return p;
}
```

Listado 1.1: Cálculo de una potencia entera.

complejidad sigue siendo $O(\log(n))$. PotenciaSQM es el algoritmo que suele utilizarse para obtener potencias de números muy grandes en campos finitos, lo que es frecuente en criptografía.

La moraleja no es ninguna novedad, ya sabíamos que podía haber diferentes algoritmos con diferentes complejidades para resolver el mismo problema. Pero espero que este ejercicio haya resultado divertido e ilustrativo en lo que refiere al cálculo de complejidad y de lo que significa lo que hemos denominado *operación esencial*.

◊

EJEMPLO 1.7

Buscando subcadenas

Existe toda un área de la computación dedicada a lo que se denomina *empate de*

cadenas o *string matching*, de hecho en la actualidad es una de las áreas con mayor auge por sus aplicaciones a la biología molecular. Uno de los problemas fundacionales del área es el de buscar un empate perfecto entre una cadena dada y una subcadena de otra, es decir, buscar alguna o todas las apariciones de una cadena P en otra más larga S.

Pongamos un ejemplo. Supongamos que tenemos un segmento de ADN que, como nos enseñaron en nuestros cursos elementales de biología, está hecho de cuatro posibles bases, a saber: adenina (A), timina (T), citosina (C) y guanina (G). Nuestro segmento S es el siguiente:

$$ATGCATTGCG$$

está constituido por una secuencia de 10 bases. Ahora supongamos que queremos buscar en él todas las apariciones de la tripleta $p = TGC$ que, como sabemos, codifica algún aminoácido que nos interesa. Seguramente ya el lector se percató de que P aparece dos veces en S, en las posiciones (empezando con 0 en el extremo izquierdo de S) 1 y 6. Si S o P fueran mucho más largos, como en sucede en la realidad, encontrar los empates ya no es tan sencillo.

Podemos pensar en un algoritmo trivial para encontrar las apariciones de P en S. Ponemos en correspondencia a P con el inicio de S y comparamos, a lo largo de todo P si coinciden o no, mientras coincidan continuamos comparando. Una vez que hayamos determinado si P se empató con el inicio de S, desplazamos P una posición a la derecha y procedemos a comparar nuevamente. Continuando así podemos garantizar que encontramos todas las apariciones de P en S.

La complejidad del algoritmo que acabamos de describir está determinada por el número de comparaciones que hay que hacer, nuestra operación esencial. Cada vez que ponemos a P en correspondencia con S tenemos que hacer, a lo más tantas comparaciones como la longitud de P, a la que denominaremos m. Ahora bien, ¿cuantas veces debemos desplazar P sobre S? si denotamos con n la longitud de S, entonces podemos decir que debemos hacer $n - m$ empates diferentes. En síntesis debemos hacer a lo más $O(m\,(n - m))$ comparaciones, como en general n es mucho mayor que m, el término dominante en esta expresión será $n\,m$, así que podemos decir que nuestro algoritmo trivial hace $O(n\,m)$ comparaciones para encontrar todas las apariciones de P en S.

Pero resulta que hay métodos mucho mejores para buscar empates. Algunos de los mejores están basados en la construcción de una estructura de datos llamada *árbol de sufijos*. En la figura 1.6 se muestra el correspondiente a la cadena usada como ejemplo anteriormente.

Un sufijo de una cadena S es una subcadena final de ella, es decir, que comienza a partir de cierta posición de S y a de allí en adelante es idéntica a S, un sufijo de nuestra cadena de ejemplo es:$ATTGCG$, por ejemplo. La cualidad mas importante del árbol es que todos los sufijos están representados en él, basta con descender por sus ramas hasta una hoja y las cadenas asociadas a las aristas que se recorren forman un sufijo de S al concatenarse en su orden de aparición.

Si uno posee el árbol de sufijos de una cadena S entonces buscar una subcadena P es

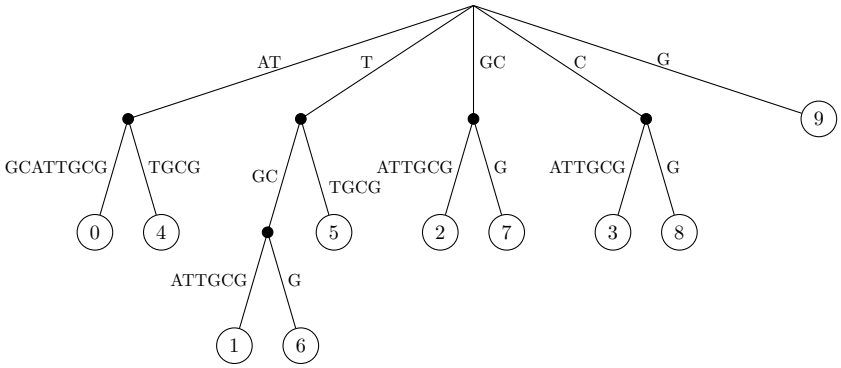

Figura 1.6: Árbol de sufijos para la cadena $ATGCATTGCG$.

simple: sólo hay que posicionarse en la raíz del árbol y descender por la arista etiquetada con el caracter inicial o una subcadena inicial de P y continuar descendiendo por aristas que contienen trozos sucesivos de P. Cuando P se termine estaremos en algún nodo del árbol que es raíz de un subárbol, las hojas que penden de él tienen asociadas las posiciones donde P aparece en S. La complejidad de este descenso es $O(m)$, porque a lo más hay que hacer ese número de comparaciones para completar P.

Bueno, pero nuestro algoritmo trivial no necesitaba ninguna estructura de datos compleja como el árbol de sufijos y este sí, así que también debemos considerar el tiempo que se tarda en construir el árbol como parte de la complejidad. Hay diversos algoritmos para construirlo, los triviales toman $O(n^3)$ comparaciones, pero en 1973 Weiner [3, 2] descubrió el primer algoritmo para construir el árbol de sufijos en tiempo lineal, es decir en $O(n)$ comparaciones, más tarde McCreight y Ukkonen [6, 7] descubrieron algoritmos diferentes, pero igualmente eficientes. Como el proceso completo de solución requiere construir el árbol y luego descender por él, la complejidad total, por el teorema 1.2 resulta ser $O(n)$.

La moraleja en este caso es más útil. Una estructura de datos adecuada puede simplificar significativamente la complejidad de los algoritmos para resolver un problema.

$\Diamond$

1.4 Pruebas de corrección.

Por supuesto no sólo estamos interesados en la eficiencia de nuestros algoritmos. De nada sirve un algoritmo muy eficiente que no hace lo que debería. Así que también debemos analizar la corrección, al menos de manera elemental. Para ello echaremos mano de los axiomas de los TDAs involucrados y de otro concepto denominado *invariante*.

A	**B**	**A** $\oplus$ **B**
FALSO	FALSO	FALSO
FALSO	VERDADERO	VERDADERO
VERDADERO	FALSO	VERDADERO
VERDADERO	VERDADERO	FALSO

Tabla 1.1: Tabla de verdad de la disyunción exclusiva.

Ya habíamos mencionado el hecho de que podemos encadenar los resultados y postcondiciones de una operación de un TDA, con los argumentos y precondiciones de otra, para formar así una cadena de deducciones lógicas que nos lleven a certificar el correcto funcionamiento de un algoritmo basado en esas operaciones. Los resultados y postcondiciones de una operación se convierten en los argumentos y precondiciones de otra que a su vez genera los de una tercera y así sucesivamente hasta la terminación del algoritmo. Veamos un ejemplo.

EJEMPLO 1.8

Supóngase que se desea implementar la operación de disyunción exclusiva (OR exclusivo, o bien XOR en la jerga usual de computación) usando alguna implementación correcta del TDA Booleano especificado anteriormente (TDA 1.1). En la tabla 1.1 se muestra la tabla de verdad de la función, denotada con $\oplus$. Es claro que la tabla especifica completamente la disyunción exclusiva: para cada posible pareja de valores de los dos argumentos se dice cuanto vale la función. El rasgo distintivo de la disyunción exclusiva es que es falsa sólo si sus dos argumentos son iguales y verdadera sólo si son diferentes.

El algoritmo para calcular la función es, simplemente:

