

Algoritmos iluminados

Tim Roughgarden

Algoritmos iluminados

Primera parte: Conceptos básicos

OJBooks
Valladolid

Algoritmos iluminados
Primera parte: Conceptos básicos

Primera edición en castellano

ISBN: 978-84-122380-5-1

Depósito legal: VA 678-2021

Traductor Miguel Revilla Rodríguez

© 2021 by Miguel Revilla Rodríguez

Traducido de la primera edición en inglés de:

Algorithms Illuminated – Part 1: The basics

Tim Roughgarden

Soundlikeyourself Publishing, LLC

ISBN: 978-9992829-0-8

© 2017 by Tim Roughgarden

Compuesto con \LaTeX

Este libro no puede ser reproducido en todo o en parte, ni transmitido por ningún medio mecánico, electrónico o análogo, incluyendo fotocopiado, escaneado o almacenado, sin autorización expresa y por escrito de los titulares de los derechos del texto original y de la traducción.

Índice general

Prefacio	xiii
1 Introducción	1
1.1 <i>¿Por qué estudiar algoritmos?</i>	1
1.2 <i>Multiplicación de enteros</i>	3
1.2.1 <i>Problemas y soluciones</i>	3
1.2.2 <i>El problema de la multiplicación de enteros</i>	4
1.2.3 <i>El algoritmo de la escuela</i>	4
1.2.4 <i>Análisis del número de operaciones</i>	5
1.2.5 <i>¿Podemos hacerlo mejor?</i>	6
1.3 <i>Multiplicación de Karatsuba</i>	6
1.3.1 <i>Un ejemplo concreto</i>	7
1.3.2 <i>Un algoritmo recursivo</i>	8
1.3.3 <i>Multiplicación de Karatsuba</i>	10
1.4 <i>MERGESORT: el algoritmo</i>	12
1.4.1 <i>Motivación</i>	12
1.4.2 <i>Ordenación</i>	13
1.4.3 <i>Un ejemplo</i>	15
1.4.4 <i>Pseudocódigo</i>	16
1.4.5 <i>La subrutina MERGE</i>	17
1.5 <i>MERGESORT: el análisis</i>	18
1.5.1 <i>Tiempo de ejecución de MERGE</i>	18
1.5.2 <i>Tiempo de ejecución de MERGESORT</i>	20
1.5.3 <i>Demostración del teorema 1.2</i>	21
1.5.4 <i>Soluciones a los cuestionarios 1.1–1.2</i>	25
1.6 <i>Principios rectores del análisis de algoritmos</i>	26
1.6.1 <i>Principio 1: análisis del peor caso</i>	26
1.6.2 <i>Principio 2: análisis global</i>	27
1.6.3 <i>Principio 3: análisis asintótico</i>	29
1.6.4 <i>¿Qué es un algoritmo “rápido”?</i>	31
Problemas	33

2 Notación asintótica	37
2.1 <u>La esencia</u>	37
2.1.1 <u>Motivación</u>	37
2.1.2 <u>La idea de alto nivel</u>	38
2.1.3 <u>Cuatro ejemplos</u>	39
2.1.4 <u>Soluciones a los cuestionarios 2.1–2.4</u>	44
2.2 <u>Notación Big-O</u>	46
2.2.1 <u>Definición en lenguaje convencional</u>	46
2.2.2 <u>Definición visual</u>	46
2.2.3 <u>Definición matemática</u>	47
2.3 <u>Dos ejemplos básicos</u>	49
2.3.1 <u>Los polinomios de grado k son $O(n^k)$</u>	49
2.3.2 <u>Los polinomios de grado k no son $O(n^{k-1})$</u>	50
2.4 <u>Notaciones Big-Omega y Big-Theta</u>	51
2.4.1 <u>Notación Big-Omega</u>	51
2.4.2 <u>Notación Big-Theta</u>	52
2.4.3 <u>Notación Little-o</u>	54
2.4.4 <u>¿De dónde viene la notación?</u>	55
2.4.5 <u>Solución al cuestionario 2.5</u>	55
2.5 <u>Ejemplos adicionales</u>	56
2.5.1 <u>Suma de una constante a un exponente</u>	56
2.5.2 <u>Multiplicación de un exponente por una constante</u>	57
2.5.3 <u>Máximo frente a suma</u>	58
Problemas	59
3 Algoritmos de divide y vencerás	63
3.1 <u>El paradigma divide y vencerás</u>	63
3.2 <u>Conteo de inversiones en tiempo $O(n \log n)$</u>	64
3.2.1 <u>El problema</u>	64
3.2.2 <u>Un ejemplo</u>	65
3.2.3 <u>Filtrado colaborativo</u>	65
3.2.4 <u>Búsqueda exhaustiva</u>	66
3.2.5 <u>Una solución de divide y vencerás</u>	67
3.2.6 <u>Algoritmo de alto nivel</u>	67
3.2.7 <u>Idea clave: a cuestas de MERGESORT</u>	68
3.2.8 <u>MERGE revisitado</u>	69
3.2.9 <u>MERGE y las inversiones separadas</u>	71
3.2.10 <u>MERGE-AND-COUNTSPLITINV</u>	72
3.2.11 <u>Corrección</u>	73
3.2.12 <u>Tiempo de ejecución</u>	73
3.2.13 <u>Soluciones a los cuestionarios 3.1–3.2</u>	74
3.3 <u>Algoritmo de multiplicación de matrices de Strassen</u>	74
3.3.1 <u>Multiplicación de matrices</u>	74

3.3.2	Ejemplo ($n = 2$)	75
3.3.3	El algoritmo directo	75
3.3.4	Una solución de divide y vencerás	77
3.3.5	Cómo ahorrar una llamada recursiva	79
3.3.6	Los detalles	80
3.3.7	Solución al cuestionario 3.3	81
*3.4	Un algoritmo en $O(n \log n)$ para el par más cercano	81
3.4.1	El problema	82
3.4.2	¿Nos ayuda en algo la ordenación?	83
3.4.3	Una solución de divide y vencerás	83
3.4.4	Un cambio sutil	86
3.4.5	CLOSESTSPLITPAIR	86
3.4.6	Corrección	89
3.4.7	Demostración del lema 3.3(a)	90
3.4.8	Demostración del lema 3.3(b)	91
3.4.9	Solución al cuestionario 3.4	93
	<u>Problemas</u>	94
4	El método maestro	99
4.1	Multiplicación de enteros revisitada	99
4.1.1	El algoritmo RECINTMULT	100
4.1.2	El algoritmo KARATSUBA	100
4.1.3	Comparativa de las recurrencias	101
4.2	Declaración formal	102
4.2.1	Recurrencias estándar	102
4.2.2	Declaración y tratamiento del método maestro	104
4.3	Seis ejemplos	105
4.3.1	MERGESORT revisitado	105
4.3.2	Búsqueda binaria	105
4.3.3	Multiplicación recursiva de enteros	106
4.3.4	Multiplicación de Karatsuba	107
4.3.5	Multiplicación de matrices	107
4.3.6	Una recurrencia ficticia	108
4.3.7	Soluciones a los cuestionarios 4.2–4.3	109
*4.4	Demostración del método maestro	110
4.4.1	Preámbulo	111
4.4.2	Árboles de recursión revisitados	112
4.4.3	Trabajo realizado en un solo nivel	113
4.4.4	Suma de todos los niveles	114
4.4.5	El bien frente al mal: la necesidad de los tres casos	114
4.4.6	Predicción de los límites del tiempo de ejecución	116
4.4.7	Los cálculos finales: caso 1	117
4.4.8	Desvío: series geométricas	118

4.4.9	Los cálculos finales: casos 2 y 3	119
4.4.10	Soluciones a los cuestionarios 4.4–4.5	120
<u>Problemas</u>		122
5 Ordenación QuickSort		125
5.1	<u>Introducción</u>	125
5.1.1	<u>Ordenación</u>	126
5.1.2	<u>Particiones en torno a un pivote</u>	126
5.1.3	<u>Descripción de alto nivel</u>	128
5.1.4	<u>Sequimos avanzando</u>	129
5.2	<u>Partición en torno a un elemento pivote</u>	130
5.2.1	<u>La vía rápida</u>	130
5.2.2	<u>Implementación <i>in situ</i>: el plan de alto nivel</u>	130
5.2.3	<u>Ejemplo</u>	131
5.2.4	<u>Pseudocódigo de PARTITION</u>	134
5.2.5	<u>Pseudocódigo de QUICKSORT</u>	136
5.3	<u>La importancia de los buenos pivotes</u>	136
5.3.1	<u>Implementación ingenua de CHOOSEPIVOT</u>	137
5.3.2	<u>Implementación excesiva de CHOOSEPIVOT</u>	137
5.3.3	<u>Soluciones a los cuestionarios 5.1–5.2</u>	139
5.4	<u>QUICKSORT aleatorizado</u>	140
5.4.1	<u>Implementación aleatorizada de CHOOSEPIVOT</u>	141
5.4.2	<u>Tiempo de ejecución del QUICKSORT aleatorizado</u>	142
5.4.3	<u>Intuición: ¿por qué son buenos los pivotes aleatorios?</u>	142
*5.5	<u>Análisis del QUICKSORT aleatorizado</u>	144
5.5.1	<u>Preliminares</u>	144
5.5.2	<u>Un esquema de descomposición</u>	146
5.5.3	<u>Aplicación del esquema</u>	148
5.5.4	<u>Cálculo de las probabilidades de comparación</u>	150
5.5.5	<u>Cálculos finales</u>	153
5.5.6	<u>Solución al cuestionario 5.3</u>	154
*5.6	<u>La ordenación necesita $\Omega(n \log n)$ comparaciones</u>	155
5.6.1	<u>Algoritmos de ordenación basados en comparaciones</u>	155
5.6.2	<u>Ordenación más rápida bajo condiciones estrictas</u>	156
5.6.3	<u>Demostración del teorema 5.5</u>	158
<u>Problemas</u>		161
6 Selección en tiempo lineal		167
6.1	<u>El algoritmo RSELECT</u>	168
6.1.1	<u>El problema de selección</u>	168
6.1.2	<u>Reducción a ordenación</u>	169
6.1.3	<u>Un método de divide y vencerás</u>	170
6.1.4	<u>Pseudocódigo de RSELECT</u>	171

<u>6.1.5</u>	Tiempo de ejecución de RSELECT	172
<u>6.1.6</u>	Soluciones a los cuestionarios 6.1–6.2	175
*6.2	<u>Ánalisis de RSELECT</u>	175
6.2.1	Sequimiento del progreso en fases	176
6.2.2	Reducción al lanzamiento de monedas	177
6.2.3	Lo juntamos todo	179
*6.3	<u>El algoritmo DSELECT</u>	180
6.3.1	La gran idea: mediana de medianas	181
6.3.2	Pseudocódigo de DSELECT	182
6.3.3	Entender DSELECT	183
6.3.4	Tiempo de ejecución de DSELECT	184
*6.4	<u>Ánalisis de DSELECT</u>	185
6.4.1	Trabajo fuera de las llamadas recursivas	185
6.4.2	Una recurrencia un poco tosca	186
6.4.3	El lema del 30-70	187
6.4.4	Solución de la recurrencia	191
6.4.5	El método probar–comprobar	192
<u>Problemas</u>		194
A	<u>Un vistazo a las demostraciones por inducción</u>	197
A.1	<u>Plantilla para demostraciones por inducción</u>	197
A.2	<u>Ejemplo: una fórmula cerrada</u>	199
A.3	<u>Ejemplo: el tamaño de un árbol binario completo</u>	199
B	<u>Un vistazo a la probabilidad discreta</u>	201
B.1	<u>Espacios de muestra</u>	201
B.2	<u>Eventos</u>	202
B.2.1	<u>Solución al cuestionario B.1</u>	204
B.2.2	<u>Solución al cuestionario B.2</u>	204
B.3	<u>Variables aleatorias</u>	204
B.4	<u>Expectativa</u>	205
B.4.1	<u>Solución al cuestionario B.3</u>	206
B.4.2	<u>Solución al cuestionario B.4</u>	207
B.5	<u>Linealidad de la expectativa</u>	207
B.5.1	<u>Declaración formal y casos de uso</u>	207
B.5.2	<u>La demostración</u>	209
B.6	<u>Ejemplo: balanceo de carga</u>	210
<u>Pistas y soluciones a problemas seleccionados</u>		215
<u>Índice alfabético</u>		221

Prefacio

Este libro es el primero de una serie basada en mis cursos en línea sobre algoritmia, que he venido impartiendo regularmente desde 2012 y que, a su vez, están basados en un curso que imparti en innumerables ocasiones en la Universidad de Stanford.

De qué vamos a hablar en este libro

La primera parte de *Algoritmos iluminados* ofrece una introducción y aspectos básicos de los siguientes cuatro temas.

Análisis asintótico y notación Big-O. La notación asintótica nos ofrece un vocabulario básico para tratar el diseño y análisis de los algoritmos. El concepto clave es la notación *Big-O*, que supone en mecanismo de modelado de la granularidad con la que medimos el tiempo de ejecución de un algoritmo. Veremos cómo el aspecto fundamental del pensamiento de alto nivel sobre el diseño de algoritmos consiste en ignorar los factores constantes y los términos de menor orden, concentrándonos así en la escala que adquiere el algoritmo en relación al tamaño de los datos de entrada.

Algoritmos de divide y vencerás y el método maestro. En el diseño de algoritmos no existe una fórmula mágica, es decir, no hay un método único de solución de todos los problemas computacionales. Sin embargo, contamos con algunas técnicas generales de diseño de algoritmos que resultan exitosas en un rango amplio de dominios diferentes. Aquí nos referiremos a la técnica de “divide y vencerás”. La idea consiste en dividir un problema principal en subproblemas más pequeños, resolver estos recursivamente y combinar de forma eficiente sus soluciones para solventar el problema original. Veremos algoritmos rápidos de divide y vencerás para ordenar, multiplicar enteros y matrices y para un problema de geometría computacional básico. También trataremos el método maestro, una potente herramienta para analizar el tiempo de ejecución de los algoritmos de divide y vencerás.

Algoritmos aleatorizados. Un algoritmo aleatorizado “lanza monedas al aire” a medida que se ejecuta, y su resultado puede depender de esos lanzamientos de monedas. Sorprendentemente, esta aleatorización lleva a construir algoritmos sencillos, elegantes y prácticos. El ejemplo canónico lo encontramos en la ordenación *quick* aleatorizada, y explicaremos en detalle tanto este algoritmo como su análisis de tiempo de ejecución.

Ordenación y selección. Como resultado sobrevenido del estudio de los tres primeros temas, aprenderemos varios algoritmos famosos de ordenación y selección, como la ordenación por mezcla, la ordenación *quick* y la selección en tiempo lineal (tanto aleatorizados como deterministas). Estas técnicas elementales son espectacularmente rápidas, casi tanto como la simple lectura de los datos de entrada. Es importante cultivar una colección de estas “técnicas de coste cero”, tanto para aplicarlas directamente a la información como para utilizarlas como ladrillos en la construcción de soluciones a problemas más complejos.

Para obtener una visión detallada del contenido del libro, puedes consultar las secciones “Conclusiones”, que ponen fin a cada capítulo y destacan los aspectos más importantes. Las secciones destacadas (indicadas con un asterisco) son las más avanzadas y, si el lector tiene prisa, puede ignorarlas en una primera lectura, sin perder continuidad en el contenido global.

Temas tratados en las otras partes de la serie. La *segunda parte* trata de estructuras de datos (montículos, árboles de búsqueda equilibrados, tablas de *hash*, filtros flor), técnicas para grafos (búsqueda en anchura y profundidad, conectividad, caminos más cortos) y sus aplicaciones (desde la deduplicación hasta el análisis de redes sociales). La *tercera parte* se centra en algoritmos voraces (programación, árboles de expansión mínimos, agrupación, códigos de Huffman) y programación dinámica (mochila, alineación de secuencias, caminos más cortos, árboles de búsqueda óptimos). La *cuarta parte* trata sobre los problemas NP-complejos: cómo identificarlos (utilizando reducciones), formas de ponderar tus ambiciones algorítmicas en relación a ellos (mediante la aproximación o el tiempo de ejecución exponencial del peor caso) y herramientas algorítmicas para alcanzar esas ambiciones ya matizadas (algoritmos heurísticos voraces, búsqueda local, programación dinámica avanzada y soluciones MIP y SAT).

