ANTONIO GARRIDO CARRILLO

PROGRAMACIÓN GENÉRICA EN C++

la biblioteca estándar

GRANADA
2017

Programación Genérica en C++: la biblioteca estándar
ISBN: 978-84-338-6078-1.
Depósito legal: GR.805-2017
Edita: Editorial Universidad de Granada.
Campus Universitario de Cartuja. Granada.
Imprime: Gráficas La Madraza. Albolote. Granada.

Printed in Spain *Impreso en España*.

Índice general

1 Análisis de la eficiencia 1

1.1 Introducción 1

1.1.1 Tamaño del problema 1

1.1.2 Algoritmos vs implementaciones 2

1.2 Eficiencia de algoritmos 2

1.2.1 Familias de órdenes de eficiencia 2

1.2.2 Notación asintótica 3

1.2.3 Eficiencia en tiempo y espacio 9

1.2.4 Elección del mejor algoritmo 9

1.3 Análisis de algoritmos 10

1.3.1 Operación elemental 10

1.3.2 Caso peor, caso promedio y análisis amortizado 11

1.3.3 Reglas para el cálculo de la eficiencia 14

1.4 Ejemplos 17

1.4.1 Algoritmo de multiplicación de matrices 18

1.4.2 Algoritmo de búsqueda binaria 18

1.4.3 Algoritmo de ordenación por selección 19

1.4.4 Mejor caso de ordenación por inserción 21

1.4.5 Algoritmo de mezcla de dos vectores 21

1.4.6 Algoritmo de ordenación por mezcla 22

1.5 Problemas 24

2 Encapsulación 27

2.1 Abstracción 27

2.1.1 Mecanismos de abstracción 28

2.2 C++98 vs C++11/14 28

2.2.1 Inicialización homogénea con llaves 29

2.2.2 Deducción del tipo con auto 32

2.2.3 Semántica de movimiento 33

2.3 Clases en C++ 40

2.3.1 Abstracción e implementación de un vector dinámico 40

2.3.2 Ocultar o no ocultar 44

2.3.3 Funciones miembro especiales 46

2.3.4 Generación o eliminación explícita con *delete* 51

2.4 Ejemplos 54

2.4.1 El tipo *Vector* 55

2.4.2 El tipo *Conjunto* 58

2.4.3 El tipo *Polinomio* 61

2.5 Problemas 63

3 Generalización: Plantillas 65

3.1 Introducción 65

3.2 Funciones plantilla en C++ 66

3.2.1 Tipo genérico de la plantilla: typename y class 67

3.2.2 Encadenar llamadas de funciones plantilla 67

3.2.3 Especificación explícita del tipo *T* 68

3.3 Clases plantilla 69

3.3.1 Definición de los métodos de la clase 70

3.3.2 Tipos asociados 72

3.3.3 Funciones miembro plantilla 75

3.3.4 Funciones y clases plantilla amigas de una clase plantilla 76

3.4 Plantillas y compilación separada 77

3.4.1 Inclusión de las definiciones 79

3.4.2 Instanciación explícita 83

3.4.3 Modelo mixto de inclusión e instanciación explícita 85

3.4.4 Compilación separada con *export* 89

3.5 Compatibilidad del tipo base en la instanciación 89

3.5.1 Conceptos 90

3.6 Número y tipo de parámetros plantilla 92

3.6.1 Número de parámetros 92

3.6.2 Tipo de parámetros 94

3.6.3 Valores por defecto 97

3.6.4 Especializaciones 98

3.7 Otros aspectos relacionados con C++14 100
3.7.1 Intercambiar y máximo en la biblioteca estándar . . . 100
3.7.2 Constructores y miembros *constexpr* . . . 102
3.7.3 Alias con *using* . . . 103
3.7.4 Eliminando alternativas con *delete* . . . 104
3.7.5 Listas de inicialización . . . 105
3.7.6 Constructor delegado . . . 107
3.7.7 Literales definidos por el usuario . . . 108
3.8 Problemas 109

4 Contenedores, iteradores y algoritmos genéricos . . . 111
4.1 Introducción 111
4.2 Posición: Un primer diseño 112
4.2.1 Acceso con índices . . . 113
4.2.2 Abstracción de la posición . . . 117
4.3 Iteradores: conceptos básicos con vectores-C 123
4.3.1 Punteros e iteración . . . 123
4.3.2 Iteración de sólo lectura: const . . . 125
4.3.3 Conversión de iteradores . . . 127
4.4 Diseño de iteradores para contenedores 128
4.4.1 Iteración sobre *Vector* y *Conjunto* . . . 128
4.4.2 Contenedores no modificables con iteradores . . . 130
4.4.3 Invalidación de iteradores . . . 131
4.4.4 Tipos de iteradores . . . 132
4.5 Algoritmos genéricos 135
4.5.1 Rangos con iteradores . . . 137
4.5.2 Características de tipos: *Type-traits* . . . 141
4.5.3 Conceptos e iteradores . . . 148
4.6 Algunos aspectos relacionados con C++14 148
4.6.1 Simplificar con *auto* . . . 148
4.6.2 Bucles basados en rango . . . 149
4.7 Problemas 150

5 Objetos llamables . . . 153
5.1 Introducción 153
5.1.1 Punteros a función . . . 154
5.1.2 Punteros a miembro . . . 154
5.2 Objetos función 156
5.2.1 Objetos función con estado . . . 157
5.2.2 Parámetros de plantilla . . . 160
5.2.3 Paso y devolución de objetos función . . . 166
5.2.4 Conceptos y objetos función . . . 168
5.2.5 Objetos función de la biblioteca estándar . . . 170

5.3 Expresiones lambda 171
5.3.1 Sintaxis de una expresión lambda . . . 172
5.3.2 Captura del entorno . . . 174
5.4 Nuevas formas de definir funciones 179
5.4.1 Uso de *decltype* . . . 180
5.4.2 Deducción automática del tipo devuelto . . . 182
5.4.3 Lambdas genéricos . . . 182
5.4.4 Deducción *auto* vs *decltype* . . . 183
5.5 Utilidades de la biblioteca estándar 185
5.5.1 Empaquetar un puntero a miembro en un *functor* . . . 185
5.5.2 Empaquetar funciones y argumentos: *std::bind* . . . 187
5.5.3 Un tipo para *gobernarlos* a todos: *std::function* . . . 190
5.6 Problemas 193

6 Contenedores y algoritmos del estándar . . . 195
6.1 Introducción 195
6.1.1 Claves de la referencia . . . 195
6.2 Contenedores 198
6.2.1 Tipos de contenedores . . . 199
6.2.2 Contenedores secuenciales . . . 201
6.2.3 Contenedores asociativos con orden . . . 205
6.2.4 Contenedores asociativos sin orden . . . 211
6.3 Iteradores: de contenedores a algoritmos 217
6.3.1 Contenedores y tipos de iteradores . . . 217
6.3.2 Iteradores: inserciones, borrados, búsquedas... . . . 221
6.3.3 Funciones útiles para iteradores . . . 223
6.4 Comparación de contenedores: operaciones 224
6.4.1 Contenedor como concepto . . . 226
6.4.2 Contenedores secuenciales . . . 229
6.4.3 Contenedores asociativos . . . 231
6.4.4 Otras operaciones . . . 234
6.5 Algoritmos 237
6.5.1 Algoritmos de la biblioteca estándar de C++ . . . 238
6.6 Problemas 250

