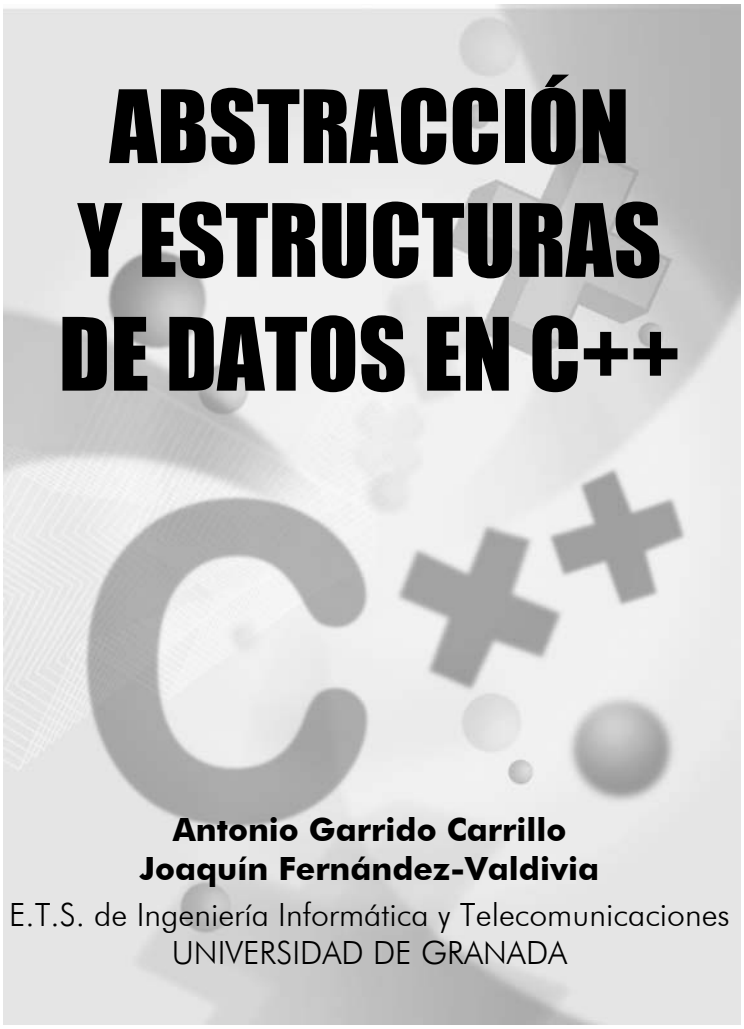
DELTA PUBLICACIONES

ABSTRACCIÓN Y ESTRUCTURAS DE DATOS EN C++

Antonio Garrido Carrillo
Joaquín Fernández Valdivia

ABSTRACCIÓN
Y ESTRUCTURAS
DE DATOS EN C++

ABSTRACCIÓN Y ESTRUCTURAS DE DATOS EN C++



Antonio Garrido Carrillo
Joaquín Fernández-Valdivia
E.T.S. de Ingeniería Informática y Telecomunicaciones
UNIVERSIDAD DE GRANADA

DELTA PUBLICACIONES

ABSTRACCIÓN Y ESTRUCTURAS DE DATOS EN C++

ANTONIO GARRIDO CARRILLO
JOAQUÍN FERNÁNDEZ VALDIVIA

Editor gerente Fernando M. García Tomé
Diseño de cubierta Mizar Publicidad, S.L.
Preimpresión Delta Publicaciones, S.L.
Impresión Grefol, S.A.
Pol. Ind. La Fuensanta. Móstoles. Madrid (España)

ISBN 84-96477-26-6
Depósito Legal

(0106-60)

Prólogo

Este libro es el resultado de un esfuerzo que se ha realizado durante años como respuesta al reto de enseñar los conceptos más importantes de programación en unas condiciones realmente exigentes:

- Los alumnos no poseen una gran experiencia en programación.
- Los temas se desarrollan durante un tiempo relativamente corto.
- Se debe mostrar en profundidad el concepto de abstracción así como las estructuras de datos fundamentales.
- Conceptos de bajo nivel resultan muy aconsejables para facilitar otras asignaturas.
- Los lenguajes de programación y la tecnología actual nos obligan a diseñar un curso que incluya los conceptos fundamentales y prepare al alumno para poder profundizar en conceptos más avanzados.

Concretamente, se ha hecho un esfuerzo para impartir la mayor parte de los contenidos de este libro durante un cuatrimestre, suponiendo que los alumnos habían asimilado y tenían una experiencia acumulada de un primer cuatrimestre de fundamentos de programación (véase Garrido[10]).

Teniendo en cuenta que los actuales planes de estudios son muy exigentes en los contenidos, es recomendable que el alumno pueda disponer de un libro donde apoyarse. Sin embargo, aunque en el mercado ya existen libros sobre C++, pueden resultar muy incómodos, puesto que tienden a presentar los contenidos guiados por el lenguaje. Así, cuando se aborda un tema se suele estudiar con todos los detalles, considerando que el lector ya tiene cierta experiencia. Lógicamente, si intentamos mostrar aspectos tan avanzados como las clases, el encapsulamiento o la programación genérica, y el lector es un programador inexperto, es difícil encontrar un libro que permita asimilarlos fácilmente.

Inicialmente, la idea era generar algunos temas puntuales sobre los aspectos más delicados, a fin de que el alumno tuviera un texto de referencia donde consultar sus dudas. Sin embargo, el material se ha ido acumulando y se ha conseguido terminar un curso sobre abstracción y estructuras de datos.

El objetivo de este libro es profundizar en los conceptos fundamentales de programación, conocer las estructuras de datos más importantes, y preparar al lector para profundizar en los paradigmas de la programación más avanzados. Para ello, se muestran los contenidos con una dificultad incremental, de forma justificada y razonada. No sólo se pretende enseñar el lenguaje C++ sino también los conceptos fundamentales que permiten que el lector vaya formándose como programador.

Por otro lado, en el mercado encontramos muchos libros dedicados a la exposición de los detalles de algún lenguaje, además de otros libros que se dedican a mostrar las estructuras de datos más importantes, suponiendo que el lector ya es un programador con cierta experiencia. En este libro, no se han querido separar el estudio del lenguaje, la metodología de la programación y las estructura de datos. Aunque se podrían seleccionar temas que podrían asignarse a una u otra parte, lo cierto es que la separación que encontramos en la bibliografía es algo poco natural, puesto que el desarrollo de programas conlleva la aplicación de todos esos conocimientos de forma conjunta.

Esta es una de las razones por las que se ha discutido tanto acerca de la posibilidad de aprender el lenguaje C++ sin conocimientos previos de programación. Éste integra muchos conceptos avanzados que resultan más difíciles de comprender para una persona sin experiencia.

Este libro pretende ser una aportación para que el lector pueda estudiar este lenguaje, incluso, aunque no tenga mucha experiencia. De esta forma, esperamos que tanto los estudiantes como los lectores autodidactas encuentren en este libro una forma mucho más simple para llegar a ser buenos programadores.

Conocimiento previos para leer este libro

Este libro se ha escrito pensando en programadores con poca experiencia. De hecho, se utiliza como libro de texto para alumnos que sólo acumulan 4 meses de trabajo con el lenguaje C++. Por tanto, suponemos que el lector ya conoce los fundamentos de programación en C++, donde podemos destacar los siguientes conocimientos:

- Programación modular en C++. Incluye la capacidad de desarrollar soluciones con varias funciones y ficheros:
  - Respecto a funciones. Es necesario conocer el concepto de referencia (paso de parámetros y devolución de resultados en funciones), así como otros conceptos más simples como funciones *inline*, o sobrecarga de funciones. Además, debe ser capaz de crear funciones recursivas.
  - Respecto a ficheros. Es necesario conocer la compilación separada, así como el concepto de espacio de nombres, al menos, para usar adecuadamente el espacio *std*.
- Tipos de datos simples y estructurados en C++. Incluye tipos de datos como punteros, vectores o estructuras.

- Memoria dinámica. Incluye el uso de los operadores *new* y *delete*.

Si no tiene ninguna experiencia en programación, puede consultar Garrido[10], referencia básica que usamos para un primer curso en programación estructurada con C++.

Por otro lado, si ya es programador en lenguaje C, le resultará bastante fácil incorporarse a este curso. Para ello, puede consultar los conocimientos adicionales que necesita para comenzarlo. Realmente, sólo tendrá que revisar algunos puntos de los que hemos indicado, entre los que cabe destacar la nueva sintaxis para *new* y *delete*, la forma de usar la entrada y salida estándar (*cin, cout*), el tipo referencia para las funciones, y el concepto de espacio de nombres.

Organización del libro

Este libro presenta una serie de capítulos que, en general, se organizan con un orden de complejidad incremental. A pesar de ello, debido a la variedad de conceptos que se presentan y a que se ha querido minimizar las dependencias entre ellos, podrían estudiarse con algunos cambios en su orden, facilitando de esta forma el uso del libro en distintos temarios. Más concretamente, los contenidos son los siguientes:

En el **capítulo 1** se introduce el concepto de la eficiencia y los métodos básicos para poder calcularla. El objetivo es que el lector sea capaz de comparar los distintos algoritmos que se proponen a lo largo del libro. A pesar de ser el primero, y que constituye una base importante para muchos comentarios posteriores, su uso no se hace fundamental hasta los capítulos dedicados a las estructuras de datos (tema 5 y siguientes).

No se pretende hacer una exposición detallada de los distintos métodos para calcular la eficiencia, sino una introducción que permita al estudiante entender las discusiones que se presentan en los temas siguientes.

En el **capítulo 2** se introduce al lector en los tipos de datos abstractos. La idea es comentar los conceptos fundamentales sin complicar los contenidos con detalles del lenguaje. Así, este tema se desarrolla sin hacer referencia a las clases en C++. Podría corresponder a un tema de tipos de datos abstractos en lenguaje C.

En el **capítulo 3** se presentan las clases en C++, el método más adecuado para desarrollar tipos de datos abstractos en este lenguaje. Cuando el lector ya es consciente de su necesidad, se pueden presentar las herramientas que ofrece el lenguaje C++ de una forma razonada, y así, facilitar su asimilación por parte del estudiante.

Se discuten todos los aspectos fundamentales para desarrollar los primeros tipos de datos abstactos, incluyendo todos los detalles como para que el lector pueda empezar a crear sus propias aplicaciones.

En el **capítulo 4** se presenta la sobrecarga de operadores, que podría considerarse una extensión del capítulo 3, ya que facilita la implementación de algunas operaciones de los tipos de datos. A pesar de ello, no resulta un tema fundamental para entender la abstracción de datos y sacar partido de ella. En este sentido, no se hará un uso intensivo

de ella, sino que se aplicará en ejemplos concretos para los que resulta especialmente útil (por ejemplo, para sobrecargar la indexación de la clase vector dinámico).

El caso del operador de asignación se presenta en el tema anterior, ya que es fundamental para que una clase se comporte de forma similar a los tipos más básicos del lenguaje.

En el **capítulo 5** se introducen las estructuras de datos lineales *Pila*, *Cola* y *Lista*. El objetivo es que el alumno asimile los conceptos asociados, y se acostumbre a usar los correspondientes tipos de datos para resolver problemas. Para ello, se evita complicar las interfaces con un número excesivo de operaciones o con la sobrecarga de operadores. Además, se presenta la necesidad de iterar sobre los elementos de un contenedor, aunque de una forma simple, y evitando la complejidad del tema de abstracción por iteración.

Se incluye también el tipo *Cola con prioridad*, ya que aunque habitualmente no se trata hasta que no se dispone de la implementación basada en árboles, se puede presentar el tipo de dato con una primera solución basada en una estructura de datos lineal.

En el **capítulo 6** el lector ya ha usado distintos contenedores para resolver algunos problemas, y ya es consciente de la incomodidad que implica restringir los tipos de datos desarrollados sólo para un determinado tipo base. En este tema se introduce el concepto de plantilla, como una herramienta que permite generalizar los tipos de datos que se han estudiado de forma que puedan contener elementos de distinto tipo.

En el **capítulo 7** se introducen los árboles como un caso de estructura de datos no lineal. Para compensar la novedad del tema y la complejidad de los nuevos conceptos, buena parte del tema se desarrolla independientemente de los temas anteriores. Así, es posible entender una buena parte del tema incluso sin los capítulos relacionados con tipos de datos abstractos. Es en los ejemplos de la clase conjunto y cola con prioridad cuando se desarrollan soluciones basadas en tipos de datos abstractos.

En el **capítulo 8** el lector se ha habituado a distintas estructuras de datos y a problemas en los que es necesario organizar la información en contenedores. En este tema es fácil introducir la idea de *iterador*, como un método común que se puede usar para recorrer e incluso modificar los elementos de un contenedor. Para justificar mejor su conveniencia, se presenta el concepto de programación genérica, y se desarrollan algunos algoritmos genéricos para distintos contenedores. En esta capítulo, el estudiante descubre que la abstracción nos permite tratar de la misma forma a estructuras de datos tan distintas como un vector, una lista de celdas enlazadas o un árbol binario de búsqueda equilibrado. Para aplicarlo en problemas concretos se desarrollan los tipos *Conjunto* y *Diccionario*.