Qué aprenderás en esta serie de libros

Dominar los algoritmos requiere tiempo y esfuerzo. ¿Por qué molestarse?

Serás un mejor programador. Aprenderás varias subrutinas espectacularmente rápidas para procesar datos, así como varias estructuras útiles para organizar información, que podrás desplegar directamente en tus programas. La implementación y uso de estos algoritmos expandirá y mejorará tus capacidades como programador. También conocerás los paradigmas generales de diseño de algoritmos, relevantes para muchos problemas en una amplia variedad de dominios, así como herramientas para predecir el rendimientos de dichos algoritmos. Estos “patrones de diseño de algoritmos” te ayudarán a diseñar nuevos algoritmos para resolver los problemas que se te vayan presentando en el futuro.

Agudizarás tu capacidad de análisis. Practicarás mucho la descripción y el razonamiento sobre algoritmos. A través del análisis matemático, lograrás una profunda comprensión de los algoritmos y estructuras de datos específicos tratados en estos libros. Ganarás agilidad en el uso de varias técnicas matemáticas, cuyo uso está ampliamente extendido en el análisis de algoritmos.

Pensarás en términos algorítmicos. Una vez comiences a conocer los algoritmos, los encontrarás en cualquier parte, ya sea al subir a un ascensor, al observar una bandada de pájaros, en la administración de tu cartera de inversiones o, incluso, viendo cómo aprenden los niños. El pensamiento algorítmico es cada vez más útil y está cada vez más presente en disciplinas ajenas a las ciencias de la computación, como en la biología, la estadística y la economía.

Estarás al día de los grandes éxitos de las ciencias de la computación. El estudio de los algoritmos se puede comparar a contemplar un resumen de muchos de los grandes éxitos de los últimos sesenta años en el campo de las ciencias de la computación. Ya no te sentirás excluido cuando, en una fiesta de informáticos, alguien cuente un chiste sobre el algoritmo de Dijkstra. Después de leer estos libros, entrarás de lleno en la conversación.

Arrasarás en tus entrevistas técnicas. A lo largo de los años, muchos estudiantes me han obsequiado con historias sobre cómo el conocimiento de los conceptos de estos libros les han llevado a dar una respuesta excelente a cada pregunta que les han realizado en una entrevista de trabajo.

En qué son diferentes estos libros

Esta serie de libros tiene un único objetivo: *enseñar los conceptos algorítmicos de la forma más sencilla posible*. Puedes verlos como una transcripción de las palabras que un tutor experto en algoritmos te diría durante una lección cara a cara.

Existe un buen número de libros de texto tradicionales sobre algoritmos que son excelentes y cualquiera de ellos será un buen complemento a esta serie de libros, al plantear problemas y temas adicionales. Te animo a que los conozcas y elijas tus favoritos. También existen otros materiales que, a diferencia de estos libros, alimentan al programador que busca implementaciones listas para usar en algún lenguaje de programación específico. Muchas de esas implementaciones también se pueden encontrar disponibles en internet.

¿Quién eres?

El objetivo principal de estos libros, y de los cursos en línea en los que se basan, radica en que sean lo más accesibles posible. En mis cursos en línea he encontrado a personas de todas las edades, orígenes y trayectos vitales y, en todos los rincones de mundo, existe una impresionante cantidad de estudiantes (de secundaria, universidad, etc.), ingenieros de software (presentes o futuros), científicos y profesionales deseosos de aprender.

Este libro no es una introducción a la programación. Lo ideal sería que ya tuvieras unas nociones básicas en este ámbito, como el uso de *arrays* y la recursión, en algún lenguaje de programación (ya sea Java, Python, C, Scala, Haskell, etc.). Si quieres una prueba de fuego, consulta la sección [1.4](#) y, si lo que encuentras ahí tiene sentido, estarás listo para afrontar el resto del libro. Si crees que necesitas refrescar tus capacidades de programación, existen varios cursos gratuitos en internet, que tienen una calidad extraordinaria y que te darán la base necesaria.

También utilizaremos el análisis matemático, según resulte necesario, para entender cómo y por qué funcionan los algoritmos. El libro gratuito “*Mathematics for Computer Science*”, de Eric Lehman, F. Thomson Leighton y Albert R. Meyer, supone un contenido excepcional y ameno para recordar la notación matemática (como \sum o \forall), los conceptos de las demostraciones (inducción, contradicción, etc.), la probabilidad discreta y mucho más. Los apéndices [A](#) y [B](#) también proporcionan un vistazo rápido a las demostraciones por inducción y a la probabilidad discreta, respectivamente.

Recursos adicionales

Estos libros están basados en cursos en línea que, en la actualidad, se encuentra disponibles en las plataformas Coursera y edX. He dejado varios recursos disponibles para ayudarte a replicar, según tus necesidades, la experiencia del curso en línea.

Vídeos. Si prefieres ver y escuchar, puedes utilizar las listas de reproducción de YouTube disponibles en www.algorithmsilluminated.org. Estos videos cubren todos los temas de esta serie de libros, así como algunos temas avanzados adicionales. Espero que te transmitan un entusiasmo contagioso por los algoritmos que, francamente, es difícil de comunicar mediante la palabra impresa.

Cuestionarios. ¿Cómo podrías saber si estás asimilando realmente los conceptos de este libro? A lo largo del texto encontrarás cuestionarios con soluciones y explicaciones. Cuando llegues a uno de ellos, te animo a que te detengas en tu lectura y trates de responderlo antes de continuar.

Problemas al final del capítulo. Al final de cada capítulo podrás encontrar varias preguntas relativamente sencillas, para comprobar tu nivel de comprensión, seguidas por problemas y desafíos más complejos. Podrás encontrar pistas o soluciones para la mayoría de estos problemas (indicados, respectivamente, por "(P)" o "(S)") al final del libro. Los lectores pueden comentar conmigo, o con otros, cualquier cuestión relativa a estos problemas, a través del foro de discusión del libro (ver más adelante).

Problemas de programación. Muchos de los capítulos finalizan proponiendo un proyecto de programación, cuyo objetivo es el de ayudarte a desarrollar una comprensión detallada de un algoritmo, mediante la creación de tu propia implementación funcional. En www.algorithmsilluminated.org podrás encontrar conjuntos de datos, junto a los casos de prueba y las soluciones.

Foros de discusión. Una de las razones más importantes del éxito de los cursos en línea, es la oportunidad que estos ofrecen a los participantes de ayudarse entre ellos para entender el material y depurar las implementaciones. Los lectores de estos libros gozarán de esas mismas oportunidades gracias a los foros disponibles en www.algorithmsilluminated.org.

Agradecimientos

Estos libros no existirían sin la pasión y el impulso aportados por los miles de participantes de mis cursos de algoritmia a lo largo de los años. Estoy especialmente agradecido a aquellos que han proporcionado comentarios y aportaciones al borrador de este libro: Tonya Blust, Yuan Cao, Jim Hume, Bayram Kuliyev, Patrick Monkelban, Kyle Schiller, Nissanka Wickremasinghe y Daniel Zingaro.

Siempre agradezco nuevas sugerencias y correcciones por parte de los lectores. El mejor canal de comunicación es el foro ya mencionado.

TIM ROUGHGARDEN
Stanford, California
Septiembre de 2017

Capítulo 1

Introducción

El objetivo de este capítulo es ilusionarte con el estudio de los algoritmos. Comenzaremos hablando de los algoritmos en general y del porqué de su importancia. Después, utilizaremos el problema de la multiplicación de dos enteros para ilustrar cómo un algoritmo ingenioso puede mejorar soluciones más directas o ingenuas. A continuación, estudiaremos el algoritmo MERGESORT en detalle, por varias razones: es un algoritmo famoso y práctico que deberías conocer, supone un buen calentamiento antes de abordar otros algoritmos más complejos y es la introducción canónica al paradigma de diseño de algoritmos “divide y vencerás”. Finalizaremos describiendo varios principios rectores sobre cómo analizaremos los algoritmos del resto del libro.

1.1 ¿Por qué estudiar algoritmos?

Permíteme que comience justificando la existencia de este libro e indicando varias razones por las que tu motivación para aprender sobre algoritmos debería ser alta. De todas formas, ¿qué es un algoritmo? Es un conjunto de reglas bien definidas (de hecho, es una receta) para resolver algunos problemas de computación. Quizá estés interesado en tomar un conjunto de números y alterar sus posiciones para que aparezcan ordenados. Quizá tengas un mapa de carreteras y quieras calcular el camino más corto desde un origen hasta un destino. Quizá necesites completar algunas tareas dentro de cierto periodo de tiempo y quieras saber en qué orden deberías realizarlas para cumplir con sus respectivos plazos.

Entonces, ¿por qué estudiar algoritmos?

Es importante para el resto de las ciencias de la computación. En primer lugar, entender los conceptos de los algoritmos y el campo relacionado de las estructuras de datos resulta esencial para realizar un trabajo serio en prácticamente cualquier rama de las ciencias de la computación. Por ejemplo, durante mis años como profesor en la Universidad de Stanford, todos los grados ofrecidos por el departamento de informática (grado, máster o doctorado) implicaban un curso de algoritmia. Por citar algunos ejemplos:

1. Los protocolos de enrutamiento en las redes de comunicaciones implican algoritmos clásicos de caminos más cortos.
2. La criptografía basada en claves públicas se basa en algoritmos eficientes sobre teoría de números.
3. Los gráficos generados por ordenador necesitan de los mecanismos computacionales proporcionados por algoritmos geométricos.
4. Los índices de las bases de datos implican estructuras de datos de árboles de búsqueda equilibrados.
5. La biología computacional utiliza algoritmos de programación dinámica para cuantificar la similitud del genoma.
6. Los algoritmos de agrupación son una de las herramientas omnipresentes en el aprendizaje de máquinas sin supervisión.

Y la lista sigue.

Son la guía de la innovación tecnológica. En segundo lugar, los algoritmos juegan un papel fundamental en la innovación tecnológica moderna. Por utilizar un ejemplo obvio, los motores de búsqueda utilizan un ramillete de algoritmos para calcular eficientemente la relevancia de varias páginas web en relación a una consulta. El más famoso de estos algoritmos es PAGE RANK, del que surgió Google. De hecho, en un informe de la Casa Blanca de los Estados Unidos de diciembre de 2010, el consejo científico y tecnológico del presidente redactó lo siguiente:

“Todo el mundo conoce la Ley de Moore, una predicción realizada en 1965 por Gordon Moore, cofundador de Intel, según la cual la densidad de los transistores en los circuitos integrados seguiría duplicándose cada 1 o 2 años [...] en muchas áreas, las mejoras de rendimiento derivadas de las

mejoras en los algoritmos han superado con creces a las inmensas mejoras de rendimiento atribuibles al aumento de la velocidad de los procesadores.”¹

Foco sobre otras ciencias. En tercer lugar, aunque queda fuera del ámbito de este libro, los algoritmos se utilizan cada vez más para procurar un “enfoque” novedoso para procesos externos a las ciencias de la computación y la tecnología. Por ejemplo, el estudio de la computación cuántica ofrece un nuevo punto de vista computacional para la mecánica cuántica. La variabilidad de precios en los mercados económicos se puede interpretar oportunamente como un proceso algorítmico. Incluso podemos ver la evolución como un algoritmo de búsqueda sorprendentemente efectivo.

Es bueno para el cerebro. Cuando era estudiante, mis clases favoritas siempre eran aquellas que me resultaban más desafiantes y que, tras esforzarme en ellas, me hacían sentir un poco más inteligente. Espero que este material te cause la misma sensación.

Es divertido. Por último, espero que, cuando llegues al final del libro, entiendas por qué el diseño y análisis de algoritmos es muy divertido. Estamos ante un esfuerzo que requiere de una extraña combinación de precisión y creatividad. Es cierto que, en ocasiones, puede resultar frustrante, pero también adictivo. Y no olvidemos que has estado aprendiendo algoritmos desde que eras un niño.

1.2 Multiplicación de enteros

1.2.1 Problemas y soluciones

Cuando estabas en, posiblemente, tercero de primaria, es casi seguro que aprendiste un algoritmo para multiplicar dos números, un conjunto de reglas bien definido para transformar una entrada (dos números) en una salida (su producto). Es importante distinguir entre dos aspectos diferentes: el enunciado del *problema a resolver* y el *método de solución* (es decir, el algoritmo que resuelve el problema). A lo largo de este libro, seguiremos de forma reiterada el patrón de presentar primero un problema informático (las entrada y salida deseadas) y, después, describir uno o más algoritmos que lo resuelvan.

¹Extracto del informe al Presidente y al Congreso: *Designing a Digital Future*, diciembre de 2010 (página 71).

1.2.2 El problema de la multiplicación de enteros

En el problema de la multiplicación de enteros, la entrada consta de dos números de n dígitos, que llamaremos x e y . La longitud n de x e y puede ser la de cualquier entero positivo, pero te animo a que pienses en un n verdaderamente grande, en el orden de los miles o, incluso, más² (es posible que estés implementando una aplicación criptográfica que deba manipular enteros muy grandes). La salida deseada en el problema de multiplicación de enteros, será el producto $x \cdot y$.

Problema: multiplicación de enteros

Entrada: Dos enteros no negativos de n dígitos, x e y .

Salida: El producto $x \cdot y$.

1.2.3 El algoritmo de la escuela

Una vez definido el problema computacional con precisión, describimos un algoritmo que lo resuelve, el mismo que aprendiste en la escuela. Valoraremos el rendimiento de este algoritmo a través del número de “operaciones básicas” que realiza, como una función del número n de dígitos de cada número de la entrada. De momento, pensemos en una operación básica como cualquiera de las siguientes: (i) sumar dos números de un solo dígito, (ii) multiplicar dos números de un solo dígito o (iii) añadir un cero al principio o al final de un número.

Para refrescarte la memoria, considera el ejemplo concreto de multiplicar $x = 5678$ por $y = 1234$ (por lo que $n = 4$). Puedes consultar la figura 1.1. El algoritmo comienza calculando el “producto parcial” del primer número y el último dígito del segundo número: $5678 \cdot 4 = 22\,712$. Calcular este producto parcial se reduce a multiplicar cada uno de los dígitos del primer número por 4, sumando los valores que “nos llevamos” según sea necesario³. Al calcular el siguiente producto parcial ($5678 \cdot 3 = 17\,034$), realizamos la misma operación, desplazando el resultado un dígito a la

²Si quieres multiplicar números de longitudes diferentes (como 1234 y 56), un truco muy sencillo consiste en añadir ceros al principio del número menor (por ejemplo, expresar 56 como 0056). Alternativamente, podremos modificar los algoritmos tratados para permitir su operación con números de longitudes diferentes.

³ $8 \cdot 4 = 32$, nos llevamos 3, $7 \cdot 4 = 28$, más 3 es 31, nos llevamos 3, ...

$$\begin{array}{r}
 5678 \\
 \times 1234 \\
 \hline
 22712 \\
 17034 \\
 11356 \\
 5678 \\
 \hline
 7006652
 \end{array}$$

n filas

≤ 3n operaciones
 (por fila)

Figura 1.1: Algoritmo de multiplicación de enteros de la escuela.

izquierda lo que, de hecho, es igual a añadir un “0” al final. Y seguimos repitiendo la operación en los dos productos parciales que nos quedan. El paso final consiste en sumar todos los productos parciales.

De vuelta al tercer curso de la escuela, probablemente aceptaste este algoritmo como *correcto*, en el sentido de que, con independencia de cuáles fuesen los números x e y , siempre que las operaciones intermedias se realizaran correctamente, terminarías obteniendo el producto $x \cdot y$ de los dos números de la entrada. Es decir, nunca obtendrías una respuesta incorrecta y el algoritmo no entraría en un bucle infinito.