```
ORExclusivo(A, B)
1  return  or(and(not(A), B), and(A, not(B)))
2  end
```

Analicemos, supongamos alternativamente que:

- A y B son iguales. En este caso el argumento del primer *not* es complementario del argumento del segundo: si el primero regresa VERDADERO el segundo regresa FALSO y viceversa. Así que los argumentos de los *and* corresponden a los del axioma B4a (usando B2a), la postcondición es que el resultado es FALSO, así que

los argumentos del *or* son ambos FALSO, ante esto el *or*, cómo lo especifica el axioma B1b, regresa FALSO, que es el resultado final.

- A y B son complementarios. Podemos suponer que A es VERDADERO y por tanto B FALSO. Ante estos argumentos ambos *not* regresan FALSO. Para el segundo *and* entonces, los argumentos corresponden a los del axioma B1a, así que regresa el valor de verdad de $not(B)$, es decir VERDADERO. Por el teorema de idempotencia el primer *and* regresa FALSO, así que el *or* recibe argumentos que empatan con los del axioma B1b, por tanto el resultado total es VERDADERO.

$\Diamond$

Claro está que el ejemplo mostrado aquí es bastante simple, en general es bastante más laborioso demostrar formalmente la corrección de un algoritmo basado en las operaciones de un TDA. Para terminar de complicar las cosas estamos suponiendo que poseemos una implementación correcta del TDA. En general deberíamos demostrar primero que las implementaciones de nuestros TDAs son correctas para luego demostrar que nuestro programa, basado en ellas, es también correcto encadenando resultados con argumentos y precondiciones con postcondiciones. Suena complicado, lo es.

Otro recurso que contribuye a garantizar el correcto funcionamiento de nuestros algoritmos es el encontrar condiciones que permanecen siendo válidas a lo largo del algoritmo. A esto se le denomina *invariante*. Nuevamente procederemos con un ejemplo.

EJEMPLO 1.9

En el primer algoritmo para el problema del subvector de suma máxima, tenemos tres ciclos anidados. En el ciclo más interno (líneas 5–7), se itera sobre un índice llamado k, recorriendo los valores desde i hasta j del vector de entrada. Es decir k comienza valiendo i y termina valiendo j. En la línea 4 se establece el valor de la variable *sumactual* en cero, así que podemos decir que *sumactual* contiene la suma de las primeras *cero* entradas del subvector comprendido entre los índices i y j. En cada iteración del ciclo se añade el valor de la entrada k al valor de *sumactual*, así que podemos asegurar que, siempre al terminar el ciclo, en la línea 7, la variable *sumactual* contiene la suma de las primeras k entradas entre la i y la j. Este es el invariante del ciclo.

Ahora bien, al terminar la última iteración del ciclo, $k = j$, así que, por el invariante del ciclo concluimos que la variable *sumactual* contiene, ciertamente, la suma de todas las entradas comprendidas entre i y j inclusive.

Luego el *if* de la línea 8 garantiza que si la suma de las entradas entre i y j excede el máximo conocido entonces este se establece como el valor recientemente encontrado.

Razonando de igual manera podemos decir que el invariante del ciclo de las líneas 3–11 es que en la variable *sumamax* está contenida la suma máxima de entradas conocida

a partir del índice i y hasta el índice j (variable sobre la que se itera), es decir *sumamax* contiene la suma máxima de entradas encontrada en todos los subvectores que comienzan en el índice i y que terminan, a lo más, en el índice j.

En la última iteración del ciclo, estando en la línea 11, sabemos entonces dado que $j = n - 1$, que en la variable *sumamax* tenemos la suma máxima encontrada en todos los subvectores que comienzan en i.

Por último, el invariante del ciclo más externo (líneas 2–12), es que en la variable *sumamax* se almacena la suma máxima de entradas encontradas en los subvectores que comienzan en una posición menor o igual a i. Cuando terminamos el algoritmo, en la línea 13, podemos asegurar entonces, que como $i = n-1$, en *sumamax* tenemos la suma máxima de entradas en todos los subvectores, que es justamente lo que buscábamos.

$\Diamond$

La definición correcta de invariantes no siempre es fácil, sobre todo si no es un ciclo contado (i.e. repetido un número fijo de veces) sino uno que verifica una condición lógica compleja basada en múltiples variables cuyo valor, por supuesto, se altera en las iteraciones del ciclo. Pero además de ser un recurso poderoso para probar la corrección de un algoritmo, el pensar en los invariantes durante el diseño del algoritmo contribuye a generar código elegante y limpio, en el que no se verifican condiciones superfluas que nunca ocurren o bien, que ocurren si y sólo si alguna otra ocurre, lo que las hace redundantes.

1.5 Resumen.

TDA

Un TDA o *Tipo de Dato Abstracto* es un sistema formal constituido por:

- Un conjunto dominio de posibles valores.
- Un conjunto de operaciones definidas sobre los elementos del dominio.

Las operaciones del TDA se rigen por una serie de *axiomas* que determinan qué se puede hacer con los objetos del tipo especificado. Las condiciones que deben satisfacer los datos para efectuar una operación determinada constituyen las *precondiciones* de dicha operación, las condiciones que los datos satisfacen luego de efectuarse la operación constituyen las *postcondiciones* de la misma.

Estructura de datos

Una *estructura de datos* está determinada por:

- La manera de organizar los datos en la memoria de la computadora y de establecer vínculos entre ellos y
- El *mecanismo de acceso* que, como su nombre lo indica, regula la manera en que los datos de la estructura son accedidos.

Clase En los lenguajes de programación orientados a objetos una clase es la definición de un tipo de objetos estableciendo los datos que estos contendrán como atributos y las operaciones que constituyen sus métodos. Una clase es una de las posibles implementaciones de un TDA.

Notación asintótica Si $f(n)$ es una función que determina la complejidad en espacio o tiempo de un algoritmo cuya complejidad de entrada está determinada por n y $g(n)$ es una función usada como patrón de medida de complejidad. Diremos que f es *del orden de* g, o en notación $f = O(g)$ si existe una cierta complejidad de entrada N a partir de la cual podemos acotar a f por arriba, con algún múltiplo de g. Esto significa que f, la complejidad del algoritmo en cuestión, no crece más rápido que el patrón g. O bien que el algoritmo con complejidad f es al menos tan eficiente como uno de complejidad g.

Invariantes En un ciclo el invariante es una condición que es verdadera antes de ejecutar el ciclo por primera vez, antes de repetir el ciclo e inmediatamente después de la última iteración de él. Son fundamentales para probar la corrección de un algoritmo.

Paréntesis: Complejidad computacional.

El estudio de la complejidad computacional comenzó a principios de la década de los 60 en el siglo XX. Ya antes los matemáticos dedicados a la teoría de números habían analizado la complejidad de muchos de los algoritmos vinculados con esa área, de hecho fueron ellos quienes iniciaron el uso de la notación asintótica[a].

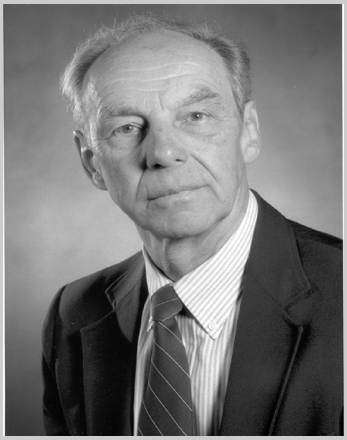

Figura 1.7: Juris Hartmanis.

En 1960 M.O. Rabin escribió acerca de los diferentes grados de dificultad para calcular una función y un orden parcial en conjuntos definidos recursivamente. Pero fue Juris Hartmanis (fig. 1.7) junto con R.E. Stearns quienes hicieron el primer estudio sistemático de la complejidad de las funciones calculables y bautizaron el área como *complejidad computacional* [4]. Hartmanis estableció los requisitos que debía cumplir una buena medida de complejidad y el vínculo entre la función de medida y el tipo de máquina de Turing asociado con el cálculo del algoritmo cuya complejidad se mide [4, 5].

Durante toda la década de los 60 y los 70 el área se vió fortalecida con resultados teóricos acerca de las clases de complejidad que es posible derivar dada la función de medida de complejidad. Además de que se volvió indispensable analizar la complejidad de los algoritmos nuevos y viejos que se ponían en el camino. A fines de los 70 la notación y la terminología del área quedó tal como la conocemos hoy día.

[a] $O()$ fué usado por primera vez en: Bachmann, P. *Analytische Zahlentheorie, Bd. 2: Die Analytische Zahlentheorie.* Leipzig, Teubner, 1894. Poco después se añadió $o()$ en: Landau, E. *Handbuch der Lehre von der Verteilung der Primzahlen.* Leipzig, Teubner, 1909. A partir de entonces a esta notación también se le conoce como los *símbolos de Landau.*

Referencias

[1] Bentley, Jon, *Programming Pearls*, 2a Ed., Addison-Wesley, 2000. `http://www.cs.bell-labs.com/cm/cs/pearls/`

[2] Giegerich, R. and S. Kurtz, "From Ukkonen to McCreight and Weiner: A Unifying view of linear time suffix tree construction", *Algorithmica*, Vol. 19, No. 3, 1997, pp. 331-353.

[3] Gusfield, Dan, *Algorithms on Strings, Trees and Sequences. Computer Science and Computational Biology*, Cambridge University Press, 1997.

[4] Hartmanis, J. y J.E. Hopcroft, "Overview of the Theory of Computational Complexity", *Journal of the ACM*, Vol. 18, No. 3, 1971, pp. 444–475.

[5] Hartmanis, J., "Observations About the Development of Theoretical Computer Science", *Annals of the History of Computing*, Vol. 3, No. 1, 1981, pp. 43–51.

[6] McCreight, Edward, "A Space-Economical Suffix Tree Construction Algorithm", *Journal of the ACM*, Vol. 23, No. 2, 1976, pp. 262-272.

[7] Ukkonen, Esko, "On-line Construction of Suffix Trees", *Algorithmica*, Vol. 14, No. 3, pp 249-260, 1995.

Ejercicios

1.1 Usando los axiomas de Huntington del TDA 1.1 demuestre:

a) $or(x, \text{VERDADERO}) = \text{VERDADERO}$
o equivalentemente
$and(x, \text{FALSO}) = \text{FALSO}$.

b) $or(x, and(not(x), y)) = or(x, y)$
o equivalentemente
$and(x, or(not(x), y)) = and(x, y)$.

1.2 Usando nuevamente los axiomas de Booleano y el ejercicio anterior demuestre las leyes de *de Morgan*:
$not(or(x, y)) = and(not(x), not(y))$
o bien
$not(and(x, y)) = or(not(x), not(y))$.

1.3 Una especificación equivalente a la de booleano mostrada en el TDA 1.1 es la mostrada en el TDA 1.3. Se han excluido las clausulas referentes al comportamiento del *or* y en su lugar se ha puesto la expresión de una de las leyes de *de Morgan*. Demuestre las clausulas faltantes con base en los nuevos axiomas.

1.4 Un polinomio de grado n es una expresión de la forma:

$$a_0 + a_1x + a_2x^2 + \cdots + a_nx^n$$

TDA 1.3 Especificación alternativa del TDA Booleano.

TDA **Booleano**
  CONSTANTES
    VERDADERO, FALSO : Booleano
  OPERACIONES
    and : Booleano × Booleano ↦ Booleano
    or : Booleano × Booleano ↦ Booleano
    not : Booleano ↦ Booleano
  VARIABLES
    x, y : Booleano
  AXIOMAS
    [B1] $and(x, y) = and(y, x)$
    [B2] $and(\text{VERDADERO}, x) = x$
    [B3] $and(\text{FALSO}, x) = \text{FALSO}$
    [B4] $not(\text{VERDADERO}) = \text{FALSO}$
    [B5] $not(\text{FALSO}) = \text{VERDADERO}$
    [B6] $or(x, y) = not(and(not(x), not(y)))$
FIN **Booleano**

Para especificar completamente el polinomio sólo se requieren los $n + 1$ coeficientes $a_0, a_1, \ldots, a_n$. Para los propósitos de esta práctica todos los coeficientes serán números naturales.

Un polinomio se evalúa asignando un valor a su variable independiente, denotada con x ($x \in \mathbb{R}$) y calculando:

$$p(x) = \sum_{i=0}^{n} a_i \, x^i \tag{1.8}$$

La manera más ingenua de hacer esto es calculando cada uno de los términos de la suma en la expresión 1.8. El siguiente algoritmo lleva a cabo este procedimiento para evaluar, en el punto x, el polinomio de grado n, cuyos coeficientes se suponen almacenados en el vector **a**.

```
EvaluaPolinomio(x, n, a)
1  resultado ← a[0]
2  for i ∈ {1, ..., n} do
3      resultado ← resultado + a[i] · Potencia(x, i)
4  endfor
5  return resultado
6  end
```

La i-ésima entrada del vector $\mathbf{a}$, es el coeficiente asociado con la i-ésima potencia de x. Presuponemos la existencia de la función Potencia(x, p), que eleva x a la p-ésima potencia ($p \in \mathbb{N}$, véase los algoritmos 1.1 y 1.2).

Existe otra manera de evaluar un polinomio conocida como la *regla de Horner*. Consiste en acumular el resultado actual, multiplicado por el valor de la variable independiente y sumado con el siguiente coeficiente del polinomio, comenzando por el coeficiente asociado con la potencia mayor. El algoritmo, puesto en términos más explícitos, es el siguiente:

```
EvaluaPolinomioHorner(x, n, a)
1  resultado ← a[n]
2  for i ∈ {n − 1, ..., 0} do
3      resultado ← resultado · x + a[i]
4  endfor
5  return resultado
6  end
```

Hay que hacer notar que la variable índice i, recorre sus valores decrecientemente.

a) Elabore una clase `Polinomio` en Java que modele polinomios de grado arbitrario, variable real (`double`) y coeficientes naturales. Que posea sendos métodos para evaluar el polinomio mediante el algoritmo ingenuo y el de Horner. El constructor de la clase sólo requiere al vector de coeficientes.

b) Elabore un programa que genere polinomios con coeficientes aleatorios distintos de cero (pero no muy grandes, digamos de un dígito) y con grados entre 1 y 20, los evalué con cada uno de los procedimientos descritos y calcule el tiempo que tarda la evaluación. A la salida se debe obtener una tabla del tiempo de cada método de evaluación Vs. el grado del polinomio, cada tiempo en la tabla debe ser resultado de promediar, al menos, 10 observaciones. Esta es una tabla con tres columnas: grado, tiempo de Horner y tiempo del ingenuo[4].

[4]Seguramente será necesario añadir tiempo para poder observar algo significativo, se debe introducir un retardo en cada multiplicación de cada uno de los métodos para poder afectarlos con justicia. Para hacer esto se debe añadir una variable de clase `Thread` como atributo de `Polinomio` e invocar `sleep`.

c) Grafique los datos obtenidos y observe cuál de los dos procedimientos es mejor.

d) Modifique la tabla obtenida, obtenga una cuarta columna con la raíz cuadrada de los valores de la que contiene los tiempos del algoritmo ingenuo. ¿Qué se obtiene ahora al volver a graficar? ¿Que significa esto? formule hipótesis acerca de la complejidad temporal de cada método.

e) Demuestre que las hipótesis que formuló luego de los experimentos son ciertas analizando la complejidad de los algoritmos.

1.5 Obtenga los invariantes para los ciclos de los algoritmos 2 y 4 del problema del subvector de suma máxima. Pruebe que los algoritmos son correctos haciendo uso de los invariantes.

1.6 No hay un ciclo en el algoritmo 3 para el problema del subvector de suma máxima, sin embargo podría hablarse de una especie de *invariante de recursión*, una condición que sea garantizada cada vez que se termina el sub-algoritmo SumParticion. Determine esta condición. Con base en ella demuestre, por inducción, que el algoritmo 3 es correcto.

Arreglos 2

... los arreglos siempre se manejan de forma clara y precisa, como debe ser, anteponiendo los intereses de la nación a los intereses personales o partidistas.

—Diego Fernández de Cevallos en *Reforma*, 23 de julio 2002.

2.1 Definiciones, características.

Sin duda en cursos previos el lector tuvo contacto con los arreglos y probablemente le parezca extraño reencontrarlos aquí considerando la simplicidad de su manejo, ¿qué más podría decirse de los arreglos que no sea evidente? A lo mejor no mucho, pero como los arreglos son tan útiles y por tanto tan usuales, conviene tratarlos con la debida formalidad con la que trataremos al resto de las estructuras de datos.

Podemos definir formalmente un arreglo n-dimensional como sigue:

Definición 2.1 Un arreglo de dimensión n, de elementos de tipo X y de tamaño $T = t_1 \times t_2 \cdots \times t_n$, donde t_j es el tamaño del arreglo en la j-ésima dimensión, es un conjunto de T elementos de tipo X, en el que cada uno de ellos es identificado unívocamente por un vector coordenado de n índices $(i_1, i_2, \ldots, i_n)$, donde $0 \leq i_j < t_j$.

Esta definición es general, aplicable a cualquier leguaje de programación. Es, lo que

podríamos llamar, una definición de *alto nivel de abstracción*. Pero en general nos serán más cotidianas las que formularemos posteriormente.

El caso más simple de arreglo es, por supuesto, el de un arreglo unidimensional. La simplicidad de arreglo unidimensional proviene de que en él la organización de los datos coincide con la de la memoria misma. A fin de cuentas la memoria de una computadora es una secuencia de celdas consecutivas, cada una de ellas es identificada unívocamente por un número al que se denomina *la dirección*. En cada celda de memoria caben exactamente ocho bits (un byte), por lo que se dice que el byte es la unidad de memoria direccionable mínima. Todos los demás tipos de datos elementales ocupan un número entero de bytes consecutivos en la memoria, un entero (`int`) de Java, por ejemplo, ocupa cuatro bytes consecutivos (32 bits). Resultaría en extremo ineficiente almacanar estos tipos de datos en bytes dispersos o en un número no entero de bytes.

Del mismo modo un arreglo unidimensional de tipo X es una secuencia de celdas consecutivas en las que caben datos de tipo X, por supuesto cada dato de tipo X es, a su vez, una secuencia de celdas de memoria del tamaño de un byte. Así que un arreglo unidimensional es realmente un bloque contiguo de memoria dividido virtualmente en un número entero de celdas adecuadas para almacenar elementos del tipo del arreglo. Esto tiene implicaciones importantes: cuando se define un arreglo de tipo X, se debe solicitar un área de memoria para almacenarlo, así que es necesario decir de antemano cuantas celdas se requieren, digamos n. El sistema operativo busca entonces un bloque de tamaño $n \times s$ donde s es el tamaño en bytes del tipo X. Esta es la causa de que sea tan sencillo acceder a una celda arbitraria de arreglo, si se desea el dato de la celda i basta que el compilador acceda a las s celdas de memoria de tamño byte que están a partir de la dirección:

$$d_i = d_0 + i \times s \tag{2.1}$$

donde d_0 es la dirección que el sistema operativo asigno para el inicio del arreglo (dirección del primer byte de la primera celda del arreglo), esto no se podría hacer si el arraglo no estuviera en un sólo bloque de memoria. Pero esto también trae consigo una desventaja: una vez que el bloque es asignado al programa que lo solicitó este puede usarlo como desee, pero no puede exceder los límites de dicho bloque porque estaría accediendo, potencialmente, a áreas de memoria asignadas a otros datos, así que si se requieren almacenar más de n datos en el arreglo se debe solicitar nuevamente un bloque de memoria contigua y copiar el contenido del bloque anterior al nuevo.

Seamos formales:

Definición 2.2 Una *celda* es un bloque contiguo de memoria. El tamaño de la celda está determinado por el tipo de datos que se almacenan en ella. Al tipo del elemento contenido en la celda le llamaremos, abusando de la terminología, el *tipo de la celda*.

Definición 2.3 Un *arreglo unidimensional* de tamaño N de elementos de tipo X, es un bloque contiguo de memoria, dividido en una colección de N celdas consecutivas, todas ellas

de tipo X, accesibles a través de un *índice* en el conjunto $\{0, \ldots, N-1\}$ que las identifica unívocamente[1]. Se puede acceder a cualquier elemento de un arreglo en cualquier momento haciendo referencia a él a través del índice de la celda en que está contenido. Una vez establecido, el tamaño del arreglo es fijo e inalterable, no es posible eliminar celdas ni insertarlas.

En síntesis, los datos de un arreglo están en celdas consecutivas que a su vez están formadas de celdas de memoria consecutivas. Los datos en el arreglo deben ser del mismo tipo (porque entonces seguro tienen el mismo tamaño) y una vez establecido el tamaño del arreglo este no se puede cambiar.

A una expresión como 2.1 se le denomina *polinomio de direccionamiento*, Dada la localización de un dato en una estructura C, el polinomio de direccionamiento permite conocer su ubicación en una estructura S subyacente a C. En general S está en un nivel de abstracción inferior a C, es decir, los polinomios de direccionamiento mapean una estructura virtual compleja en otra más simple con la que se ha implementado aquella. En el caso de 2.1, dado el índice del dato en el arreglo del elementos de tipo X, se obtiene la dirección de memoria del primer byte del dato, no puede haber nada menos abstracto que una celda de memoria.

Nosotros también nos daremos a la tarea de calcular polinomios de direccionamiento un poco más adelante, no precisamente del tipo de los calculados por los compiladores porque a nosotros no nos interesa saber la dirección efectiva en memoria física de nuestros datos, pero si nos interesará conocer la posición (índice) de un dato en un arreglo lineal cuando la estructura virtual que estemos utilizando no sea lineal.

Los arreglos de más de una dimensión son una generalización del caso unidimensional, en ellos para acceder a un dato particular hay que especificar su posición diciendo más de un índice. El caso más simple es, por supuesto, el de un arreglo bidimensional, que podemos imaginar como una matriz en la que un dato específico se identifica diciendo el renglón y la columna en la que se encuentra, es decir, dos índices. Claro que esto se generaliza a más dimensiones, en un arreglo n dimensional se requieren n índices para especificar un dato particular.

Hay dos maneras diferentes en las que los lenguajes de programación implementan los arreglos multidimensionales. Lenguajes como, el ahora olvidado, Pascal y Fortran, el más antiguo de los lenguajes, implementaban los arreglos multidimensionales como un arreglo unidimensional en el que, para localizar un dato, el compilador calculaba el polinomio de direccionamiento adecuado. En este esquema un arreglo bidimensional de R reglones y C columnas realmente se implementa como un arreglo unidimensional de $R \times C$ celdas, acceder al elemento en la posición (r, c), donde r es el renglón y c la columna, consiste en acceder a la celda i-ésima del arreglo unidimensional, donde $i = r \times C + c$. Por supuesto para reservar memoria suficiente para el arreglo deben darse, de una sóla vez, los tamaños en *todas* sus dimensiones.

[1]La decisión acerca de cuál debe ser el valor del índice inicial y del final es arbitraria. Algunos lenguajes establecen el primer índice en 1, otros en cero y podría ser cualquier otro valor siempre y cuando los índices sean consecutivos.

En languajes como C, C++ y Java la situación es diferente. En estos lenguajes un arreglo n dimensional es un arreglo de arreglos de arreglos ... (n veces). Es decir, un arreglo bidimensional es un arreglo de arreglos unidimensionales, uno tridimensional es un arreglo de arreglos bidimensioales, o sea un arreglo de arreglos de arreglos. Esto tiene la ventaja de que puede diferirse la reservación de memoria para las dimensiones superiores siempre y cuando se reserven las primeras. Podríamos decir que un arreglo bidimensional tendrá 20 renglones y después definir cuantas columnas, de hecho podríamos definir un número diferente de columnas para cada renglón.

Definición 2.4 Un arreglo de dimensión n de elementos de tipo X es:

- Si $n = 1$ es un arreglo unidimensional de acuerdo con la definición 2.3.
- Si $n > 1$ es un arreglo unidimensional de arreglos de dimensión $n - 1$ de tipo X.

Usando esta definición podemos visualizar un arreglo tridimensional de $L \times N \times M$, como un arreglo unidimensional con L entradas, una por cada "piso" del arreglo tridimensional, cada una de estas entradas, digamos la i-ésima, hace referencia a su vez, a un arreglo unidimensional de N entradas, cada entrada de este arreglo hace referencia a un último arreglo unidimensional de M entradas, que contiene los elementos de un renglón del piso i del arreglo tridimensional. Esto es mostrado esquemáticamente en la figura 2.1. De hecho podríamos pensar en un arreglo similar al mostrado, pero con pisos de tamaño diferente o con tamaños variables arbitrarios en los renglones de cada piso.

Mencionamos ya el hecho de que es imposible hacer crecer un arreglo, algo similar ocurre si se desea eliminar datos de él. Borrar un dato de un arreglo, esto es, *eliminar su celda*, también es imposible porque la dirección de la celda está dentro del conjunto de direcciones asociadas al arreglo y estas deben ser consecutivas. Esta última operación puede, sin embargo, ser simulada como sigue: si se quiere eliminar la i-ésima celda del arreglo, en realidad lo que se pretende es que en el i-ésimo sitio quede el dato actualmente almacenado en el lugar $i + 1$, así que basta con copiar el dato del lugar $i + 1$ al lugar i, el del lugar $i + 2$ al $i + 1$ y asi sucesivamente hasta copiar el dato en el último lugar en el penúltimo. Algo costoso porque implica hacer $n - i - 1$ operaciones de copia, donde n es el tamaño del arreglo.

2.2 Creación de un arreglo.

Hay dos momentos fundamentales en la creación de un arreglo y no debemos confundirlos. El primero es la *declaración*, es el momento en que se informa al compilador la existencia de una nueva variable que sevirá para referirse al arreglo y sólo eso, por

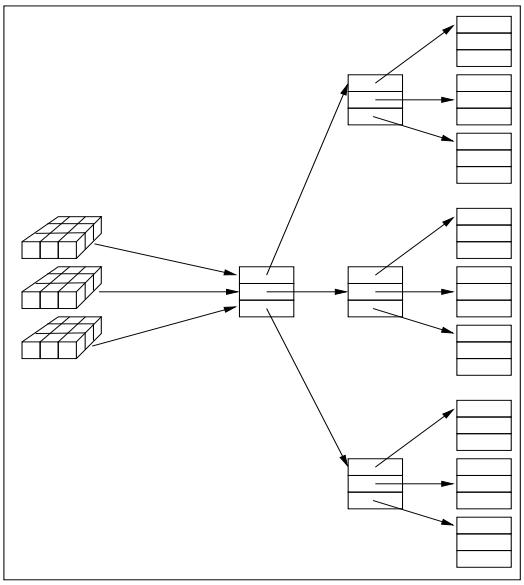

Figura 2.1: Un arreglo tridimensional representado como un arreglo de arreglos de arreglos. Cada piso es un arreglo unidimensional donde cada entrada es una referencia a un arreglo unidimensional que contiene referencias a tres arreglos unidimensionales que contienen las entradas del piso.

ahora nada tiene que ver con el tamaño del arreglo ni con sus celdas, sólo define una caja en la que se guardará, después, la dirección de inicio del arreglo. En la declaración:

```
int[][]  matriz;
```

se informa que, de ahora en adelante `matriz` será la manera de referirse a la caja donde se guardará la dirección de inicio de un arreglo que aún no existe. Por el momento la caja está vacía.

El segundo momento de la creación de un arreglo es la *solicitud de memoria* para él o, al menos, para alguna de sus dimensiones. Por ejemplo:

```
matriz = new int[nreng][];
```

es una solicitud válida, le pide al sistema operativo que proporcione la dirección de inicio de un bloque de memoria de tamaño igual al contenido de la variable `nreng`. Cuando la dirección del bloque es entregada por el sistema, esta se almacena en la caja etiquetada `matriz`, que definimos antes. Ahora hay un bloque de `nreng` celdas consecutivas que servirán, igual que `matriz` como cajas para guardar direcciones de bloques.

Más tarde se hace:

```
for (i = 0; i < nreng; i++) {
   matriz[i] = new int[i + 1];
}
```

esto hace `nreng` solicitudes de bloque al sistema operativo: la primera pide una sóla celda del tamaño de un entero (4 bytes) y la dirección de el bloque se guarda en el lugar 0 del bloque de `nreng` celdas pedidas anteriormente. La siguiente pide un bloque de dos celdas de tamaño `int`, luego se piden tres y así sucesivamente. Al final cada celda de la primera dimensión del arreglo tiene la dirección de inicio de un bloque contiguo en donde se almacenarán realmente las entradas del arreglo.

En lenguajes como Pascal o Fortran la declaración y la reservación de memoria se debían hacer al mismo tiempo, como cuando en Java se hace[2]:

```
int[][] matriz = new int[nreng][4];
```

uniendo los dos momentos mencionados en uno solo. Esto permite conocer de una sola vez el tamaño total del arreglo y por tanto reservar memoria para todo él.

A diferencia de Java, en los lenguajes que no permiten la reservación diferida de memoria se obtiene un sólo bloque contiguo para todo el arreglo, sin importar si es unidimensional o no. Para proveer al arreglo de su estructura virtual multidimensional se hace necesario, el uso de los mencionados *polinomios de direccionamiento*, calculados por el compilador. Si suponemos que los renglones de un arreglo bidimensional son colocados uno tras de otro consecutivamente, de tal forma que la primer celda del renglon i suceda a la última del renglón $i-1$, entonces, para obtener la dirección del byte inicial de la celda en la posición (ren, col) de un arreglo bidimensional de m renglones y n columnas se tendría que calcular:

$$dir(ren, col) = \texttt{dir\_matriz} + (n \cdot ren + col) \cdot \texttt{tam}$$

donde *tam* denota el tamaño (en bytes) del tipo de dato que se está almacenando. El producto $n \cdot ren$ es, de hecho, el número total de celdas de la matriz[3] en los renglones previos al indicado por ren. La celda de la matriz en la posición determinada por este producto es la que encabeza el renglón ren, una vez allí es necesario desplazarse *col* celdas más para llegar a la columna deseada.

Usando el polinomio de direccionamiento que acabamos de calcular, podríamos pensar en almacenar un arreglo bidimensional en uno lineal, por nuestra propia cuenta sin dejar que el compilador lo haga por nosotros. En ese caso podemos decir que la celda en la posición (ren, col) de nuestro arreglo "virtual" bidimensional de m renglones y n columnas se guarda en la celda $n\ ren + col$ del arreglo lineal que realmente usamos. De hecho podríamos encapsular esto dentro de una clase que manipula matrices numéricas bidimensionales, digamos, de tal forma que el usuario de la clase pueda solicitar la entrada (i, j) de la matriz sin que para él sea relevante el hecho de que la matriz está almacenada en un arreglo lineal y no en uno bidimensional. Otro programador

[2]De hecho en Pascal ni siquiera era posible poner variables como especificadores de tamaño en las dimensiones

[3]Conviene hacer énfasis en que son celdas de la matriz y no de la memoria. En el caso del ejemplo serían celdas del tamaño de cuatro bytes (tamaño del tipo `int`) y no de un byte.

podría no complicarse la vida inútilmente y usar el arreglo bidimensional evidente proporcionando los mismos servicios. Ambas clases serían intercambiables, la esencia del encapsulamiento.

En este caso no tiene mucho sentido almacenar en un arreglo lineal y calcular "a pie" el polinomio de direccionamiento, pero esta breve introducción nos será útil más adelante.

Por supuesto se pueden definir arreglos de un número arbitrario de dimensiones y se pueden acceder sus elementos haciendo uso de polinomios de direccionamiento más complicados. En una matriz tridimensional de $\ell \times m \times n$ (alto, ancho y largo) acomodada linealmente habría que desplazarse i m n celdas para llegar al piso i, luego desplazarse j n celdas para llegar al renglón j en ese piso y finalmente recorrer k celdas más para llegar a la celda de coordenadas (i, j, k), es decir:

$$idx = (i \cdot m \cdot n + j \cdot n + k) \tag{2.2}$$

sería el polinomio de direccionamiento para obtener el índice, en un arreglo lineal, de la celda en cuestión. En general el polinomio de direccionamiento de un arreglo n dimensional es de grado n.

2.3 Polinomios de direccionamiento.

Hasta ahora hemos calculado polinomios de direccionamiento que no tienen mucho caso. Primero lo hicimos para ejemplificar una labor propia del compilador y luego sólo como un medio para tener una representación alternativa, ni más ni menos eficiente, de arreglos multidimensionales.

En esta sección adquieren mayor sentido aunque no dejan de parecer una alternativa, no más eficiente que otras, para acceder a datos almacenados de manera óptima. La importancia real estriba en que hay problemas en los que la otra alternativa no existe (recuerdo por ejemplo algunos malabares que tuve que hacer con polinomios de direccionamiento para manipular imágenes en la pantalla y para analizarlas o algunos otros para manipular estructuras de datos no lineales en lenguaje ensamblador) y siempre es bueno saber echar mano de estas cosas.

Supongamos que estamos trabajando en un programa que involucra el manejo de gráficas no dirigidas pesadas y completas. Conviene recordar que una gráfica es una pareja de conjuntos $G = (V, E)$, V es el conjunto de vértices de la gráfica y E el de aristas, cada arista $v \in V$ une dos vértices, y puede ser recorrida en cualquiera de sus dos sentidos, es decir: $v = (e_1, e_2) = (e_2, e_1)$; además cada arista tiene asociado un número real no negativo llamado *peso* y que denotamos como $w(v)$.

Como las aristas unen parejas de vértices lo natural es representarlas usando el producto cartesiano de estos $E \times E$. Además cada arista tiene un peso, así que podemos

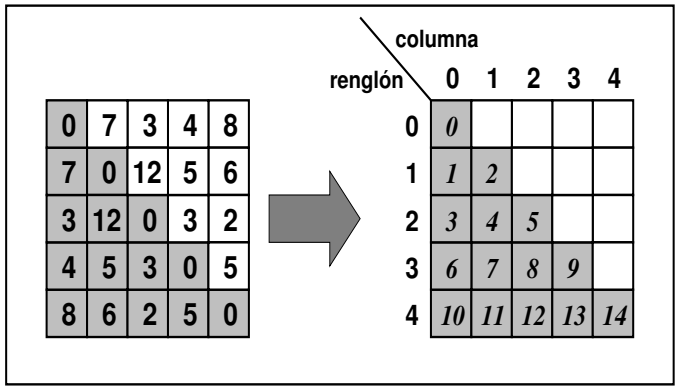

Figura 2.2: De una matriz simétrica puede almacenarse sólo el triángulo inferior. De hecho puede almacenarse en un arreglo lineal haciendo el mapeo adecuado entre las posiciones de la matriz original, en términos de renglones y columnas, y las posiciones en el arreglo lineal (en itálicas dentro de las celdas).

pensar en almacenar la información de una gráfica en una matriz de pesos de $n \times n$ donde n es el número de vértices, en la entrada del renglón i, columna j de la matriz se guarda el peso de la arista $v = (e_i, e_j)$, a esto se le llama la *matriz de adyacencia* de la gráfica. Ahora bien, como la gráfica es no dirigida y el peso para ir de e_i a e_j es el mismo sin importar el sentido en el que se recorra la arista, entonces en la matriz de pesos m se tiene que $m[i, j] = m[j, i]$, es decir es una matriz simétrica. Esto significa que no es necesario más que almacenar la mitad de la matriz, digamos el triángulo inferior como se muestra en la figura 2.2.

Podemos pensar, de hecho, en programar una clase que represente gráficas mediante sus matrices de adyacencia y que en caso de toparse con una gráfica no dirigida sólo guarde las entradas del triángulo inferior de dicha matriz. Podemos hacer esto de dos formas diferentes.

Podríamos aprovechar las facilidades de Java y definir un arreglo bidimensional cuyos renglones tengan el tamaño adecuado. Por ejemplo:

```
double[][] matriz;
   ...
matriz = new double[dim][];
   ...
for (i = 0; i < dim; i++) {
   matriz[i] = new double[i+1];
}
```

O bien podríamos usar un polinomio de direccionamiento, para así almacenar en un arreglo lineal únicamente el triángulo inferior de la matriz de adyacencia y sin embargo saber cuál es la entrada en una posición dada de la matriz de completa.

Aparentemente ambas alternativas son igualmente buenas (en relidad no, como veremos más adelante), la primera es directa y simple, la segunda requiere de más trabajo.

Para hacer funcional el almacenamiento en un arreglo lineal necesitamos calcular el índice efectivo en dicho arreglo, digamos idx, de la celda que conceptualmente, está en la columna col del renglón ren de la matriz de adyacencia. Nuevamente lo primero que necesitamos es ponernos en la primera celda del renglón donde se encuentra la entrada (ren, col), lo que significa contar el número de celdas previas. Antes del renglón 0 hay cero entradas, antes del 1 hay una, antes del 2 hay $1+2=3$, antes del 3 hay $1+2+3=6$, en general antes del renglón k hay:

$$\sum_{i=0}^{k} i = \frac{k\,(k+1)}{2}$$

celdas previas. Así que la celda del arreglo lineal que corresponde a la primera celda del renglón ren es la de índice:

$$\frac{ren\,(ren+1)}{2}$$

Ahora sólo resta desplazarnos col lugares a partir de ella para llegar a la celda en la columna con ese índice, esto para aquellas celdas en el triángulo inferior. Si pretendemos llegar a una celda del triángulo superior no podemos hacer esto porque estas no se guardan en el arreglo lineal, en este caso haremos uso de la propiedad de simetría de la matriz invirtiendo los índices ren y col. En síntesis, podemos almacenar sólo en triángulo inferior de una matriz simétrica m de $n \times n$ en un arreglo lineal a, el valor de la entrada en el renglón ren columna col de m estará dado por:

$$m[ren, col] = \begin{cases} a\left[\frac{ren\ (ren+1)}{2} + col\right] & \text{Si } ren \geq col \\ \\ a\left[\frac{col\ (col+1)}{2} + ren\right] & \text{Si } ren < col \end{cases}$$

El número de celdas necesarias es:

$$\frac{n\ (n+1)}{2}$$

donde n es la dimensión de la matriz original.

Con lo anterior logramos una representación compacta de una matriz simétrica cualquiera o de una matriz triangular inferior, pero en nuestro caso hay una característica adicional que no hemos considerado: la matriz de adyacencia tiene ceros en la diagonal, es decir el costo de ir del vértice i a él mismo es cero, esto no necesariamente es así siempre, pero en nuestro caso podemos suponer que así es. esto significa que aún desperdiciamos espacio almacenando todos lo ceros de la diagonal. Podríamos no guardarlos, en ese caso reduciremos en n el número de celdas necesarias, que ahora sería:

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

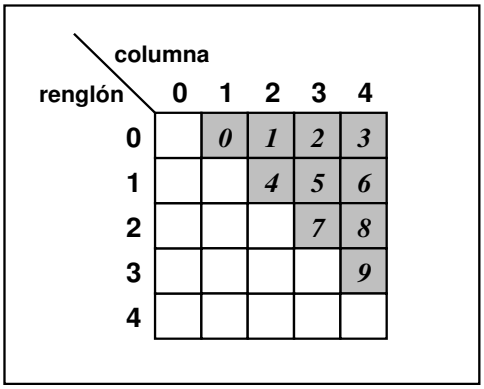

Figura 2.3: Almacenamiento del triángulo superior de una matriz en un arreglo lineal

El polinomio de direccionamiento ahora debe calcular la suma de los primeros $k-1$ números naturales para determinar el número de celdas previas al renglón k, por lo que ahora tendríamos:

$$m[ren, col] = \begin{cases} a\left[\frac{ren\ (ren-1)}{2} + col\right] & \text{Si } ren > col \\ 0 & \text{Si } ren = col \\ a\left[\frac{col\ (col-1)}{2} + ren\right] & \text{Si } ren < col \end{cases}$$

No siempre es tan sencillo calcular el polinomio de direccionamiento, de hecho a a veces una mala elección del punto de vista puede traer complicaciones innecesarias. Siempre es bueno pensar en más de una alternativa para guardar los datos para poder decidir cuál genera cálculos más simples. Pensemos por ejemplo en nuestro mismo caso: almacenar la matriz de adyacencia en un arreglo lineal, sin la diagonal de ceros, pero ahora en vez de guardar el triángulo inferior se nos ocurre, sin ver nuestros resultados previos, guardar el triángulo superior, como en la figura 2.3

Ahora antes del renglón k hay $(n-1)+(n-2)+\cdots+(n-k)$ celdas previas, es decir la celda del arreglo lineal que corresponde a la primera que hay que guardar del renglón ren de la matriz original es:

$$\begin{aligned} \sum_{j=1}^{n-1} j - \sum_{j=1}^{n-ren-1} j &= \frac{n(n-1)}{2} - \frac{(n-ren)(n-ren-1)}{2} \\ &= \frac{ren(2n-ren-1)}{2} \end{aligned}$$

Luego, estando ya en el renglón ren, habría que llegar hasta la columna col, pero en el renglón hay ren celdas que pertenecen al triángulo inferior y una más que pertenece a la diagonal, así que esas no cuentan, en total habría que desplazarse $col-(ren+1)$ celdas más. En síntesis, si guardamos el triángulo superior de la matriz m de $n \times n$, sin

la diagonal, en el arreglo lineal a:

$$m[ren, col] = \begin{cases} a\left[\frac{ren\ (2n-ren-1)}{2} + (col - ren - 1)\right] & \text{Si } ren < col \\ 0 & \text{Si } ren = col \\ a\left[\frac{col\ (2n-col-1)}{2} + (ren - col - 1)\right] & \text{Si } ren > col \end{cases}$$

La expresión es un poco más complicada que las anteriores y deducirla fue también un poco más laborioso. El costo de la deducción no sería muy relevante, después de todo sólo se hace una vez, pero en este caso el costo extra invertido nos dio un balance negativo.

Sin nuestras experiencia previas bien podría ocurrir que se nos hubiera ocurrido esta última alternativa en vez de las más simples. La moraleja obvia es que siempre hay que buscar formas alternativas para organizar los datos y en muchos casos una de ellas parecerá la más natural y por tanto la más simple; a esa se le debe dar preferencia a menos que posea inconvenientes insospechados. Un principio similar al de la navaja de Occam: el modelo más simple es casi siempre el correcto.

2.4 Arreglos empacados.

Otro de los casos en los que es importante optimizar el espacio utilizado en un arreglo es cuando los datos que se guardan en él son de un tamaño que no es múltiplo entero de un byte. Es decir, las celdas que contienen los datos en el arreglo poseen un tamaño que no se ajusta a un número entero de bytes, la unidad mínima direccionable en memoria. Esto significa que para poder almacenar de manera óptima los datos se debería poder direccionar la memoria usando unidades más pequeñas, lo que es imposible, o bien simulándolo haciendo cálculos como los que hemos hecho con nuestros polinomios de direccionamiento.

EJEMPLO 2.1

Para ilustrar esto vamos a poner un ejemplo interesante: códigos correctores de error de Hamming, en particular el código de Hamming de 7 bits.

Este código está constituido de palabras de siete bits de longitud y se utiliza para transmitir datos a través del espacio o del tiempo de tal forma que estos puedan ser recuperados en otro lugar o más tarde con la garantía de que, si ha ocurrido a lo más un error de transmisión por palabra, lo recuperado coincidirá exactamente con lo transmitido eliminando así los errores. Veamos cómo se hace posible esto.

Decimal	$\mathbf{b_2}$	$\mathbf{b_1}$	$\mathbf{b_0}$
0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1

Tabla 2.1: Las posibles posiciones de error en una palabra de siete bits de largo.

el código con el que trataremos posee palabras de 7 bits de longitud. Así que debe poder decirnos, en caso de error, el índice del bit incorrecto. Diremos que los bits de la palabra están numerados del uno al siete, así que usando el código debemos ser capaces de decir un número en el conjunto $\{0, 1, \ldots, 7\}$, el cero lo usaremos para decir que no hay error (al menos no un sólo error) y el resto para decir la posición donde se ha detectado el bit erróneo.

En la tabla 2.1 se muestran, en binario todos los números del 0 al 7. Observándola nos percatamos de que:

- El bit etiquetado b_0 se "prende", es decir, vale 1, en 1, 3, 5 y 7.
- El bit b_1 vale 1 en 2, 3, 6 y 7 y
- El bit b_2 se prende en 4, 5, 6 y 7.

Así que los utilizaremos para verificar esas posiciones dentro de la palabra de código de siete bits.

La verificación consistirá en algo que probablemente el lector ya conozca: verificación de paridad par. El esquema de paridad par consiste en completar siempre un número par de unos en una transmisión binaria, si se envía un dato, por ejemplo 1010001 y se añade un bit de paridad par para completar un número par de unos entonces de enviará 10100011, si en cambio se pretende enviar 1011010 entonces la transmisión será 10110100 porque el dato ya tenía por sí mismo un número par de unos.

Por lo que dedujimos de la tabla 2.1 vamos a agregar tres bits de verificación de paridad par, eso nos permite decir una posición errónea entre 1 y 7 más el cero que significa "no hay error detectable". Así que de nuestra palabra de 7 bits tres de ellos son

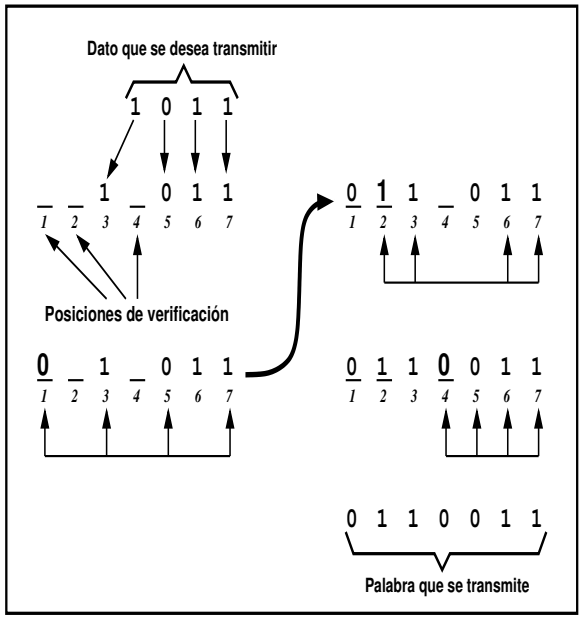

Figura 2.4: Construcción de la palabra de código de Hamming de 7 bits para el dato 1011.

de verificación de paridad, los cuatro restantes serán el dato que pretendemos transmitir. Las posiciones dentro de la palabra para los bits de verificación de paridad serán los índices iniciales de las posiciones verificadas por cada uno de ellos, es decir: 1, 2 y 4.

Supongamos que pretendemos transmitir el dato 1011. En la figura 2.4 se muestra lo que se debe hacer para transformarlo en una palabra de Hamming de 7 bits. El bit 1 se pone en cero porque las posiciones 3, 5 y 7 tienen ya un número par de unos, el bit 2 en cambio se pone en 1 porque las posiciones 3, 6 y 7 tienen en conjunto tres unos, que es impar; el bit 4 se pone en cero porque las posiciones 5, 6 y 7 tienen en conjunto dos unos.

Supongamos que ocurre un error en la posición tres de la palabra de 7 bits. Es decir el receptor recibe 0100011 en vez del 0110011 que le fue transmitido. En ese caso el verificador se dará cuenta de que hay un error en alguna de las posiciones siguientes 1, 3, 5 o 7 o bien en 2, 3, 6 o 7, estos conjuntos tienen en común al 3 y al 7, pero si el bit erróneo fuera el 7 también habría error de paridad en el grupo 4, 5, 6 o 7, así que la única posibilidad es que esté mal el bit 3. La manera en que esto se hace en la práctica es multiplicando una matriz, llamada *matriz de verificación de paridad* por el vector

columna constituido por la palabra recibida:

$$\begin{pmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \end{pmatrix} = (011)$$

el resultado de la multiplicación es 011, que es un 3, la posición del bit erróneo, escrito en binario. Así que el receptor puede ahora modificar el tercer bit de la palabra recibida 0100011 y obtener 0110011 que es la palabra que realmente le fue enviada.

Ahora bien, supongamos que tenemos una gran cantidad de datos codificados usando el esquema de Hamming de 7 bits. Si colocamos cada palabra en un byte estaremos desperdiciando un bit por cada palabra, no es mucho, pero si se tienen 2 millones de palabras, el monto de espacio desperdiciado ya asciende a 244 KB. lo que pudiera no ser despreciable. Así que nos interesa almacenar el arreglo de palabras de Hamming de la manera más compacta posible, una tras de otra consecutivamente sin espacios vacíos entre ellas. Por supuesto nuestro arreglo debe de ser de elementos de algun tipo, digamos bytes, para aproximarnos lo mejor posible. Ahora el problema es que, dado el índice de una palabra de Hamming en el arreglo, debemos calcular el índice del byte donde comienza dicha palabra, además de determinar cuantos bits de ese byte son ocupados por ella y cuantos del siguiente en caso de haberlos. En la tabla 2.2 se muestran estos datos para las primeras posiciones del arreglo.

Por supuesto notamos que la i-ésima palabra de hamming guardada en el arreglo de bytes se encuentra en la posición que resulta de dividir $7 \cdot i$ (segunda columna de la tabla) entre 8. Es decir, el índice del byte donde comienza la palabra i es:

$$\left\lfloor \frac{7i}{8} \right\rfloor$$

Dentro de este byte, el índice del bit (el bit cero es el más significativo) donde comienza la palabra es:

$$7i \pmod 8$$

Así que hay exactamente:

$$8 - [7i \pmod 8]$$

bits de ese byte ocupados por la palabra i-ésima y los restantes:

$$7 - (8 - [7i \pmod 8]) = 7i \pmod 8 - 1$$

estan en el byte de índice

$$\left\lfloor \frac{7i}{8} \right\rfloor + 1$$

i	**bit**	**byte**	**bit en el byte**
0	0	0	0
1	7	0	7
2	14	1	6
3	21	2	5
4	28	3	4
5	35	4	3
6	42	5	2
7	49	6	1
8	56	7	0

Tabla 2.2: Posiciones en un arreglo de bytes de las palabras de Hamming de 7 bits.

Esto ocurre sólo si $7i \pmod 8 > 1$ es decir si la palabra de 7 bits comienza después del segundo bit.

Una vez calculadas las posiciones de bytes y bits basta con filtrar los bits necesarios de los bytes del arreglo y luego pegarlos, de ser necesario, para formarla palabra de código de Hamming.

$\Diamond$

En general hay que hacer esencialmente lo mismo para datos de otros tamaños. Sean:

arr El arreglo empacado de palabras binarias.

idx El índice de la palabra deseada en el arreglo empacado.

longpal La longitud en bits de las palabras.

numbits Número de bits en el arreglo previos al primero asociado a la palabra de índice *idx*.

idxbyte Índice del byte del arreglo donde comienza la palabra deseada.

leidos El número de bits leidos de la palabra deseada.

res Palabra binaria donde se copiará la deseada, inicialmente todos los bits de *res* están en cero.

res.setBit(i) Método que pone en 1 el bit de índice i de la palabra binaria *res*.

E_i El i-ésimo bit del byte E.

El procedimiento es:

```
getPalabra(idx)
 1  numbits ← idx * longpal
 2  idxbyte ← ⌊numbits/8⌋
 3  idxbit ← numbits (mod 8)
 4  leidos ← 0
 5  while leidos < longpal do
 6      while (leidos < longpal) ∧ (idxbit ≠ 8) do
 7          if arr[idxbyte]_idxbit = 1 then
 8              res.setBit(longpal − leidos − 1)
 9          endif
10          leidos ← leidos + 1
11          idxbit ← idxbit + 1
12      endwhile
13      idxbyte ← idxbyte + 1
14  endwhile
15  return res
16  end
```

Siempre hay que calcular en número de bits p, previos a la palabra deseada, luego el cociente $p/8$ es el índice del byte donde dicha palabra comienza y $p \pmod 8$ es el índice del primer bit de ese byte dedicado a la palabra deseada.

Paréntesis: Vectores de Iliffe.

En 1956 un grupo de profesores de la Universidad de Rice en Texas decidieron que necesitaban una computadora electrónica [2]. Para financiar el proyecto apelaron a la Comisión de Energía Atómica, que recientemente que había financiado el "cerebro electrónico" (como solían llamarse las computadoras entonces) del laboratorio de Los Alamos. Así en 1957 inició el proyecto de construcción de la R1 o *The Rice University Computer*, puesta en operación en 1959. El hardware de la máquina fue diseñado mayormente por Martin Graham y el software del sistema por John K. Iliffe. Una parte importante del software lo constituía un sistema llamado *Genie*, una especie de compilador encargado de la codificación de expresiones simbólicas y de definir estructuras de datos en memoria. Con el propósito de independizar, en la medida de lo posible, la manipulación de los arreglos del tamaño que estos tenían ([1], pag. 25) Iliffe decidió implementarlos en Genie como una cascada de referencias, tal cómo los describimos a propósito de languajes como Java y C++, es decir, como arreglos de arreglos. De allí que a ésta técnica se le denomine *vectores de Iliffe*. En general, si en la arquitectura de la máquina es más costoso multiplicar que el manejar múltiples indirecciones y la memoria no es un recurso crítico, los vectores de Iliffe son más eficientes, además de ofrecer la flexibilidad de diferir el alojamiento de memoria.

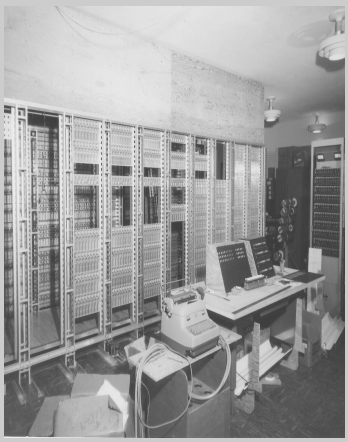

Figura 2.5: Rice R1

Referencias

[1] Iliffe, John K., "The Use of The Genie System in Numerical Calculations". *Annual Review in Automatic Programming*, Vol. 2, 1961, pp. 1–28.

[2] Thornton, Adam, *A Brief History of the Rice Computer 1959-1971* `http://www.cs.rice.edu/History/R1/`

Ejercicios

2.1 Suponga que desea almacenar eficientemente una matriz triangular inferior (incluyendo la diagonal) en la que, además, todas las entradas de los renglones de índice impar son cero. Obtenga el polinomio de direccionamiento para almacenar esta matriz en un arreglo unidimensional. (Ayuda: la suma de los primeros k números naturales impares es k^2).

2.2 Si se pretende almacenar en memoria un arreglo k dimensional de dimensiones $\{n_1, n_2, \ldots n_k\}$. ¿Qué cantidad de memoria se requiere para hacerlo usando *vectores de Iliffe*, o sea arreglos de arreglos de dimensión inferior?, ¿cuanta si se almacenan en un sólo bloque accesible por un polinomio de direccionamiento?, ¿cuál es el costo de hacer un "barrido" completo de los elementos del arreglo en términos de indirecciones con vectores de Iliffe y cuál el costo en multiplicaciones en el caso del polinomio de direccionamiento?, ¿cuando conviene usar cada alternativa?.

Recurrencia 3

Yo soy quien soy, / Y no me parezco a nairen, / Me cuadra el campo, / Y el chifilo de sus aigres. / Mis compañeros, / Son mis buenos animales, / Chivos y mulas, / Y uno que otro viejo buey.

—Yo soy quien soy, canción de Manuél Esperón y Felipe Bermejo, 1955.

3.1 Introducción

Este capítulo inicia con la cita de una canción que en 1956, en el papel de Pablo Saldaña, cantaba Pedro Infante, en la película de Julián Soler *La Tercera Palabra*. Analizemos un poco el primer verso. Para empezar Pedro dice: *yo soy*, es decir se está autodefiniendo y termina con: *quien soy*, esto significa que para encontrar la acepción a lo que se pretende definir hace referencia a lo que definido. En síntesis, Pedro se define a sí mismo como él mismo, añadiendo que no hay nadie más que se le parezca. Tal como está la definción no parece muy útil. Sin embargo, si añadimos algunos requisitos a una definción similar, que recurre al mismo objeto que pretende definir, el resultado es una de las técnicas más poderosas en matemáticas y computación. En matemáticas se le suele llamar inducción, en computación *recurrencia*

Principio de inducción.

La recurrencia es una técnica de programación que permite implantar funciones inductivas. Esto tiene que ver con el principio de inducción: *si se establece una propo-*

sición que es válida para el primer elemento de un conjunto numerable y si, suponiendo que esta proposición es válida para el n-ésimo elemento del conjunto, se puede deducir que es válida también para el sucesor de él, entonces se puede afirmar que es válida para cualquier elemento del conjunto.

Un ejemplo, quizá el más usual, de una definición inductiva es la del factorial de un número.

$$fact(n) = \begin{cases} 1 & si \quad n = 0 \\ n\ fact(n-1) & si \quad n > 0 \end{cases} \tag{3.1}$$

Nótese que en la definición se tiene un *caso base*, a saber, cuando $n = 0$ entonces el factorial es 1 y para definir el factorial de un número cualquiera n mayor que cero entonces, asumiendo que se tiene el cálculo de la función para un entero $n - 1$, se dice cuanto vale para el sucesor de dicho número (n).

Algoritmo recurrente.

En computación se habla de algoritmos recurrentes para referirse a aquellos que implantan el cálculo de funciones definidas de manera inductiva. El rasgo característico de un algoritmo recurrente es que en alguno de sus pasos hace referencia a sí mismo. En el caso de algoritmos recurrentes, si siempre se ejecutan todos los pasos del algoritmo y al menos uno de ellos es una referencia a la ejecución del algoritmo mismo, ¿que asegura que este no se la pasará llamándose a sí mismo infinitas veces y por tanto nunca termina?. Evidentemente no siempre se ejecutan linealmente todos los pasos del algoritmo sino que se debe tener un cambio en el control de flujo del algoritmo de tal manera que en ocasiones se haga una serie de cosas y en otros casos se haga otra diferente. Uno de estos casos debe asegurar que el algoritmo terminará, esta es la *condición de terminabilidad* del algoritmo y no es otra cosa que la consideración del *caso base* mencionado antes.

EJEMPLO 3.1

Factorial

En la ecuación 3.1 se muestra la definición del factorial de un número natural. El código en Java mostrado en la figura 3.1 implanta esta definición en una función recurrente.

Nótese que se hace la pregunta ¿es `num` igual a 0? en cuyo caso la función regresa un 1. Este es al caso base de la definición inductiva de factorial y es la condición de terminabilidad de la función recurrente. Luego se procede a considerar el caso en el que `num` es distinto de 0 y se procede a multiplicar el valor del parámetro actual, `num`, por el resultado de evaluar la función factorial (haciendo referencia a ella misma) con el parámetro `num-1`. Si se ejecuta esta función con un valor inicial de `num`, por ejemplo 3, el paso por la llamada recurrente asegura que esta valor se irá decrementando en cada nueva llamada hasta que, finalmente, llegue a ser 0, en ese momento se hace una

```
1     public static long factorial(int num) {
2        if (num == 0) {    // si se pide el factorial de cero
3           return (1);  // se regresa un 1
4        }
5        else {
6           // si no, el producto de num por el factorial de num-1
7           return num * factorial(num - 1);
8        }
9     }
```

Listado 3.1: Método recurrente para el cálculo del factorial.

última llamada a factorial que regresa inmediatamente un 1 (se cumple la condición de terminabilidad).

$\Diamond$

EJEMPLO 3.2

Fibonacci

El k-ésimo término de la sucesión de Fibonacci se define como sigue:

$$fib(k) = \begin{cases} 1 & si \quad k = 1 \ o \ k = 2 \\ fib(k-1) + fib(k-2) & si \quad k > 2 \end{cases} \tag{3.2}$$

Con base en la definición 3.2 se elaboró en Java la función mostrada en la figura 3.2, la cuál calcula el `num`-ésimo término de la sucesión.

La complejidad del cálculo hecho con base en esta definición está determinado entonces por la obvia relación de recurrencia:

$$T(n) = \begin{cases} 1 & si \quad n = 1 \ o \ n = 2 \\ T(n-1) + T(n-2) & si \quad n > 2 \end{cases} \tag{3.3}$$

Considerando que $T(n-2) \leq T(n-1)$ entonces:

$$T(n) \leq 2T(n-1)$$

Iterando el proceso:

$$\begin{aligned} T(n) &\leq 2T(n-1) \\ &= 4T(n-2) \\ &\vdots \quad \vdots \\ &= 2^n T(1) = 2^n \end{aligned}$$

```
1       /**
2        * Función recurrente para calcular el término "num"-ésimo de
3        * la sucesión de Fibonacci.
4        * @param num es el número de término requerido
5        * @return el valor del término solicitado de la sucesión
6        * de Fibonacci.
7        */
8       public static int fibonacci(int num) {
9           // si se piden el termino 1 o 2
10          if (num < 3) {
11              return (1); // se regresa un 1
12          }
13          else {
14              // si no se regresa el resultado de sumar
15              // los dos términos anteriores de la sucesión
16              return fibonacci(num - 1) + fibonacci(num - 2);
17          }
18      }
```

Listado 3.2: Método recurrente para el cálculo del *num*-ésimo término de la sucesión de Fibonacci.

Así que $T(n) = O(2^n)$, ¡Una complejidad exponencial!

Pero esta no es la única manera de calcular los términos de la sucesión. Es posible elaborar una versión no recurrente de la misma, tal y como aparece en el listado 3.3. A esta versión le denominamos iterativa dado que reemplaza la recursión por la iteración de un ciclo que va sumando los dos últimos términos de la sucesión y luego reasigna valores para calcular el término siguiente.

La versión iterativa de `fibonacci` posee una complejidad $O(n)$, se pasa una y sólo una vez por cada índice del ciclo y en cada iteración del mismo se calcula un término. ¿Cual es el problema con la versión recurrente? La respuesta es evidente al observar la figura 3.1, en ella se muestran las llamadas recurrentes necesarias para calular el sexto término de la sucesión. Podemos ver, por ejemplo que se hacen dos llamadas diferentes para calcular $fib(4)$, tres para calcular $fib(3)$, cinco para $fib(2)$ y cinco para $fib(1)$. En esencia deberían bastar con las que están presentes en la rama del extremo izquierdo del árbol, pero ocurre que el procedimiento recursivo olvida lo que ya ha calculado y lo recalcula. Podríamos decir de hecho, que el procedimiento, por el simple hecho de ser recurrente, olvida una cantidad exponencial de términos previamente calculados, tantos más cuanto mayor sea el término que se requiere. No siempre la solución recurrente, aunque resulte natural, es la más eficiente, más adelante veremos cómo *siempre* es posible transformar un algoritmo recurrente en uno que no lo es.

$\Diamond$