En el **capítulo 9** se presentan las tablas *hash*. Este tipo de dato es fundamental en cualquier libro de estructuras de datos, y ofrece una alternativa al problema de búsqueda que es muy distinta de la ofrecida en el capítulo de árboles. La primera parte se desarrolla de una forma similar a la de árboles, es decir, independientemente del problema de desarrollar clases basadas en esta estructura, centrándonos en los conceptos y los desarrollos teóricos que la justifican. Así, esta primera parte se podría también estudiar antes que el tema de clases.

En la segunda parte se desarrolla el tipo de dato *ConjuntoDesordenado*, integrándolo en la metodología que se ha desarrollado en los temas anteriores.

En el **capítulo 10** se estudia la gestión de E/S y los ficheros. Un estudio en profundidad de este tema requiere el uso de conceptos de programación dirigida a objetos. Sin embargo, sólo serían necesarios para entender la relación que hay entre las distintas clases que implementan el sistema de gestión de E/S. A pesar de que el lector no conozca la herencia y el polimorfismo, es sencillo que entienda de forma intuitiva el comportamiento de estas clases, de manera que no resulta difícil que maneje la jerarquía sin dificultad.

Dado que las aplicaciones que se pueden desarrollar a lo largo del libro pueden resultar bastante complejas, es conveniente ofrecer este tema para poder trabajar con información en disco. Se ha desarrollado evitando cualquier relación con los temas anteriores, de forma que sólo es necesario conocer el tema de clases para poder estudiarlo. De esta manera, el lector podrá disponer de los conocimientos necesarios para poder manejar grandes cantidades de información.

Finalmente, se incluyen varios apéndices para completar los conocimientos que se presentan en los temas anteriores:

- En el **apéndice A** aparecen las soluciones a los ejercicios propuestos a lo largo del libro. Para que el lector asimile más fácilmente los contenidos, se proponen ejercicios a la largo de los temas, de forma que el estudio de éstos se realice de una forma más interactiva. Cuando se propone un ejercicio, el lector debería intentar resolverlo proponiendo su solución, tras lo que debería consultar la solución propuesta en este apéndice.

- En el **apéndice B** se lista el código de los tipos *Fecha* y *Polinomio* propuestos en el tema 2. Aunque estos tipos no son definitivos, ya que en el tema 3 se propone la implementación en base a clases, resulta útil discutir los ejemplos incluyendo hasta los últimos detalles, especialmente para los programadores sin experiencia. Además, de esta forma podrán realizarse ejercicios relacionados, así como entender mejor las modificaciones que se proponen en el tema 3.

- En el **apéndice C** se presenta la clase *string* y algunos detalles relacionados. El tipo *std::string* es parte de la biblioteca estándar de C++, y su comportamiento es, en gran medida, parecido al de los tipos simples. En este sentido, este apéndice no resulta fundamental para el desarrollo del libro. Sin embargo, la cantidad de aplicaciones que usan cadenas de caracteres, así como la importancia de entender el valor añadido de esta nueva clase, hace necesario incluir este pequeño apéndice.

- En el **apéndice D** se presentan algunas tablas de interés para un programador de C++, concretamente, los operadores y las palabras reservadas del lenguaje.

Agradecimientos

En primer lugar, queremos agradecer el trabajo desinteresado de miles de personas que han contribuido al desarrollo del software libre, sobre el que se ha desarrollado este material, y que constituye una solución ideal para que el lector pueda disponer del software necesario para llevar a cabo este curso.

Por otro lado, no podemos olvidar a las personas que nos han animado a seguir trabajando para crear este documento. El trabajo y tiempo que implica escribir un libro no viene compensado fácilmente si no es por las personas que justifican ese esfuerzo. En este sentido, queremos agradecer a nuestros alumnos su paciencia, y recordar especialmente a aquellos, que con su esfuerzo e interés por aprender, han sabido entender el trabajo realizado para facilitar su aprendizaje.

Finalmente, queremos agradecer el apoyo de los compañeros que han usado este material como guía en sus propias clases, ya que con ello nos han animado a completarlo y a terminar este libro.

Los autores
Diciembre de 2005

Contenido

Capítulo 1

Análisis de la eficiencia

En este capítulo

1.1. Introducción

Cuando se desea resolver un problema es necesario seleccionar o diseñar un algoritmo. Lógicamente, podemos disponer de distintas soluciones y, en principio y a pesar de que unos sean más rápidos que otros, o que unos consuman más recursos que otros, puede parecernos irrelevante (o al menos poco importante) esta elección. Además, podemos pensar que el incremento de potencia de los computadores actuales nos permite disponer de máquinas mucho más potentes en un tiempo relativamente pequeño y, por tanto, incluso los algoritmos más lentos se ejecutarán en poco tiempo.

Sin embargo, este creencia es totalmente falsa. Consideremos un ejemplo. Tenemos el problema de asignar 50 trabajadores a 50 trabajos distintos. El perfil de cada trabajador nos indica un rendimiento distinto dependiendo del trabajo que se le asigne. Cuantificamos dicho rendimiento con un determinado valor, de forma que se nos plantea el problema de descubrir aquella asignación que sume el valor más alto.

En principio, podemos estar tentados a diseñar el algoritmo más simple, es decir, el ordenador busca la solución evaluando todas las asignaciones posibles. Si nos fijamos detenidamente, esto implica la evaluación de 50! posibilidades, es decir, del orden de 10^{64} asignaciones. Un ordenador que evaluara 1 billón de posibilidades por segundo, y hubiera empezado hace 15000 millones de años, todavía no habría acabado.

Este ejemplo nos muestra que es necesario tener en cuenta la eficiencia de los algoritmos que usamos. Ahora bien, si disponemos de varios algoritmos, ¿Cómo podemos compararlos?

1.1.1. Tamaño del problema

En primer lugar, tenemos que tener en cuenta que un algoritmo no tiene un tiempo fijo de ejecución, sino que ese tiempo depende del tamaño del problema. Así, el algoritmo que indicábamos en el ejemplo anterior, para un conjunto de 5 trabajadores y 5 puestos tiene un tiempo de ejecución equivalente a la evaluación de 120 asignaciones, es decir, un tiempo relativamente corto en un ordenador actual. Sin embargo, hemos visto que para nuestro ejemplo de tamaño 50 era impracticable.

Consideremos otro ejemplo. Tenemos dos algoritmos para ordenar un vector de enteros. Una forma simple de compararlos es implementarlos y medir el tiempo para un determinado tamaño, por ejemplo 10.000. Ahora bien, si el primero es más rápido, ¿es más eficiente?. Tal vez, una ejecución con más elementos, por ejemplo 100.000, tenga un comportamiento distinto.

Por lo tanto, debemos tener en cuenta que el tiempo de ejecución viene descrito por una función del tamaño del problema[1], que a partir de ahora denotaremos n, y por

[1] En realidad, este tiempo no sólo depende del tamaño del problema sino también del valor concreto de los datos de entrada. Véase más adelante una discusión más detallada sobre esta cuestión. Por ahora, simplemente consideremos que los datos de entrada son los peores (los que más tiempo necesitan para su procesamiento).

tanto, si queremos comparar dos algoritmos, lo que tenemos que hacer es *comparar las funciones* que describen su tiempo de ejecución.

Dado que n lo podemos considerar un número natural, y el tiempo de ejecución no puede ser negativo, consideraremos que la función que describe el tiempo de ejecución de un algoritmo está definida del conjunto de los números naturales a los reales positivos incluyendo el cero ($\Re_0^+$)

1.1.2. Algoritmos vs implementaciones

Es fundamental la distinción entre algoritmos, como un conjunto finito de pasos que nos llevan a resolver un problema, e implementaciones, como una realización de un algoritmo en un determinado lenguaje de programación. El análisis que nos ocupa se refiere a los algoritmos, y no a las implementaciones.

1.2. Eficiencia de algoritmos

1.2.1. Familias de órdenes de eficiencia

Supongamos que tenemos dos algoritmos para resolver un problema de tamaño n. Desarrollamos dos implementaciones, a las que corresponden las siguientes funciones como tiempos de ejecución:

- Algoritmo 1: $t_1(n) = n$
- Algoritmo 2: $t_2(n) = 2n$

La conclusión obvia es que la función 1 es mejor que la 2. Ahora bien, si pensamos detenidamente en la razón de esta diferencia descubrimos que hay otros factores, distintos al algoritmo propio, que pueden influir en estos tiempos. Por ejemplo:

- El tiempo no puede depender de la máquina o el sistema operativo.
- Las herramientas usadas en el desarrollo de las soluciones pueden afectar al resultado (por ejemplo, un compilador más optimizado).
- Tal vez, la segunda solución la ha propuesto un programador menos habilidoso, o incluso, el mismo programador estando más cansado.

Por tanto, no es posible asegurar que la primera solución sea la mejor. Recordemos que nuestro objetivo es el análisis de algoritmos, y no de implementaciones.

El principio de invarianza nos permite resolver el problema:

Principio de invarianza

"Dos implementaciones del mismo algoritmo no difieren en eficiencia más de una constante multiplicativa."

Es decir, desde un punto de vista matemático, dadas dos implementaciones de un mismo algoritmo, con funciones $t_1(n)$ y $t_2(n)$, existe $c \in \Re^+$ y n_0 natural tal que

$$\begin{aligned} t_1(n) &\leq c\, t_2(n) \\ t_2(n) &\leq c\, t_1(n) \end{aligned} \tag{1.1}$$

para $n \geq n_0$.

Por tanto, si queremos que el análisis de eficiencia no dependa de la implementación, tenemos que considerar a ambas funciones *equivalentes*. Es decir, hemos particionado el conjunto de las funciones que caracterizan el tiempo de ejecución de los algoritmos en *clases de equivalencia* y constituyen las denominadas *familias de órdenes de eficiencia* o, más común y simple, órdenes de eficiencia.

Algunos de estos órdenes son muy comunes, por lo que resulta más sencillo referirse a ellos con algún nombre. Así, por ejemplo, podemos distinguir:

- Orden lineal, donde se encuentra la función *f(n)=n*.
- Orden cuadrático ($f(n) = n^2$).
- Orden logarítmico.
- Orden exponencial.
- etc.

De esta forma, los algoritmos con tiempos de ejecución en cada una de esas clases se denominarán algoritmos lineales, cuadráticos, logarítmicos, etc.

Cuando queramos estudiar el tiempo de ejecución de un algoritmo independientemente de la implementación, tendremos que calcular la clase de equivalencia (representadas en la figura 1.1) a la que corresponde la función del tiempo de ejecución, es decir, determinar el orden de eficiencia.

1.2.2. Notación asintótica

Como hemos visto, la eficiencia de un algoritmo viene caracterizada por un orden de eficiencia. Ahora se nos plantea el problema de "ordenar" los órdenes, de manera que podamos indicar si un algoritmo es más eficiente que otro, así como describir una notación para poder expresarlo.

Esta notación la denominamos asintótica, porque nos interesa el comportamiento especialmente cuando *n* aumenta:

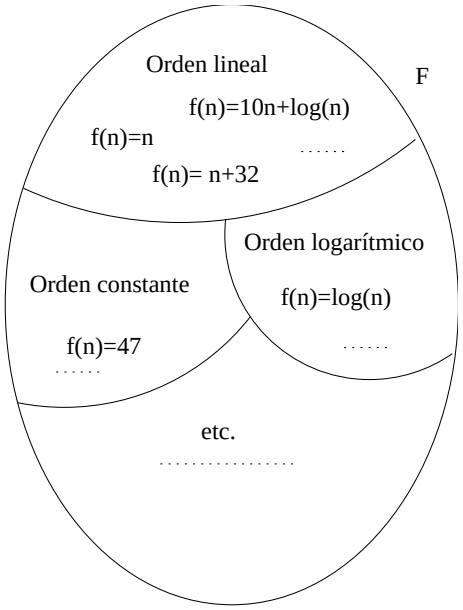


Figura 1.1
Partición del conjunto de las funciones en órdenes de eficiencia.

- Recordemos que no distinguimos entre tiempos que difieren en una constante multiplicativa. Cuando el tamaño del problema es pequeño, podemos compensar con una constante. Es decir, si tenemos una función que está por debajo de otra en un intervalo finito de valores, siempre podemos encontrar una constante multiplicativa para situarla por encima[2].

- Cuando mejora el hardware, los problemas que se resuelven son de mayor tamaño. Los algoritmos con un mejor comportamiento cuando *n* crece son más interesantes con vistas al futuro. Por ejemplo, consideremos una mejora a un ordenador el doble de rápido. Si tenemos dos algoritmos de eficiencia logarítmica y exponencial respectivamente:

  - $k_1 log_2(n)$: Si en 1 hora resuelve *n=100*, con un ordenador dos veces más rápido, puede resolver *n=10.000*.
  - $k_2 2^n$: Si en 1 hora resuelve *n=100*, con un ordenador dos veces más rápido, puede resolver *n=101*.