1.2.4 Análisis del número de operaciones

Es bastante probable que tu profesor de tercero no te hablase del número de operaciones básicas necesarias para llevar a término este método. Para calcular el primer producto parcial, hemos multiplicado 4 veces cada uno de los dígitos 5, 6, 7, 8 del primer número. Esto supone cuatro operaciones básicas. Cada vez que “nos llevamos” un valor, podríamos vernos en la necesidad de sumar un número de un dígito a otro de dos dígitos, añadiendo dos operaciones básicas más. En general, el cálculo de un producto parcial implica n multiplicaciones (una por dígito) y un máximo de $2n$ sumas (por los valores que “nos llevamos”), lo que supone un total máximo de $3n$ operaciones. Dado que tenemos n productos parciales (uno por cada dígito del segundo número), el cálculo completo necesita un máximo de $n \cdot 3n = 3n^2$ operaciones básicas. Y todavía tendremos que sumarlos todos para obtener la respuesta final, lo que implica un número similar de operaciones (como mucho otras $3n^2$, como veremos). En resumen:

$$\text{número total de operaciones} \leq \underbrace{\text{constante}}_{=6} \cdot n^2.$$

Si pensamos en que la cantidad de trabajo que debe realizar el algoritmo *aumenta* según van creciendo los números de la entrada, veremos que las operaciones requeridas crecen de forma cuadrática en relación al número de dígitos. Si duplicamos la longitud de los números de la entrada, el trabajo necesario aumenta por un factor de 4. Si cuadriplicamos su longitud tendremos un factor de 16 y, así, sucesivamente.

1.2.5 ¿Podemos hacerlo mejor?

Dependiendo del tipo de alumno de tercer curso que fueses, podrías haber aceptado este método como el único o, al menos, el óptimo para multiplicar dos números. Pero si quieres ser un diseñador de algoritmos serio, lo primero que debes hacer es escapar de esa timidez obediente. El libro de algoritmos clásico de Aho, Hopcroft y Ullman, tras iterar por una serie de paradigmas de diseño de algoritmos, dice lo siguiente:

“El principio más importante del buen diseñador de algoritmos es, quizá, negarse a estar satisfecho.⁴

O, como me gusta decirlo a mí, todo diseñador de algoritmos debe hacer suyo este mantra:

¿Podemos hacerlo mejor?

Esta pregunta resulta muy pertinente cuando te enfrentas a la solución más ingenua o evidente de un problema de computación. Quizá tercero de primaria no era el mejor momento para preguntar si existía una forma mejor de multiplicar. Pero ahora es la ocasión perfecta para plantear, y contestar, esta cuestión.

1.3 Multiplicación de Karatsuba

El ámbito del diseño de algoritmos es sorprendentemente rico y, desde luego, existen otros métodos interesantes para multiplicar dos enteros, más allá de lo que aprendiste en tercero. Esta sección describe una técnica denominada *multiplicación de Karatsuba*⁵

⁴Alfred V. Aho, John E. Hopcroft y Jeffrey D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974, página 70.

⁵Descubierta en 1960 por Anatoly Karatsuba, quien, en aquel momento, era un estudiante de 23 años.

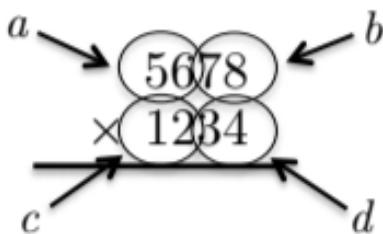


Figura 1.2: Números de cuatro dígitos como pares de dos dígitos.

1.3.1 Un ejemplo concreto

Para hacernos una idea de cómo funciona la multiplicación de Karatsuba, volveremos a nuestro ejemplo anterior, con $x = 5678$ e $y = 1234$. Vamos a ejecutar una sucesión de pasos, bastante diferentes a los del algoritmo de la escuela, que culminará con el producto de $x \cdot y$. Estos pasos deberían sorprenderte como lo hace un mago cuando saca un conejo de la chistera aunque, más adelante, explicaremos qué es exactamente la multiplicación de Karatsuba y por qué funciona. La clave está en apreciar que existe un abanico de opciones para resolver problemas de computación, como el de la multiplicación de enteros.

En primer lugar, para referirnos a las primera y segunda mitades de x como números por derecho propio, los llamaremos a y b (así, $a = 56$ y $b = 78$). Igualmente c y d equivaldrán a 12 y 34, respectivamente (figura 1.2).

A continuación, realizaremos una sucesión de operaciones que solo implican a los números de dos dígitos a , b , c y d y, por último, reuniremos todos los términos por arte de magia, para obtener como resultado el producto de x por y .

Paso 1: Calcular $a \cdot c = 56 \cdot 12$, que es 672 (compruébalo si quieras).

Paso 2: Calcular $b \cdot d = 78 \cdot 34 = 2652$.

Los dos siguientes pasos son, si cabe, más inescrutables.

Paso 3: Calcular $(a + b) \cdot (c + d) = 134 \cdot 46 = 6164$.

Paso 4: Restar los resultados de los dos primeros pasos del resultado del tercer paso: $6164 - 672 - 2652 = 2840$.

Por último, sumamos los resultados de los pasos 1, 2 y 4, pero únicamente después de añadir 4 ceros al final de la respuesta del primer paso y dos ceros a la respuesta del cuarto.

Paso 5: Calcular $10^4 \cdot 672 + 10^2 \cdot 2840 + 2652 = 6\,720\,000 + 284\,000 + 2652 = 7\,006\,652$.

Y, así, obtenemos exactamente el mismo resultado (correcto) que habíamos calculado mediante el algoritmo de la escuela de la sección 1.2.

En este momento no deberías tener ni idea de lo que acaba de ocurrir. Más bien te encontrarás con una sensación confusa entre el estupor y la intriga, junto al agradecimiento por descubrir que existe otro algoritmo para la multiplicación de enteros, en apariencia completamente distinto al que aprendiste de niño. Una vez que te has dado cuenta de lo rico que puede llegar a ser el ámbito de los algoritmos, llegará la pregunta: ¿podemos hacerlo mejor que con el algoritmo de tercero de primaria? ¿De verdad este algoritmo es mejor?

1.3.2 Un algoritmo recursivo

Antes de abordar la multiplicación de Karatsuba al completo, vamos a explorar un sencillo algoritmo recursivo para la multiplicación de enteros⁶. Cabe esperar que un algoritmo recursivo para la multiplicación de enteros implique multiplicaciones de números con menos dígitos (como 12, 34, 56 y 78 en el cálculo anterior).

En general, un número x con una cantidad par de n dígitos se puede expresar en términos de dos números de $n/2$ dígitos, siendo la primera mitad a y la segunda b :

$$x = 10^{n/2} \cdot a + b.$$

Igualmente, podemos escribir:

$$y = 10^{n/2} \cdot c + d.$$

Para calcular el producto de x e y , utilizaremos las dos expresiones anteriores y las multiplicaremos:

$$\begin{aligned} x \cdot y &= (10^{n/2} \cdot a + b) \cdot (10^{n/2} \cdot c + d) \\ &= 10^n \cdot (a \cdot c) + 10^{n/2} \cdot (a \cdot d + b \cdot c) + b \cdot d. \end{aligned} \tag{1.1}$$

⁶Asumo que, como programador, has oído hablar de la recursividad. Un procedimiento recursivo es aquel que se llama a sí mismo como subrutina con una entrada menor, hasta que se llega a un caso base.

Hay que notar que todas las multiplicaciones de (1.1) se producen entre pares de números de $n/2$ dígitos o implican potencias de 10 .⁷

La expresión (1.1) sugiere una técnica recursiva para la multiplicación de dos números. Para calcular el producto $x \cdot y$, calculamos la expresión (1.1). Los cuatro productos relevantes ($a \cdot c$, $a \cdot d$, $b \cdot c$ y $b \cdot d$) implican números con menos de n dígitos, por lo que podemos calcular cada uno de ellos recursivamente. Una vez que las cuatro llamadas recursivas nos devuelvan sus respuestas, podemos procesar la expresión (1.1) de la forma más obvia: añadimos n ceros al final de $a \cdot c$, sumamos $a \cdot d$ y $b \cdot c$ (utilizando la suma que aprendimos en la escuela) y añadimos $n/2$ ceros al final del resultado para, finalmente, sumar estas dos expresiones a $b \cdot d$.⁸ Resumimos este algoritmo, que denominaremos RECINTMULT, mediante el siguiente pseudocódigo⁹.

RECINTMULT

Entrada: dos enteros positivos x e y de n dígitos.

Salida: el producto $x \cdot y$.

Asunción: n es una potencia de 2.

```
si  $n = 1$  entonces // caso base
    calcular  $x \cdot y$  en un paso y devolver el resultado
en otro caso // caso recursivo
     $a, b :=$  primera y segunda mitades de  $x$ 
     $c, d :=$  primera y segunda mitades de  $y$ 
    calcular recursivamente  $ac := a \cdot c$ ,  $ad := a \cdot d$ ,  $bc := b \cdot c$  y
         $bd := b \cdot d$ 
    calcular  $10^n \cdot ac + 10^{n/2} \cdot (ad + bc) + bd$  utilizando la suma de
        la escuela y devolver el resultado
```

⁷Para simplificar, asumimos que n es una potencia de 2. Un truco sencillo para garantizar esta asunción consiste en añadir el número adecuado de ceros al principio de x e y , lo que, como mucho, duplicará su longitud. Alternativamente, si n es impar, también es válido separar x e y en dos números que tengan casi la misma longitud.

⁸Los algoritmos recursivos necesitan uno o más casos base, para que no sigan llamándose a sí mismos hasta el final de los tiempos. En esta ocasión, el caso base es: si x e y son números de 1 dígito, multiplicarlos en una operación básica y devolver el resultado.

⁹En nuestro pseudocódigo, utilizaremos “=” para indicar una comprobación de igualdad y “:=” para indicar la asignación de variables.

¿Es el algoritmo RECINTMULT más rápido o más lento que el algoritmo de la escuela? Posiblemente no tengas una respuesta clara a la pregunta. Para resolver el enigma tendrás que esperar al capítulo 4.

1.3.3 Multiplicación de Karatsuba

La multiplicación de Karatsuba es una versión optimizada del algoritmo RECINTMULT. Volvemos a empezar desde la expansión (1.1) de $x \cdot y$ en términos de a, b, c y d . El algoritmo RECINTMULT utiliza cuatro llamadas recursivas, una por cada uno de los productos de (1.1) entre números de $n/2$ dígitos. Pero *no nos preocupamos por $a \cdot d$ o $b \cdot c$* , salvo por el interés que tenemos en su suma $a \cdot d + b \cdot c$. Con solo tres cantidades en las que pensar ($a \cdot c, a \cdot d + b \cdot c$ y $b \cdot d$), ¿podremos salir airosos con solo tres llamadas recursivas?

Para comprobarlo, utilizamos inicialmente dos llamadas recursivas para calcular $a \cdot c$ y $b \cdot d$, igual que hicimos antes.

Paso 1: Calcular recursivamente $a \cdot c$.

Paso 2: Calcular recursivamente $b \cdot d$.

En vez de calcular recursivamente $a \cdot d$ o $b \cdot c$, lo que calculamos recursivamente es el producto de $a + b$ y $c + d$.¹⁰

Paso 3: Calcular $a + b$ y $c + d$, utilizando la suma de la escuela, y calcular recursivamente $(a + b) \cdot (c + d)$.

La clave de la multiplicación de Karatsuba la encontramos en el trabajo del matemático de principios del siglo XIX Carl Friedrich Gauss, que pensaba en cómo multiplicar números complejos. Restar los resultados de los dos primeros pasos del resultado del tercero, nos dará lo que buscamos, el coeficiente central de $a \cdot d + b \cdot c$ en (1.1):

$$\underbrace{(a + b) \cdot (c + d) - a \cdot c - b \cdot d}_{=a \cdot c + a \cdot d + b \cdot c + b \cdot d} = a \cdot d + b \cdot c.$$

Paso 4: Restar los resultados de los dos primeros pasos del resultado del tercero, para obtener $a \cdot d + b \cdot c$.

El paso final calcula (1.1), igual que con el algoritmo RECINTMULT.

¹⁰Los números $a + b$ y $c + d$ podrían tener hasta $(n/2) + 1$ dígitos, pero el algoritmo seguirá funcionando.

Paso 5: Calcular (1.1) sumando los resultados de los pasos 1, 2 y 4, después de añadir n ceros al final de la respuesta del paso 1 y $n/2$ ceros al final de la respuesta del paso 4.

KARATSUBA

Entrada: dos enteros positivos x e y de n dígitos.

Salida: el producto $x \cdot y$.

Asunción: n es una potencia de 2.

```
si  $n = 1$  entonces // caso base
    calcular  $x \cdot y$  en un paso y devolver el resultado
en otro caso // caso recursivo
     $a, b :=$  primera y segunda mitades de  $x$ 
     $c, d :=$  primera y segunda mitades de  $y$ 
    calcular  $p := a + b$  y  $q := c + d$  usando la suma de la escuela
    calcular recursivamente  $ac := a \cdot c$ ,  $bd := b \cdot d$  y  $pq := p \cdot q$ 
    calcular  $adbc := pq - ac - bd$  usando la suma de la escuela
    calcular  $10^n \cdot ac + 10^{n/2} \cdot adbc + bd$  usando la suma de la
        escuela y devolver el resultado
```

Pues sí, la multiplicación de Karatsuba solo realiza tres llamadas recursivas. Ahorrarnos una de las llamadas sin duda tendrá un impacto en el tiempo de ejecución global pero, ¿de qué magnitud? ¿Es el algoritmo KARATSUBA más rápido que el algoritmo de multiplicación de la escuela? La respuesta no es evidente en absoluto, pero resulta una aplicación sencilla de las herramientas que aprenderás en el capítulo 4, para el análisis del tiempo de ejecución de este tipo de algoritmos de “divide y vencerás”.

Sobre el pseudocódigo

Este libro explica los algoritmos utilizando una combinación de pseudocódigo de alto nivel y lenguaje natural (como hemos visto en esta sección). Doy por hecho que cuentas con los conocimientos necesarios para traducir estas descripciones de alto nivel a código funcional, utilizando tu lenguaje de programación favorito. Otros libros y re-

cursos disponibles en internet proporcionan implementaciones específicas de diversos algoritmos en lenguajes de programación concretos.

La ventaja más notable de destacar las descripciones de alto nivel frente a implementaciones específicas, reside en su flexibilidad: mientras asumo una cierta familiaridad con *algún* lenguaje de programación, me resulta irrelevante cuál pueda ser este. Además, este método ayuda a la comprensión de los algoritmos a un nivel más profundo y conceptual, evitando los detalles de bajo nivel. Los programadores con experiencia y los científicos de la computación suelen diseñar y describir los algoritmos de forma similar.

Aun así, nada puede reemplazar a la compresión detallada de un algoritmo que surge de la creación de una implementación funcional. Te animo a que programes tantos algoritmos de este libro como te sea posible (lo que, además, es una excusa perfecta para aprender un lenguaje de programación nuevo). Encontrarás ayuda para ello en los problemas de programación del final de cada capítulo y en sus casos de prueba.

1.4 MERGESORT: el algoritmo

Esta sección y la siguiente nos ofrecerán un primer contacto con el análisis del tiempo de ejecución de un algoritmo no trivial, el famoso algoritmo MERGESORT, para ordenación por mezcla.

1.4.1 Motivación

MERGESORT es un algoritmo relativamente antiguo y, sin ninguna duda, ya era conocido por John von Neumann en 1945. ¿Por qué comenzar un curso de algoritmia moderno con un ejemplo tan primitivo?

Viejo, pero bueno. Aunque tiene más de 70 años, MERGESORT sigue siendo unos de los métodos preferidos para la ordenación. La realidad es que se utiliza constantemente y es el algoritmo de ordenación predeterminado en un buen número de bibliotecas de programación.

Algoritmo canónico de divide y vencerás. El paradigma de diseño de algoritmos de “divide y vencerás” es un método generalizado para resolución de problemas, con aplicaciones en muchos ámbitos diferentes. La idea básica consiste en dividir el problema en subproblemas más pequeños, resolver recursivamente esos subproblemas y, por último, combinar sus soluciones en otra para el problema original. MERGESORT resulta ser una presentación perfecta del paradigma de divide y vencerás, los beneficios que ofrece y los retos que presenta su análisis.

Medida de tu preparación. Nuestro tratamiento de MERGESORT te servirá como una buena indicación de si tus conocimientos actuales están acorde al nivel de este libro. Doy por hecho de que cuentas con las habilidades necesarias, tanto en programación como en matemáticas, para traducir (con algo de esfuerzo) la idea de alto nivel de MERGESORT a una implementación funcional, utilizando tu lenguaje de programación favorito, y de que serás capaz de seguir el análisis del tiempo de ejecución que realizaremos. Si esta sección y la siguiente tienen sentido para ti, significa que estás en buena forma para afrontar el resto del libro.