```java
/**
 * Función iterativa para calcular el término "num"-ésimo de
 * la sucesión de Fibonacci.
 * @param num es el número de término requerido
 * @return el valor del término solicitado de la sucesión
 * de Fibonacci.
 */
public static int fibonacci(int num) {
    // si el término solicitado es
    // el primero o el segundo
    if (num < 3) {
        return 1;
    }
    int ultimo = 1;
    int penultimo = 1;
    int suma = 0;
    int j;
    // se suman el penúltimo y el último
    for (j = 3; j <= num; j++) {
        suma = ultimo + penultimo;
        penultimo = ultimo; // actualizandolos en
        ultimo = suma; // cada iteración
    }
    return suma;
}
```

Listado 3.3: Método iterativo para el cálculo del `num`-ésimo término de la sucesión de Fibonacci.

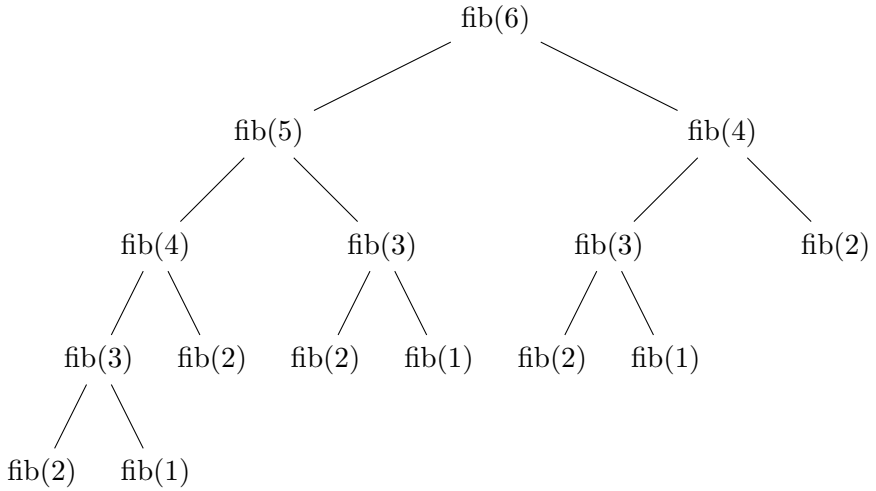

Figura 3.1: Árbol de llamadas recurrentes para calcular el sexto término de la sucesión de Fibonacci.

EJEMPLO 3.3

Las torres de Hanoi

Cuenta la leyenda[1] que bajo la gran cúpula colocada en el centro del mundo, en el templo de Benarés, en India, hay una placa de latón en la que se han fijado tres agujas idénticas, cada una de unos 50 cm. de largo. Ensartados en una de estas agujas, Dios colocó, el día de la creación, sesenta y cuatro discos de oro macizo horadados en su centro para dar lugar a la aguja. El disco que se encuentra apoyado en la placa de latón es el más grande de todos, el diámetro de los discos va disminuyendo conforme se acercan a la parte superior. Esta es la torre de Brahma. Día y noche, el sacerdote en turno transfiere los discos de una aguja a otra de acuerdo a las inmutables leyes de Brahma:

1. Solo un disco puede moverse a la vez.
2. Nunca debe colocarse un disco mayor sobre uno más pequeño.

Cuando se hayan transferido los sesenta y cuatro discos de la aguja donde los colocó Dios a alguna de las otras dos, la torre, el templo y los brahamanes se convertirán en polvo y el mundo entero desaparecerá en medio de un trueno.

Este es uno de los problemas más mencionados a propósito de funciones recurrentes. Se tienen tres varillas de igual tamaño, en una de esas varillas a la que llamaremos *origen* se encuentra un cierto número de discos horadados (n) insertos en la varilla. Los discos son de diferente tamaño y están ordenados del piso hacia arriba de mayor a menor diámetro. El problema consiste en pasar todos los discos de la varilla origen a otra, a la que llamaremos *destino*, usando la otra como medio de almacenamiento temporal (a esta le llamaremos varilla *auxiliar*), acatando siempre las dos leyes de Brahma mencionadas arriba.

La solución a este problema es en esencia la siguiente: Para pasar n discos del disco origen al destino usando aux

1. si n es cero ya terminamos y si no
2. Movemos $n-1$ discos del origen al aux
3. Movemos el disco siguiente del origen al destino
4. Movemos los restantes $n-1$ discos del aux al destino

Esta solución ha sido implantada en el código mostrado en la figura 3.4

Existe también un algoritmo no recurrente para resolver el problema. Es conocido como el algoritmo de Buneman-Levy, quienes lo formularon en 1980. Supóngase que

[1] Atribuida al escritor inglés W. Rouse-Ball.

```
1       /**
2        * Función recurrente para resolver el problema de las
3        * torres de Hanoi.
4        * Esta función indica todos los movimientos que hay que
5        * hacer para transferir <code>ndisc</code> discos
6        * ordenados por tamaño, de la torre origen a la destino
7        * utilizando temporal mente la torre aux.
8        * @param ndisc   número de discos a mover
9        * @param origen  cadena que identifica la varilla origen
10       * @param destino cadena que identifica la varilla
11       *                destino
12       * @param aux     cadena que identifica la varilla
13       *                auxiliar
14       * @return el número de movimientos realizados.
15       */
16      public static int hanoi(int ndisc, String origen,
17                              String destino, String aux) {
18          int mov_ida;
19          int mov_reg;
20
21          // si hay discos que mover "n"
22          if (ndisc > 0) {
23              // movemos los n-1 de arriba al aux
24              mov_ida = hanoi(ndisc - 1, origen, aux, destino);
25
26              // luego el de hasta abajo del origen al destino
27              System.out.print("Mueve disco " + ndisc);
28              System.out.print(" de " + origen);
29              System.out.println(" a " + destino);
30
31              // y por ultimo los n-1 del aux al destino
32              mov_reg = hanoi(ndisc - 1, aux, destino, origen);
33
34              return mov_ida + mov_reg + 1;
35          }
36          return 0;
37      }
```

Listado 3.4: Método recurrente para resolver el problema de las torres de Hanoi.

los discos están ordenados correctamente en la varilla del extremo izquierdo y que se desean pasar todos a la del extremo derecho. Numeremos entonces las varillas como 0 (la de la izquierda, donde están los discos al inicio), 1 (la auxiliar de enmedio) y 2 (la del extremo derecho, la varilla destino). El algoritmo se basa en la observación de que el disco más pequeño de todos debe moverse siempre en los movimientos impares y luego de eso hay siempre otro disco (que por supuesto es más grande) que puede moverse a un único sitio. Formalmente:

1. Mover el disco más pequeño P de la varilla i a la $i+1 \pmod 3$.
2. Mover el disco expuesto más pequeño distinto de P, al único lugar disponible para ello.
3. Repetir 1 y 2 hasta que todos los discos estén en la varilla destino (la 2).

Tanto la complejidad del algoritmo recurrente como la del iterativo debe medirse, claro está, por el número de movimientos de discos que se deben hacer. En el caso del primero ésta está determinada por la recurrencia: $T(n) = 2T(n-1) + 1$. Es decir:

$$\begin{aligned} T(n) &= 2T(n-1)+1 \\ &= 4T(n-2)+3 \\ &= 8T(n-3)+7 \\ &\vdots \quad \vdots \\ &= 2^k T(n-k) + 2^k - 1 \\ &= 2^k(T(n-k)+1) - 1 \end{aligned} \tag{3.4}$$

cuando $n = 1$ se tiene que $T(1) = 1$ y cuando $k = n-1$, la expresión 3.4 se convierte en: $T(n) = 2^{n-1}(T(1)+1) - 1 = 2^n - 1$. Así que $T(n) = O(2^n)$. Esta es, por cierto la misma complejidad del algoritmo no recurrente y de hecho la complejidad del problema.

Si se tienen $n = 1$ discos, es claro que el número de movimentos requerido es $1 = 2^1 - 1$. Si suponemos que para $n = k-1$ discos se requieren $2^{k-1} - 1$ movimientos, veamos cuantos son necesarios para $n = k$. Para mover los k discos del origen al destino debemos, poder pasar el disco más grande, para descubrirlo hay que quitarle de encima los $k-1$ discos menores que él y dejar disponible la varilla destino, así que debemos pasar los $k-1$ discos a la auxiliar, eso, por hipótesis de inducción nos toma $2^{k-1} - 1$ movimientos. A esto hay que añadir el movimiento del disco mayor de origen a destino y después hay que pasar los $k-1$ restantes del auxiliar al destino, lo que nos lleva, otra vez, $2^{k-1}-1$ movimientos. En total hicimos $2(2^{k-1}-1)+1 = 2^k-2+1 = 2^k-1$ movimientos. Lo que queríamos demostrar. Claro que no es fortuito que lo hayamos desmostrado por inducción, después de todo es lo que nos dicta nuestro algoritmo recurrente, pero ahora sabemos que no es posible resolver el problema en menos movimientos. Eso es siempre algo que se anhela en computación, determinar la complejidad *del problema*, la que le es inherente, la que le correspondería al mejor de los algoritmos que lo resuelve. Cuando

se descubre algo así, se ha determinado algo que tiene que ver con la estructura misma del espacio de búsqueda de las soluciones.

◊

3.2 Retroceso mínimo

Imaginemos que estamos atrapados en un laberinto, condenados por el malvado rey de Creta a deambular el resto de nuestras vidas entre los interminables pasillos. Luego de un tiempo, nuestras mentes de computólogos elaboran un esquema para recorrer metódicamente el laberinto: caminamos, en cuanto llegamos a un sitio del que salen varios caminos alternativos, elegimos uno, digamos el primero a la derecha, ponemos una marca en el muro y seguimos caminando; si en algún momento llegamos a un callejón sin salida o nos topamos con una marca dejada antes por nosotros mismos, damos la vuelta y desandamos el camino hasta llegar al lugar donde tomamos la decisión más reciente y elegimos ahora otra alternativa.

Este esquema no nos garantiza que no tengamos que recorrer todo el laberinto para encontrar la salida, pero ciertamente nos permite no pasar dos veces por el mismo camino. Si suponemos además que el camino recorrido hasta cierto momento es parte del camino que hay que seguir para salir, nuestros retrocesos nos permiten deshacer lo mínimo indispensable para buscar un camino alternativo, conservamos siempre lo más posible de la propuesta de solución que hemos construido hasta el momento. Bien pudiera ser que en algún momento tengamos que retroceder tantas veces que regresemos al punto de partida, en efecto, pero nuestro pensamiento optimista nos dice que eso no será lo común y si llegara a ocurrir, en todo caso el recorrido hecho no ha sido en vano, hemos descartado una buena parte de posibles caminos que ya nunca volveremos a considerar como posibles soluciones.

El esquema de exploración descrito arriba tiene nombre propio en la computación: retroceso mínimo o bien, en inglés *backtrack*.

A grandes rasgos el mecanismo consiste en lo siguiente:

1. Se divide el problema en etapas similares que hay que resolver para llegar a la solución final.

2. Se procede a resolver la primera etapa y luego cada una de las siguientes tomando en consideración lo ya hecho.

3. Si en alguna etapa no hay solución posible entonces se debe retroceder a la etapa inmediata anterior y buscar otra posible solución en ella para partir de una premisa diferente.

EJEMPLO 3.4

Supongamos que tenemos un tablero de ajedrez y que en él deseamos colocar exactamente ocho reinas, sin que puedan atacarse mutuamente bajo las reglas comunes de movimiento y ataque para esta pieza del ajedrez: la reina se puede mover en linea recta horizontal, vertical o diagonalmente (en cualquiera de las dos diagonales), un número arbitrario de casillas hasta encontrar el borde del tablero o toparse con alguna otra pieza, en cuyo caso la reina toma el lugar que esta ocupaba y la elimina del tablero.

Como son justamente ocho reinas y siempre que se colocan dos de ellas en la misma columna o renglón, se atacan, es evidente que debe colocarse exactamente una sola reina por columna. Así que lo único que necesitamos saber para tener completamente determinada una configuración del tablero es en que renglón quedó la reina de la columna i. Entonces, en vez de utilizar la representación más obvia del tablero, con un arreglo bidimensional de 64 casillas, podemos hacer todo con un solo arreglo unidimensional de ocho posiciones (0, 1, ..., 7), donde en la entrada i del arreglo guardamos el número de renglón donde quedo la i-ésima reina (es decir la de la columna i). Y para decidir si una posición propuesta para una reina es atacada o no por otras reinas, podemos utilizar el hecho de que en toda diagonal positiva la suma de los índices del renglón y la columna permanecen constantes para cualquier elemento y que para toda diagonal negativa lo que permanece constante es la diferencia del índice de la columna menos el del renglón. Esto puede verse en las tablas 3.1 y 3.2. Es posible entonces representar a cada diagonal positiva y negativa como una entrada en el arreglo respectivo, solo son necesarias 15 entradas para guardar todas las diagonales positivas y 15 para las negativas, además necesitaremos un arreglo de 8 entradas para representar los renglones. Ahora cada vez que es colocada una reina en la posición (ren, col) basta con poner un uno en las entradas que corresponden a la diagonales positiva $(ren + col)$ y negativa $(col - ren + 7)$ donde está contenida la casilla (ren, col) y poner un uno también en la posición ren del arreglo de renglones. Verificar si una casilla es atacada o no por otras reinas consiste en verificar si los arreglos de diagonales y el de renglones tienen ceros o unos en ciertas posiciones bien definidas en vez del engorroso procedimiento que tendríamos que llevar a cabo si desearamos verificar si una casilla es atacada explorando, casilla por casilla, todas las lineas del tablero que concurren en ella.

Lo que hemos logrado con este análisis es reducir el monto de memoria necesario para almacenar los datos necesarios (en vez de un tablero de 64 casillas representamos el tablero con cuatro arreglos, dos de 30 casillas y dos de 8 lo que hace un total de 46 casillas).

Además nuestro análisis ha hecho posible que hagamos la función `ataca` como es mostrada en la fig. 3.7 y que no es, ciertamente, el procedimiento que a uno le viene a la mente en primera instancia, sino uno mucho más breve. Nuestro análisis nos

	0	1	2	3	4	5	6	7
0	0+0=0	0+1=1	0+2=2	0+3=3	0+4=4	0+5=5	0+6=6	0+7=7
1	1+0=1	1+1=2	1+2=3	1+3=4	1+4=5	1+5=6	1+6=7	1+7=8
2	2+0=2	2+1=3	2+2=4	2+3=5	2+4=6	2+5=7	2+6=8	2+7=9
3	3+0=3	3+1=4	3+2=5	3+3=6	3+4=7	3+5=8	3+6=9	3+7=10
4	4+0=4	4+1=5	4+2=6	4+3=7	4+4=8	4+5=9	4+6=10	4+7=11
5	5+0=5	5+1=6	5+2=7	5+3=8	5+4=9	5+5=10	5+6=11	5+7=12
6	6+0=6	6+1=7	6+2=8	6+3=9	6+4=10	6+5=11	6+6=12	6+7=13
7	7+0=7	7+1=8	7+2=9	7+3=10	7+4=11	7+5=12	7+6=13	7+7=14

Tabla 3.1: Diagonales positivas (de pendiente positiva), de un tablero de ajedrez. En cada diagonal es constante la suma columna+renglón.

ha permitido también hacer más eficiente, con menos instrucciones, el algoritmo de verificación de ataque. Hemos reducido la *complejidad del algoritmo*.

El algoritmo 3.1 es el que hemos seguido para encontrar una solución al problema de las ocho reinas.

$\Diamond$

	0	1	2	3	4	5	6	7
0	0-0=0	1-0=1	2-0=2	3-0=3	4-0=4	5-0=5	6-0=6	7-0=7
1	0-1=-1	1-1=0	2-1=1	3-1=2	4-1=3	5-1=4	6-1=5	7-1=6
2	0-2=-2	1-2=-1	2-2=0	3-2=1	4-2=2	5-2=3	6-2=4	7-2=5
3	0-3=-3	1-3=-2	2-3=-1	3-3=0	4-3=1	5-3=2	6-3=3	7-3=4
4	0-4=-4	1-4=-3	2-4=-2	3-4=-1	4-4=0	5-4=1	6-4=2	7-4=3
5	0-5=-5	1-5=-4	2-5=-3	3-5=-2	4-5=-1	5-5=0	6-5=1	7-5=2
6	0-6=-6	1-6=-5	2-6=-4	3-6=-3	4-6=-2	5-6=-1	6-6=0	7-6=1
7	0-7=-7	1-7=-6	2-7=-5	3-7=-4	4-7=-3	5-7=-2	6-7=-1	7-7=0

Tabla 3.2: Diagonales negativas (de pendiente negativa), de un tablero de ajedrez. En cada diagonal es constante la diferencia columna-renglón.

Algoritmo 3.1 Algoritmo para encontrar una solución al problema de las 8 reinas.