  Es decir, un problema 100 veces más grande puede resolverlo el primer algoritmo con sólo un ordenador 2 veces más rápido. Sin embargo, el tiempo que requiere

[2]En realidad, esto no es posible si consideramos que las funciones pueden valer cero. Sin embargo, puesto que manejamos funciones que miden el tiempo de ejecución de un algoritmo, podemos considerar que los casos que nos interesan tienen valores siempre positivos.

el segundo lo hace inútil. La bondad del primer algoritmo es consecuencia del "buen" comportamiento cuando el tamaño del problema crece.

Por tanto, la decisión sobre la mayor o menor eficiencia de un algoritmo dependerá del comportamiento del orden de eficiencia cuando n crece, es decir, cuando tiende a infinito. En este sentido, lo que hacemos es comparar *"perfiles de crecimiento"*, es decir, un algoritmo es más eficiente si crece más lentamente.

Comparando órdenes

Para poder comparar dos algoritmos, tenemos que ser capaces de decir que un orden de eficiencia es mayor que otro. Para poder hacerlo, tenemos en cuenta que:

- El resultado no puede depender de lo que ocurre en un intervalo finito de valores de la función.
- Para comparar dos órdenes de eficiencia, podemos usar dos funciones concretas que representen las correspondientes clases. El resultado no puede depender de esta elección.

Con estas consideraciones, resulta natural la siguiente definición:

Orden entre funciones
Sean dos funciones $f, g : N \to \Re_0^+$.
Diremos que la función *g(n)* es menor o igual que *f(n)* si:
$\exists\, c \in \Re^+, n_0 \in N$ tal que $\forall\, n \geq n_0\ g(n) \leq c \cdot f(n)$

A partir de esta definición podemos deducir las relaciones de igual[3], mayor y menor.

En la figura 1.2 podemos observar la gráfica[4] de las funciones n^2 y $100n$. En este caso, vemos que la función cuadrática es superior a la lineal a partir del cien. Según la definición, con la constante $c = 100$ y valor $n_0 = 1$, la función cuadrática queda por encima de la función lineal. Por tanto es mayor o igual que la lineal. Nótese que lo contrario, situar la lineal por encima, no es posible (si lo fuese, las dos funciones pertenecerían a la misma clase, serían iguales).

Nótese que el orden que estamos estableciendo es sólo parcial, es decir, podemos encontrar dos funciones que no puedan ser ordenadas. Consideremos por ejemplo las funciones:

[3] Recordemos el principio de invarianza y la equivalencia de funciones. Véase la coincidencia para definir que dos funciones son iguales(están en la misma clase).

[4] Se han dibujado como lineas continuas para mayor claridad, aunque sabemos que son funciones definidas sobre los naturales.

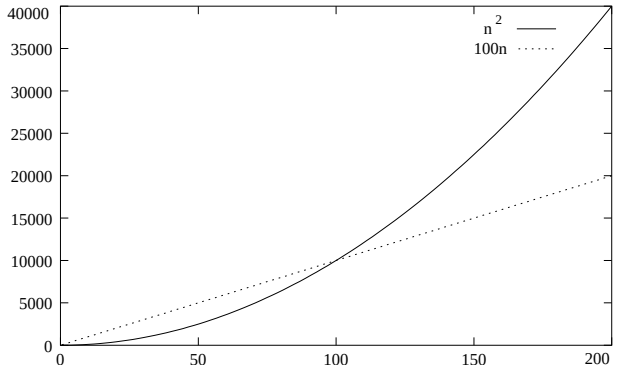


Figura 1.2
Comparación de n^2 y de $100n$.

$$\begin{aligned} f(n) &= \begin{cases} n^2 & \text{si } n \text{ par} \\ n^4 & \text{si } n \text{ impar} \end{cases} \\ g(n) &= n^3 \end{aligned} \tag{1.2}$$

En este caso, la función *g(n)* no está por debajo, ni es de la misma clase, ni está por encima de la función *f(n)*. Sin embargo, en nuestras discusiones sobre eficiencia de algoritmos, haremos uso de funciones más simples que pueden ordenarse entre sí y, por tanto, siempre será posible decidir qué algoritmo es el más eficiente.

Notaciones O, Ω, Θ

Para poder manejar el concepto de eficiencia de un algoritmo se desarrollan las notaciones O, Ω, Θ. El objetivo es poder indicar, de forma clara y sin ambigüedad, el grado de eficiencia que hemos obtenido en el análisis de un algoritmo.

En principio, la forma ideal de analizar un algoritmo consiste en obtener la función que determina el tiempo necesario para ejecutarse, y que nos lleva a una clase de eficiencia. Si tenemos ordenadas las distintas clases, tenemos ordenados, por eficiencia, los distintos algoritmos. En la figura 1.3 se muestra gráficamente esta idea. Podemos distinguir dos pasos:

1. Análisis del tiempo de ejecución. Tenemos que estudiar la función que indica el tiempo de ejecución necesario para cada tamaño de problema. Este problema lo tratamos posteriormente en este tema.

2. Análisis de eficiencia. Clasificamos ese tiempo de ejecución en una familia de funciones.

Si tenemos ordenadas las distintas familias de órdenes de eficiencia, podemos comparar algoritmos, una vez que conocemos la eficiencia de éstos.

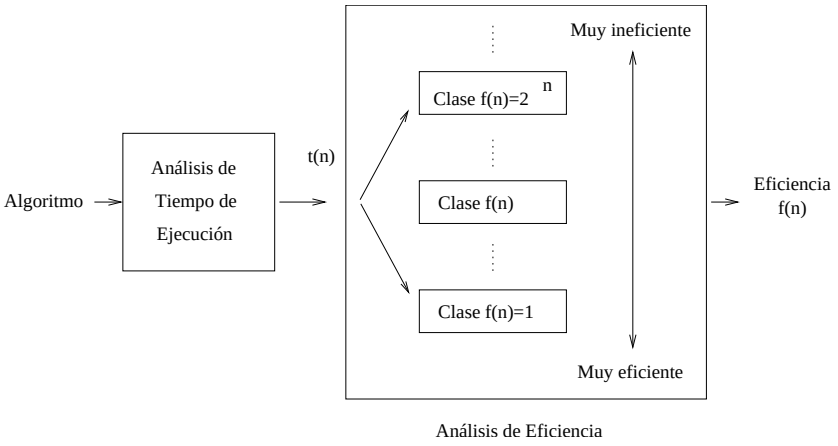


Figura 1.3
Análisis de eficiencia de algoritmos.

Por tanto, este tipo de análisis consiste en estudiar el *orden exacto* de la función de tiempo de ejecución. Para denotar esta eficiencia, se utiliza la notación Θ. Denominamos $\Theta(f(n))$ al conjunto de funciones de la familia *f(n)*. Por ejemplo, si analizamos un algoritmo que tiene un tiempo de ejecución de *t(n)*, decimos que es $\Theta(f(n))$ si *t(n)* es de la familia de *f(n)* ($(t(n) \in \Theta(f(n))$). Ejemplos:

- La función $t(n) = (n+1)^2$ es $\Theta(n^2)$
- La función $t(n) = 2^n$ no es $\Theta(3^n)$
- La función $t(n) = \log_2 n$ es $\Theta(\log_3 n)$

Sin embargo, tenemos que tener en cuenta que tal vez sea muy complejo el análisis del tiempo de ejecución. En este caso es muy difícil determinar la eficiencia del algoritmo. Por otro lado, es posible que nos interese demostrar que un algoritmo tiene una eficiencia mínima, aunque no conozcamos su eficiencia exacta.

Por ejemplo: imaginemos que la escala de eficiencia es un valor entero del 1 al 10. El 1 es muy bueno y el 10 es muy malo. Tal vez no es necesario estudiar la eficiencia de un algoritmo para indicar que es de eficiencia, digamos, 3. Puede bastarnos el decir que como mucho es 5 para indicar que es bastante bueno, o por ejemplo decir que al menos es 7, para indicar que es bastante malo. Es decir, indicar una cota superior o inferior del valor de eficiencia.

Para denotar esta situación, utilizamos las notaciones O, Ω. La primera nos sirve para indicar una cota superior y la segunda inferior. Así, $O(f(n))$ es el conjunto de todas las funciones que son iguales o menores que *f(n)*, y $\Omega(f(n))$ es el conjunto de todas las funciones que son iguales o superiores a *f(n)*. Algunos ejemplos son:

- La función $t(n) = (n+1)^2$ es $O(n^2)$ ($(n+1)^2 \in O(n^2)$)

- La función $t(n) = 2^n$ es $O(3^n)$
- La función $t(n) = n^2$ es $\Omega(n)$
- La función $t(n) = \log_2 n$ es $O(2^n)$

En la figura 1.4 se muestra gráficamente el significado de las tres notaciones. Como se puede observar, se han representado como elipses distintas clases de eficiencia de forma ordenada, desde la más eficiente a la menos eficiente. Como se indica,

- $O(f(n))$ es el conjunto de todas las funciones que hay en la clase de *f(n)* más las que están por debajo. Es decir, cualquier función de este conjunto es menor o igual que la clase *f(n)* (ésta es una cota superior).
- De forma similar $\Omega(f(n))$ es el conjunto de las funciones en la clase *f(n)* más las que están por encima (es una cota inferior).
- Y finalmente, $\Theta(f(n))$ es el conjunto de funciones que están exactamente en la misma clase que *f(n)*.

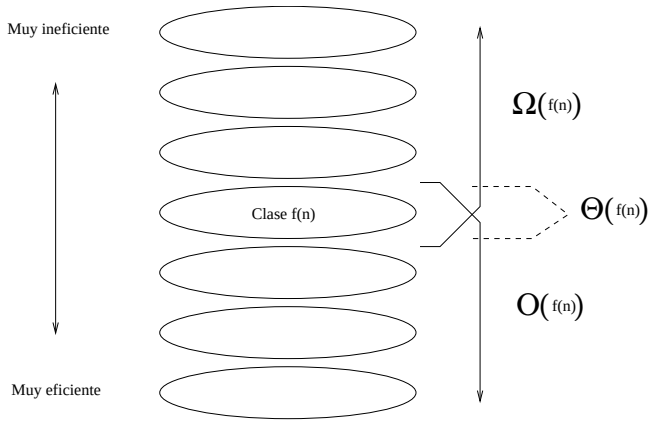


Figura 1.4
Notaciones O, Ω, Θ.

Más formalmente, la notación O (leída O-mayúscula) se puede definir:

Definición 1.1 (Notación O-mayúscula) *Sea una función* $f : N \to \Re_0^+$. *El conjunto de las funciones "del orden de f(n)", denotado por* $O(f(n))$ *se define como:*
$O(f(n)) = \{g : N \to \Re_0^+ / \exists\, c \in \Re^+, n_0 \in N \ \textit{tal que}\ \forall\, n \geq n_0\ g(n) \leq c \cdot f(n)\}$

De forma similar, la notación Ω:

Definición 1.2 (Notación Ω-mayúscula) *Sea una función* $f : N \rightarrow \Re_0^+$. *El conjunto de las funciones* $\Omega(f(n))$ *se define como:*
$\Omega(f(n)) = \{g : N \rightarrow \Re_0^+ / \exists\, c \in \Re^+, n_0 \in N \text{ tal que } \forall\, n \geq n_0\ g(n) \geq c \cdot f(n)\}$

Finalmente, la notación Θ a partir de las definiciones 1.1 y 1.2

Definición 1.3 (Notación Θ-mayúscula)

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

Es importante indicar que el concepto de eficiencia se ha mostrado más desde un punto de vista intuitivo que formal. De ahí, que hayamos abusado en la presentación del conjunto de clases como un grupo de funciones totalmente ordenado. Es necesario volver a insistir en el orden parcial que hemos establecido. Un ejemplo ilustrativo con las notaciones que hemos comentado es:

$$f(n) = \begin{cases} n & \text{si } n \text{ par} \\ 2^n & \text{si } n \text{ impar} \end{cases} \tag{1.3}$$

De la que podemos decir por ejemplo que es más eficiente que $\Theta(2^n)$ y menos eficiente que $\Theta(n)$, es decir, es $O(2^n)$ y $\Omega(n)$, pero nada con respecto a la relación con funciones como n^3, $n \log n$, $n\sqrt{n}$, etc. Este ejemplo nos sirve, asimismo, para mostrar una vez más la importancia de las notaciones O, Ω.

Operaciones entre órdenes de eficiencia

Se definen las operaciones de suma y producto para cada uno de las tres notaciones que hemos expuesto.