Motiva los principios rectores del análisis de algoritmos. Nuestro análisis del tiempo de ejecución de MERGESORT pondrá de relevancia un buen número de principios rectores más generales, como la identificación de los límites del tiempo de ejecución que corresponden a cada entrada de un tamaño determinado, así como la importancia de la tasa de crecimiento del tiempo de ejecución de un algoritmo (como una función del tamaño de la entrada).

Calentamiento para el método maestro. Analizaremos MERGESORT utilizando el “método del árbol de recursión”, que es una forma de identificar las operaciones realizadas por un algoritmo recursivo. El capítulo 4 se basa en estas ideas y culmina con el “método maestro”, una herramienta potente y sencilla de utilizar para delimitar el tiempo de ejecución de muchos algoritmos diferentes de divide y vencerás, incluyendo aquí a los algoritmos RECINTMULT y KARATSUBA, que hemos visto en la sección 1.3.

1.4.2 Ordenación

Si te fijas un poco, te darás cuenta de que los ordenadores son bastante eficaces a la hora de ordenar cosas de forma instantánea. Piensa en la lista de contactos de tu teléfono. ¿Aparecen en el orden en que los añadiste? Es evi-

dente que no, eso sería una locura. En su lugar, para facilitar la búsqueda, están en orden alfabético. Al mismo tiempo, tu hoja de cálculo favorita no tiene ningún problema en ordenar un archivo grande en base a cualquier criterio que se te ocurra. ¿Cómo lo hacen?

Esta es una versión canónica del problema de ordenación:

Problema: ordenación

Entrada: Un *array* de n números, en orden arbitrario.

Salida: Un *array* con los mismos números, ordenados de menor a mayor.

Por ejemplo, dado el *array* de entrada

5	4	1	8	7	2	6	3
---	---	---	---	---	---	---	---

el *array* de salida deseado es

1	2	3	4	5	6	7	8
---	---	---	---	---	---	---	---

En el ejemplo anterior, los ocho números del *array* de entrada son diferentes. La ordenación no tiene por qué ser más difícil cuando existen duplicados y, según las circunstancias, podría incluso ser más sencillo. Pero para mantener la discusión lo más sencilla posible, vamos a asumir (estamos entre amigos) que los números del *array* de entrada son siempre diferentes. Te animo a que pienses en cómo habría que modificar nuestros algoritmos de ordenación (si es que fuese necesario) para ocuparse de los duplicados¹¹.

Idear un algoritmo de ordenación no es muy difícil, siempre que no estés preocupado por optimizar el tiempo de ejecución. Quizá la técnica más

¹¹En la práctica, suele haber información (llamada *valor*) asociada a cada número (llamado *clave*). Por ejemplo, quizás quieras ordenar registros de empleados (por nombre, salario, etc.), utilizando sus números de la seguridad social como claves. Nos centraríamos en ordenar las claves, entendiendo que cada una de ellas conservaría su información asociada.

sencilla consiste en primero explorar el *array* de entrada, para identificar al elemento más pequeño y, después, copiarlo sobre el primer elemento del *array* de salida, luego realizarás otra exploración para identificar al segundo elemento y, así, sucesivamente. Este algoritmo se conoce como **SELECTIONSORT** (ordenación por selección). Quizá hayas oído hablar de **INSERTIONSORT** (ordenación por inserción), que podría verse como una implementación más hábil de la misma idea de aumentar iterativamente el prefijo del *array* de salida ordenado. Es posible que también conozcas **BUBBLESORT** (ordenación de burbuja), en el que identificas pares adyacentes de elementos desordenados y realizas intercambios sucesivos, hasta que el *array* completo queda ordenado. Todos estos algoritmos tienen tiempos de ejecución cuadráticos, lo que significa que (de forma análoga al algoritmo de multiplicación de la escuela, de la sección 1.2.3) el número de operaciones realizadas sobre *arrays* de longitud n crece en el orden de n^2 , el cuadrado de la longitud de la entrada.

¿Podemos hacerlo mejor? Gracias al paradigma de diseño de algoritmos de divide y vencerás, el algoritmo **MERGESORT** mejora extraordinariamente a cualquiera de los otros algoritmos de ordenación más directos¹².

1.4.3 Un ejemplo

La mejor forma de entender **MERGESORT** es utilizar la imagen de un ejemplo concreto (figura 1.3). Usaremos el *array* de entrada de la sección 1.4.2.

Como algoritmo de divide y vencerás recursivo, **MERGESORT** se llama a sí mismo con *arrays* más pequeños. La forma más sencilla de descomponer un problema de ordenación en otros problemas más pequeños, consiste en dividir el *array* de entrada por la mitad. Las primera y segunda mitades se ordenarán recursivamente. Por ejemplo, en la figura 1.3, las primera y segunda mitades del *array* de entrada son [5, 4, 1, 8] y [7, 2, 6, 3]. El paso final de “combinación” vuelve a unir los dos *arrays* ordenados de longitud 4 en uno solo que contiene los 8 números. A continuación, veremos más detalles de este paso, en el que la idea básica consiste en recorrer los índices de cada uno de los *subarrays* ordenados, completando el *array* de salida en orden de izquierda a derecha.

¹²Aunque generalmente superado por **MERGESORT**, **INSERTIONSORT** resulta útil en algunos casos, especialmente si el tamaño de la entrada es pequeño.

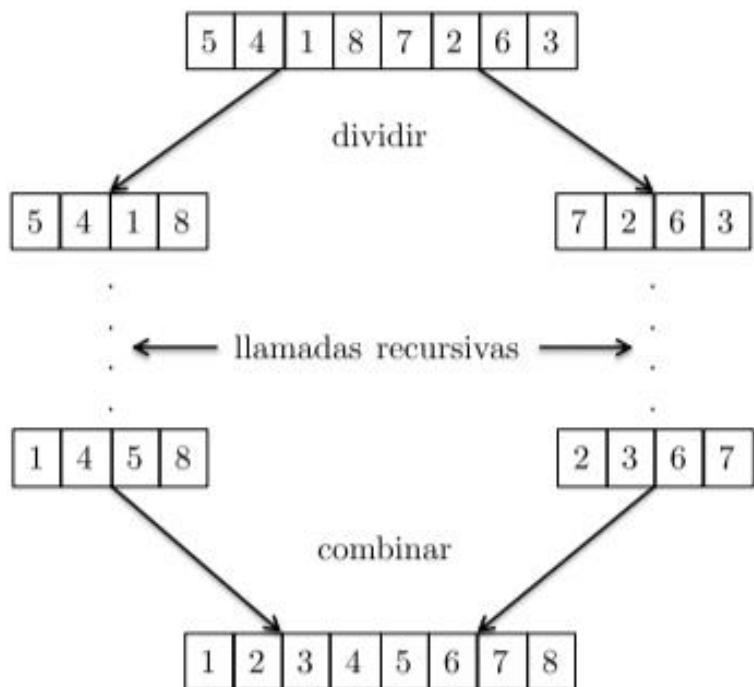


Figura 1.3: Vista general de MERGESORT sobre un ejemplo concreto.

1.4.4 Pseudocódigo

La imagen de la figura 1.3 sugiere el siguiente pseudocódigo para el problema general, con dos llamadas recursivas y un paso de combinación. Como es habitual, nuestra descripción no será, necesariamente, trasladable línea por línea a un código funcional (aunque está muy cerca).

MERGESORT

Entrada: array A con n enteros diferentes.

Salida: array con los mismos enteros, ordenados de menor a mayor.

```

// casos base ignorados
C := ordenar recursivamente la primera mitad de A
D := ordenar recursivamente la segunda mitad de A
devolver MERGE (C,D)
  
```

Hay algunas omisiones deliberadas en el pseudocódigo, que merecen un comentario. Como algoritmo recursivo, debería haber uno o más casos base, en los que ya no haya lugar a la recursividad y la respuesta se devuelva

directamente. Así, si el *array* de entrada *A* solo contiene 0 o 1 elementos, MERGESORT lo devolverá (ya está ordenado). El pseudocódigo no entra en consideración de cuáles son las “primera” y “segunda” mitades cuando *n* es impar, pero la interpretación más obvia (en la que una “mitad” tiene un elemento más que la otra) funcionará correctamente. Por último, el pseudocódigo ignora los detalles de implementación sobre cómo entregar los dos *subarrays* a sus respectivas llamadas recursivas. Esos detalles dependen del lenguaje de programación elegido. La ventaja del pseudocódigo de alto nivel es que ignora las cuestiones relativas a la implementación y se centra en los conceptos que trascienden a una implementación concreta.

1.4.5 La subrutina MERGE

¿Cómo implementamos el paso MERGE (combinación)? Llegados a este punto, las dos llamadas recursivas ya han hecho su trabajo y tenemos en nuestro poder dos *subarrays* *C* y *D*, de longitud $n/2$, ordenados. La idea consiste en recorrer ambos *subarrays* ordenados y llenar el *array* de salida ordenado de izquierda a derecha¹³.

MERGE

Entrada: arrays ordenados *C* y *D* (ambos de longitud $2/n$).

Salida: array ordenado *B* (de longitud *n*).

Asunción para simplificar: *n* es par.

```
1 i := 1
2 j := 1
3 para k := 1 hasta n hacer
4   si C[i] < D[j] entonces
5     B[k] := C[i]           // llenar el array de salida
6     i := i + 1                // incrementar i
7   en otro caso                  // D[j] < C[i]
8     B[k] := D[j]
9     j := j + 1
```

¹³Salvo que se indique lo contrario, numeramos las entradas de nuestros *arrays* comenzando en 1 (en vez de en 0), y utilizamos la sintaxis “*A*[*i*]” para referirnos a la entrada *i*-ésima de un *array* *A*. Estos detalles varían entre diferentes lenguajes de programación.

Recorremos el *array* de salida utilizando el índice k , y los *subarrays* ordenados con los índices i y j . Los tres *arrays* se recorren de izquierda a derecha. El bucle de la línea 3 implementa la pasada sobre el *array* de salida. En la primera iteración, la subrutina identifica el elemento mínimo entre C y D y lo copia a la primera posición del *array* de salida B . El elemento mínimo global estará, bien en C (en cuyo caso será $C[1]$, porque C ya está ordenado), o bien en D (en cuyo caso será $D[1]$, porque D también está ordenado). El incremento del índice correspondiente (i o j) evita que volvamos a tener en consideración el elemento que acabamos de copiar, y el proceso se repite para identificar el elemento mínimo que quede en C o en D (el segundo más pequeño globalmente). En general, el elemento más pequeño que todavía no se haya copiado a B será $C[i]$ o $D[j]$. La subrutina comprueba explícitamente cuál de los dos es menor y procede según corresponda. Como, con cada iteración, se copia el elemento mínimo de los considerados entre C y D , el *array* de salida se completará en orden.

Como es habitual, nuestro pseudocódigo es un tanto abstracto, para destacar el bosque sobre los árboles. Una implementación completa debería guardar un registro del momento en el que el recorrido de C o D supere el final del *array*, momento en el que los elementos restantes del otro *array* serían copiados (en orden) a las entradas finales de B . Ahora sería un buen momento para que intentases realizar tu propia implementación del algoritmo MERGESORT.

1.5 MERGESORT: el análisis

¿Cuál es el tiempo de ejecución del algoritmo MERGESORT, en función de la longitud n del *array* de entrada? ¿Es más rápido que otros métodos de ordenación como SELECTIONSORT, INSERTIONSORT o BUBBLESORT, más directos? Con “tiempo de ejecución” nos referimos al número de líneas de código que se ejecutarán en una implementación completa del algoritmo. Piensa en cómo sería recorrer la implementación, línea por línea, utilizando un depurador, con una “operación básica” cada vez. Nos interesa el número de pasos que realizaría el depurador antes de finalizar la ejecución.

1.5.1 Tiempo de ejecución de MERGE

Analizar el tiempo de ejecución del algoritmo MERGESORT parece una cuestión intimidante, ya que estamos ante un algoritmo recursivo que se llama a sí mismo una y otra vez. Así que vamos a practicar un poco con la

tarea, más sencilla, de entender el número de operaciones realizadas por una sola invocación a la subrutina MERGE, cuando se le llama sobre dos arrays ordenados, de longitud $\ell/2$ cada uno. Podemos hacerlo directamente, inspeccionando el código de la sección 1.4.5 (donde n corresponde a ℓ). En primer lugar, las líneas 1 y 2 realizan una inicialización, y lo contamos como dos operaciones. Después, tenemos un bucle que se ejecuta un total de ℓ veces. Cada iteración del bucle realiza una comparación en la línea 4, una asignación en las líneas 5 u 8, y un incremento en las líneas 6 o 9. El índice k del bucle también debe incrementarse en cada iteración del mismo. Esto significa que se realizan 4 operaciones básicas por cada una de las ℓ iteraciones del bucle¹⁴. Por ir concretando, hemos llegado a la conclusión de que la subrutina MERGE realiza un máximo de $4\ell + 2$ operaciones para combinar dos arrays ordenados, de longitud $\ell/2$ cada uno. Permíteme que abuse de nuestra amistad con una desigualdad cierta, aunque un tanto trámposa, que nos hará la vida más fácil: por cada entero positivo $\ell \geq 1$, tenemos $4\ell + 2 \leq 4\ell + 2\ell = 6\ell$. Es decir, 6ℓ es también un límite superior válido para el número de operaciones realizadas por la subrutina MERGE.

Lema 1.1 (Tiempo de ejecución de MERGE) *Por cada par de arrays de entrada ordenados C, D , de longitud $\ell/2$, la subrutina MERGE realiza un máximo de 6ℓ operaciones.*

Sobre lemas, teoremas y derivados

Cuando se escribe en términos matemáticos, las afirmaciones técnicas más importantes se denominan *teoremas*. Un *lema* es una afirmación técnica que ayuda a la demostración de un teorema (de la misma forma que MERGE ayuda con la implementación de MERGESORT). Un *corolario* es una afirmación que sigue inmediatamente a un resultado ya demostrado, como un caso especial de un teorema. Utilizaremos el término *proposición* para referirnos a afirmaciones técnicas independientes, que no resultan especialmente importantes por sí mismas.

¹⁴Podríamos discutir esta afirmación de las 4 operaciones. ¿Cuenta la comparación del índice k del bucle con su límite superior en cada una de las iteraciones? En ese caso serían 5. La sección 1.6 explica por qué esas diferencias no tienen, en realidad, ninguna importancia. Así que, como estamos entre amigos, vamos a acordar que la cifra de operaciones básicas por iteración es de 4.

1.5.2 Tiempo de ejecución de MERGESORT

¿Cómo podríamos avanzar desde el análisis directo de la subrutina MERGE a un análisis de MERGESORT, un algoritmo recursivo que se extiende en continuas invocaciones a sí mismo? Resulta especialmente aterradora la rápida proliferación de llamadas recursivas, cuyo número literalmente explota exponencialmente según aumenta la profundidad de la recursión. Lo único que juega a nuestro favor es el hecho de que cada llamada recursiva recibe una entrada sustancialmente menor a la de la llamada que la origina. Existe un enfrentamiento entre dos fuerzas opuestas: por un lado, la explosión de los diferentes subproblemas que se deben resolver y, por el otro, las entradas siempre menores de las que son responsables esos subproblemas. La reconciliación de estas dos fuerzas nos conducirá a nuestro análisis de MERGESORT. Al final, demostraremos el siguiente límite superior, específico y útil, en el número de operaciones realizadas por MERGESORT (a lo largo de todas sus llamadas recursivas).

Teorema 1.2 (Tiempo de ejecución de MERGESORT) *Por cada array de entrada de longitud $n \geq 1$, el algoritmo MERGESORT realiza un máximo de*

$$6n \log_2 n + 6n$$

operaciones, donde \log_2 indica el logaritmo en base 2.

Sobre logaritmos

Hay quien se siente innecesariamente intimidado ante la aparición de un logaritmo, que es, en realidad, un concepto muy terrenal. Para un entero positivo n , $\log_2 n$ solo significa lo siguiente: escribe n en una calculadora y cuenta el número de veces que tendrás que dividirlo entre 2 hasta que el resultado sea 1 o menos¹⁵. Por ejemplo, hay que realizar cinco operaciones de división entre 2 para llevar 32 a 1, por lo que $\log_2 32 = 5$. Diez divisiones entre 2 para llevar 1024 a 1, por lo que $\log_2 1024 = 10$. Estos ejemplos dejan patente el hecho de que $\log_2 n$ es mucho menor que n (compara 10 con 1024), especialmente cuando n es muy grande. Una gráfica puede confirmar esta intuición (figura 1.4).