7 Adaptadores . . . 253
7.1 Introducción 253
7.2 Adaptadores de contenedores 254
7.2.1 El adaptador *stack* . . . 254
7.2.2 El adaptador *queue* . . . 256
7.2.3 El adaptador *priority_queue* . . . 258

7.3 Adaptar a la interfaz de iterador **261**
7.3.1 Iteradores *istream*/*ostream* . . . 261
7.3.2 Iteradores *reverse* . . . 265
7.3.3 Iteradores de inserción . . . 267
7.3.4 Iteradores de movimiento . . . 270
7.3.5 Funciones para generación de adaptadores . . . 270
7.4 Problemas **271**

8 Metaprogramación . . . 273
8.1 Introducción **273**
8.2 Metafunciones **274**
8.2.1 Metaprogramación y *plantillas* . . . 274
8.2.2 Metaprogramación y *constexpr* . . . 275
8.2.3 Devolución con plantillas: *value* y *type* . . . 276
8.3 Plantillas... revisitado **279**
8.3.1 Deduciendo el tipo . . . 279
8.3.2 Referencias universales . . . 282
8.3.3 Paquetes de parámetros . . . 283
8.3.4 SFINAE . . . 287
8.4 Condicionales y bucles **288**
8.4.1 Condicionales . . . 289
8.4.2 Bucles . . . 295

A Solución a los ejercicios . . . 299
A.1 Eficiencia **299**
A.2 Encapsulación **302**
A.3 Plantillas **306**
A.4 Contenedores **309**
A.5 Objetos llamables **314**
A.6 Contenedores y algoritmos **320**
A.7 Adaptadores **324**

B Tablas . . . 329
B.1 Tabla ASCII **329**
B.2 Operadores C++ **330**
B.3 Palabras reservadas de C89, C99, C11, C++ y C++11 **332**
B.4 Referencia de contenedores **332**
B.4.1 Contenedores vs métodos . . . 333
B.4.2 Operaciones de los contenedores . . . 336

B.5 Referencia de algoritmos 351
B.5.1 Algoritmos clasificados 351
B.5.2 Algoritmos por orden alfabético 358
B.6 Referencia de objetos función 365

Bibliografía 367
Referencias electrónicas 370

Índice alfabético 371

Prólogo

El objetivo de este libro es presentar conceptos sobre programación genérica en un contexto basado en la abstracción. La exposición se hará en base al lenguaje C++ y la biblioteca estándar de C++, lo que permite que el lector conozca mejor este lenguaje y los detalles de cómo se podrían implementar los tipos que ofrece.

Como consecuencia, el estudiante entenderá mejor no sólo la interfaz de la biblioteca, sino también los detalles sobre cómo puede sacar el máximo rendimiento de ésta. Tener conocimientos básicos sobre las estructuras de datos y la metodología que se ha usado para crearla facilita en gran medida su uso, ya que:

- Conocer detalles sobre qué hace una operación permite usarla con precisión, siendo conscientes de las ventajas y desventajas que puede conllevar.
- Estudiar la metodología en el desarrollo de la STL permite entender mejor el porqué de la interfaz. No sólo descubrimos que es cómoda sino que incluso somos capaces de inferir qué otras capacidades probablemente nos ofrecerá.

Por tanto, abandonamos en parte la necesidad de estudiar *de memoria* especificaciones o manuales sobre su uso, siendo nuestros conocimientos y nuestra lógica la que nos guíe para obtener lo mejor de la STL.

Lo que no es este libro es un manual de la biblioteca estándar de C++. Existen muchos manuales de referencia que el lector puede usar y que pueden resultar más simples y eficaces, como son las referencias en la red (véanse las referencias [55] o [54], por ejemplo). Es absurdo dedicar esfuerzo y espacio a algo que ya está escrito en otro lugar. En este texto, nos centramos en los fundamentos para entender esas referencias. A pesar de ello, se incluirá una información mínima que haga del libro una referencia autocontenida. Es posible desarrollar con la información limitada que se ofrece aunque si tiene alguna duda sobre algún detalle, deberá recurrir a una referencia completa.

Biblioteca estándar de C++ vs STL

Existe cierta controversia en la comunidad de programadores C++ sobre un uso correcto de las dos expresiones: *C++ Standard Library* y *Standard Template Library (STL)*. Normalmente no me

gusta incluir discusiones sobre nombres y clasificaciones, son más adecuados para esquemas o manuales de referencia, pero es necesario puntualizar —aunque sea en el prólogo— sobre un cierto abuso del acrónimo STL en este libro.

Ciertamente, la STL se creó hace décadas —fundamentalmente por Alexander Stepanov— como una gran contribución a la programación genérica y que fue la base para buena parte del contenido del estándar. Es interesante indicar que son dos cosas distintas, especialmente después del último estándar de C++, en el que las nuevas contribuciones a la biblioteca estándar de C++ hacen que el nombre STL sea aún más distante.

La confusión es consecuencia de que la biblioteca estándar hace un uso intensivo de las plantillas en particular y la programación genérica en general, ideas que ya se estaban explotando en las implementaciones de la STL. Teniendo en cuenta que el diseño original de la biblioteca estándar se realizó, en parte, casi duplicando buena parte de las primeras implementaciones de la STL —tenga en cuenta que ya se conocía y usaba por un gran número de programadores C++— el resultado es que muchos programadores usan STL como una forma simple y corta de expresarse.

En este libro, seguiremos usando STL por simplicidad. Aunque de alguna forma se pueda considerar una expresión incorrecta, es importante enfatizar que muchos de los conceptos que se presentan cuando estudiamos contenedores, algoritmos o iteradores son los que originalmente se incluyeron en las primeras implementaciones de la biblioteca, antes de que se completaran y se hicieran parte de la biblioteca estándar.

En general, el acrónimo STL sigue siendo válido y debería considerarse como una forma adecuada para referirse tanto a la tecnología que se creó originalmente como a la versión ampliada que actualmente se está usando (véase, por ejemplo, Josuttis [21]).

A quién va dirigido el libro

Este libro se ha planteado para un estudiante de programación, más que para un programador experimentado; no es un libro para que un programador que conoce conceptos avanzados en otros lenguajes conozca la sintaxis de C++. Para eso existen excelentes referencias, entre las que pueden incluirse manuales de referencia que no entran a explicar los conceptos que sustentan el diseño de la biblioteca.

Por tanto, se plantearán las lecciones persiguiendo que el lector amplíe conocimientos con conceptos y aspectos que mejoren su capacidad para resolver problemas de programación. Por supuesto, un efecto colateral será que el estudiante podrá entender mejor cómo usar C++ en general y la STL en particular.

En concreto, estudiar nuevas formas de abstracción como son la *programación genérica* y la *abstracción por iteración* confiere al lector una nueva perspectiva para el diseño de soluciones. Constituye un nuevo paso para un estudiante que tiene conocimientos básicos de programación y necesita añadir nuevas bases que le permitan avanzar hacia un nivel de programación más elevado.

Tenga en cuenta que un buen libro sobre la biblioteca estándar de C++ podría ser muy amplio, pues requiere manejar múltiples campos. Por ejemplo, el libro Josuttis[21] es una buena referencia, pero requiere una formación mínima, lo que permite al autor exponer detalles sobre la biblioteca sin necesidad de explicar conceptos que, aun siendo avanzados, se suponen conocidos por el lector.