Definición 1.4 (Suma de Órdenes de complejidad) *Sean dos conjuntos O(f(n)), O(g(n)), la suma* $O(f(n)) + O(g(n))$, *se define como el conjunto:*

$$O(f(n)) + O(g(n)) = \{h : N \rightarrow \Re_0^+ / \exists\ f\prime \in O(f(n)),\ g\prime \in O(g(n),\ n_0 \in N \text{ tal que } \forall\, n \geq n_0\ h(n) = f\prime(n) + g\prime(n)\}$$

Definición 1.5 (Producto de Órdenes de complejidad) *Sean dos conjuntos O(f(n)), O(g(n)), el producto* $O(f(n)) \cdot O(g(n))$, *se define como el conjunto:*

$$O(f(n)) \cdot O(g(n)) = \{h : N \rightarrow \Re_0^+ / \exists\ f\prime \in O(f(n)),\ g\prime \in O(g(n),\ n_0 \in N \text{ tal que } \forall\, n \geq n_0\ h(n) = f\prime(n) \cdot g\prime(n)\}$$

y sus correspondientes extensiones a la suma y producto de órdenes Ω, Θ.

A partir de ellas, se definen las reglas de la suma y el producto como sigue:

Regla 1.1 (Regla de la suma)

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

Regla 1.2 (Regla del producto)

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

y sus correspondientes extensiones para las notaciones Ω, Θ.

Estas reglas tienen una aplicación directa en el análisis de algoritmos que veremos más adelante. Ahora mismo sólo indicaremos que, para calcular la eficiencia de algoritmos con estructuras secuenciales, se utilizará la regla de la suma, y para el cálculo con estructuras iterativas, la del producto.

1.2.3. Eficiencia en tiempo y espacio

A lo largo de las secciones anteriores se ha mostrado el concepto de eficiencia en relación al tiempo de ejecución requerido. Sin embargo, un algoritmo no sólo consume recursos de CPU, sino también de memoria. De hecho, en muchos casos podemos ver cómo resulta sencillo mejorar el tiempo de ejecución del algoritmo a costa de aumentar la cantidad de memoria requerida. Por consiguiente, sería un terrible error evaluar la bondad de un algoritmo únicamente a partir del tiempo.

Es necesario, por tanto, indicar también la cantidad de espacio que necesita un algoritmo. Al igual que hemos hecho con respecto al tiempo, podemos plantear una discusión similar con respecto al espacio. Si embargo, sólo indicaremos que las notaciones que hemos propuesto para el primero se utilizarán para el segundo. Por supuesto, el significado será el mismo, utilizaremos una función del tamaño del problema, en este caso referido a unidades de almacenamiento y no de tiempo de ejecución.

1.2.4. Elección del mejor algoritmo

Durante las secciones anteriores se ha insistido en la definición de eficiencia como un perfil de crecimiento independiente de cualquier constante. En principio, podemos considerar que la elección de un algoritmo se basa en su eficiencia. Sin embargo, en la práctica debemos considerar varios factores:

- El tamaño de los problemas que nuestro software va a resolver.
- Los requisitos de tiempo y espacio de nuestro sistema.
- Complejidad de implementación y mantenimiento de los algoritmos.

Por ejemplo, podemos considerar dos algoritmos con un tiempo de ejecución de $t_1(n) = 100n$ y $t_2(n) = n^2/5$. Obviamente, el primero es más eficiente ya que es lineal ($\Theta(n)$) mientras que el segundo es cuadrático ($\Theta(n^2)$). A pesar de ello, es posible que el primero pueda ser más recomendable en la práctica, por ejemplo si ocurre que:

- El tamaño de los problemas a resolver no va a pasar de 100. En este caso, el tiempo del primero es menor que el segundo. A pesar de tener una eficiencia teórica mejor, tiene un comportamiento práctico, para esos tamaños, peor. En la elección de un algoritmo, no podemos olvidar la constante multiplicativa que hemos considerado en el análisis de la eficiencia.

- El primer algoritmo es mucho más rápido, sin embargo, requiere una cantidad de espacio superior. Por ejemplo, puede darse el caso de que requiera una cantidad de espacio cuadrática con respecto al tamaño, mientras que el segundo requiera una cantidad constante. Si en nuestra aplicación es importante el espacio y los requisitos no nos permiten ese gasto, tendremos que escoger el algoritmo de tiempo cuadrático.

- El algoritmo lineal requiere un coste de implementación y mantenimiento muy altos. Si nuestra aplicación tiene unos requisitos, de tiempo y espacio, que cumple perfectamente el algoritmo de tiempo cuadrático, esta opción puede ser más conveniente para reducir costes sin que la calidad del producto final se vea afectada.

1.3. Análisis de algoritmos

Una vez que conocemos el concepto de eficiencia teórica, debemos estudiar métodos que nos permitan obtener, a partir de un algoritmo escrito, por ejemplo en C++, la eficiencia que tiene asociada. En esta sección abordamos este problema.

1.3.1. Operación elemental

Definición 1.6 (Operación elemental) *Una operación elemental es cualquier operación cuyo tiempo de ejecución se puede acotar superiormente por una constante.*

Este concepto es fundamental en el análisis de algoritmos, porque el problema de estimar el tiempo de ejecución de un algoritmo se traduce en el problema de estimar el número de operaciones elementales que ejecuta un algoritmo.

Recordemos que en el análisis de eficiencia, nuestro objetivo es calcular una función que indique el orden de eficiencia al que pertenece nuestro algoritmo. Por tanto, en principio, no nos va a interesar conocer la función exacta que indica el tiempo de ejecución, sino cualquiera que corresponda a la misma eficiencia. No importa obtener una eficiencia $\frac{5}{2}n^2 + 7n + 18$ o de $100n^2$ ya que, en ambos casos, nos referimos a eficiencia n^2. Esta condición resulta muy útil, ya que nos permite hacer las simplificaciones que queramos a los cálculos, siempre que no afecten al orden de eficiencia obtenido.

Veamos un ejemplo que ilustra estas ideas. Supongamos un algoritmo que consiste en n_s sumas, n_m multiplicaciones y n_a asignaciones de un número real. Estas tres ope-

raciones podemos considerarlas elementales, porque si el tamaño de los operandos es limitado, el tiempo de ejecución de cada una de ellas está acotado[5].

Consideremos que el tiempo de ejecución es t_s, t_m y t_a segundos respectivamente para cada una de las operaciones. Es fácil ver, que si nuestro tiempo de ejecución total es t,

$$min(t_s, t_m, t_a) \cdot (n_s + n_m + n_a) \leq t = n_s \cdot t_s + n_m \cdot t_m + n_a \cdot t_a \leq max(t_s, t_m, t_a) \cdot (n_s + n_m + n_a)$$

Y por tanto el tiempo exacto de ejecución se puede acotar con:

$$c_1 \cdot n \leq t \leq c_2 n$$

donde $n = n_s + n_n + n_a$ es el número de operaciones del algoritmo. Como las constantes no afectan a la eficiencia, es claro que el tiempo es una función lineal, ya que la hemos puesto encima y debajo de dos funciones lineales. Por tanto, el algoritmo es $\Theta(n)$.

Veamos ahora un ejemplo sobre un trozo de código simple, la asignación de cero a todas las posiciones de un vector:

```
for (i=0;i<n;++i)
  A[i]=0;
```

La eficiencia viene determinada por el número de operaciones elementales. Si nos fijamos en el código, el número de operaciones elementales podría contabilizarse como:

- Una operación de asignación antes de la ejecución del bucle.
- Una operación de evaluación de la condición antes de entrar en el bucle.
- Para cada iteración:
  - Dos operaciones para el cuerpo del bucle (indexación+asignación).
  - Operación de incremento y evaluación de la condición.

Por lo tanto, el número total de operaciones sería $1 + 1 + n \cdot (2 + 2)$, es decir, $4n + 2$.

Sin embargo, este análisis resulta muy incómodo, pues obliga a estudiar en detalle el algoritmo[6]. Podemos simplificar aún más el problema. Por ejemplo, podemos decir que antes del bucle realiza un número finito o constante de operaciones elementales, que es lo mismo que si realiza una sola (no afectará al resultado del análisis). Dentro del bucle

[5] Estrictamente hablando, estas operaciones no son realmente elementales, pues dependen del tamaño de los operandos. Cuanto más bits se utilizan en la representación del número, más tiempo se requiere para ejecutarla y, por tanto, no se puede limitar. Sin embargo, estos valores probablemente se traducirán en variables simples del lenguaje de programación que usemos, para las que existe claramente un tiempo límite de ejecución.

[6] Nótese que podría incluso complicarse si intentamos estudiar los pasos que realiza el algoritmo a un nivel más bajo (por ejemplo en ensamblador).

realiza dos operaciones elementales y dos para el incremento y evaluación de la condición, pero también podemos decir que para cada iteración del bucle realiza una operación elemental, ya que tampoco afectará al resultado (nótese además, que un número constante de operaciones elementales puede considerarse, según nuestra definición, una sola operación).

Con este planteamiento simplificado, es fácil ver que el código realiza un número de operaciones de $1 \cdot n + 1$ que corresponde a una eficiencia $\Theta(n)$, al igual que en el análisis detallado que acabamos de comentar.

1.3.2. Caso peor, caso promedio y análisis amortizado

El tiempo de ejecución de un algoritmo depende del tamaño de la entrada y, como indicamos al principio de este capítulo, de los valores de entrada. Por ejemplo, consideremos un algoritmo que busca un elemento en un vector. Si la entrada es un vector que contiene el elemento en la primera posición buscada, la ejecución consumirá muy poco tiempo, sin embargo, si el elemento está en la última, el algoritmo consumirá mucho más. Por tanto, antes de indicar la eficiencia de un algoritmo, tenemos que decidir cómo vamos a considerar la entrada.

Análisis del peor caso

El primer lugar, consideramos el análisis más habitual en la mayor parte de la bibliografía. Se refiere, como era de esperar, al análisis del peor caso. Dado que el algoritmo puede tener multitud de posibles entradas, a las que corresponden múltiples tiempos de ejecución, escogemos una muy concreta: la que provoca un tiempo de ejecución máximo, es decir, el *peor caso*.

Consideremos la búsqueda de la posición de un elemento en un vector de tamaño n, como un algoritmo que recorre una a una las posiciones del vector desde la primera hasta la última (algoritmo de *búsqueda lineal*).

Podemos analizar desde el mejor caso, es decir, que esté en la primera posición buscada, y por lo tanto en un paso el algoritmo termine, hasta el peor caso, es decir que el algoritmo busque la primera posición, no esté el elemento y continúe hasta que recorra todas las posiciones, encontrándolo exactamente en la última posición consultada. Como vemos, este caso es el de mayor tiempo de ejecución, para el que se necesitan del orden de n pasos.

Como es de esperar, parece que el caso más interesante es el de mayor tiempo, y por tanto, diremos que el algoritmo de búsqueda necesita, en el peor caso, del orden de n pasos ($\Theta(n)$, ya que cada paso se puede considerar una operación elemental).

Finalmente, recordemos que la notación O-mayúscula indica una cota superior de la eficiencia del algoritmo. Como el peor caso lo podemos considerar el valor más alto para cualquier ejecución, resulta natural indicar la eficiencia con esta notación, incluso sin decir que se trata de análisis de peor caso. Así, el algoritmo de búsqueda de un elemento es, en peor caso, $\Theta(n)$, o simplemente es un algoritmo $O(n)$.

La mayor parte de las referencias a eficiencia que encontramos en la literatura utilizan esta notación, asumiendo cuando no se dice nada, que el análisis se hace en peor caso[7].

Análisis del caso promedio

En muchos algoritmos resulta mucho más interesante medir, no el tiempo de ejecución del peor caso, sino el tiempo medio que necesita una llamada cualquiera. Por ejemplo, consideremos que para resolver un problema se van a realizar un número muy grande de llamadas a cierto algoritmo, para el que es de esperar que el tipo de las entradas sea variado. En este ejemplo, resulta más práctico considerar el tiempo que en media tarda el algoritmo, ya que el tiempo en peor caso multiplicado por el número de llamadas puede resultar un valor mucho más alto que el real.

Un análisis en caso promedio es más complejo que en peor caso, puesto que para éste sólo analizamos una entrada concreta, mientras que para el tiempo medio necesitamos considerar que puede darse cualquier entrada. Básicamente, la forma de realizarlo es calcular una media del número de operaciones que tenemos que llevar a cabo para cada posible entrada. Nótese que no todas las entradas tienen que ser igualmente probables, por lo que la media será ponderada por la probabilidad de cada entrada. Por tanto, el primer problema a resolver para este tipo de análisis es determinar una distribución de probabilidad para las posibles entradas.

Volvamos al ejemplo simple de la búsqueda lineal, donde localizamos la posición de un elemento que se encuentra en un vector:

- *Posibles entradas*: Corresponden a todos los casos, desde el más favorable que corresponde a que el elemento está en la primera posición, hasta el pero caso, cuando está en la última. Por tanto *n* posibles entradas. Si el elemento está en la primera posición, el algoritmo requiere un paso para localizarlo, si está en la segunda, dos pasos, etc.