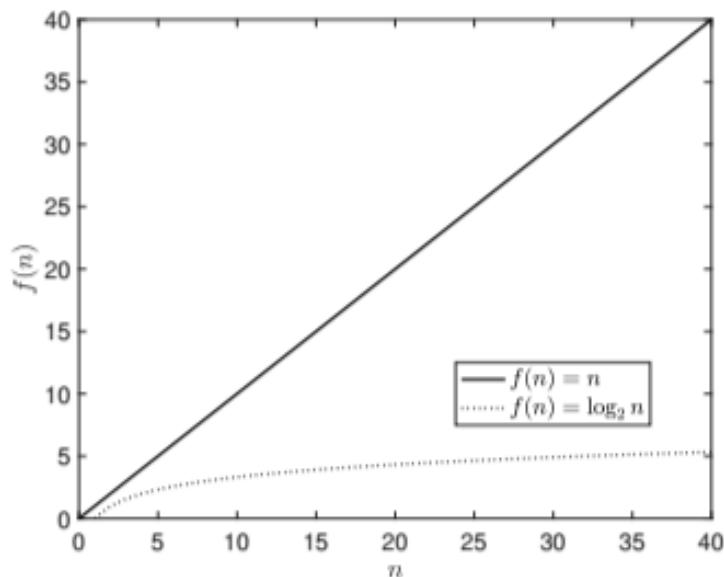


Figura 1.4: La función del logaritmo en base 2, $f(n) = \log_2 n$, crece mucho más despacio que la función lineal $f(n) = n$. Otras bases logarítmicas nos llevarían a gráficas muy similares.

El teorema 1.2 supone una victoria rotunda para el algoritmo MERGESORT y demuestra los beneficios del paradigma de diseño de algoritmos de divide y vencerás. Ya hemos mencionado que los tiempos de ejecución de los algoritmos de ordenación más sencillos, como SELECTIONSORT, INSERTIONSORT y BUBBLESORT, dependen *cuadráticamente* del tamaño de entrada n , lo que significa que el número de operaciones requeridas crece por una constante multiplicada por n^2 . En el teorema 1.2, uno de los factores de n es sustituido por $\log_2 n$. Como nos sugiere la figura 1.4, esto significa que MERGESORT se ejecutará, normalmente, mucho más rápido que los algoritmos de ordenación más sencillos, especialmente a medida que n vaya creciendo¹⁶.

1.5.3 Demostración del teorema 1.2

Ahora realizaremos un análisis completo del tiempo de ejecución de MERGESORT, sustanciando con ello la afirmación de que una técnica recursiva de divide y vencerás resulta en un algoritmo de ordenación más rápido que

¹⁵Siendo un poco pedantes, $\log_2 n$ no será un entero si n no es una potencia de 2, por tanto, lo que hemos descrito es, en realidad, $\log_2 n$ redondeado al entero más cercano. Podemos ignorar esta pequeña diferencia.

¹⁶Volveremos sobre este punto en la sección 1.6.3.

el obtenido mediante métodos más directos. Para simplificar, asumiremos que la longitud n del *array* de entrada es una potencia de 2. Esta asunción se podría evitar modificando pequeños detalles.

Para demostrar el límite del tiempo de ejecución del teorema 1.2, utilizaremos un *árbol de recursión*¹⁷ (ver la figura 1.5). La idea del método del árbol de recursión consiste en escribir, sobre una estructura de árbol, todo el trabajo realizado por un algoritmo recursivo, donde los nodos representan a las llamadas recursivas y los hijos de esos nodos a las llamadas hechas por los mismos. Esta estructura de árbol nos proporcionará un mecanismo de recuento de todo el trabajo realizado por MERGESORT, a lo largo de todas sus llamadas recursivas.

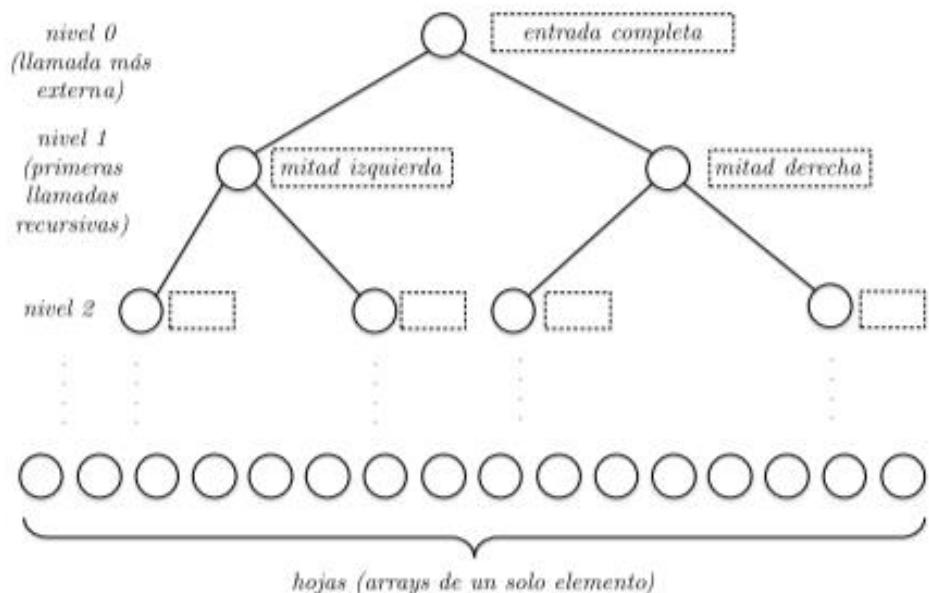


Figura 1.5: Árbol de recursión para MERGESORT. Los nodos corresponden a las llamadas recursivas. El nivel 0 representa la llamada más exterior a MERGESORT, el nivel 1 a sus llamadas recursivas, etc.

La raíz del árbol de recursión corresponde a la llamada más exterior a MERGESORT, en la que la entrada es el *array* de entrada original. Lo denominaremos como nivel 0 del árbol. Como cada invocación a MERGESORT se divide en dos llamadas recursivas, el árbol es binario (es decir, tiene dos hijos por nodo). El nivel 1 del árbol tiene dos nodos, que corresponden a las dos llamadas recursivas realizadas por la llamada más exterior, una pa-

¹⁷Por alguna razón, los científicos de la computación tienden a pensar que los árboles crecen hacia abajo.

ra la mitad izquierda del *array* de entrada y la otra para la mitad derecha. Cada llamada recursiva del nivel 1 realiza, a su vez, otras dos llamadas recursivas, que operarán, cada una de ellas, sobre una cuarta parte diferente del *array* de entrada original. Este proceso continuará hasta que, en algún momento, la recursión finalice con *arrays* de longitud 1 (los casos base).

Cuestionario 1.1

¿Cuántos niveles tendrá, aproximadamente, este árbol de recursión, en función de la longitud n del *array* de entrada?

- a) Un número constante (independiente de n)
- b) $\log_2 n$
- c) \sqrt{n}
- d) n

Solución y aclaraciones en la sección 1.5.4

Este árbol de recursión sugiere un método particularmente cómodo para contabilizar el trabajo realizado por MERGESORT, al presentarse nivel por nivel. Para implementar esta idea, necesitamos entender dos cosas: el número de subproblemas diferentes en un nivel de recursión j dado y la longitud de la entrada en cada uno de estos subproblemas.

Cuestionario 1.2

¿Cuál es el patrón? Rellena los espacios en blanco de la siguiente afirmación: en cada nivel $j = 0, 1, 2, \dots$ del árbol de recursión, hay [espacio en blanco] subproblemas, cada uno de ellos operando sobre un *subarray* de longitud [espacio en blanco].

- a) 2^j y 2^j , respectivamente
- b) $n/2^j$ y $n/2^j$, respectivamente
- c) 2^j y $n/2^j$, respectivamente
- d) $n/2^j$ y 2^j , respectivamente

Solución y aclaraciones en la sección 1.5.4

Ahora le daremos uso a este patrón y calcularemos el número de operaciones que realiza MERGESORT. Procederemos nivel por nivel, por lo que establecemos un nivel j del árbol de recursión. ¿Cuánto trabajo realizan las llamadas recursivas del nivel j , sin tener en cuenta el correspondiente a sus propias llamadas recursivas de niveles inferiores? Al estudiar el código de MERGESORT, vemos que solo hace tres cosas: dos llamadas recursivas y una llamada a la subrutina MERGE para combinar los resultados. Por lo tanto, si ignoramos el trabajo realizado por las llamadas recursivas, la única operación del subproblema del nivel j consiste en la invocación de MERGE. Gracias al lema 1.1, ya hemos entendido que habrá un máximo de 6ℓ operaciones, donde ℓ es la longitud del *array* de entrada de este subproblema.

Para obtener una visión completa, podemos expresar el trabajo total realizado por las llamadas recursivas del nivel j (sin tener en cuenta las llamadas recursivas posteriores) como

$$\frac{\# \text{ de subproblemas del nivel } j}{=2^j} \times \frac{\text{trabajo por subproblema del nivel } j}{=6n/2^j}$$

Utilizando la solución al cuestionario 1.2, sabemos que el primer término es igual a 2^j y que la longitud de la entrada de cada subproblema es $n/2^j$. Al tomar $\ell = n/2^j$, el lema 1.1 implica que cada subproblema del nivel j realiza un máximo de $6n/2^j$ operaciones. Podemos concluir que se realizarán un máximo de

$$2^j \cdot \frac{6n}{2^j} = 6n$$

operaciones, a lo largo de todas las llamadas recursivas en el nivel de recursión j -ésimo.

Sorprendentemente, el límite del trabajo realizado en un nivel j dado es independiente de j . Es decir, cada nivel del árbol de recursión aporta el mismo número de operaciones al análisis. El motivo de esto lo encontramos en el perfecto equilibrio entre dos fuerzas opuestas: el número de subproblemas se duplica en cada nivel, mientras que la cantidad de trabajo realizada por cada subproblema se divide a la mitad.

Nos interesa el número de operaciones realizadas a lo largo de *todos* los niveles del árbol de recursión. Según la solución del cuestionario 1.1, el árbol de recursión tiene $\log_2 n + 1$ niveles (del 0 al $\log_2 n$, ambos inclusive). Utilizando nuestro límite de $6n$ operaciones por nivel, podemos limitar el número total de operaciones a

$$\underbrace{\text{número de niveles}}_{=\log_2 n + 1} \times \underbrace{\text{trabajo por nivel}}_{\leq 6n} \leq 6n \log_2 n + 6n,$$

lo que coincide con el límite afirmado en el teorema 1.2. *QED*¹⁸

Sobre operaciones básicas

Medimos el tiempo de ejecución de un algoritmo como MERGESORT en términos del número de “operaciones básicas” realizadas. Como su nombre sugiere, una operación básica realiza una tarea simple (como sumar, comparar o copiar), empleando un conjunto reducido de variables sencillas (como enteros de 32 bits)¹⁹. Advertencia: en algunos lenguajes de programación de alto nivel, una sola línea de código puede ocultar un gran número de operaciones básicas. Por ejemplo, una expresión que implique a todos los elementos de un *array* largo se traducirá en un número de operaciones básicas proporcional a la longitud del *array*.

1.5.4 Soluciones a los cuestionarios 1.1–1.2

Solución al cuestionario 1.1

Respuesta correcta: (b). La respuesta correcta es $\approx \log_2 n$. La razón es que la entrada disminuye de tamaño por un factor de dos en cada nivel de la recursión. Si la longitud de la entrada en el nivel 0 es n , las llamadas recursivas del nivel 1 operan sobre *arrays* de longitud $n/2$, las llamadas recursivas del nivel 2 sobre *arrays* de longitud $n/4$ y, así, sucesivamente. La recursión finaliza en los casos base, con *arrays* de entrada con una longitud máxima de 1, lo que no deja lugar a nuevas llamadas recursivas. ¿Cuántos niveles de recursión son necesarios? El número de veces que haya que dividir n entre 2 para llegar a un valor de 1 como máximo. Si n es una potencia de 2 esta es, exactamente, la definición de $\log_2 n$ (de forma generalista será $\log_2 n$ redondeado al entero más cercano).

¹⁸“Q.e.d.” es la abreviatura de *quod erat demonstrandum* y significa “lo que se quería demostrar”. En terminología matemática, se utiliza al final de una demostración para indicar que esta ha quedado completa.

¹⁹Existen definiciones más precisas, pero no son necesarias.

Solución al cuestionario 1.2

Respuesta correcta: (c). La respuesta correcta es que hay 2^j subproblemas diferentes en el nivel de recursión j , y cada uno de ellos opera sobre un *subarray* de longitud $n/2^j$. En el primer punto comenzamos en el nivel 0, donde hay una llamada recursiva. En el nivel 1 habrá dos llamadas recursivas y, de forma general, ya que MERGESORT se llama a sí mismo dos veces, el número de llamadas recursivas para un nivel dado será el doble que en el nivel anterior. Esta duplicación sucesiva implica que existen 2^j subproblemas en cada nivel j del árbol de recursión. Igualmente, dado que cada llamada recursiva recibe únicamente la mitad de la entrada que la anterior, después de j niveles de recursión la longitud de la entrada se reduce a $n/2^j$. Visto de otra manera, ya sabemos que hay 2^j subproblemas en el nivel j y que el *array* de entrada original (de longitud j) se partitiona igualmente entre ellos (exactamente $n/2^j$ elementos por subproblema).

1.6 Principios rectores del análisis de algoritmos

Visto un primer análisis de algoritmos real (MERGESORT, en el teorema 1.2), ha llegado el momento de dar un paso atrás y plantear explícitamente tres elementos que han servido para realizar la valoración del tiempo de ejecución y la interpretación que podemos hacer de ella. Adoptaremos estas tres asunciones como principios rectores del razonamiento sobre algoritmos y las usaremos para fijar el concepto de “algoritmo rápido”.

El objetivo de estos tres principios es el de identificar el punto óptimo en el análisis de algoritmos, aquel que equilibre la precisión con la facilidad de manejo. Solo es posible realizar un análisis de tiempo de ejecución exacto en los algoritmos más sencillos pero, de forma más general, es necesario sacrificar algunos aspectos. Por otro lado, no nos interesa perder información valiosa por el camino, seguimos buscando un análisis matemático con suficiente poder predictivo como para determinar si un algoritmo es rápido o lento en la práctica. Una vez hallado el equilibrio correcto, podremos validar con garantía los tiempos de ejecución de docenas de algoritmos fundamentales, y esa garantía dibujará una imagen precisa de qué algoritmos suelen ser más rápidos que otros.

1.6.1 Principio 1: análisis del peor caso

El límite del tiempo de ejecución de $6n \log_2 n + 6n$, que vimos en el teorema 1.2, es válido para *cualquier array* de entrada de longitud n , con

independencia de su contenido. No damos por asumido ningún aspecto de la entrada más allá de su longitud n . Si, hipotéticamente, tuviésemos un enemigo cuyo único propósito en la vida fuese el de construir una entrada maligna, diseñada para hacer que MERGESORT se ejecutase de la forma más lenta posible, el límite de $6n \log_2 n + 6n$ permanecería intacto. Este tipo de análisis se denomina *análisis del peor caso*, ya que proporciona un límite del tiempo de ejecución válido incluso para las “peores” entradas.

Visto cómo, de forma natural, el análisis del peor caso desapareció en nuestra evaluación de MERGESORT, es lógico preguntarse qué más podemos hacer. Una técnica alternativa es la del “análisis del caso medio”, que explora el tiempo de ejecución promedio de un algoritmo, bajo la asunción de ciertas frecuencias relativas de diferentes entradas. Por ejemplo, en el problema de ordenación, podemos asumir que todos los posibles *arrays* de entrada tendrán la misma probabilidad de aparecer y, después, estudiaremos el tiempo de ejecución medio de los diferentes algoritmos de ordenación. Otra alternativa consiste en estudiar el algoritmo solamente sobre una pequeña colección de “instancias de rendimiento”, que consideremos representativas de entradas “típicas” o “del mundo real”.

Tanto el análisis del caso medio, como el análisis de instancias de rendimiento, pueden resultar útiles cuando alcances suficiente conocimiento del ámbito del problema a tratar y tengas cierta comprensión de qué entradas son más representativas que otras. El análisis del peor caso, para el que no se da nada por asumido en relación a la entrada, resulta particularmente apropiado para subrutinas de uso general, diseñadas para funcionar adecuadamente en diversos ámbitos de aplicaciones. Estos libros, con la intención de resultar útiles al mayor número de lectores posible, se centran en estas subrutinas de uso general y, en consecuencia, utilizan el análisis del peor caso para valorar el rendimiento de los algoritmos.