Por otro lado, una exposición general sobre estructuras de datos resulta muchas veces lejana a casos reales, no porque no sea adecuada, sino porque el estudiante no entiende bien cómo podría llevarse a la práctica y cómo podrían mejorar su forma de programar. Un estudio más concreto asociado a una biblioteca como la STL permite aportar una visión muy pedagógica pero también muy real de cómo se podrían usar complejas estructuras de una forma eficaz y simple. Sin duda, es una aportación muy positiva a la hora de entender conceptos de diseño y metodología de programación.

El desarrollo de estos contenidos, donde se mezclan conceptos básicos, aspectos pedagógicos y detalles de implementación está especialmente adaptado a un estudiante de programación al que se le suponen unos conocimientos mínimos de C++, incluyendo el diseño y desarrollo de clases. Si este no es su caso, puede consultar las referencias Garrido[1] y Garrido[2], con las que podrá adquirirlos.

Es posible que esté pensando en este libro como una segunda parte de las referencias mencionadas y con el que podrá completar su formación en este lenguaje. No, no es el caso. Ser un experto programador en C++ es una tarea muy compleja que requiere de un tiempo de formación relativamente alto. Este libro no es más que una nueva etapa, una nueva capa en la formación del estudiante que cubre aspectos tan importantes como la abstracción y las estructuras de datos y que permite abrir nuevas puertas para abordar, con más garantías, temas más avanzados.

Los diseños que se proponen en este libro esquivarán la programación orientada a objetos (POO) , más concretamente, la herencia y el polimorfismo. Es probable que encuentre implementaciones de estos tipos, por ejemplo, aprovechando la herencia para reutilizar código en distintas representaciones. Desde un punto de vista práctico, es una mejor solución, aunque desde un punto de vista pedagógico no resulta especialmente relevante para los objetivos de este libro. Aunque podemos añadir algún comentario sobre una solución con herencia, no son necesarios para realizar este curso, se incluyen solamente para completar la discusión y que el lector sea consciente de una visión global del problema.

Cuando estudie temas de POO, le resultará sencillo revisar y entender esas posibles alternativas. Verá que los fundamentos son los mismos, aunque los detalles de la implementación la harían más difícil de seguir y muy poco apropiada para un libro. En cualquier caso, si desea echarle un vistazo —por ejemplo, a la implementación de la biblioteca estándar de C++ del compilador de la *GNU*— le recomiendo que antes lea algo sobre POO en C++.

Organización del libro

El libro tiene un objetivo claramente docente; se ha organizado como continuación de las referencias Garrido[1] y Garrido[2], del mismo autor. No es un libro en el que recopilar múltiples listas de tipos, algoritmos, utilidades, etc. de la biblioteca estándar de C++. Es necesario incluir buena parte de esta información para que el lector comprenda la amplitud y posibilidades de lo que ofrece, pero no es el objetivo principal. Se incluirá información, pero la mínima para que el estudiante asimile los contenidos.

Tampoco es objetivo de este curso obtener implementaciones ideales, puesto que los ejemplos deben ser ilustrativos de los conceptos que se muestran, evitando dar detalles innecesarios ni complicaciones que no aportarían mucho a la lección. En este punto, no aporta demasiado ver los detalles que optimizan las clases hasta el límite. Tenga en cuenta que una implementación real de los tipos aúna la experiencia, no sólo de muchos programadores, sino también de un nivel excepcional.

Con este libro no se cierra el estudio del lenguaje C++. No es objetivo revisar todo el lenguaje. Por ejemplo, se van a evitar las jerarquías de clases —que se dejan para un curso posterior— o gestión de errores mediante excepciones. Aun así, se irán añadiendo algunas anotaciones que permitan al estudiante ir adaptándose y familiarizándose con distintos aspectos del lenguaje que en principio podrían ser demasiado confusos. Al final del curso, le será mucho más fácil entender implementaciones reales, probablemente añadiendo algunos conocimientos mínimos sobre programación dirigida a objetos.

Los capítulos se han organizado para que el lector adquiera gradualmente los conocimientos relacionados con la programación genérica en C++, comenzando con temas simples de creación de clases y terminando con la abstracción propia que requiere la metaprogramación en este lenguaje.

En el **capítulo 1** se presentan las bases para entender las discusiones sobre eficiencia que se darán a lo largo de todo el libro. Uno de los objetivos del diseño de tipos genéricos es crear implementaciones eficientes. Es fundamental que el estudiante tengo unos conocimientos básicos, aunque no se presentarán problemas de análisis de eficiencia especialmente complejos.

En el **capítulo 2** se revisan los conocimientos sobre desarrollo de clases que se usarán en los temas siguientes. Suponemos que el lector tiene cierta experiencia, por lo que la lección no incluirá todos los detalles. Los contenidos se centran especialmente en los puntos más delicados —los que normalmente generan más dudas— así como en la exposición de algunos ejemplos sencillos que sirven como soporte para los temas siguientes. Además, se revisan las aportaciones de C++14 para poder ofrecer una visión más actualizada, especialmente si se incluyen conocimientos más avanzados sobre metaprogramación, pero necesarios también para poder usar correctamente las implementaciones actuales de la STL.

En el **capítulo 3** entramos en uno de los grandes pilares de la programación genérica: las plantillas. Suponemos que el estudiante no sabe nada sobre ellas. La exposición comienza desde lo más básico. Es absolutamente necesario entender perfectamente cómo funcionan. Es un tema largo, con muchos detalles y que requiere que el estudiante practique ampliamente creando programas con plantillas. A pesar de su longitud, no se abarcarán todos los posibles detalles, sólo los más importantes y suficientes para entender cómo funciona la STL.

En el **capítulo 4** se presenta el diseño de la STL como la unión de contenedores, iteradores y algoritmos genéricos. Los primeros como plantillas que parametrizan los tipos que contienen, los segundos como un resultado de la abstracción por iteración que comparten los contenedores y los terceros como plantillas que aprovechan esta abstracción para crear soluciones que son válidas e independientes de los contenedores. Se aportan discusiones para reflexionar sobre el diseño y entender el porqué de la solución propuesta en la biblioteca.

En el **capítulo 5** se han reunido las distintas formas de objetos llamables que se proponen en el lenguaje. Es difícil situar estos contenidos en otros temas, porque necesitan de conocimientos que van desde la sobrecarga de operadores básica para una clase hasta discusiones sobre generalización que aparecen en el último estándar. Sin embargo, es en este punto —después de revisar el diseño de los algoritmos genéricos— donde resulta más fácil apreciar las posibilidades que ofrecen y situarlos en el lugar adecuado en el contexto de la biblioteca estándar. Tenga en cuenta que se incluyen comentarios avanzados sobre las funciones como un objeto más a manejar, incluyendo las expresiones lambda de C++11.

En el **capítulo 6** volvemos al estudio de los contenedores y los algoritmos de la biblioteca estándar. Ahora no nos centramos en el diseño de las distintas partes de la biblioteca como en el capítulo 4, sino en una exposición razonada sobre lo que ofrece. Suponemos que los conocimientos del estudiante sobre estructuras de datos son mínimos, por lo que se razona sobre la variedad de estructuras que ofrecen los contenedores, sobre cómo resuelven las implementaciones y cómo afectan a los tipos y operaciones con iteradores. Gran parte de la aportación se relaciona con estructuras de datos, aunque sin entrar en detalles de implementación. En este punto, el lector ya sabe cómo se ha diseñado la STL, qué contenedores ofrece y con qué capacidades, así como algunas ideas básicas que le permiten decidir sobre el diseño de sus programas. Resta mostrar de forma muy resumida los algoritmos que ofrece la biblioteca estándar de C++ para disponer de todos los ingredientes para crear sus primeras soluciones a problemas relativamente complejos.