- *Distribución de probabilidad*: Parece lógico que para un caso general, el valor que buscamos puede estar en cualquier lugar del vector con igual probabilidad. Esto es, que la probabilidad de encontrarlo en la posición i-ésima es *1/n*, para cualquier posición *i* del vector.

Por consiguiente, la media se puede calcular como

$$T(n) = \sum_{i=1}^{n} t_i \cdot p_i = 1 \cdot \frac{1}{n} + 2 \cdot \frac{1}{n} + \cdots + n \cdot \frac{1}{n} = \frac{1+n}{2}$$

donde t_i es el número de operaciones requeridas para resolver la entrada i, que tiene probabilidad p_i.

[7] A pesar de estas consideraciones, no podemos olvidar que una afirmación del tipo: "la búsqueda es $O(n^2)$" es correcta, sin que esto indique que el peor caso sea $\Theta(n^2)$, como hemos podido comprobar.

Como vemos, el resultado en este algoritmo es que en media la eficiencia también es lineal. Nótese que el número de operaciones que hemos calculado en este caso nos aparece dividido por dos, porque realmente, en media se realizan menos operaciones (el tiempo en media siempre será menor o igual que el peor caso). Existen otros algoritmos más complejos, para los que comprobaremos que el caso medio es más eficiente que el peor caso.

Finalmente, es importante destacar que en este análisis hemos supuesto una distribución uniforme en todas las entradas. Para aplicaciones concretas se puede realizar un estudio de esta distribución, ya que para que los resultados sean válidos, ésta debe ser conocida.

Análisis amortizado

En las secciones anteriores hemos considerado el tiempo de *peor caso en una ejecución* de un algoritmo, y el *tiempo medio de m ejecuciones independientes*. Consideramos un tipo de análisis adicional: *tiempo medio de m ejecuciones consecutivas*. Es importante destacar que:

1. El análisis amortizado es una media. Debido a que medimos el tiempo de *m* ejecuciones, y nuestro interés es el tiempo de una ejecución de un algoritmo, obviamente, el valor que buscamos es el cociente de ese tiempo entre el número de ejecuciones que realizamos.

2. El análisis amortizado es un estudio en peor caso. Aunque hacemos una media, no suponemos ninguna distribución de probabilidad sobre las entradas, sino que contamos el tiempo total, únicamente considerando que son ejecuciones consecutivas. Podríamos decir que el análisis amortizado es el tiempo, en peor caso, de un algoritmo que se ejecuta *m* veces consecutivas.

Inicialmente, podríamos pensar que es una situación muy extraña, ya que si tenemos que un algoritmo necesita un tiempo *f(n)* en peor caso, si lo ejecutamos *m* veces consecutivas, necesitaremos un tiempo *m f(n)*. Sin embargo, ahora introducimos un elemento más en nuestro estudio, ya que consideramos que cada vez que se llama, el algoritmo actúa sobre un mismo conjunto de datos, de manera que distintas llamadas necesitan distintos tiempos de ejecución y las operaciones realizadas en una de ellas afecta a las siguientes.

El hecho de realizar este tipo de análisis se debe a que existen muchos casos en los que un algoritmo puede ser muy costoso, pero este tiempo se puede compensar con otras llamadas que son muy rápidas. Por ejemplo, consideremos el llenado de una caja de libros. El problema consiste en que se dispone de una caja con capacidad para *n* libros. La operación que queremos analizar es la de *introducir* un libro. Esta operación requiere el movimiento de 1 libro desde un montón al interior de la caja, y la podemos suponer de tiempo 1 unidad (operación elemental). En la figura 1.5 se representa esta operación.

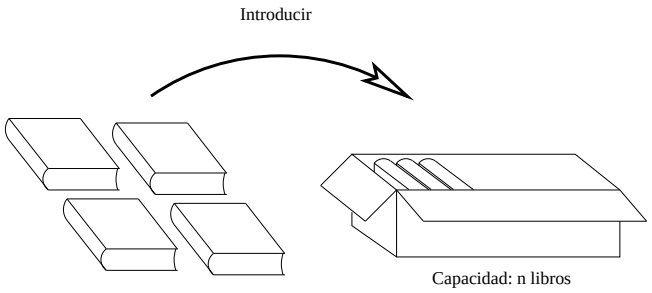


Figura 1.5
Operación *introducir* un libro.

Cuando la caja se llena, no se puede realizar la siguiente operación, así que la operación de *introducir* un nuevo libro necesita reemplazar la caja con otra que tenga el doble de capacidad. En este caso, tendrá que mover, uno a uno, todos los libros de la primera hasta la nueva caja para poder seguir introduciendo nuevos libros en la siguiente operación.

Un análisis en peor caso nos indica que la llamada a la operación de *introducir* libro es *O(n)*, ya que en el peor caso, cuando la caja se llena, hay que trasladar cada libro a la nueva caja (*n* operaciones elementales). Por lo tanto, podríamos decir que *m* operaciones consecutivas necesitan *mn* operaciones elementales.

Sin embargo, esta estimación es demasiado pesimista, ya que si son operaciones consecutivas, sabemos que para que ocurra una operación costosa, tienen que haber ocurrido muchas operaciones rápidas (cada ejecución que no llena la caja necesita sólo una operación elemental).

Para entenderlo, considere que se realizan *n* operaciones *introducción* de un libro. Obviamente, al llegar a *n* la última operación tendrá que usar otra caja (de capacidad *2n*) y trasladar uno a uno todos los libros. Para que ocurra esto, antes se han realizado *n* introducciones con coste 1. ¿Cuál ha sido el coste de las *n* inclusiones? Es fácil ver que en total, se han necesitado *2n* movimientos de libros (*n* primeras inclusiones más *n* movimientos a la nueva caja). Por tanto, cada operación ha necesitado *2* movimientos, es decir, la operación tiene un coste amortizado de *O(1)*.

En este ejemplo, se ha querido mostrar cómo la inclusión de un nuevo elemento (la caja y los libros), afecta al tiempo de la operación de *introducción*. El análisis se ha realizado de una manera intuitiva para que se entienda la importancia y el sentido de este tipo de análisis, mostrando cómo el tiempo calculado es en peor caso, y que corresponde a una media de ejecuciones consecutivas (¡que no son independientes!). Es un ejemplo claro de que una operación que puede parecernos costosa realmente se comporta con un tiempo de ejecución mucho mejor.

Este tipo de análisis es muy importante en el estudio de estructuras de datos, ya que es fácil encontrar operaciones que no son independientes. Operaciones muy costosas, que organizan las estructuras de datos, pueden compensarse con operaciones muy rápidas que van desorganizándolas poco a poco.

Los métodos para el cálculo del tiempo amortizado se salen de los objetivos de este libro. Se puede encontrar una buena introducción en Brassard[3] y algunos ejemplos sobre estructuras de datos avanzadas en Weiss[42], así como resultados prácticos en Mehlhorn[17]. Más adelante mostraremos ejemplos concretos en los que haremos referencia a este tipo de análisis.

1.3.3. Reglas para el cálculo de la eficiencia

Una vez establecido el problema del cálculo de eficiencia en las secciones anteriores, las reglas para realizarlo se pueden derivar de forma directa a partir de la semántica de las sentencias de nuestros algoritmos.

Nos centramos en el problema del cálculo de una cota superior (*O*-mayúscula) para el peor caso, ya que será el que utilizaremos en mayor medida a lo largo de este libro (las notaciones Θ, Ω se pueden analizar utilizando razonamientos similares).

Consideraremos el análisis de algoritmos descritos en C++, sin pérdida de generalidad. Es interesante observar que las reglas de cálculo no son más que contar el número de operaciones elementales del algoritmo, con la ventaja añadida de las simplificaciones derivadas del hecho de que queremos obtener un resultado en notación *O*-mayúscula.

Secuencia

Sean S_1 y S_2 dos fragmentos de código, con eficiencia $O(f_1(n)), O(f_2(n))$ respectivamente. La eficiencia de la unión secuencial de ambos fragmentos, "$S_1; S_2$", es $O(f_1(n)) + O(f_2(n))$ que, por la regla de la suma (página 10), es igual a $O(\text{máx}\{f_1(n), f_2(n)\})$.

Selección

En este apartado analizamos las sentencias *if,if-else,switch*, que corresponden a la selección simple, doble y múltiple, respectivamente.

En todas ellas es necesario evaluar una expresión para escoger el camino de ejecución correspondiente. Esta evaluación requiere un tiempo *O(E(n))*. Por supuesto, distintos caminos pueden tener asociados distintos órdenes de eficiencia. Nuestro interés se centra en el peor caso, por lo que tendremos que seleccionar, de entre los posibles caminos, el de peor eficiencia *O(f(n))*, y considerar que la eficiencia total es la suma de ambas, que como hemos visto, es $O(\text{máx}\{E(n), f(n)\})$.

Repetición

En este caso analizamos los bucles, es decir, las sentencias iterativas *for,while* y *do-while* de C++.

En todas ellas, el bucle se ejecuta un número de iteraciones determinado que puede depender del tamaño del problema. Es decir, podemos decir que el bucle se ejecuta un número *O(Ite(n))*. Téngase en cuenta que este valor depende de la condición de parada,

así que deberemos considerar el peor caso, es decir, el máximo valor de iteraciones que se pueden llevar a cabo.

Igualmente, es necesario evaluar para cada iteración la condición de parada, para lo que es necesario un tiempo *O(Con(n))*, y ejecutar el cuerpo del bucle con un tiempo *O(Cu(n))*.

El tiempo de ejecución total para cada bucle es, por tanto,

- *for*: En este caso, existe una inicialización previa al bucle, que necesita un tiempo *O(Ini(n))*, así como un incremento para cada iteración con tiempo *O(Inc(n))*. Conociendo la semántica de este tipo de bucle, podemos concluir que el tiempo total es:

$$O(Ini(n)) + O(Con(n)) + O(Ite(n)) \cdot [O(Cu(n)) + O(Inc(n)) + O(Con(n))]$$

  que corresponden, respectivamente, a la inicialización, la evaluación inicial de la condición, y el tiempo total de las iteraciones (para cada una hay que ejecutar el cuerpo, realizar el incremento y comprobar la condición).

- *while*: Utilizando la misma notación, el tiempo es:

$$O(Con(n)) + O(Ite(n)) \cdot [O(Cu(n)) + O(Con(n))]$$

- *do-while*: De una forma similar, el tiempo es:

$$O(Ite(n)) \cdot [O(Cu(n)) + O(Con(n))]$$

Para obtener la expresión final se aplicarán las reglas de la suma y el producto (página 10).

Funciones

El orden de eficiencia de una función viene determinado por el orden de eficiencia de las sentencias que la componen. Sin embargo, debemos tener en cuenta que también se realizan operaciones de paso de parámetros y devolución de resultados.

Tanto la llamada (el paso del control a la función) como el paso de parámetros por referencia (sólo es necesario pasar una dirección) se realizan, obviamente, en un tiempo constante y por tanto no afectan a la eficiencia de la función. Sin embargo, es de especial interés el paso por valor, ya que implica una operación de copia. Para evaluar el coste de la función se debe calcular el coste de esta operación. Por ejemplo, podríamos tener una función cuyo cuerpo es de orden constante pero que en la llamada tiene que copiar un parámetro que es del orden del tamaño del problema. Entonces, la función, por la regla de la suma, sería de orden lineal.

Por otro lado, el análisis de funciones tiene una dificultad añadida: la *recursividad*. Efectivamente, el coste de la función se puede calcular a partir del código de ésta, sin embargo, en una función recursiva necesitamos conocer su costo antes de calcularlo.

Para resolver este problema, debemos tener en cuenta que la llamada recursiva se realiza para un problema de tamaño menor (n más pequeño), por lo que podemos escribir la función de costo como una ecuación en términos de ella misma. La solución a esta ecuación es la eficiencia que buscamos. Por ejemplo, supongamos que queremos analizar la eficiencia de la siguiente función de cálculo del factorial:

```
1 int Factorial (int n)
2 {
3   if (n<=1)
4     return 1;
5   else
6     return n*Factorial(n-1);
7 }
```

El problema está en que no conocemos la eficiencia de la línea 6, ya que incluye una llamada a la misma función que estamos analizando. Para resolverlo, asignamos una función $T(n)$ desconocida, como tiempo de ejecución de la función. De esta forma, podemos traducir el problema a una ecuación recurrente que resolvemos con algún método matemático.

Para formular esta ecuación tenemos en cuenta dos casos:

1. El valor de n es uno, y por lo tanto la ejecución de la función es de orden constante. Por ejemplo, supongamos que el tiempo de ejecución es c.

2. El valor de n es mayor que uno. En este caso, el trozo de código que se ejecuta corresponde a un grupo de operaciones constantes más la llamada recursiva. Si denotamos d al tiempo que necesitan esas operaciones $O(1)$, el tiempo total de ejecución será $T(n-1)+d$.