Además, el análisis del peor caso suele ser mucho más manejable matemáticamente que sus alternativas. Es una de las razones por las que el análisis del peor caso surgió durante el estudio de MERGESORT, aunque no estuviésemos centrándonos específicamente en las entradas con los peores casos.

1.6.2 Principio 2: análisis global

Los segundo y tercer principios rectores están íntimamente relacionados. Llamaremos al segundo *análisis global* (cuidado, porque este término no está generalizado). Este principio establece que no debemos preocuparnos excesivamente de pequeños factores constantes o términos de orden bajo al

valorar los límites del tiempo de ejecución. Ya hemos visto cómo funciona esta filosofía durante el análisis de MERGESORT: al analizar el tiempo de ejecución de la subrutina MERGE (lema 1.1), comenzamos demostrando un límite superior de $4\ell + 2$ sobre el número de operaciones (donde ℓ es la longitud del *array* de salida) y, después, establecimos el límite superior más sencillo de 6ℓ , incluso aunque se viese afectado por un factor constante muy grande. ¿Cómo podemos justificar relajarnos tanto en relación a los factores constantes?

Manejable matemáticamente. El primer motivo que justifica el análisis global lo encontramos en que, desde el punto de vista matemático, resulta más sencillo que perdernos entre factores constantes o términos de orden bajo. Este punto ya se hizo evidente en el análisis del tiempo de ejecución de MERGESORT.

Las constantes dependen de factores específicos del entorno. La segunda justificación es menos obvia, pero extraordinariamente importante. En el nivel de granularidad que utilizaremos para describir los algoritmos, como hicimos con MERGESORT, resultaría muy poco práctico obsesionarse con la exactitud de los factores constantes. Por ejemplo, durante nuestro análisis de la subrutina MERGE, existía una ambigüedad a la hora de determinar cuántas “operaciones básicas” se realizaban, exactamente, en cada iteración del bucle (4, 5 o, quizás, otro valor). Por lo tanto, diferentes interpretaciones del mismo pseudocódigo pueden llevar a factores constantes distintos. La ambigüedad crecerá aún más una vez que el pseudocódigo se traduzca a una implementación concreta en un lenguaje de programación de alto nivel y, después, se vuelva a traducir a código máquina (los factores constantes dependerán, inevitablemente, del lenguaje de programación utilizado, de la implementación específica y de las características concretas del compilador y del procesador). Nuestro objetivo es centrarnos en aquellas propiedades del algoritmo que trasciendan al lenguaje de programación y a la arquitectura específicos, y deben ser independientes, en el cálculo del límite del tiempo de ejecución, de pequeños cambios en los factores constantes.

Pérdida de poder predictivo. La tercera justificación es, sencillamente, que todo esto funcionará. Puede que te preocupe que el ignorar factores constantes haga que nos perdamos completamente, confundiéndonos en la creencia de que un algoritmo es rápido cuando, en la práctica, es lento, o viceversa. Afortunadamente, este caso no se dará en los algoritmos trata-

dos en este libro²⁰. Aunque no mantendremos un registro de los términos de orden bajo y de los factores constantes, las predicciones cualitativas de nuestros análisis matemáticos serán muy precisas (cuando el análisis sugiera que un algoritmo debe ser rápido, lo será en la práctica, y también al contrario). Así, aunque el análisis global descarta cierta información, conserva la que realmente nos importa: será una guía precisa de qué algoritmos suelen ser más rápidos que otros²¹.

1.6.3 Principio 3: análisis asintótico

Nuestro tercer y último principio rector es el uso del *análisis asintótico* y la observación de la tasa de crecimiento del tiempo de ejecución de un algoritmo, a medida que crece el tamaño de entrada n . Este prejuicio hacia las entradas grandes ya se puso de relevancia cuando interpretamos nuestro límite de tiempo de ejecución para MERGESORT (teorema 1.2), de $6n \log_2 n + 6n$ operaciones. En ese momento declaramos, con cierta arrogancia, que MERGESORT es “mejor que” otros algoritmos de ordenación más sencillos pero con tiempo cuadrático en relación al tamaño de la entrada, como INSERTIONSORT. Pero, ¿es esto cierto?

Para concretar, supongamos que tenemos un algoritmo de ordenación que realiza un máximo de $\frac{1}{2}n^2$ operaciones cuando ordena un *array* de longitud n , y consideremos la comparación

$$6n \log_2 n + 6n \text{ frente a } \frac{1}{2}n^2.$$

Si observamos, en la figura 1.6, el comportamiento de estas dos funciones, veremos que $\frac{1}{2}n^2$ es la expresión menor cuando n es pequeño (con un máximo aproximado de 90), mientras que $6n \log_2 n + 6n$ es más pequeño para todos los n más grandes. Así, cuando afirmemos que MERGESORT es más rápido que otros métodos de ordenación más sencillos, lo que implicamos, en realidad, es que lo es con *instancias lo suficientemente grandes*.

¿Por qué nos preocupan más las instancias grandes que las pequeñas? Porque las grandes son las únicas que precisan de ingenio algorítmico. Cual-

²⁰Con una posible excepción, el algoritmo determinista de selección en tiempo lineal de la sección opcional 6.3.

²¹Sin embargo, sigue siendo útil tener una idea general de cuáles podrían ser los factores constantes relevantes. Por ejemplo, en las versiones más optimizadas de MERGESORT, que encontrarás en muchas bibliotecas de programación, el algoritmo pasa de MERGESORT a INSERTIONSORT (debido a su mejor factor constante) una vez que la longitud del *array* de entrada es lo suficientemente pequeña (con, como mucho, siete elementos).

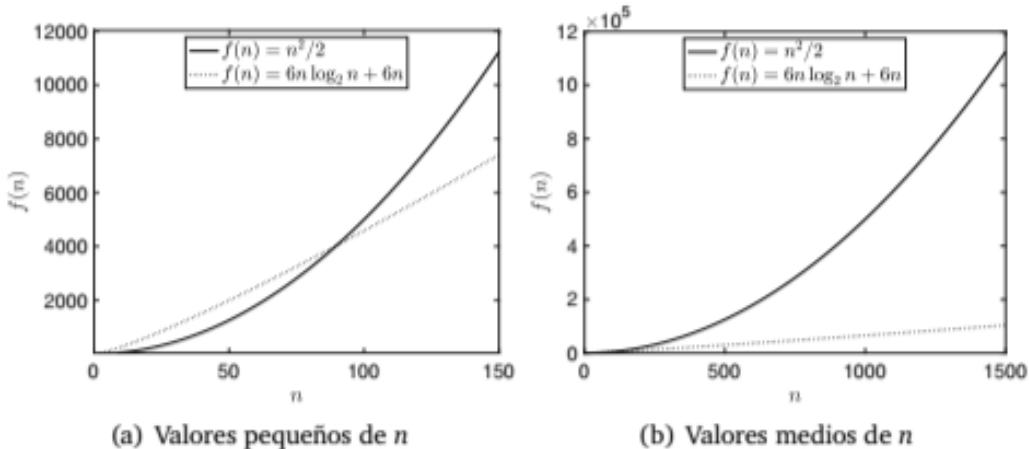


Figura 1.6: La función $\frac{1}{2}n^2$ crece mucho más rápido que $6n \log_2 n + 6n$, a medida que n aumenta. Las escalas de los ejes x e y en (b) son mayores que las de (a) por uno y dos órdenes de magnitud, respectivamente.

quier método de ordenación que se te pudiese ocurrir, ordenaría instantáneamente un *array* de longitud 1000 en un ordenador moderno, caso en el que no sería necesario aprender algoritmos de divide y vencerás.

Dado que la velocidad de los ordenadores crece de forma constante, quizás estés preguntando si llegará el día en que todos los problemas informáticos tendrán una solución trivial. La realidad es que disponer de ordenadores más rápidos hace que el análisis asintótico cobre más relevancia que nunca. Nuestras ambiciones de cálculo han ido creciendo a la par que la potencia computacional de la que disponemos y así, según va pasando el tiempo, nos enfrentamos con problemas de tamaños cada vez más grandes. Y la brecha de rendimiento entre algoritmos con diferentes tiempos de ejecución asintóticos se va ensanchando según crece el tamaño de la entrada. Por ejemplo, la figura 1.6.(b) muestra la diferencia entre las funciones $6n \log_2 n + 6n$ y $\frac{1}{2}n^2$ para valores grandes (aunque todavía modestos) de n , y ya al llegar a $n = 1500$ existe una diferencia con un factor aproximado de 10. Si aumentamos n por otro factor de 10, o 100 o, incluso, 1000, para acercarnos a problemas con un tamaño interesante, la diferencia entre las dos funciones será gigantesca.

Para ver el análisis asintótico desde otra perspectiva, vamos a suponer que tienes un presupuesto de tiempo fijo de, digamos, una hora al día. ¿Cómo escala el tamaño de los problemas resolubles en relación al aumento de la capacidad de cálculo? Con un algoritmo cuyo tiempo de ejecución sea

proporcional al tamaño de la entrada, un aumento de cuatro veces en la potencia de cálculo supondrá que podrás resolver problemas cuatro veces más grandes. Pero, con un algoritmo que crezca de forma proporcional al cuadrado del tamaño de la entrada, solo podrás resolver problemas el doble de grandes.

1.6.4 ¿Qué es un algoritmo “rápido”?

Nuestros tres principios rectores nos llevan a la siguiente definición de “algoritmo rápido”:

Un “algoritmo rápido” es aquel cuyo tiempo de ejecución en el peor caso crece lentamente en relación al tamaño de la entrada.

Nuestro primer principio rector, por el que buscamos los tiempos de ejecución que garanticen que no debemos asumir ningún conocimiento del ámbito, es la razón por la que nos centramos en el tiempo de ejecución del peor caso de un algoritmo. Nuestros segundo y tercer principios rectores, por los que los factores constantes son dependientes del lenguaje de programación y la potencia de cálculo, y los problemas grandes son los interesantes, son las razones por las que nos centramos en la tasa de crecimiento del tiempo de ejecución de un algoritmo.

¿A qué nos referimos al decir que el tiempo de ejecución de un algoritmo “crece lentamente”? El Santo Grial, para casi todos los problemas que trataremos, es un *algoritmo de tiempo lineal*, es decir, un algoritmo cuyo tiempo de ejecución sea proporcional al tamaño de la entrada. El tiempo lineal es incluso mejor que nuestro límite del tiempo de ejecución de MERGESORT, que es proporcional a $n \log_2 n$ y, en consecuencia, modestamente superlineal. Tendremos éxito en el diseño de algoritmos de tiempo lineal para algunos problemas pero, para otros, no será así. En cualquier caso, es la mejor situación a la que debemos aspirar.

Técnicas de coste cero

Se puede pensar en un algoritmo con tiempo de ejecución lineal, o casi lineal, como una técnica que tendrá, en esencia, “coste cero” (la cantidad de tiempo requerida superará escasamente a la necesaria para leer la entrada). La ordenación es un ejemplo canónico de una técnica de

coste cero y, más adelante, aprenderemos otras. Cuando cuentas con una técnica, aplicable a un problema, tan increíblemente rápida, ¿por qué no utilizarla? Por ejemplo, siempre puedes ordenar los datos en un paso de procesamiento previo, aunque no estés muy seguro de la utilidad que tendrá posteriormente. Uno de los objetivos de esta serie de libros es el de equipar tu caja de herramientas algorítmicas con tantas técnicas de coste cero como sea posible para, después, aplicarlas cuando sea conveniente.

Conclusiones

- ★ Un algoritmo es un conjunto de reglas bien definidas, para resolver problemas computacionales.
- ★ El número de operaciones básicas realizadas por el algoritmo que aprendiste en la escuela, para multiplicar dos enteros de n dígitos, crece como una función cuadrática del número n .
- ★ La multiplicación de Karatsuba es un algoritmo recursivo para la multiplicación de enteros y utiliza el truco de Gauss para ahorrar una llamada recursiva en relación a otro algoritmo recursivo más directo.
- ★ Los programadores experimentados y los científicos de la computación suelen diseñar y comunicar sus algoritmos utilizando descripciones de alto nivel en vez de implementaciones detalladas.
- ★ El algoritmo MERGESORT es un algoritmo de “divide y vencerás” que divide el *array* de entrada en dos mitades, ordena recursivamente cada una de ellas y combina el resultado mediante la subrutina MERGE.
- ★ Ignorando factores constantes y términos de orden bajo, el número de operaciones realizadas por MERGESORT, para ordenar n elementos, crece de acuerdo a

la función $n \log_2 n$. El análisis utiliza un árbol de recursión para organizar adecuadamente el trabajo realizado por cada llamada recursiva.

- ★ Debido a que la función $\log_2 n$ crece lentamente junto a n , MERGESORT suele ser más rápido que otros algoritmos de ordenación más sencillos, que necesitan un número cuadrático de operaciones. Para un n grande, la mejora es espectacular.
- ★ Los tres principios rectores del análisis de algoritmos son: (i) análisis del peor caso, para diseñar algoritmos de uso general que funcionan bien sin asumir ningún aspecto de la entrada; (ii) análisis global, que equilibra el poder predictivo con el manejo matemático, al ignorar factores constantes y términos de orden bajo; y (iii) análisis asintótico, que tiene prejuicios sobre las entradas grandes, que son las que requieren ingenio algorítmico.
- ★ Un “algoritmo rápido” es aquel en el que el tiempo de ejecución del peor caso crece lentamente junto al tamaño de la entrada.
- ★ Una “técnica de coste cero” es un algoritmo que se ejecuta en tiempo lineal o casi lineal, suponiendo un incremento mínimo al empleado en la propia lectura de la entrada.

Comprueba que lo has entendido

Problema 1.1 (S) Supongamos que ejecutamos MERGESORT sobre el siguiente *array* de entrada:

5	3	8	9	1	7	0	2	6	4
---	---	---	---	---	---	---	---	---	---

Avanzamos hasta el momento en el que las dos llamadas recursivas más exteriores se hayan completado, pero antes del último paso MERGE. Al

pensar en los dos *arrays* de salida de 5 elementos de las llamadas recursivas, como si de un solo *array* unido de 10 elementos se tratase, ¿qué número ocupará la séptima posición?

Problema 1.2 (P) Consideremos esta modificación del algoritmo MERGE-SORT: dividir el *array* de entrada en tercios (en vez de en mitades), ordenar recursivamente cada tercio y, por último, combinar los resultados utilizando una subrutina MERGE de tres canales. ¿Cuál será el tiempo de ejecución de este algoritmo como función de la longitud n del *array* de entrada, ignorando factores constantes y términos de orden bajo?

- a) n
- b) $n \log n$
- c) $n(\log n)^2$
- d) $n^2 \log n$

Problema 1.3 (P) Supongamos que tenemos k *arrays* ordenados, cada uno de ellos con n elementos, y queremos combinarlos en un único *array* de kn elementos. Una opción es utilizar la subrutina MERGE de la sección 1.4.5 repetidas veces, combinando primero los dos primeros *arrays*, después combinando ese resultado con el tercer *array*, después con el cuarto y, así, sucesivamente, hasta combinar el array k -ésimo y último. ¿Cuál es el tiempo de ejecución necesario para este algoritmo de combinación sucesiva, como función de k y n , ignorando factores constantes y términos de orden bajo?

- a) $n \log k$
- b) nk
- c) $nk \log k$
- d) $nk \log n$
- e) nk^2
- f) n^2k

Problema 1.4 (S) Consideremos nuevamente el problema de combinar k *arrays* ordenados de longitud n en un único *array* ordenado de longitud kn . Pensemos en el algoritmo que comience dividiendo los k *arrays* en $k/2$

pares de *arrays* y utilice la subrutina *MERGE* para combinar cada par, resultando en $k/2$ *arrays* ordenados de longitud $2n$. El algoritmo repite este paso hasta que solo quede un *array* ordenado de longitud kn . ¿Cuál es el tiempo de ejecución de esta técnica, como función de k y n , ignorando factores constantes y términos de orden bajo?

- a) $n \log k$
- b) nk
- c) $nk \log k$
- d) $nk \log n$
- e) nk^2
- f) n^2k

Problema 1.5 (S) Los siguientes problemas se refieren a un *array* de entrada de enteros, con posibles entradas duplicadas. ¿Cuáles de ellos se pueden resolver utilizando una única invocación a una subrutina de ordenación seguida de una sola pasada sobre el *array* ordenado (selecciona todas en las que sea posible)?