```
ColocaReina(col, tablero, fin)
 1  if col = 8 then
 2      fin ← true
 3  else
 4      fin ← false
 5      ren ← 0
 6      while (ren < 8) ∧ (¬fin) do
 7          if noAtacada(ren, col) then
 8              ponReina(ren.col)
 9              ColocaReina(col + 1, tablero, fin)
10              if ¬fin then
11                  quitaReina(ren, col)
12                  ren ← ren + 1
13              endif
14          else
15              ren ← ren + 1
16          endif
17      endwhile
18  endif
19  end
```

```
1     /**
2      * Variable booleana para determinar si se ha encontrado una
3      * solución (true) o no (false).
4      */
5      static boolean yastuvo = false;

7     /**
8      * Tamaño del tablero, 8 por omisión.
9      */
10     static int TAM_TABLERO;

12    /**
13     * Tamaño del arreglo de diagonales (número de diagonales).
14     * Por omisión 15.
15     * T_DIAG = (2 * TAM_TABLERO) - 1;
16     */
17     static int T_DIAG;

19     /**
20      * Arreglo para guardar el estado de las diagonales con
21      * pendiente positiva.
22      * diagpos[i] = 0 si y sólo si no hay reinas en la i-ésima
23      * diagonal.
24      * Se inicializa con ceros.
25      */
26     static int[] diagpos;

28     /**
29      * Arreglo para guardar el estado de las diagonales con
30      * pendiente negativa.
31      * diagneg[i] = 0 si y sólo si no hay reinas en la i-ésima
32      * diagonal.
33      * Se inicializa con ceros.
34      */
35     static int[] diagneg;

37     /**
38      * Arreglo para guardar el estado de los renglones.
39      * renglon[i] = 0 si no hay reinas en el renglón i-ésimo,
40      * 1  en otro caso.
41      * Se inicializa con ceros.
42      */
43     static int[] renglon;

45     /**
46      * Arreglo para guardar el estado del tablero.
47      * tablero[i] = renglón donde va la reina de la i-ésima
48      * columna. Se inicializa con -1.
49      */
50     static int[] tablero;
```

Listado 3.5: Variables usadas para resolver el problema de las 8 reinas.

```
1       /**
2        * Método recurrente para resolver el problema de las 8
3        * reinas haciendo "backtrack".
4        * @param columna es el índice de la columna en la que
5        * debe colocarse la siguiente reina.
6        */
7       public static void coloca_reina(int columna) {
8           int renglon;

10          if (columna == TAM_TABLERO) { // si ya se colocaron todas
11              yastuvo = true; // reinas se termina
12          }
13          else {
14              yastuvo = false; // si aún hay reinas por colocar

16              // se prueba en el primer renglón (renglón cero)
17              renglon = 0;

19              // Mientras no se terminen los renglones y no
20              // se haya encontrado solución
21              while ((renglon < TAM_TABLERO) && !yastuvo) {
22                  // si es atacada
23                  if (ataca(renglon, columna)) {
24                      renglon++; // se prueba en el sig. renglón
25                  }
26                  else {
27                      // si no es atacada, se coloca en ese renglón
28                      ponreina(renglon, columna);

30                      // y se trata de encontrar posiciones seguras
31                      // para las restantes
32                      coloca_reina(columna + 1);

34                      if (!yastuvo) { // si no hay solución aún

36                          // cambiar la propuesta:
37                          // se quita la reina del renglón seguro
38                          quitareina(renglon, columna);

40                          // y se prueba en el siguiente
41                          renglon++;
42                      }
43                  } // else
44              } // while
45          }
46      }
```

Listado 3.6: Código para resolver el problema de las 8 reinas.

```
1       /**
2        * Determina si una posición del tablero es atacada o no.
3        * @param ren índice del renglón del tablero.
4        * @param col índice de la columna del tablero.
5        * @return <code>true</code> si la posición dada como
6        * entrada es atacada, <code>false</code> en otro caso.
7        */
8       public static boolean ataca(int ren, int col) {
9           int resul;
10          int i;
11          int j;

13          resul = 0;
14          i = ren + col; // índice de la diagonal positiva
15          j = col - ren + (TAM_TABLERO - 1); // diagonal negativa

17          resul = diagpos[i] + diagneg[j] + renglon[ren];

19          return (resul != 0); // si todos 0 entonces falso
20      }
```

Listado 3.7: Determina si una reina es atacada en una posición dada.

```
1       /**
2        * Coloca una reina en una posición del tablero.
3        * @param ren renglón del tablero en el que se coloca la
4        * reina.
5        * @param col columna del tablero en la que se coloca la
6        * reina.
7        */
8       public static void ponreina(int ren, int col) {
9           int i;
10          int j;

12          i = ren + col; // índice de la diagonal positiva

14          // donde se encuentra la casilla (ren, col)
15          j = col - ren + (TAM_TABLERO - 1); // diagonal negativa

17          diagpos[i] = 1; // se marcan las diagonales ocupadas
18          diagneg[j] = 1;
19          renglon[ren] = 1; // y el renglón
20          tablero[col] = ren; // y se coloca la reina
21      }

23      /**
24       * Quita una reina de una posición del tablero.
25       * @param ren renglón del tablero en el que se quita la
26       * reina.
27       * @param col columna del tablero en la que se quita la
28       * reina.
29       */
30      public static void quitareina(int ren, int col) {
31          int i;
32          int j;

34          i = ren + col;
35          j = col - ren + (TAM_TABLERO - 1);

37          diagpos[i] = 0; // se marcan las diagonales desocupadas
38          diagneg[j] = 0;
39          renglon[ren] = 0; // al igual que el renglón
40          tablero[col] = -1; // y se quita la reina
41      }
```

Listado 3.8: Métodos auxiliares para el problema de las 8 reinas.

Paréntesis: Recursión y computabilidad.

En 1934, en un seminario que tenía lugar en el Institute for Advanced Studies en Princeton, Kurt Gödel habló acerca de lo que el llamó *funciones recurrentes* (*rekursiv*), como un modelo que permitiría definir las funciones calculables. La idea había surgido en un intercambio de correspondencia entre Gödel y el matemático francés Jacques Herbrand y luego de ser expuesta en el seminario fué retomada por uno de los asistentes: Stephen Cole Klenne (véase fig. 3.2), en ese entonces alumno de doctorado de Alonzo Church.

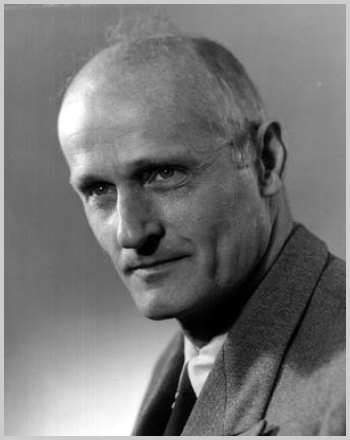

Figura 3.2: Stephen Kleene.

Kleene renombró las funciones de Gödel como *recurrentes primitivas* y les dio un tratamiento axiomático similar al que Peano le dio a los números naturales: la función constante cero es recurrente primitiva así como la función sucesor; se añade además una función que selecciona uno y sólo uno de todos sus argumentos (proyección) y se especifica que la composición y la recursión de funciones primitivas recurrentes da lugar a funciones del mismo tipo [1, 5, 3]. Las definiciones de suma y producto que formulamos anteriormente con base en los axiomas de Peano son, esencialmente, las definiciones que tendrían en el contexto de las funciones recurrentes primitivas. El conjunto de funciones recurrentes primitivas puede ser extendido [2, 3], añadiendo una operacion más denominada *minimización* u *operador de búsqueda*, dando lugar al conjunto de las *funciones recurrentes* y que, en el marco de la teoría de Kleene resultan ser las funciones computables [4]. En 1937, un año después de la formulación de Kleene, Alan Turing demostró que este concepto de computabilidad es equivalente al que él mismo formuló también en 1936 [6, 3].

Referencias

[1] "Recursive function", en *Wikipedia*, Wikimedia Foundation Inc.
`http://en.wikipedia.org/wiki/Recursive_function`

[2] "Primitive recursive function", en *Wikipedia*, Wikimedia Foundation Inc.
`http://en.wikipedia.org/wiki/Primitive_recursive_function`

[3] Boolos, G.S. y R.C. Jefferey, *Computability and Logic*, 3a Ed., Cambridge University Press, 1999.

[4] Kleene, S.C., "Origins of Recursive Function Theory", *Annals of the History of Computing*, IEEE, Vol. 3, No. 1, enero 1981.

[5] Kleene, S.C., "Turing's Analysis of Computability, and Major Applications of it", en R. Herken (editor), *The Universal Turing Machine, A Half-Centory Survey*, 2a Ed., Springer Verlag, 1995.

[6] Gandy, R., "The Confluence of Ideas in 1936", en R. Herken (editor), *The Universal Turing Machine, A Half-Centory Survey*, 2a Ed., Springer Verlag, 1995.

Ejercicios

3.1 Formule un algoritmo recurrente con retroceso mínimo para encontrar la salida de un laberinto.

3.2 Programe el algoritmo de Buneman-Levy para resolver el problema de las torres de Hanoi. Compare los tiempos de ejecución promedio para n en un cierto rango. Discuta los resultados con sus compañeros.

3.3 Averigue que es *recurrencia de cola* (*tail recursion*). Entre los ejemplos analizados en este capítulo ¿hay alguno que corresponda con la definición?, ¿cómo optimizan los compiladores estos casos?, ¿por qué se puede hacer esta optimización?

3.4 El algoritmo 3.1 permite encontrar sólo una solución al problema de las 8 reinas. Formule un algoritmo para encontrar todas las soluciones, para hacerlo simple no haga consideraciones de simetría (es decir considere dos configuraciones como diferentes aún cuando una de ellas sea resultado de rotar o reflejar la otra). ¿Cuantas soluciones obtuvo?, si en vez de ocho reinas se usan n en un tablero de $n \times n$ escaques, ¿cuál es la complejidad del problema?