Por tanto, podemos escribir la siguiente ecuación:

$$T(n) = \begin{cases} c + T(n-1) & n > 1 \\ d & n \leq 1 \end{cases}$$

Para resolver esta ecuación lo podemos hacer por expansiones sucesivas. Concretamente, podemos escribir que:

$$T(n) = c + \underbrace{T(n-1)}_{[a]} \quad n > 1$$

De igual forma, podemos expandir $[a]$:

$$[a] = c + \underbrace{T(n-2)}_{[b]} \quad n > 2 \quad (T(n) = 2c + T(n-2) \quad n > 2)$$

De nuevo expandimos [b]:

$$[b] = c + \underbrace{T(n-3)}_{[c]} \quad n > 3 \quad (T(n) = 3c + T(n-3) \quad n > 3)$$

$$[c] = ...$$

De esta forma, podemos extraer la forma general que corresponde a la expansión i-ésima:

$$T(n) = ic + T(n-i) \quad n > i$$

Dado que esta ecuación es válida para cualquier $n > i$, en particular para $i = n - 1$, tenemos:

$$T(n) = (n-1)c + T(n-(n-1)) = (n-1)c + T(1) = (n-1)c + d$$

En conclusión, podemos decir que $T(n)$ es $O(n)$.

Finalmente, indicar que este método de resolución de ecuaciones recurrentes no es el más eficaz. De hecho, para muchas ecuaciones recurrentes resulta imposible aplicarlo con éxito. En Peña[21],Brassard[3], etc. se pueden consultar distintos métodos de resolución.

Análisis de eficiencia en la práctica

En las secciones anteriores hemos presentamos algunas reglas para el análisis de distintos fragmentos de código. Téngase en cuenta que en la práctica, y para casos particulares, se pueden hacer consideraciones adicionales teniendo en cuenta la semántica del algoritmo. Por ejemplo, en el caso más simple de la secuencia, hemos analizado la eficiencia de dos segmentos de código de manera independiente. Sin embargo, es posible que en la práctica su comportamiento no sea independiente porque la primera parte modifica algún parámetro que afecta al tiempo de ejecución de la segunda.

De esta forma, podemos decir que el análisis de la eficiencia de un algoritmo no es una simple aplicación de reglas, sino que requiere que el analista entienda el comportamiento del algoritmo, incorporando este conocimiento para obtener un mejor resultado.

De hecho, en la práctica no se aplican de forma ciega las reglas descritas, sino que se estudia de forma razonada los pasos que realizará el algoritmo, acumulándolos en una función que describe el orden de eficiencia global. Incluso, se simplifica aún más, analizando la instrucción elemental que se ejecutan más veces en un segmento de código (denominada *instrucción crítica* o *instrucción barómetro*), por lo que un simple vistazo a un segmento de código, nos puede indicar la eficiencia que buscamos.

1.4. Ejemplos

1.4.1. Algoritmo de multiplicación de matrices

Un elemento $C_{i,j}$ de una matriz C (de dimensiones $n \times m$), resultado del producto de A ($n \times l$) por B ($l \times m$), se puede calcular como:

$$C_{i,j} = \sum_{k=1}^{k=l} A_{i,k} B_{k,j}$$

Esto nos permite escribir de una forma muy simple y directa un algoritmo de multiplicación de matrices con tres bucles anidados (dos para recorrer los elementos de C) y uno para el cálculo de cada sumatoria. Si suponemos que todas las matrices son de dimensiones $n \times n$, se puede escribir el siguiente código:

```
1 for (i=0;i<n;i++)
2   for (j=0;j<n;j++) {
3     C[i][j]= 0;
4     for (k=0;k<n;k++)
5       C[i][j]+=A[i][k]*B[k][j];
6   }
```

Para analizar este código estudiamos las sentencias más internas y vamos añadiendo líneas hasta calcular la eficiencia total:

1. En primer lugar podemos considerar la línea 5, donde acumulamos un valor en la posición *(i,j)* de la matriz resultado. Todas las operaciones, desde el acceso a cada posición hasta el producto y suma acumulada, son de orden constante.

2. Esta sentencia se encuentra en un bucle (línea 4) que itera n veces. La inicialización, incremento y condición son de orden constante, así que el bucle en conjunto tiene una eficiencia $O(n)$.

3. La asignación de la línea 3 y el bucle de la línea 4 se ejecutan de forma consecutiva, y por tanto, su eficiencia corresponde a la suma $O(1) + O(n)$, es decir, de orden lineal.

4. Estas sentencias se encuentran dentro del bucle de la línea 2, que itera n veces. La inicialización, incremento y condiciones son de orden constante. La eficiencia por tanto será el producto del número de iteraciones por el orden de las sentencias del cuerpo. En este caso, $O(n^2)$.

5. Finalmente, y de forma similar al caso anterior, tenemos un bucle (línea 1) que contiene como cuerpo un código de orden cuadrático. Este bucle se ejecuta n veces, y por tanto, el algoritmo tiene una eficiencia de $O(n^3)$.

Este mismo código podríamos haberlo analizado considerando que la línea 5 (de orden constante) es la que más veces se ejecuta. Ésta está dentro de tres bucles anidados

que iteran n veces. Por tanto, la línea se ejecuta del orden de n^3 veces, o lo que es lo mismo, que el código es de eficiencia cúbica.

1.4.2. Algoritmo de búsqueda binaria

Un algoritmo clásico sobre un vector ordenado es el de búsqueda binaria o dicotómica. Para buscar un determinado elemento se divide el vector en dos partes y se consulta si el elemento está en la parte izquierda o derecha (dependiendo del orden con el elemento central). El algoritmo selecciona la parte del vector donde debería estar el elemento, y vuelve a repetir el proceso hasta que el vector se hace de tamaño cero o se encuentra el elemento. Una función que lo implementa es la siguiente:

```
1  int Busqueda (const int v[], int n, int x)
2  {
3    int izq=0, der=n-1;
4    int centro;
5
6    while (izq<=der) {
7      centro= (izq+der)/2;
8      if (x<v[centro])
9        der= centro-1;
10     else if (x>v[centro])
11             izq= centro+1;
12           else return centro;
13   }
14   return -1;
15 }
```

Para analizar este código tenemos que tener en cuenta que la ejecución puede dar lugar a distintos tiempos. Por un lado, y como era de esperar, el algoritmo depende del tamaño n del problema. Por otro, para un mismo tamaño podemos tener distintos resultados, ya que el número de pasos que son necesarios para ejecutar la función depende del elemento x buscado. Por ejemplo, si x se encuentra justo en el centro del vector, el bucle se ejecuta sólo una vez, ya que alcanza la sentencia *return* de la línea 12. En este caso, sólo se han necesitado unas pocas operaciones de orden constante, y el tiempo del algoritmo sería $O(1)$.

Sin embargo, este análisis correspondería al *mejor caso*. Por supuesto, nosotros estamos especialmente interesados en el *peor caso*. Para ello, supondremos que el bucle *while* se ejecuta el máximo número de iteraciones posible, es decir, que el algoritmo no llega a encontrar el elemento.

Las líneas internas del bucle (*7-12*) son $O(1)$. Por tanto, la eficiencia del bucle viene determinada por el número de iteraciones. El problema consiste en calcular este número. En principio, parece que no es un problema trivial, pero es fácil ver que en cada iteración del bucle, el tamaño del *subvector [izq,der]* se reduce a la mitad. Si consideramos que cada iteración implica una reducción del vector a la mitad, y suponemos que el tamaño del vector inicial es una potencia de *2* (por ejemplo, 2^m), la secuencia de iteraciones provoca unos tamaños de $2^m, 2^{m-1}, 2^{m-2}, \cdots, 1$. Por tanto, el número de iteraciones para

que el tamaño se reduzca hasta que el bucle termine es del orden de *m*, o lo que es lo mismo, $log_2(n)$.

Finalmente, teniendo en cuenta que las sentencias externas al bucle son de orden constante, podemos decir que la función es de orden logarítmico.

1.4.3. Algoritmo de ordenación por selección

El siguiente código implementa el algoritmo de ordenación de un vector por el método de selección:

```
1 for (i=0;i<n-1;++i) {
2   minimo= i;
3   for (j=i+1;j<n;++j)
4     if (A[j]<A[minimo])
5       minimo= j;
6   temporal= A[minimo];
7   A[minimo]= A[i];
8   A[i]= temporal;
9 }
```

Para analizar su eficiencia, realizamos un análisis similar a los ejemplos anteriores. Sin embargo, en este ejemplo tenemos una dificultad añadida, ya que el bucle interior (línea 3) no tiene un número de iteraciones fijo.

La sentencia condicional de la línea 4 es de orden constante, y por tanto, la eficiencia del bucle interior corresponde directamente al número de iteraciones que realiza. El problema es que el número de iteraciones es variable, ya que depende de *i*, es decir, de la variable contador del primer bucle. Cuando *i=0*, el número de iteraciones es del orden de *n*, y cuando vale *n-2* es de orden constante.

Si queremos realizar un análisis rápido, y puesto que deseamos obtener una cota superior en peor caso, podemos suponer que el número de iteraciones del bucle es de orden *n*. Teniendo en cuenta que el resto de operaciones que componen el cuerpo del bucle principal (líneas 2-8) son de orden constante, la eficiencia de ese trozo de código es de orden lineal ($O(1) + O(n)$). Puesto que el bucle de la línea 1 itera del orden de *n* veces, el algoritmo es de orden cuadrático ($O(n) \cdot O(n)$).

Ahora bien, sabemos que esto es una cota superior, ya que hemos supuesto que el bucle interno es $O(n)$ en cualquier caso. Si estamos interesados en conocer la eficiencia exacta (Θ), debemos contar exactamente el número de operaciones que realmente se realiza. Para esta análisis, podemos calcular el número de veces que se ejecuta la sentencia condicional de la línea 4:

1. El cuerpo del bucle interno (que contiene la sentencia condicional) se ejecuta $n - (i + 1)$ veces.

2. El cuerpo del bucle externo (que contiene el bucle anterior) se ejecuta para todos los valores de i desde el 0 hasta el $n - 2$.

Por tanto, la línea 4 se ejecuta:

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

Es decir, la eficiencia del trozo de código es $\Theta(n^2)$.

1.5. Problemas

Problema 1.1 *Pruebe que las siguientes sentencias son verdad:*

- 29 *es* $O(1)$
- $\frac{n(n-1)}{2}$ *es* $O(n^2)$ *y* $\Omega(n^2)$ *(es decir,* $\Theta(n^2)$*)*
- $max(n^3, c \cdot n^2)$ *es* $O(n^3)$ *(*$c \in \Re^+$*)*
- $log_2 n$ *es* $\Theta(log_3 n)$
- $f(n) \in O(g(n))$ *y* $g(n) \in O(h(n)) \Longrightarrow f(n) \in O(h(n))$

Problema 1.2 *Ordene de menor a mayor los siguientes órdenes de eficiencia:*
n, $\sqrt{n}$, n^3+1, n^2, $nlog_2(n^2)$, $nlog_2log_2(n^2)$, $3^{log_2(n)}$, 3^n, 2^n, 20000, $n+100$, $n2^n$

Problema 1.3 *Supongamos que* $T_1(n) \in O(f(n))$ *y* $T_2(n) \in O(f(n))$*. Razone la verdad o falsedad de las siguientes afirmaciones:*

a.- $T_1(n) + T_2(n) \in O(f(n))$

b.- $T_1(n) \in O(f^2(n))$

c.- $T_1(n)/T_2(n) \in O(1)$

Problema 1.4 *Demuestre la siguiente jerarquía de órdenes de complejidad:*

$$O(1) \subset O(log(n)) \subset O(n) \subset O(n^2) \subset O(2^n) \subset O(n!)$$

Problema 1.5 *Obtenga, usando la notación O-mayúscula, la eficiencia de la función* Insercion *(ordena un vector usando el método de inserción).*

```
void Intercambiar(int& a, int& b)
{
  int temp=a; a=b; b=temp;
}
void Insercion(const int v[], int n)
{
  for (int i=1;i<n;++i)
    if (v[i]<v[0]) intercambia(v[0],v[i]);
```

```
  for (int i=2;i<n;++i)
  {
    int j=i;
    int aux= v[i];
    while (aux<v[j-1])
    {
      v[j]= v[j-1];
      j--;
    }
    v[j]= aux;
  }
}
```

Problema 1.6 *Obtenga, usando la notación O-mayúscula, la eficiencia de la función* Burbuja *(ordena un vector usando el método de la burbuja).*

```
void Burbuja (const int v[], int n)
{
  for (int i=0;i<n-1;++i)
    for (int j=n-1; j>i; --j)
      if (v[j-1]>v[j]) {
        int aux= v[j-1];
        v[j-1]=v[j];
        v[j]=aux;
      }
}
```

Capítulo 2

Tipos de datos abstractos en programación

En este capítulo

2.1. Introducción

La abstracción es un proceso mental que consiste en realzar los detalles relevantes, es decir, los que nos interesan en un momento sobre el objeto de estudio, mientras se ignoran los detalles irrelevantes. Esto nos lleva a una simplificación del problema, ya que:

- la cantidad de información que es necesario manejar en un momento dado disminuye y
- podemos tratar cosas diferentes como si fueran la misma.