En el **capítulo 7** se presentan los adaptadores. En los temas anteriores han aparecido algunas pinceladas sobre ellos, puesto que son tipos disponibles en la biblioteca, pero sin entrar en cómo se han diseñado, puesto que no son propiamente contenedores. No se han integrado en ese diseño genérico que les permite su integración en la iteración y los algoritmos de la STL. Sin embargo, son una excelente oportunidad de enfatizar la adaptación como un patrón de diseño que encontrará repetidamente a lo largo de su carrera como programador.

En el **capítulo 8** se presenta la metaprogramación. En principio, es un tema algo avanzado para un libro docente de este nivel. Sin embargo, no podemos exponer los componentes más importantes de la programación genérica excluyendo algunas de las más importantes aportaciones del estándar. Sin este tema, el estudiante podría encontrarse perdido en algunos códigos o textos que tratan sobre programación en C++14. No es objetivo exponer con detalle este tipo de programación, pero es conveniente que se muestren algunas de las tecnologías que la facilitan así como unas guías generales que muestren en qué consiste esta forma de programar.

Además, se incluyen dos apéndices que completan el libro:

- En el **Apéndice A** se incluyen las soluciones de los ejercicios propuestos a lo largo de los temas. Es recomendable que se estudie cada tema y se revisen estos ejercicios, primero intentando solucionarlos y segundo, independientemente del resultado, consultando la solución que se propone. Los problemas que aparecen al final de los temas son propuestos para el estudiante interesando en practicar, sin soluciones en el texto.
- En el **Apéndice B** se incluyen tablas y referencias. Encontrará algunas tablas básicas, como el código *ASCII* o los operadores de C++, pero también la referencia a contenedores y algoritmos de la biblioteca estándar. De nuevo, de una forma reducida, pero especialmente útil para que el lector disponga de un material autocontenido. Por un lado se amplía los detalles sobre las operaciones de los contenedores y por otro se presentan los algoritmos clasificados y por orden alfabético. La intención es apoyar al estudiante en la práctica, ofreciendo una referencia a modo de *chuleta* que le facilite crear programas basados en la STL.

Finalmente, se presenta la bibliografía, donde aparece la lista de referencias que se han usado para la creación de este libro o que resultan de especial interés para que el lector amplíe conocimientos. Además, se incluye un índice alfabético que facilite al lector navegar no sólo por los conceptos más relevantes, sino también por la gran cantidad de nombres que ofrece la biblioteca estándar de C++.

Sobre la ortografía

El libro se ha escrito cuidando la redacción y la ortografía; no podría ser de otra forma. Se ha intentado cuidar especialmente la expresión y la claridad en las explicaciones. Como parte de este cuidado, es necesario incluir en este prólogo una disculpa por no respetar todas y cada una de las normas que actualmente establece la Real Academia Española (RAE) sobre el español.

En primer lugar, es difícil concretar un lenguaje claro y duradero en el contexto de la informática, especialmente porque cambia e incorpora nuevos conceptos continuamente. Unido al retraso que requiere el análisis y la decisión por parte de la RAE para incluirlos en el diccionario de la lengua española, siempre podemos encontrar alguna palabra que se adapta desde el inglés y no está aún aceptada. Por ejemplo, usaremos la palabra *unario* como traducción de *unary*. Otros palabras que usamos de forma natural cuando comentamos código en C++ pueden ser *booleano, token, buffer, precondición, endianness, log, argumento, preincremento, prompt,* etc.

Por otro lado, es habitual que el programador trabaje con documentación técnica que está escrita en inglés. Esto hace que un uso de palabras similares en español sea una forma de facilitar la lectura rápida y cómoda. Por ejemplo, la palabra *indentación* en lugar de *sangrado*, *casting* en lugar de *moldeado* o *array* en lugar de *arreglo*.

Sin duda, mis cambios son un error; pero desde un punto de vista técnico prefiero no darle excesiva importancia. Es más importante superar la dificultad de los contenidos técnicos, dejando cuestiones de la lengua en un segundo plano.

A pesar de todo, he querido ajustarme en lo posible a lo que nos dice la RAE; por respeto al estupendo y complejo trabajo que realizan, pero sobre todo por respecto a nuestra lengua materna, el principal legado común que nuestros ancestros nos han dejado.

Una vez certificado mi respeto a la lengua y a la RAE como institución, tengo que disculparme por introducir alguna palabra u ortografía que no es actual. Esta disculpa se refiere, especialmente, al uso de la tilde diacrítica en el adverbio *sólo* —note la tilde— y los *pronombres demostrativos*. En mi humilde opinión, la distinción gráfica con respecto al adjetivo *solo* y los *determinantes demostrativos*, respectivamente, justifican claramente su uso. Si quiero priorizar la legibilidad, la claridad, el énfasis de lo que comunico, creo que su uso está justificado.

Por supuesto, seguramente habrá otros errores ortográficos o gramaticales; éstos son fruto de mi torpeza e ignorancia, por lo que pido humildemente disculpas.

Agradecimientos

En primer lugar, quiero 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 puedo olvidar a las personas que me 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, quiero agradecer a mis estudiantes 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.

Tengo que confesar que son ellos, los alumnos que desean aprender, los que llegan con una inmensa ilusión por recibir los conocimientos de un supuesto experto en un tema, los que me animan a seguir trabajando en este tipo de material; especialmente si tenemos en cuenta que no sólo no se premia, sino que prácticamente no se reconoce este trabajo. Sirva como una humilde aportación a lo que me hubiera gustado encontrar cuando llegué a la universidad.

A. Garrido
Junio de 2017

1 Análisis de la eficiencia

Introducción 1
Eficiencia de algoritmos 2
Análisis de algoritmos 10
Ejemplos 17
Problemas 24

1.1 Introducción

Cuando se desea resolver un problema es necesario seleccionar o diseñar un algoritmo, ya sea porque disponemos de varias soluciones o porque podemos diseñar una nueva. Todas ellas funcionarán; por lo que es posible que parezca irrelevante —o al menos poco importante— seleccionar una u otra.

Unos algoritmos serán más lentos que otros, consumirán más memoria que otros o, en general, requerirán más recursos que otros. Incluso con esta distinción, nos puede parecer indiferente la solución finalmente escogida, pues 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 lo analizamos 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. Por ejemplo, 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. Por 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 no negativos ($\Re_0^+$).

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, no a las implementaciones.

1.2 Eficiencia de algoritmos

El tiempo de ejecución de un algoritmo depende del tamaño del problema. Además, si queremos medir cuánto tiempo necesita cierto tamaño tendremos que realizar una implementación. Ésta implementación tardará un tiempo que depende del programador, del programa e incluso del sistema donde se ejecuta. Es necesario establecer exactamente a qué nos referimos con *eficiencia de un algoritmo*.

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 va a 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 actualizado puede optimizar mejor el resultado de la compilación.
- 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, no de implementaciones.

[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).

El principio de invarianza nos permite resolver el problema:

Definición 1.1 — 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*. La consecuencia es que 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 habituales, 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*, que incluye a $f(n) = n^2$.
- Orden *logarítmico*, que incluye funciones de la forma $f(n) = \log n$.
- Orden *exponencial*, que se refiere a que el orden incluye alguna función del tipo X^n, con $X > 1$.
- 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.