- a) Calcular el espacio mínimo entre cualquier par de elementos del *array*.
- b) Calcular el número de enteros distintos contenidos en el *array*.
- c) Calcular una versión de “eliminación de duplicados” del *array* de entrada, con lo que el *array* de salida contendrá exactamente una copia de cada uno de los distintos enteros del *array* de entrada.
- d) Calcular la moda (el entero más común) del *array*. Si se produce un empate y existen dos o más modas, el algoritmo debe devolverlas todas.
- e) Para esta parte, asumimos que los enteros del *array* son todos distintos y que la longitud del mismo es impar. Calcular la mediana del *array* (el “elemento central”, para el que los elementos menores igualan en número a los mayores).

Problemas más difíciles

Problema 1.6 (P) De forma intuitiva, cada algoritmo que verifica si un *array A* de n enteros contiene un entero t dado, debe examinar cada elemento del *array y*, en consecuencia, necesitará ejecutar, al menos, n operaciones básicas. Supongamos ahora que la entrada incluye no uno, sino k enteros t_1, t_2, \dots, t_k , y que el objetivo es identificar cuál de los t_i se encuentra en *A*. ¿Necesitará todo algoritmo que solucione este problema un mínimo de kn operaciones? ¿Depende la respuesta de k ?

Problema 1.7 (P) Recibes como entrada un *array* no ordenado de n números diferentes, donde n es una potencia de 2. Escribe un algoritmo que identifique el segundo número más grande del *array*, utilizando un máximo de $n + \log_2 n - 2$ comparaciones

Problemas de programación

Problema 1.8 Implementa el algoritmo de multiplicación de enteros de Karatsuba utilizando tu lenguaje de programación favorito²². Para lograr el mejor resultado posible, el programa solo debería invocar al operador de multiplicación del lenguaje sobre pares de números de un dígito.

Para concretar el reto, ¿cuál es el producto de los dos siguientes números de 64 dígitos²³?

3141592653589793238462643383279502884197169399375105820974944592

2718281828459045235360287471352662497757247093699959574966967627

²²Una reflexión: ¿te hará la vida más fácil el hecho de que el número de dígitos de cada entero sea una potencia de 2?

²³Si necesitas ayuda, o deseas comparar tus notas con las de otros lectores, puedes visitar el foro de discusión en www.algorithmsilluminated.org.

Notación asintótica

Este capítulo desarrolla los formalismos matemáticos que codifican nuestros principios rectores para el análisis de algoritmos (sección 1.6). El objetivo es identificar un punto óptimo de granularidad en el razonamiento sobre algoritmos (nos interesa suprimir detalles de segundo orden, como los factores constantes y los términos de orden bajo, y centrarnos en cómo escala el algoritmo en relación al crecimiento del tamaño de la entrada). Esto se hace, formalmente, a través de la notación *Big-O*, o cota superior asintótica, y sus derivados (conceptos que forman parte del vocabulario de todo programador o científico de la computación serio).

2.1 La esencia

Antes de entrar en los formalismos matemáticos de la notación asintótica, vamos a asegurarnos de que el tema está correctamente motivado, que tienes una visión clara de los que se intenta conseguir y que has visto un par de ejemplo sencillos e intuitivos.

2.1.1 Motivación

La notación asintótica nos ofrece el vocabulario elemental para tratar del diseño y análisis de los algoritmos. Es importante que entiendas a qué se refiere un programador cuando dice que una sección de código se ejecuta en “tiempo *Big-O* de n ”, mientras que otra lo hace en “tiempo *Big-O* de n cuadrado”.

Este vocabulario es tan ubicuo porque identifica ese “punto óptimo” del razonamiento sobre algoritmos. La notación asintótica es lo suficientemente

generalista como para eliminar todos aquellos detalles que queremos ignorar (detalles que dependen de la elección de la arquitectura, del lenguaje de programación, del compilador, etc.). Por otro lado, es lo suficientemente precisa como para permitirnos realizar comparaciones entre diferentes técnicas algorítmicas de alto nivel para la solución de un problema, especialmente en el caso de entradas grandes (aquellas que necesitan ingenio algorítmico). Por ejemplo, el análisis asintótico nos ayuda a diferenciar entre aproximaciones mejores y peores a la ordenación, a la multiplicación de dos enteros y, en general, a cualquier otra operación.

2.1.2 La idea de alto nivel

Si le pides a un programador en activo que te explique el sentido de la notación asintótica, probablemente te dirá algo así:

Notación asintótica en ocho palabras

eliminar factores constantes y términos de orden bajo
dependientes del sistema irrelevantes en entradas grandes

Enseguida veremos que la notación asintótica implica más que estas ocho palabras, pero dentro de diez años esto es lo que recordarás, la verdadera esencia.

Al analizar el tiempo de ejecución de un algoritmo, ¿qué interés podemos tener en prescindir de información como factores constantes y términos de orden bajo? Los términos de orden bajo, por definición, se van volviendo más irrelevantes a medida que nos centramos en entradas más grandes, que son las que requieren ingenio algorítmico. Al mismo tiempo, los factores constantes son, normalmente, muy dependientes de los detalles del entorno. Si no queremos comprometernos, al analizar un algoritmo, con un lenguaje de programación, una arquitectura o un compilador específicos, tiene todo el sentido que utilicemos formalismos que no se centren en factores constantes.

Por ejemplo, ¿recuerdas nuestro análisis de MERGESORT (secciones [1.4](#)–[1.5](#))? Le dimos a su tiempo de ejecución un límite superior de

$$6n \log_2 n + 6n$$

operaciones básicas, donde n es la longitud del array de entrada. En es-

te caso, el término de orden bajo es $6n$, ya que n crece más despacio que $n \log_2 n$, por lo que será suprimido en la notación asintótica. El factor constante de 6 también desaparece, dejándonos con la expresión $n \log n$, mucho más sencilla. Podríamos decir que el tiempo de ejecución de MERGESORT es el “Big-O de $n \log n$ ”, expresado como $O(n \log n)$, o que MERGESORT es un “algoritmo de tiempo $O(n \log n)$ ”¹. Como consecuencia de lo anterior, decir que algo es $O(f(n))$, para una función $f(n)$, significa que $f(n)$ es lo que nos queda después de eliminar los factores constantes y los términos de orden bajo². Esta “notación Big-O” clasifica los algoritmos en grupos, según sus tiempos de ejecución asintóticos del peor caso: algoritmos de tiempo lineal ($O(n)$), algoritmos de tiempo $O(n \log n)$, algoritmos de tiempo cuadrático ($O(n^2)$), algoritmos de tiempo constante ($O(1)$), etc.

Para ser totalmente claro, no estoy afirmando que los factores constantes no tengan ninguna influencia en el diseño de algoritmos. A lo que me refiero es a que, cuando quieras comparar técnicas fundamentalmente distintas de resolver un problema, el análisis asintótico suele ser la mejor herramienta para entender cuál de ellas tiene un mejor rendimiento, especialmente con entradas razonablemente grandes. Una vez que has determinado cuál es la mejor aproximación algorítmica de alto nivel para resolver un problema, puedes seguir trabajando sobre ella para mejorar el principal factor constante y, quizás, incluso los términos de orden bajo. En cualquier caso, si el futuro de tu empresa recién creada depende de la eficiencia en la implementación de un segmento de código en particular, no dejes de mejorarlo hasta que sea lo más rápido posible.

2.1.3 Cuatro ejemplos

Terminamos esta sección con cuatro ejemplos muy sencillos. Son tan sencillos que, si ya tienes experiencia con la notación Big-O, quizás deberías pasar directamente a la sección 2.2, para comenzar a aprender los formalismos matemáticos. Pero, si dichos conceptos te resultan completamente nuevos, estos sencillos ejemplos deberían ser una buena orientación.

Comencemos considerando el problema de buscar, dentro de un *array*, un entero t dado. Vamos a analizar el algoritmo directo, que realiza una ex-

¹Al ignorar los factores constantes, ni siquiera es necesario indicar la base del logaritmo (ya que las diferentes funciones logarítmicas se diferencian únicamente por un factor constante). Hablamos más de ello en la sección 4.2.2.

²Por ejemplo, incluso la función $10^{100} \cdot n$ es, técnicamente, $O(n)$. En estos libros estudiaremos, únicamente, los límites de tiempos de ejecución en los que el factor constante eliminado es razonablemente pequeño.

ploración lineal de todo el *array*, comprobando cada entrada para ver si contiene el entero t deseado.

Búsqueda en un *array*

Entrada: *array A* de n enteros y un entero t .

Salida: si *A* contiene a t o no.

```
para  $i := 1$  hasta  $n$  hacer
    si  $A[i] = t$  entonces
        devolver VERDADERO
    devolver FALSO
```

Este código se limita a comprobar, una por una, todas las entradas del *array*. Si, en algún momento, encuentra el entero t , devuelve VERDADERO. Pero si llega al final del *array* sin haber encontrado t , la respuesta es FALSO.

Todavía no hemos definido formalmente el significado de la notación *Big-O*, pero partiendo de nuestros comentarios quizás ya seas capaz de adivinar el tiempo de ejecución asintótico del código anterior.

Cuestionario 2.1

¿Cuál es el tiempo de ejecución asintótico del código anterior para buscar en un *array*, como función de la longitud n del mismo?

- a) $O(1)$
- b) $O(\log n)$
- c) $O(n)$
- d) $O(n^2)$

Solución y aclaraciones en la sección 2.1.4

Nuestros tres últimos ejemplos implican diferentes métodos de combinación de dos bucles. En primer lugar, pensemos en un bucle seguido de otro. Vamos a suponer que tenemos dos *arrays* de enteros *A* y *B*, ambos de lon-

gitud n , y que queremos saber si un entero t determinado se encuentra en alguno de ellos. Volvamos a pensar en el algoritmo directo, en el que buscamos en A y, en caso de no encontrar nada, continuamos buscando en B . Si tampoco encontramos t en B , devolveremos FALSO.

Búsqueda en dos arrays

Entrada: arrays A y B , de n enteros cada uno, y un entero t .

Salida: si A o B contienen a t o no.

```
para i := 1 hasta n hacer
    si A[i] = t entonces
        devolver VERDADERO
    para i := 1 hasta n hacer
        si B[i] = t entonces
            devolver VERDADERO
        devolver FALSO
```

¿Cuál es, según Big-O, el tiempo de ejecución de este código más largo?

Cuestionario 2.2

¿Cuál es el tiempo de ejecución asintótico del código anterior para buscar en dos arrays, como función de las longitudes n de los mismos?

- a) $O(1)$
- b) $O(\log n)$
- c) $O(n)$
- d) $O(n^2)$

Solución y aclaraciones en la sección 2.1.4

Veamos ahora un ejemplo más interesante, en el que los dos bucles están anidados en vez de ser secuenciales. Supongamos que queremos verificar

si dos arrays de longitud n tienen un número en común. La solución más sencilla consiste en comprobar todas las posibilidades. Es decir, por cada índice i del array A y cada índice j del array B , verificamos si $A[i]$ es el mismo número que $B[j]$. Si lo es, devolvemos VERDADERO. En caso de agotar todas las posibilidades sin haber hallado dos elementos iguales, podemos devolver FALSO.

Comprobación de un elemento común

Entrada: arrays A y B de n enteros cada uno.

Salida: si existe, o no, un entero t contenido tanto en A como en B .

```
// bucle exterior  
para i := 1 hasta n hacer  
    // bucle interior  
    para j := 1 hasta n hacer  
        si A[i] = B[j] entonces  
            devolver VERDADERO  
    devolver FALSO
```

La pregunta es la habitual: utilizando notación Big-O, ¿cuál es el tiempo de ejecución de este código?

Cuestionario 2.3

¿Cuál es el tiempo de ejecución asintótico del código anterior para la comprobación de un elemento común, como función de las longitudes n de los arrays?

- a) $O(1)$
- b) $O(\log n)$
- c) $O(n)$
- d) $O(n^2)$

Solución y aclaraciones en la sección [2.1.4](#)

Nuestro último ejemplo vuelve a utilizar bucles anidados, pero esta vez lo que estamos buscando son entradas duplicadas en un único *array A*, en vez de en dos diferentes. Este es el código que debemos analizar.

Comprobación de duplicados

Entrada: *array A* de n enteros.
Salida: si *A* contiene, o no, un mismo entero más de una vez.

```
para i := 1 hasta n hacer
    para j := i + 1 hasta n hacer
        si A[i] = A[j] entonces
            devolver VERDADERO
    devolver FALSO
```

Podemos encontrar dos pequeñas diferencias entre este código y el anterior. La primera, y más evidente, es que estamos comparando el i -ésimo elemento de *A* con el j -ésimo elemento de *A*, en vez de con el j -ésimo elemento de un supuesto *array B*. La segunda, más sutil, es que el bucle interior comienza en el índice $i + 1$ en vez de en 1. ¿Por qué no empezamos en 1, como hacíamos antes? Porque entonces devolveríamos VERDADERO en la primera iteración (evidentemente, $A[1] = A[1]$), independientemente de que hubiera duplicados o no. Para obtener el resultado correcto, podríamos evitar todas las iteraciones en las que i y j fuesen iguales, pero esto seguiría siendo un desperdicio de tiempo: cada par de elementos $A[h]$ y $A[k]$ de *A* sería comparado dos veces (una vez cuando $i = h$ y $j = k$ y la otra cuando $i = k$ y $j = h$), mientras que nuestro código solo realiza la comparación una vez.

Y, nuevamente, la pregunta es la habitual: utilizando notación *Big-O*, ¿cuál es el tiempo de ejecución de este código?

Cuestionario 2.4

¿Cuál es el tiempo de ejecución asintótico del código anterior para la comprobación de duplicados, como función de la longitud n del *array*?

- a) $O(1)$

- b) $O(\log n)$
- c) $O(n)$
- d) $O(n^2)$

Solución y aclaraciones en la sección [2.1.4](#)

Estos ejemplos básicos deberían haberte dado una idea bastante precisa de cómo se define la notación *Big-O* y qué trata de lograr. A continuación, pasaremos a ver el desarrollo matemático de la notación asintótica y algunos algoritmos interesantes más.

2.1.4 Soluciones a los cuestionarios 2.1–2.4

Solución al cuestionario 2.1

Respuesta correcta: (c). La respuesta correcta es $O(n)$. De forma equivalente, podemos decir que el algoritmo tiene un tiempo de ejecución *lineal en n*. ¿Por qué es así? El número exacto de operaciones realizadas depende de la entrada: si el número objetivo t se encuentra, o no, en el *array A* y, de ser así, en qué posición del mismo. En el peor caso, cuando t no está en el *array*, el código realizará una búsqueda sin éxito, explorando todo el *array* (en n iteraciones del bucle) y devolviendo FALSO. La observación clave se encuentra en que el código realiza un número constante de operaciones para cada entrada del *array* (comparar $A[i]$ con t , incrementar el índice i del bucle, etc.). En este caso, “constante” responde a algún número diferente de n , como 2 o 3. Podemos discutir, viendo el código, cuál es exactamente esta constante pero, sea la que sea, queda oportunamente eliminada en la notación *Big-O*. Debido a que ignorar los factores constantes y los términos de orden bajo nos deja con un límite de n para el total de operaciones, el tiempo de ejecución asintótico de este código es $O(n)$.

Solución al cuestionario 2.2

Respuesta correcta: (c). La respuesta es la misma que antes, $O(n)$. El motivo es que el número de operaciones realizado en el peor caso (una búsqueda sin éxito) es el doble que en el código anterior: una búsqueda en el primer *array* y, después, otra en el segundo. Este factor adicional de

`2` solo aporta a la constante principal del límite del tiempo de ejecución y, en consecuencia, se elimina al utilizar la notación *Big-O*. Por lo tanto este algoritmo, al igual que el anterior, es de tiempo lineal.

Solución al cuestionario 2.3

Respuesta correcta: (d). En esta ocasión la respuesta ha cambiado. En el caso de este código, el tiempo de ejecución no es $O(n)$, sino $O(n^2)$ (“*Big-O* de n cuadrado”, también llamado “algoritmo de tiempo cuadrático”). Con este algoritmo, si multiplicas las longitudes de los *arrays* de entrada por 10, el tiempo de ejecución crecerá por un factor de 100 (en vez de 10, como ocurriría en un algoritmo de tiempo lineal).