Este proceso de la mente humana es fundamental para comprender y manejar complejos sistemas que contienen múltiples detalles y relaciones. Por ello, y dada la complejidad de los programas actuales, la evolución de los paradigmas y lenguajes de programación marca un uso creciente de la abstracción.

La única forma de abordar un problema complejo es separar la solución en distintas partes, es decir, distintos módulos que resuelven subproblemas. El programa será más sencillo de manejar cuanto mayor sea la independencia entre estos módulos, es decir, minimizando la relación de unos módulos con otros. Para cada uno de ellos, determinamos la forma en que podemos hacer uso de su funcionalidad, indicando la interfaz para poder acceder a ella, y ocultando las partes internas que no son necesarias para utilizarla.

En la figura 2.1 se presenta gráficamente esta idea. En la parte izquierda representamos un módulo, que corresponde a un programa donde no se ha aplicado la abstracción. Por tanto, todos los elementos que lo componen pueden estar relacionados. Su desarrollo y mantenimiento es mucho más complejo, ya que en cada punto tenemos que tener en cuenta múltiples detalles que pueden afectar a lo que se esté haciendo.

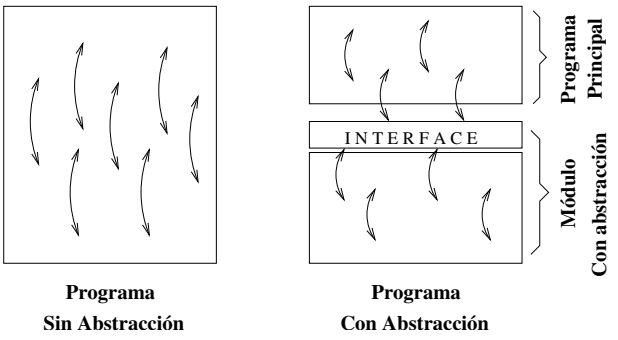


Figura 2.1
Minimización de relaciones por medio de la abstracción.

Sin embargo, en la parte derecha se ha aplicado la abstracción, desarrollando un módulo que oculta sus detalles del programa principal por medio del uso de una interfaz. Tanto el programa principal como la parte interna del módulo se pueden modificar de forma independiente si se mantiene la misma interfaz.

Así, una vez determinada la interfaz de un módulo, el desarrollo interno de éste es independiente del resto del programa. Además, este programa puede usar la funcionalidad del módulo, independientemente de los detalles internos de su implementación. El programa que usa el módulo sólo debe conocer la interfaz (también lo denominaremos la parte pública), mientras no necesita conocer la parte interna de implementación (también la denominaremos la parte privada).

Lógicamente, esta metodología constituye una aplicación directa de la abstracción, en donde el programa utiliza los detalles relevantes de un módulo, obviando los detalles internos de cómo se ha desarrollado.

Finalmente, en el desarrollo de software y en concreto de módulos, debemos documentar el desarrollo para poder usarlo y mantenerlo. Recordemos que tenemos que diferenciar dos partes, la parte *pública* (interfaz) y la parte *privada* (implementación). En este sentido, el programador debería crear dos documentos diferenciados:

1. **Especificación**. Corresponde al documento que presenta las características sintácticas y semánticas que describen la parte pública. Éste debiera ser suficiente (no necesitamos más información para poder usar todas las posibilidades del módulo) e independiente de los detalles internos de implementación.

2. **Implementación**. Corresponde al documento que presenta las características internas del módulo. Es importante destacar que el software es algo dinámico, por tanto, es necesario mantener un sistema de documentación que permita el mantenimiento, tanto en lo que respecta a diseño como a implementación.

En esta lección se presenta un estudio más profundo de la abstracción de datos, enmarcada en una metodología de descomposición y desarrollo de módulos basada en la abstracción, consiguiendo con ello mejorar la calidad de los resultados y facilitar el desarrollo, modificación y mantenimiento del software.

2.2. Abstracción funcional

El primer tipo de abstracción que históricamente aparece es la **funcional o procedimental**, que surge a partir de separar el propósito de una función de su implementación.

Se considera el *qué* de una función, obviando el *cómo* se implementa esa función. Así, los algoritmos complejos que requieren muchas operaciones se engloban en una sola, eliminando todos los detalles de cómo se implementan. En la abstracción funcional o procedimental abstraemos un conjunto preciso de operaciones (detalles de *cómo* se realiza) como una única operación en forma de función o procedimiento.

Por ejemplo, si en el programa que se desarrolla es necesario calcular el índice del elemento máximo que se almacena en un vector, podemos olvidar todos los detalles relativos a cómo se debe calcular, las variables que se deben usar, etc (detalles irrelevantes

para el uso de la operación) y considerar que disponemos de una operación *IndiceMaximo* que se ocupa de resolverlo (detalles relevantes). Así, el código correspondiente podría ser el presentado en la tabla 2.1.

Tabla 2.1
Operación IndiceMaximo

Operación	`int IndiceMaximo (const int vector[], int nelem)`

Detalles	`{`
	`  int i,max;`
	`  max=0;`
	`  for (i=1;i<nelem;i++)`
	`  if (vector[max]<vector[i])`
	`  max= i;`
	`  return max;`
	`}`

Para usar este módulo sólo necesitamos conocer la sintaxis y semántica de su interfaz, las cuales vienen documentadas en la especificación. De esta forma podemos ignorar los detalles de la implementación (parte privada).

Un ejemplo de la interfaz completa (sintaxis más semántica) de la función *IndiceMaximo* es el siguiente:

```
int IndiceMaximo (const int vec[], int nelem)
/*
  Argumentos:
    vec: array 1–D que contiene los elementos donde encontrar
         el valor máximo.
    nelem:  número de elementos en el vector 'vec'.
  Devuelve:
    índice en el vector 'vec'
  Precondiciones:
    nelem>0
    vec tiene al menos 'nelem' elementos
  Efecto:
    Busca el elemento más grande en el vector 'vec' y devuelve la
    posición de éste, es decir, el valor i tal que v[i]<=v[j]
    para 0<=j<nelem
*/
```

Si especificamos el módulo para que pueda ser procesado por alguna herramienta automática, como *doxygen*, podemos escribirlo de la siguiente forma:

```
/**
*  @brief Calcula índice del elemento máximo de un vector
*  @param vec vector con los elementos donde encontrar el máximo.
*  @param nelem número de elementos en el vector \a vec. \a nelem>0
*  @pre \a vec es un vector de al menos \a nelem elementos
```

```
*  @return el índice del elemento más grande en el vector \a vec ,
*          es decir, el valor \e i tal que \e vec[i]<=vec[j]
*          para \e 0<=j<nelem
*/
int IndiceMaximo (const int vec[], int nelem)
```

2.2.1. Un ejemplo: Motivación

Considere que se desea desarrollar un programa que lea dos polinomios de la entrada estándar y escriba la suma en la salida estándar.

Una posible solución, de un programador sin experiencia, podría ser escribir directamente todo el código en la función *main*, para resolver el problema de la forma más "rápida", sin preocuparse del diseño.

Para ello, en primer lugar, decide representar un polinomio por medio de un vector, de forma que el elemento i-ésimo almacena el coeficiente del monomio de grado *i*. Por ejemplo, el polinomio $x^6 - 3x^2 + 7$ se representa en un vector *p*, como muestra la figura 2.2.

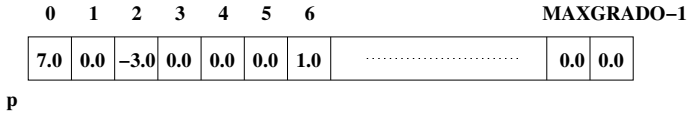


Figura 2.2
Representación de un polinomio en un vector.

La solución del problema consiste en declarar dos vectores que almacenarán los dos polinomios que se leen, y un tercer vector para almacenar el resultado. El código podría ser el que sigue:

```
#include <iostream>
using namespace std;

int main()
{
  const int MAXGRADO = 100;
  float p[MAXGRADO],q[MAXGRADO],res[MAXGRADO];

  // ------- Inicializamos -------
  for (int i=0;i<MAXGRADO;++i)
    p[i]= q[i]= res[i]= 0; // Inicialmente valen cero

  // --------- Leemos ---------
  cout << "Introduzca Primer Polinomio" << endl;
  for (int i=0;i<MAXGRADO;++i) {
    cout << "Coeficiente de grado " << i << ':';
    cin >> p[i];
  }
```

```
  cout << "Introduzca Segundo Polinomio" << endl;
  for (int i=0;i<MAXGRADO;++i) {
    cout << "Coeficiente de grado " << i << ':';
    cin >> q[i];
  }

  // --------- Sumamos ---------
  for (int i=0;i<MAXGRADO;++i)
    res[i]= p[i]+q[i];

  // --------- Resultado ---------
  cout << "Polinomio Suma: "
  int grado=0;
  for (int i=MAXGRADO-1;i>=0 && grado==0;i--)
    if (res[i]!=0.0)
      grado= i;
  cout << "Grado: " << i << endl;

  for (int i=0;i<MAXGRADO;++i)
    if (res[i]!=0.0)
      cout << i << ' ' << res[i] << endl;
}
```

donde podemos ver que *p*,*q* son los polinomios de entrada, y *res* el de salida. Seguro que el lector puede proponer varias modificaciones para mejorar este programa. Algunas características a tener en cuenta son:

- El programa es poco legible. Si queremos estudiar el programa para entender el algoritmo que implementa, es necesario un tiempo considerable (teniendo en cuenta la simplicidad del problema) ya que aparecen todos los detalles "mezclados" en una misma función.
- Si el programa falla, es más difícil encontrar el error. Todo el código está relacionado, y por tanto, buscarlo requiere estudiar todo lo que hemos escrito, teniendo en cuenta todos los detalles que existen.
- Es más difícil modificar el programa. Por ejemplo, si queremos usar otro algoritmo para realizar la suma, de nuevo tenemos que "enfrentarnos" a todos los detalles de este código.
- Si queremos escribir otro programa, tenemos que escribirlo desde cero. Por ejemplo, tal vez necesitemos hacer la suma de dos polinomios, que tendremos que reescribir a pesar de haberlo resuelto ya.

Por tanto, podemos pensar en otra solución que resuelva, al menos en parte, algunos de estos problemas. Por ejemplo, podemos dividir la solución en varias funciones, de forma que creamos un módulo (*polinomio.cpp*) que contiene la implementación de las operaciones sobre polinomios. Un posible listado de este módulo sería:

```
#include <iostream>
#include "polinomio.h"
using namespace std;

// Hace p cero
void Anular (float p[], int n)
{
  for (int i=0;i<n;++i) p[i]= 0;
}

// Lee p con n coeficientes
void Leer (float p[], int n)
{
  for (int i=0;i<n;++i) {
    cout << "Coeficiente de grado " << i << ':';
    cin >> p[i];
  }
}

// res=p+q
void Sumar (float res[], const float p[], const float q[], int n)
{
  for (int i=0;i<n;++i) res[i]= p[i]+q[i];
}

// Escribe p que tiene n coeficientes
void Escribir (const float p[], int n)
{
for (int i=0;i<n;++i)
    if (p[i]!=0.0) cout << i << ' ' << p[i] << endl;
}

// Devuelve el grado de p.
int Grado (const float p[], int n)
{
  int grado=0;
  for (int i=n-1;i>=0 && grado==0;i--)
    if (p[i]!=0.0) grado= i;
  return grado;
}
```

Así, nuestro problema se podría resolver de la siguiente forma:

```
#include "polinomio.h"

int main()
{
  const int MAXGRADO = 100;
  float p[MAXGRADO],q[MAXGRADO],res[MAXGRADO];

  Anular(p); Anular(q); Anular(res);
```

```
    cout << "Introduzca Primer Polinomio" << endl;
    Leer (p,MAXGRADO);
    cout << "Introduzca Segundo Polinomio" << endl;
    Leer (q,MAXGRADO);

    Sumar (res,p,q,MAXGRADO);

    cout << "Polinomio Suma: "
    cout << "Grado: " << Grado(p,MAXGRADO) << endl;
    Escribir(res);
}
```

donde obtenemos un mejor diseño. Por ejemplo, revisemos algunos de los problemas comentados:

- El código es más fácil de leer. Por ejemplo, la función *main* presenta una lectura mucho más sencilla.
- Es más fácil buscar errores. Podemos estudiar las funciones de forma independiente, de manera que probarlas o confirmar si son correctas es mucho más simple.
- Facilitamos la modificación. Por ejemplo, si deseamos un algoritmo distinto para la suma (imagine el uso de aritmética de punteros), podemos centrarnos en la función que lo implementa sin necesidad de tener en cuenta el resto del programa.
- Es más fácil reusar el código. Si otro programa necesita realizar alguna de las operaciones que usamos, no tenemos más que enlazar con el primero de los módulos.