Si lo que nos interesa es decir que un algoritmo es mejor que otro, tal vez no sea necesario determinar exactamente qué eficiencia tienen, sino simplemente que uno está por encima del otro. Para poder expresar esta relación cómodamente, creamos una notación. Además, esta notación la denominamos *asintótica*, porque esta diseñada especialmente para expresar el comportamiento cuando n —el tamaño del problema— aumenta. Se diseña así teniendo en cuenta que:

- No distinguimos entre tiempos que difieren en una constante multiplicativa (recuerde el principio de invarianza). 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].

[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 siempre valores positivos.

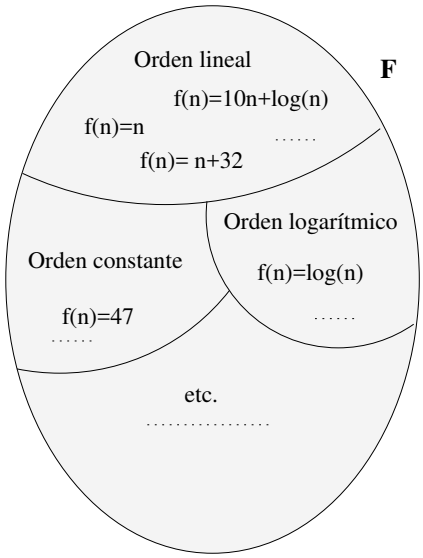


Figura 1.1
Partición del conjunto de las funciones en órdenes de eficiencia.

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

  Es decir, con un ordenador dos veces más rápido, el primer algoritmo nos permitiría resolver —en el mismo tiempo— un problema 100 veces más grande. Sin embargo, el segundo algoritmo prácticamente no aprovecha esta mejora en la velocidad. 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:

Definición 1.2 — 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 $\forall n \geq n_0 \quad g(n) \leq c \cdot f(n)$

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 valor 100. 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.

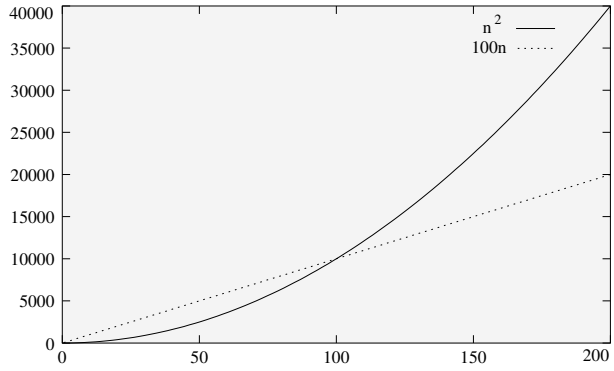


Figura 1.2
Comparación de n^2 y de $100n$.

Es importante darse cuenta de que el orden que estamos estableciendo es sólo parcial, es decir, podemos encontrar dos funciones que no pueden ser ordenadas. Consideremos por ejemplo las funciones:

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

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, lo que nos lleva a su clase de eficiencia. Si tenemos

[3] Recuerde el principio de invarianza y la equivalencia de funciones. Note que esta definición coincide con ese principio cuando determina que dos funciones son iguales (es decir, están en la misma clase).

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

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 queremos comparar dos algoritmos, tendremos que analizarlos para determinar a qué clases pertenecen. Una vez analizados, será posible decir que un algoritmo es mejor que otro puesto que tenemos ordenadas las distintas familias de órdenes de eficiencia.

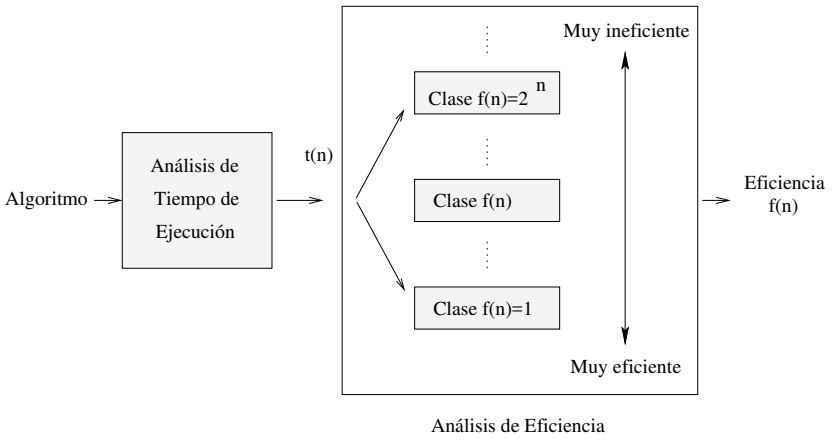


Figura 1.3
Análisis de eficiencia de algoritmos.

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 $\Theta(\log_3 n)$

Desafortunadamente, en la práctica es posible que el análisis del tiempo de ejecución de algunos algoritmos sea muy complejo. 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, sin que nos importe demasiado 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 $O(n^2)$. Usando la notación de conjuntos: $(n+1)^2 \in O(n^2)$.

- 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 puede ver, se han representado como elipses distintas clases de eficiencia, desde la más eficiente a la menos eficiente. Observe que:

- $O(f(n))$ 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 ($f(n)$ es una cota inferior).
- Y finalmente, $\Theta(f(n))$ es el conjunto de funciones que están exactamente en la misma clase que $f(n)$.

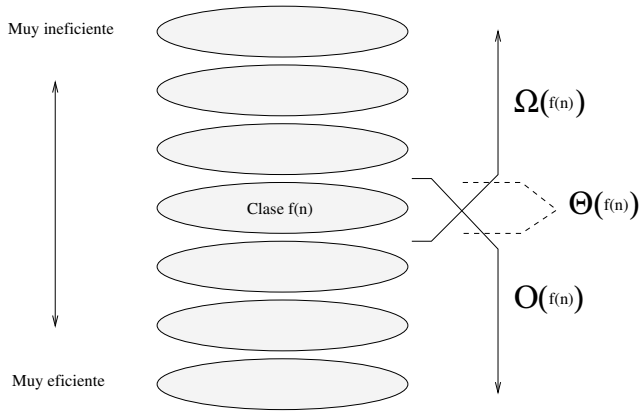


Figura 1.4
Notaciones O, Ω, Θ.

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

Definición 1.3 — 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:

$$O(f(n)) = \{g : N \to \Re_0^+ / \exists c \in \Re^+, n_0 \in N \text{ tal que } \forall n \geq n_0 \; g(n) \leq c \cdot f(n)\}$$

De forma similar, la notación Ω:

Definición 1.4 — Notación Ω-mayúscula. Sea una función $f : N \to \Re_0^+$. El conjunto de las funciones $\Omega(f(n))$ se define como:

$$\Omega(f(n)) = \{g : N \to \Re_0^+ / \exists c \in \Re^+, n_0 \in N \text{ tal que } \forall n \geq n_0 \; g(n) \geq c \cdot f(n)\}$$

Finalmente, la notación Θ a partir de las definiciones 1.3 y 1.4

Definición 1.5 — Notación Θ-mayúscula. Sea una función $f : N \to \Re_0^+$. El conjunto de las funciones "del orden exacto f(n)", denotado por $\Theta(f(n))$ se define como:

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

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.