Sin duda, y a pesar de las mejoras introducidas, se pueden proponer nuevas modificaciones. Para motivar un nuevo diseño, considere que se desea realizar una modificación en la forma de representar un polinomio. Por ejemplo, podemos plantear una aplicación donde se realizan operaciones con polinomios de alto grado. Imagine que queremos sumar x^{2457} y x^{5372}. Esta operación parece trivial y muy rápida, sin embargo, con el código antes propuesto:

1. Es muy ineficiente en tiempo, pues tenemos que recorrer todos los grados (incluso los que no afectan al valer cero).
2. La cantidad de memoria requerida es muy alta para almacenar un simple monomio.

En este sentido, se hace necesaria una modificación del código para poder realizar este tipo de operaciones en un tiempo y espacio más pequeño. Para ello, podríamos proponer que para manejar un polinomio sólo sea necesario almacenar los coeficientes distintos de cero. Es decir, basta con almacenar las parejas *grado-coeficiente* cuando el coeficiente sea distinto de cero. En la figura 2.3 se muestran dos ejemplos de polinomios en esta nueva representación.

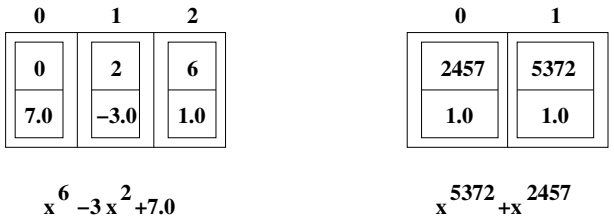


Figura 2.3
Representación de un polinomio como un vector de monomios.

Es obvio que esta nueva representación implica la reescritura de todo el código que hemos realizado. Imagine que hubiéramos desarrollado miles de programas que usan el módulo de funciones sobre polinomios. Todos ellos dejarían de ser válidos con la nueva representación.

El problema surge porque, aunque hemos creado el módulo *polinomio.cpp* para que sea independiente de nuestro problema y pueda reutilizarse, la relación entre este módulo y el resto sigue siendo muy fuerte, ya que dependen de una representación muy concreta para el polinomio. Si deseamos un diseño que facilite este tipo de modificaciones, deberíamos plantear una solución que hiciera a los programas independientes de esta representación. La solución, como veremos a continuación, la encontramos en los *tipos de datos abstractos*.

2.3. Tipos de datos abstractos

En la historia de la programación, los algoritmos son cada vez más complejos, al igual que los datos manejados. Por ello, también se aplica la abstracción a éstos, que podemos denominar **abstracción de datos**. En este caso, la abstracción consiste en un nuevo tipo de dato, junto con unas operaciones, que constituyen la única manera de manejarlo. La idea es usar el nuevo tipo de dato, ignorando detalles referentes a su representación (cómo se almacena) y su manejo (cómo se implementan las operaciones).

En primer lugar, es interesante destacar que los lenguajes de alto nivel ya ofrecen una solución con los tipos simples (por ejemplo *double*, *int*, etc) y un conjunto de operaciones asociadas a ellos. Note que:

- Para manejarlos, el usuario no necesita conocer los detalles de la representación, o la forma en que se realizan las operaciones. Sólo es necesario conocer un “manual” (que luego denominaremos, especificación).

- Para implementarlos (es decir, para que el compilador disponga del tipo de dato y las operaciones), no es necesario conocer dónde se van a usar. Sólo es importante respetar que el comportamiento sea el especificado (según el “manual” o especificación del tipo de dato).

Cuando la información a manejar en un programa es más compleja, el compilador no dispone de tipos de datos para facilitarnos el trabajo. Por ejemplo, si necesitamos usar fechas en un programa de gestión bancaria, el compilador no tiene, por ejemplo, un tipo de dato *Fecha* que facilite la programación. Surge así el desarrollo de **tipos de datos abstractos** que, recordemos, son nuevos tipos de datos con un grupo de operaciones que proporcionan la única manera de manejarlos. De esta forma, debemos conocer las operaciones que se pueden usar, pero no necesitamos saber:

- la forma en que se almacenan los datos ni
- cómo se implementan las operaciones.

Por ejemplo, si en el programa que se desarrolla es necesario gestionar fechas, es más complejo que el programador tenga que manejar, simultáneamente, los detalles del programa que se realiza junto con los detalles de cómo se almacena y se opera con fechas. En este caso, es conveniente crear un nuevo tipo de dato, que podemos denominar *Fecha*, junto con un conjunto de operaciones para manejarlo (ver tabla 2.2).

Tabla 2.2
Ejemplo de operaciones de un T.D.A. Fecha

Tipo	*Fecha*
Operaciones	*void AsignarFecha(Fecha &f, int d, int m, int a)* *int DiaFecha(Fecha f)* *int MesFecha(Fecha f)* *int AgnoFecha(Fecha f)* *Fecha IncrementarFecha(Fecha f, int dias)* *int DiferenciaFecha(Fecha f1, Fecha f2)* *int DiaSemanaFecha(Fecha f)*

Por ejemplo, supongamos que quiero saber los días que han pasado entre dos fechas. Podemos realizar un programa como el siguiente:

```
Fecha f1,f2;
int d,m,a;

cin >> d >> m >> a;
AsignarFecha(f1,d,m,a);
cin >> d >> m >> a;
AsignarFecha(f2,d,m,a);
cout << "Han pasado " << DiferenciaFecha(f1,f2) << " días" << endl;
```

En este programa no es necesario conocer los detalles internos de implementación del tipo *Fecha* y sus operaciones, únicamente es necesario un “manual” (*especificación*) de cómo usarlo.

En la figura 2.4 aparece un esquema de la solución, donde podemos ver que el programa se hace independiente de la implementación, por medio de la abstracción.

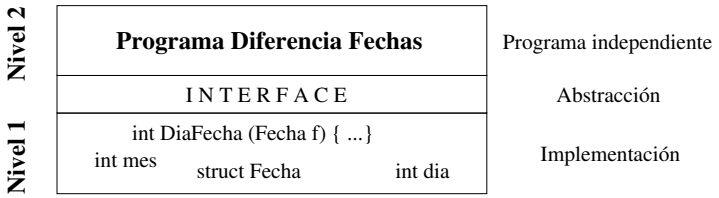


Figura 2.4
Abstracción en el programa de diferencia de fechas.

Por supuesto, la implementación (nivel 1 de la figura 2.4) tendría distintas alternativas (respectando la misma *interfaz*). Por ejemplo, una estructura:

```
struct Fecha{
  int dia;
  int mes;
  int anio;
};
```

o un simple entero codificando la distancia en días desde cierta fecha conocida:

```
typedef int Fecha;
```

Igualmente, podríamos tener una variedad de maneras para codificar cada una de las operaciones, incluso para una determinada representación.

De forma similar a la abstracción funcional, esta multitud de detalles del tipo de dato abstracto son irrelevantes para el resto del programa. Para resolver un problema que usa fechas, sólo necesito conocer el nuevo tipo de dato y una *especificación* de cómo usarlo.

Por tanto, podemos establecer la siguiente definición:

> **Definición 2.1 (Tipo de dato abstracto)** . *Un tipo de dato abstracto (T.D.A.) es una colección de valores, junto con unas operaciones sobre ellos, definidos mediante una especificación que es independiente de cualquier implementación.*

Las ventajas del uso de tipos de datos abstractos son claras:

1. *Facilidad de uso*. No necesitamos conocer los detalles internos del tipo de dato abstracto, sólo la especificación, que es mucho más simple.

2. *Desarrollo y mantenimiento*. Desarrollar y modificar el código es mucho más sencillo, ya que si determinamos una interfaz, cualquier cambio interior al tipo de dato abstracto no afecta al resto del programa. De forma similar, cualquier cambio en el programa que siga respetando la interfaz no afecta al tipo de dato abstracto. Además, localizar los errores es mucho más simple, ya que es más fácil su aislamiento.

3. *Reusabilidad*. El tipo de dato abstracto se puede usar en distintos programas.

4. *Fiabilidad*. El código es más fiable, ya que es más fácil realizar pruebas sobre los módulos de forma independiente. Por ejemplo, podemos realizarlas sobre los T.D.A. hasta un grado de fiabilidad muy alto, de forma que todos los programas que lo usen pueden suponer que esa parte de la solución es correcta.

Por supuesto, aplicar esta metodología en un problema no se limita a definir un tipo de dato abstracto, sino que podemos aplicar la abstracción múltiples veces. Así, un T.D.A. puede estar compuesto de varios módulos, hacer uso de la interfaz con otro módulo o con otro T.D.A., etc.

2.3.1. Un ejemplo. El T.D.A. *Matriz*

Ejemplo 2.3.1 Se desea calcular el valor de $(a + b) \cdot (a + b)^t$, donde a, b son dos matrices de iguales dimensiones, y las operaciones que aparecen corresponden a la suma matricial (+) el producto de matrices ($\cdot$) y la traspuesta (t) de una matriz. Proponga una modularización haciendo uso de tipos de datos abstractos.

El programa lee dos matrices y escribe el resultado de la operación indicada. Para ello, deberá leer las dimensiones de las matrices, junto con cada uno de los elementos que las componen. Parece claro que el uso de memoria dinámica es lo más indicado.

Una primera (y realmente mala) solución, es resolver el problema directamente con todo el código en la función *main*. Imagine por un momento el caos que se generaría con todos los detalles mezclados, incluyendo reservas de memoria, variables auxiliares para los cálculos, los nombres de los campos de las estructuras, etc. Si pensamos un momento en la calidad de la solución en términos de facilidad de desarrollo, mantenibilidad, reusabilidad, fiabilidad, es fácil entender que no es muy adecuada.

Podemos pensar en una segunda solución, en la que incorporamos la abstracción funcional para mejorar la primera. Ahora podemos encapsular la dificultad de algunos algoritmos en una función. Por ejemplo, para leer los datos de una matriz, podemos usar:

```
void LeerMatriz(double **& datos, int& filas, int& columnas);
```

Sin embargo, esta interfaz es bastante "incómoda", puesto que siempre estaremos arrastrando esos tres datos con cada matriz. Además, si queremos modificar la representación (por ejemplo, usando un *double* * en lugar *double* **) cambiarían las cabeceras (las interfaces) de las funciones. Así, otra solución para facilitar el manejo de matrices puede tener en cuenta un nuevo tipo de dato, por ejemplo con una estructura como:

```
struct Matriz{
    double **datos;
    int filas, columnas;
};
```

para almacenar una matriz en memoria dinámica, usando un vector de punteros que apuntan a cada una de las filas. Con ella, creamos funciones para resolver algunos algoritmos necesarios, como leer los datos de una matriz, sumar dos matrices, reservar

memoria para una matriz, etc. Sin duda, esta solución es mucho mejor que la anterior. Sin embargo, pensemos en algunas situaciones que se pueden presentar:

- Después del desarrollo, la ejecución falla porque la matriz a escribir tiene un valor del campo *columnas* que vale cero. ¿En qué parte del código se ha puesto a cero? Cualquier función de las que hemos realizado puede tener acceso a ese campo, y por tanto, cambiarlo.
- Decidimos modificar la implementación, haciendo que el campo *datos* sea un puntero a todos los datos por filas. En este caso, debemos revisar y modificar todo nuestro programa, ya que en cualquier sitio se ha podido usar el "puntero a puntero" de la anterior representación.
- Queremos usar este código para desarrollar otro programa que usa matrices. En este caso, tendremos que revisar las funciones que pueden seguir siendo válidas, tal vez modificar algunas para adaptarlas a nuestro nuevo problema, y crear otras. Además, si decidimos cambiar algo básico, como la representación interna, necesitaremos reescribir todo el código.

Para mejorar el diseño, podemos optar por disminuir aún más las relaciones entre distintas partes del código. Para ello, podemos proponer una modularización del problema en dos partes:

1. Un módulo para resolver el problema de la representación y gestión de matrices.
2. Un módulo para que, usando el anterior, resuelva el problema concreto.

El primero consiste en un tipo de dato abstracto, es decir, que ofrece una interfaz y oculta los detalles de la implementación. El segundo, usa la interfaz y es independiente de los detalles internos del primero. Más concretamente, podemos proponer una interfaz que incluye las funciones que aparecen en la tabla 2.3. Un pequeño esbozo de su especificación es:

- *CrearMatriz*. Prepara una matriz de *fxc* elementos. Por ejemplo, reserva memoria e inicializa los valores.
- *FilasMatriz*. Devuelve el número de filas de una matriz.
- *ColumnasMatriz*. Devuelve el número de columnas de una matriz.
- *SetMatriz*. Modifica un elemento de una matriz.
- *GetMatriz*. Devuelve un determinado elemento de una matriz.
- *DestruirMatriz*. Libera los recursos ocupados por una matriz.