Precisamente, el uso de estas notaciones como cotas superiores e inferiores se muestra más útil al poder expresar los órdenes de eficiencia a pesar de esta parcialidad. Un ejemplo ilustrativo para mostrar la conveniencia de las notaciones O y Ω es considerar la siguiente función:

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

De la que podemos decir —por ejemplo— que tiene al menos la eficiencia de $\Theta(2^n)$ y como mucho es tan eficiente como $\Theta(n)$, es decir, es $O(2^n)$ y $\Omega(n)$. Aunque no podemos decir nada con respecto a la relación con funciones como n^3, $n\log n$, $n\sqrt{n}$, etc.

En este punto es interesante enfatizar ya la importancia de la notación *O-mayúscula*. Con ésta, establecemos una cota superior, es decir, si decimos que un algoritmo es $O(f(n))$ estamos afirmando que como mucho tiene un perfil de tiempo de ejecución $f(n)$.

Operaciones entre órdenes de eficiencia

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

Definición 1.6 — 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:

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

Definición 1.7 — 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:

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

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 que necesita. Por consiguiente, sería un grave 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, es decir, la elección sigue la eficiencia teórica que desprecia esas constantes. 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 —es de orden exacto $\Theta(n)$— mientras que el segundo es cuadrático ($\Theta(n^2)$). A pesar de ello, es posible que en la práctica el primero pueda ser más recomendable. Podría ocurrir, por ejemplo, si:

- El tamaño de los problemas a resolver no va a pasar de 100. En este caso, el tiempo del primero es menor que el del 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. Esta situación se da, especialmente, cuando se dispone ya de un algoritmo suficientemente bueno y no se quiere invertir en diseñar nuevas alternativas.

Por tanto, la elección en la práctica no implica necesariamente la elección de la mejor eficiencia teórica. En muchos casos, podemos optar por realizar las implementaciones más simples y más fáciles de mantener, dejando para una etapa posterior el análisis del programa para localizar las partes que requieren de una mejora.

Sin embargo, a pesar de todo, cuando los tamaños de los problemas crezcan, probablemente los algoritmos más ineficientes empezarán a generar problemas de ineficiencia, siendo los más eficientes los que mejor se adapten al crecimiento del tamaño del problema. Desde este punto de

vista, podemos decir que son más fáciles de mantener pues se adaptan mejor a las necesidades futuras del programa.

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 1.8 — 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 finalmente 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 operaciones 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:

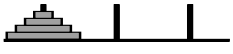
$$c_1\cdot n\leq t\leq c_2 n$$

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;
```

[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.



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 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

En 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

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

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.

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, ya que en media se realizan menos operaciones (el tiempo en media siempre será menor o igual que el peor caso).

Se ha presentado este ejemplo por simplicidad, aunque el estudio en media nos lleve a la misma eficiencia. Existen otros algoritmos más complejos, para los que descubrirá 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 debería ajustarse lo más posible a la realidad.

[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.

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 $m \cdot f(n)$. 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.

Para intentar mostrar una idea intuitiva de esto, consideremos el problema de almacenar libros en un contenedor. Este ejemplo nos permite intuir, evitando detalles sobre código, por qué el tipo **`vector`** de la STL —que hace de contenedor— tiene un tiempo amortizado de $O(1)$ para la operación de añadir elemento al final.

El problema consiste en que se dispone de una caja con capacidad para n libros y la operación que queremos analizar es la de *introducir un libro* más. 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.

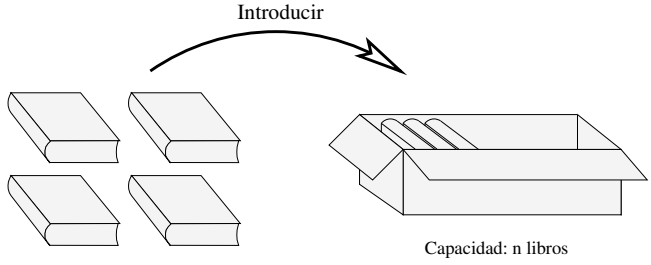


Figura 1.5
Operación *introducir* un libro.

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 está llena— hay que trasladar cada libro a la nueva caja (n operaciones elementales). Por lo tanto, podríamos decir que m operaciones consecutivas necesitan $m \cdot n$ 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. Recuerde que cada ejecución que no llena la caja necesita sólo una operación elemental.

Para entenderlo, considere que se realizan n operaciones consecutivas *introducción de un libro*. Obviamente, al llegar a n, una nueva operación tendrá que usar otra caja (de capacidad $2n$) y trasladar uno a uno todos los libros que ya habíamos introducido. Para que ocurra esto, antes ya 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 —en media— 2 movimientos. Es decir, la operación tiene un coste amortizado de $O(1)$.

En este ejemplo, el análisis se ha realizado de una manera intuitiva para que se entienda la importancia y el sentido de este tipo de cálculo, 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[6] y algunos ejemplos sobre estructuras de datos avanzadas en Weiss[53], así como resultados prácticos en Mehlhorn[25]. 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 y 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 eficiencias $O(f_1(n))$ y $O(f_2(n))$, 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 8), es igual a $O(\text{máx}\{f_1(n), f_2(n)\})$.

Selección

En este apartado analizamos las sentencias **`if`**, **`if`**−**`else`** y **`switch`**; éstas corresponden a la selección simple, doble y múltiple, respectivamente.

En las dos primeras es necesario evaluar una expresión booleana 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. Si la sentencia es **`if`** simple, supondremos que la condición es **`true`**. En el caso de **`if`**−**`else`** tendremos que seleccionar, de entre los dos posibles caminos, el de peor eficiencia $O(f(n))$.

El resultado es que la eficiencia total es la suma de ambas, que como sabemos, corresponde a $O(\max\{E(n), f(n)\})$.

Para el caso de la sentencia **`switch`** el razonamiento es similar, pues primero tendrá que evaluar la expresión del **`switch`** y luego tiene que seleccionar uno de los casos. Se encadenará, por tanto, el tiempo de esa evaluación con el peor caso de entre las distintas alternativas.

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 $O(Cpo(n))$.

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:

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

  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).

> **Ejercicio 1.1** Considere el siguiente trozo de código que anula los elementos de un vector de tamaño n:
>
> ```
> for (int i=0; i<n; ++i)
>   v[i]= 0;
> ```
>
> ¿Cuál es el tiempo de ejecución del bucle?

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

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

> **Ejercicio 1.2** Considere el siguiente trozo de código donde buscamos un elemento x en un vector de tamaño n:
>
> ```
> v[n]= x;
> int i= 0;
> while (v[i]!=x)
>   i++;
> ```
>
> ¿Cuál es su tiempo de ejecución?

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

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

Para obtener la expresión final se aplicarán las reglas de la suma y el producto (página 8).

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 d.
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 c al tiempo que necesitan esas operaciones $O(1)$, el tiempo total de ejecución será $T(n-1)+c$.

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[31], Brassard[6], etc. se pueden consultar distintos métodos de resolución.

Ejercicio 1.3 ¿Cuál es la eficiencia en espacio de la función anterior?

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.

Ejercicio 1.4 Considere el siguiente trozo de código que escribe todos los números que van desde 1 hasta $n!$:

```
for (int i=1; i<Factorial(n); ++i) {
   cout << i << ' ';
```

Si suponemos que escritura en la salida estándar es $O(1)$, ¿cuál es su tiempo de ejecución?

1.4 Ejemplos

La mejor forma de entender los conceptos que se han expuesto en las secciones anteriores es mostrar ejemplos y comprobar cómo, en la práctica, es fácil deducir la eficiencia sin muchas discusiones teóricas o deducciones matemáticas. Veamos algunos ejemplos sencillos.

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 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:

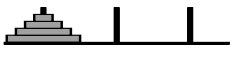
1. En primer lugar podemos considerar la línea *5*, donde acumulamos un valor en la posición `[i][j]` de la matriz resultado. Todas las operaciones, desde el acceso a cada posición hasta el producto y suma acumulada, son de orden constante.
2. Esta sentencia se encuentra en un bucle (línea 4) que itera `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. Por tanto, $O(n^2)$.
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 de multiplicación 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 respecto al 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 v, de tamaño n, 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 (v[j]<v[minimo])
5          minimo= j;
6    tmp= v[minimo];
7    v[minimo]= v[i];
8    v[i]= tmp;
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 ($O(n)\cdot O(n)$). En la figura 1.6 se muestra un esquema gráfico de este análisis.

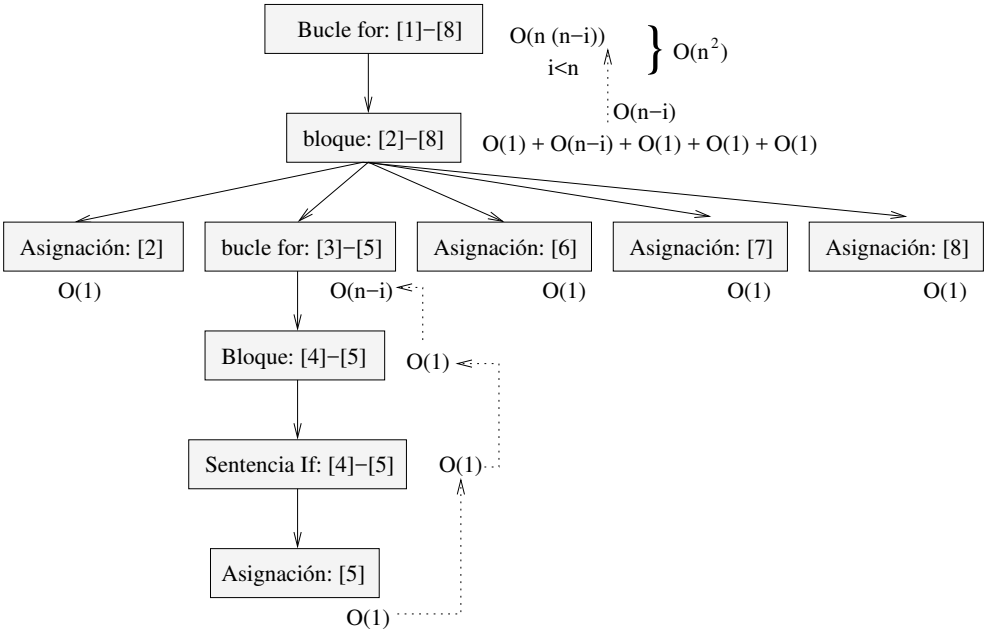


Figura 1.6
Esquema del análisis del algoritmo de selección.

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 ($\Theta(n)$), debemos contar exactamente el número de operaciones que realmente se realiza. Para este 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 $\Theta(n^2)$.

Observe que la expresión que hemos obtenido —en la que dividimos entre 2— es bastante intuitiva. El bucle interior se ejecuta desde 1 vez hasta $n-1$. Si lo hacemos para todos los valores, en media se ejecuta aproximadamente $n/2$ veces. Note la forma de exposición, donde ignoramos valores exactos, constantes y hablamos de número aproximado de iteraciones. Realmente, en la práctica nos interesa una eficiencia teórica, por lo que podemos realizar todas estas simplificaciones si sabemos que no va a afectar a la función que finalmente obtenemos como orden de eficiencia.

1.4.4 Mejor caso de ordenación por inserción

Para analizar el mejor caso de este algoritmo, estudiamos la siguiente implementación que ordena un vector de n objetos de tipo **double**:

```
1  void OrdenarInsercion (double v[], int n)
2  {
3    double x;
4    int j;
5
6    for (int i=1;i<n;++i){
7      x= v[i];
8      for (j=i;j>0 && x<v[j-1];--j)
9        v[j]= v[j-1];
10     vector[j]= x;
11   }
12 }
```

Si estudia las condiciones que deben darse para que el algoritmo haga el mínimo de operaciones, verá que la clave está en que el bucle interior —en la línea 8— se pare inmediatamente porque la condición sea **false**. Eso ocurre, concretamente, cuando el vector a ordenar ya está ordenado.

En el mejor caso, el bucle no entra, por lo que el bloque de líneas 7-10 no son más que una operación elemental, pues son varias asignaciones simples y la comprobación de la condición del bucle. Por consiguiente, la eficiencia la determina el número de iteraciones del bucle de la línea 6. En concreto, la condición del bucle se evalúa n veces, por lo que la eficiencia es del orden de n.

Note que es una evaluación en mejor caso. En este sentido, podría decirse que este algoritmo es $\Omega(n)$ para cualquier caso. También podríamos decir que es, exactamente, $\Theta(n)$ para el caso concreto de un vector ordenado. Normalmente, dado el uso habitual de la notación *O-mayúscula*, se suele indicar simplemente que el algoritmo es $O(n)$ en mejor caso.

Ejercicio 1.5 ¿Cuál es la eficiencia en mejor caso del algoritmo de selección de la sección 1.4.3?

1.4.5 Algoritmo de mezcla de dos vectores

El algoritmo de mezcla recibe dos vectores con los elementos ordenados de menor a mayor y copia todos los elementos a un vector destino donde aparecerán de forma ordenada. La clave de su eficiencia está en que aprovechamos que ya están ordenados para ir volcándolos al resultado. La implementación, usando aritmética de punteros, puede ser la siguiente:

```
1  void MezclarOrdenados (const double v1[], int n1,
2                         const double v2[], int n2.
3                         double mezcla[])
4  {
5     const double* fin1= v1+n1; // Siguiente al último de v1
6     const double* fin2= v2+n2; // Siguiente al último de v2
7
8     while (v1!=fin1 && v2!=fin2)  { // Mientras haya en los dos
9        if (*v1<*v2)
10         *mezcla++ = *v1++;
11       else *mezcla++ = *v2++;
12    }
13
14    while (v1!=fin1) // Restantes de v1
15       *mezcla++ = *v1++;
16
17    while (v2!=fin2) // Restantes de v2
18       *mezcla++ = *v2++;
19 }
```

Observe que básicamente consiste en mantener dos apuntadores —*v1* y *v2*— que controlan los dos elementos candidatos a ser volcados y otro apuntador —*mezcla*— que controla la posición donde se volcará. En cada iteración, el menor se vuelca y se adelanta. Tenga en cuenta que estos

punteros pasan por valor, por lo que pueden modificarse dentro de la función sin ningún efecto en el código que llama.

Todas las operaciones que aparecen en el código son elementales; la eficiencia dependerá del número de iteraciones de los bucles. Se puede hacer un análisis más detallado, hablando de $O(1)$ en las líneas 5-6, un bloque $O(n_1+n_2-p)$ en el primer bucle, más los dos últimos bucles que tendrían $O(1)$ y $O(p)$. La suma de todo sería un algoritmo $O(n)$, donde $n=n_1+n_2$.

Una alternativa más simple es confirmar que todas las operaciones son elementales y que cada vez que itera un bucle, el puntero `mezcla` avanza una posición, es decir, incluye un nuevo elemento en el resultado. Lógicamente, el número total de avances $n1+n2$ es el que determina la eficiencia final.

Ejercicio 1.6 Considere el algoritmo de mezcla mediante el vuelco de los dos vectores en el vector destino y la llamada a ordenar con el algoritmo se selección implementado en la sección 1.4.3. ¿Qué eficiencia tendría?

1.4.6 Algoritmo de ordenación por mezcla

El algoritmo de ordenación por mezcla se basa en la idea de que para ordenar un vector podemos ordenar la primera mitad, ordenar la segunda mitad y luego mezclarlas. Esta mezcla aprovecha un algoritmo eficiente como el presentado en la sección 1.4.5.

Se puede implementar fácilmente mediante una función recursiva, ya que las dos mitadas se ordenan exactamente con el mismo algoritmo, es decir, corresponde al mismo problema pero con un tamaño menor.

Para diseñarla, podemos proponer una cabecera de función a la que se pasa el vector a ordenar y el número de elementos que contiene:

```
void OrdenMezcla (double v[], int n);
```

Esta cabecera nos permite llamar recursivamente a cada mitad, ya que la primera correspondería al vector `v` con `n/2` elementos y la segunda es el vector `v+n/2` con `n-n/2` elementos. Sin embargo, es necesario mezclar el resultado de ordenarlas; esta mezcla no se puede hacer en el mismo vector `v`, por lo que necesitamos un vector auxiliar `tmp` que nos permita ir añadiendo los elementos conforme mezclamos. Una forma de incorporarlo es pasarlo a la función como un parámetro de forma que el código de llamada es el encargado de ofrecer esa zona para resultados intermedios.

El código que proponemos para resolverlo aprovecha la aritmética de punteros; es el siguiente:

```
1  void OrdenMezcla (double v[], int n, double tmp[])
2  {
3    if (n>1) {
4       int n1= n/2;
5       int n2= n-n/2;
6       double *v1= v;
7       double *v2= v+n1;
8
9       OrdenMezcla (v1, n1, tmp);
10      OrdenMezcla (v2, n2, tmp);
11
12      MezclarOrdenados (v1, n1, v2, n2, tmp);
13
14      double *fin= v+n;
15      while (v!=fin)
16         *v++= *tmp++;
17   }
18 }
```

Observe que en la línea 12 hacemos una llamada a la función `MezclarOrdenados` que hemos analizado en la sección 1.4.5 y que sabemos que tiene eficiencia $O(n)$. Se podría hacer

alguna modificación para simplificar líneas o incluso usar la función **`memcpy`** de **`cstring`** para resolver la última copia, pero no entraremos en detalles[8].

El análisis de eficiencia incluye llamadas recursivas. Para calcularla podemos crear una ecuación recurrente y resolverla matemáticamente. Para ello, tenga en cuenta que:

- Las líneas 4-7 son operaciones elementales.
- Las líneas 14-16 son $O(n)$.
- La línea 12 es $O(n)$.
- Las líneas 9 y 10 son dos llamadas al mismo algoritmo pero con la mitad de elementos. Es posible que una llamada tenga un elemento más, pero podemos suponer que n es una potencia de dos y que tienen el mismo coste.

La ecuación recurrente con estas consideraciones se podría escribir como sigue:

$$T(n) = \begin{cases} 2 \cdot T(n/2) + c \cdot n + d & n > 1 \\ k & n \leq 1 \end{cases}$$

donde hemos incluido constantes indicando para acotar los bloque de orden lineal y constante.

Esta ecuación recurrente se tiene que resolver con algún método matemático. No es objetivo de este tema estudiar técnicas para resolverla. Hay métodos relativamente sencillos y muy eficaces. Si desea una demostración que es fácil de entender, puede cambiar de variable y realizar expansiones sucesivas:

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

$$T(2^m) = 2 \cdot \underbrace{T(2^{m-1})}_{[a]} + c \cdot 2^m + d$$

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

$$[a] = 2 \cdot \underbrace{T(2^{m-2})}_{[b]} + c \cdot 2^{m-1} + d$$

De nuevo expandimos $[b]$:

$$[b] = 2 \cdot \underbrace{T(2^{m-3})}_{[c]} + c \cdot 2^{m-2} + d$$

$$[c] = ...$$

Si unimos las expresiones y analizamos la forma general, podemos determinar que la expansión i-ésima es la siguiente:

$$T(2^m) = 2^i \cdot T(2^{m-i}) + c \cdot i \cdot 2^m + d(2^i - 1)$$

Si suponemos que hemos realizado m expansiones sucesivas, la ecuación tras estas expansiones corresponde a sustituir m en el lugar de i, es decir:

$$T(2^m) = 2^m \cdot T(0) + c \cdot m \cdot 2^m + d \cdot (2^i - 1)$$

Y si tenemos en cuenta que $n = 2^m$, que $m = \log_2 m$ y que $T(0) = k$, la expresión queda:

$$T(n) = c \cdot n \cdot \log_2 n + (d + k) \cdot n - d$$

[8] Para más detalles sobre este algoritmo se puede consultar, por ejemplo, Brassard[6] y Sedgewick[38].

Observe que las constantes no afectan al resultado, por lo que se pueden eliminar. Realmente, podríamos haber analizado la recurrencia sin estas constantes, es decir, la siguiente ecuación:

$$T(n) = 2 \cdot T(n/2) + n$$

que nos lleva al mismo resultado (puede intentarlo para comprobar que es más simple de expandir). En la práctica, se pueden realizar muchas simplificaciones si sabemos que al final vamos a llegar al mismo resultado. Recuerde que la eficiencia teórica no se ve afectada por las constantes.

Como puede ver en la expresión obtenida, el resultado final que estábamos buscando es $O(n \cdot \log_2 n)$. Piense un momento en la diferencia tan notable entre este algoritmo y uno básico como el de *selección* que tiene una eficiencia $O(n^2)$. Si damos un valor concreto al perfil de crecimiento —por ejemplo $n = 10^6$— es fácil intuir que el algoritmo cuadrático es mucho más lento que el de mezcla.

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:

- $T_1(n) + T_2(n) \in O(f(n))$
- $T_1(n) \in O(f^2(n))$
- $T_1(n)/T_2(n) \in O(1)$

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 en peor caso de la función `Insercion`, que ordena un vector usando el método de inserción.

```
void Intercambiar(int& a, int& b)
{
   int aux= a;
   a= b;
   b= aux;
}

void Insercion(const int v[], int n)
{
  for (int i=1;i<n;++i)
    if (v[i]<v[0])
       Intercambiar (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 Considere la función `Insercion` que se lista en el problema 1.5. El primer bucle permite situar el elemento más pequeño:

- ¿Para qué se ha añadido este primer paso?

Esta función de ordenación no corresponde a un algoritmo de ordenación *estable*. Como sabe, el algoritmo de inserción incluye su condición de estable como una característica positiva. Modifique la implementación anterior para que lo sea. Para esta nueva versión que ha obtenido, responda a:

- ¿Es más rápida?
- ¿Es más eficiente?

Problema 1.7 Obtenga, usando la notación O-mayúscula, la eficiencia de la función `Burbuja`, que 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;
      }
}
```