¿Por qué tiene este código un tiempo de ejecución de $O(n^2)$? El código vuelve a realizar un número constante de operaciones por cada iteración del bucle (es decir, por cada elección de los índices i y j) y un número constante de operaciones fuera de los bucles. La diferencia es que ahora tenemos un total de n^2 iteraciones en este doble bucle: una por cada elección de $i \in \{1, 2, \dots, n\}$ y $j \in \{1, 2, \dots, n\}$. En nuestro primero ejemplo, solo había n iteraciones en un bucle sencillo. En el segundo, debido a que el primer bucle finalizaba antes de que comenzase el segundo, solo teníamos un total global de $2n$ iteraciones. En este caso, por *cada una* de las n iteraciones del bucle exterior, el código realiza n iteraciones en el bucle interior. Esto supone un total de $n \times n = n^2$ iteraciones.

Solución al cuestionario 2.4

Respuesta correcta: (d). La respuesta a esta cuestión vuelve a ser la misma que la anterior, $O(n^2)$. El tiempo de ejecución vuelve a ser proporcional al número de iteraciones del bucle doble (con un número constante de operaciones por iteración). Entonces, ¿cuántas iteraciones se realizan? La respuesta es de, aproximadamente, $\frac{n^2}{2}$. Una forma de verlo es recordar que este código realiza, más o menos, la mitad de trabajo que el anterior (porque solo tiene en consideración los pares i, j donde $i < j$, y nunca aquellos en los que $i \geq j$). Otra forma de interpretarlo pasa por la observación de que hay, exactamente, una iteración por cada conjunto $\{i, j\}$ de dos índices diferentes de $\{1, 2, \dots, n\}$, y existen, precisamente, $\binom{n}{2} = \frac{n(n-1)}{2}$ subconjuntos de ese tipo³.

³ $\binom{n}{2}$ se lee “ n sobre 2” y, en ocasiones, es conocido como “coeficiente binomial”. Ver también la solución al cuestionario 3.1.

2.2 Notación Big-O

Esta sección presenta la descripción formal de la notación *Big-O*. Comenzaremos con una definición en lenguaje convencional, lo ilustraremos con una imagen y, finalmente, daremos una definición matemática.

Charla motivacional

Es absolutamente normal que, la primera vez que observes la definición matemática de la notación *Big-O*, sientas cierta confusión. Pero esta confusión no debe desanimarte. No significa falta de capacidad de comprensión por tu parte, sino que es una oportunidad para ser todavía más inteligente.

2.2.1 Definición en lenguaje convencional

La notación *Big-O* se refiere a las funciones $T(n)$ definidas sobre los enteros positivos $n = 1, 2, \dots$. Para nosotros, $T(n)$ indicará, casi siempre, un límite en el tiempo de ejecución de un algoritmo en el peor caso, como función del tamaño n de la entrada. ¿Qué implica afirmar que $T(n) = O(f(n))$ para una función $f(n)$ “canónica”, como n , $n \log n$ o n^2 ? Esta es la definición en lenguaje convencional.

Notación Big-O (versión en lenguaje convencional)

$T(n) = O(f(n))$ si, y solo si, $T(n)$ llega a estar limitado en su parte superior por una constante múltiplo de $f(n)$.

2.2.2 Definición visual

La figura 2.1 ilustra gráficamente la definición de la notación *Big-O*. El eje x corresponde al parámetro n , el eje y al valor de una función. Digamos que $T(n)$ es la función correspondiente a la línea sólida, y que $f(n)$ a la línea rayada inferior. $T(n)$ no está limitado por arriba por $f(n)$, pero multiplicar $f(n)$ por 3 resultará en la línea rayada superior, que se sitúa por encima de $T(n)$ una vez que estamos suficientemente a la derecha del gráfico, después

del “punto de cruce” en n_0 . Debido a que $T(n)$ terminará por estar limitado en su parte superior por un múltiplo constante de $f(n)$, podemos afirmar que $T(n) = O(f(n))$.

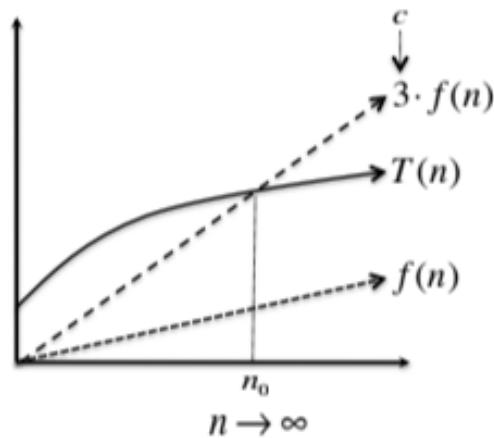


Figura 2.1: Imagen que ilustra cuándo $T(n) = O(f(n))$. La constante c cuantifica el “múltiplo constante” de $f(n)$, mientras que la constante n_0 cuantifica “terminará por”.

2.2.3 Definición matemática

Esta es la definición matemática de la notación *Big-O*, que es la que deberías utilizar en las demostraciones formales.

Notación *Big-O* (versión matemática)

$T(n) = O(f(n))$ si, y solo si, existen constantes positivas c y n_0 tales que

$$T(n) \leq c \cdot f(n) \quad (2.1)$$

para todos $n \geq n_0$.

Esto supone una traducción directa de la definición en lenguaje convencional de la sección 2.2.1. La desigualdad en (2.1) expresa que $T(n)$ debería tener un límite superior determinado por un múltiplo de $f(n)$ (múltiplo indicado por la constante c). La expresión “para todos $n \geq n_0$ ” supone que la desigualdad solo debe mantenerse en cierto momento, una vez que n sea

lo suficientemente grande (tamaño que viene especificado por la constante n_0). Por ejemplo, en la figura 2.1, la constante c corresponde a 3, mientras que n_0 corresponde al punto de cruce entre las funciones $T(n)$ y $c \cdot f(n)$.

Una perspectiva desde la teoría de juegos. Si quieras demostrar que $T(n) = O(f(n))$ para, por ejemplo, establecer que el tiempo de ejecución asintótico de un algoritmo es lineal en relación al tamaño de la entrada (correspondiente a $f(n) = n$), tu tarea consistirá en seleccionar las constantes c y n_0 de forma que (2.1) se verifique siempre que $n \geq n_0$. Una forma de plantearlo es desde la teoría de juegos, en la forma de una competición entre un oponente y tú. Tú mueves primero, y tienes que comprometerte con las constantes c y n_0 . Tu oponente moverá después y podrá elegir cualquier entero n que sea, al menos, n_0 . Ganarás si (2.1) se verifica, mientras que tu oponente ganará si lo que se verifica es la desigualdad opuesta $T(n) > c \cdot f(n)$.

Si $T(n) = O(f(n))$, existirán constantes c y n_0 tales que (2.1) se verifique para todos $n \geq n_0$ y tendrás una estrategia ganadora en la partida. En caso contrario, no importa qué c y n_0 elijas, pues tu oponente siempre podrá elegir un $n \geq n_0$ lo suficientemente grande como para invertir la desigualdad y ganar.

Una advertencia

Cuando afirmamos que c y n_0 son constantes, queremos decir que *no pueden depender de n*. Por ejemplo, en la figura 2.1, c y n_0 son números fijos (como 3 o 1000) y, en base a ello, consideramos la desigualdad (2.1), ya que n crece hasta un tamaño arbitrariamente grande (hacia infinito, si observamos la parte derecha de la figura). Si alguna vez te encuentras en la situación de decir “tomar $n_0 = n$ ” o “tomar $c = \log_2 n$ ” en una supuesta demostración con Big-O, deberás volver a empezar seleccionando unos c y n_0 que sean independientes de n .

2.3 Dos ejemplos básicos

Superado el enfrentamiento con la definición formal de la notación *Big-O*, veremos un par de ejemplos. Estos no nos aportan ninguna información que no tengamos ya, pero servirán como una buena verificación de que la notación *Big-O* logra lo que pretende, eliminar los factores constantes y los términos de orden bajo. También serán un buen calentamiento de cara a otros ejemplos, menos evidentes, que encontraremos más adelante.

2.3.1 Los polinomios de grado k son $O(n^k)$

Nuestra primera afirmación formal es que, si $T(n)$ es un polinomio con un grado k , entonces $T(n) = O(n^k)$.

Proposición 2.1 *Supongamos que*

$$T(n) = a_k n^k + \dots + a_1 n + a_0,$$

donde $k \geq 0$ es un entero no negativo y los a_i son números reales (positivos o negativos). Entonces $T(n) = O(n^k)$.

La proposición 2.1 dice que con un polinomio, en notación *Big-O*, de lo único de lo que te tienes que preocupar es del grado más alto que aparece en el mismo. En consecuencia, la notación *Big-O* sí que es, realmente, un supresor de los factores constantes y los términos de orden bajo.

Demostración de la proposición 2.1: Para demostrar esta proposición, necesitaremos utilizar la definición matemática de la notación *Big-O* (sección 2.2.3). Para satisfacer la definición, debemos encontrar un par de constantes positivas c y n_0 (ambas independientes de n), donde c cuantifica el múltiplo constante de n^k y n_0 cuantifica un “ n lo suficientemente grande”. Para lograr que el razonamiento sea fácil de seguir aunque, con ello, no menos misterioso, vamos a sacarnos de la manga los valores constantes: $n_0 = 1$ y c igual a la suma de los valores absolutos de los coeficientes⁴:

$$c = |a_k| + \dots + |a_1| + |a_0|.$$

Ambos números son independientes de n . Ahora necesitamos mostrar que estas elecciones de las constantes satisfacen la definición, implicando que $T(n) \leq c \cdot n^k$ para todos $n \geq n_0 = 1$.

⁴Recordemos que el *valor absoluto* $|x|$ de un número real x es igual a x cuando $x \geq 0$ y $-x$ cuando $x \leq 0$. En particular, $|x|$ nunca es negativo.

Para verificar esta desigualdad, establecemos arbitrariamente un entero positivo $n \geq n_0 = 1$. Necesitamos una sucesión de los límites superiores de $T(n)$, que culmine en el límite superior de $c \cdot n^k$. En primer lugar, aplicamos la definición de $T(n)$:

$$T(n) = a_k n^k + \dots + a_1 n + a_0.$$

Si tomamos el valor absoluto de cada coeficiente a_i del lado derecho, la expresión solo se hará más grande ($|a_i|$ es, al menos, igual de grande que a_i y, dado que n^i es positivo, $|a_i|n^i$ será, en consecuencia, al menos igual de grande que $a_i n^i$). Esto significa que

$$T(n) \leq |a_k|n^k + \dots + |a_1|n + |a_0|.$$

¿Por qué resulta útil este paso? Ahora que los coeficientes son no negativos, podemos utilizar un truco similar para convertir las diferentes potencias de n en una misma potencia común. Dado que $n \geq 1$, n^k es, al menos, igual de grande que n^i para cada $i \in \{0, 1, 2, \dots, k\}$. Como $|a_i|$ no es negativo, $|a_i|n^k$ es, al menos, igual de grande que $|a_i|n^i$. Esto significa que

$$T(n) \leq |a_k|n^k + \dots + |a_1|n^k + |a_0|n^k = \underbrace{(|a_k| + \dots + |a_1| + |a_0|)}_{=c} \cdot n^k.$$

Esta desigualdad se mantiene para todo $n \geq n_0 = 1$, que es, precisamente, lo que queríamos demostrar. \mathcal{QED}

¿Cómo puedes saber cómo elegir las constantes c y n_0 ? La técnica más habitual consiste en aplicarles ingeniería inversa. Esto implica pasar por una deducción como la que hemos mostrado y suponer sobre la marcha las posibles constantes que harán que la demostración sea válida. Veremos algunos ejemplos de este método en la sección [2.5](#).

2.3.2 Los polinomios de grado k no son $O(n^{k-1})$

Nuestro segundo ejemplo es, en realidad, un antiejemplo: un polinomio de grado k es $O(n^k)$ pero, normalmente, no es $O(n^{k-1})$.

Proposición 2.2 *Digamos que $k \geq 1$ es un entero positivo y define $T(n) = n^k$. En consecuencia, $T(n)$ no es $O(n^{k-1})$.*

La proposición [2.2](#) implica que los polinomios con distinto grado son también distintos en relación a la notación Big-O (si esto no fuese así, nuestra definición de la notación Big-O no valdría para nada).

Demostración de la proposición 2.2: La mejor forma de probar que una función no es la *Big-O* de otra consiste, normalmente, en una demostración por contradicción. En este tipo de demostraciones debemos asumir lo *contrario* a aquello que pretendemos probar y, a partir de ahí, construir sobre esta asunción siguiendo una serie de pasos lógicamente correctos, que culminen en una afirmación incuestionablemente falsa. Tal contradicción implica que lo asumido no puede ser cierto, lo que demuestra la afirmación buscada.

Así, asumimos que n^k es, de hecho, $O(n^{k-1})$ y procedemos a deducir una contradicción. ¿Qué significa que $n^k = O(n^{k-1})$? Pues que n^k llegará a estar limitado, en algún momento, por una constante múltiplo de n^{k-1} . Es decir, existen ciertas constantes positivas c y n_0 tales que

$$n^k \leq c \cdot n^{k-1}$$

para todos $n \geq n_0$. Como n es un número positivo, podemos cancelar n^{k-1} en ambos lados de la desigualdad, para obtener

$$n \leq c$$

para todos $n \geq n_0$. Esta desigualdad indica que la constante c es mayor que cualquier entero positivo, una afirmación incuestionablemente falsa (como contraejemplo, podemos tomar $c + 1$, redondeado al entero más cercano). Esto demuestra que nuestra asunción original de que $n^k = O(n^{k-1})$ no puede ser correcta y podemos concluir que n^k no es $O(n^{k-1})$. *QED*

2.4 Notaciones *Big-Omega* y *Big-Theta*

La notación *Big-O* es, con mucho, el concepto más importante y más utilizado en el tratamiento del tiempo de ejecución asintótico de los algoritmos. Pero también merece la pena conocer a dos parientes cercanos, las notaciones *Big-Omega* y *Big-Theta*. Si *Big-O* es análoga a decir “menor o igual que (\leq)”, *Big-Omega* y *Big-Theta* son análogas a “mayor o igual que (\geq)” e “igual que (=)”, respectivamente. Vamos a estudiarlas con un poco más de atención.

2.4.1 Notación *Big-Omega*

La definición formal de la notación *Big-Omega* guarda paralelismo con la de la notación *Big-O*. En lenguaje convencional, diremos que una función $T(n)$ es *Big-Omega* de otra función $f(n)$ si, y solo si, $T(n)$ llega a tener un

límite *inferior* de un múltiplo constante de $f(n)$. En este caso, escribimos $T(n) = \Omega(f(n))$. Al igual que antes, utilizaremos dos constantes c y n_0 para cuantificar “múltiplo constante” y “llega a tener”.

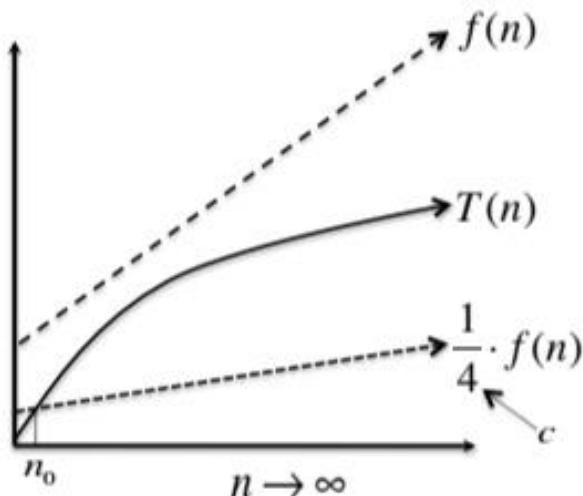
Notación Big-Omega (versión matemática)

$T(n) = \Omega(f(n))$ si, y solo si, existen dos constantes positivas c y n_0 tales que

$$T(n) \geq c \cdot f(n)$$

para todos $n \geq n_0$.

No es difícil imaginar el aspecto que tendrá la ilustración correspondiente:



$T(n)$ vuelve a corresponder a la función con la línea sólida. La función $f(n)$ está indicada por la línea rayada superior. Esta función no limita a $T(n)$ por debajo, pero si la multiplicamos por la constante $c = \frac{1}{4}$, el resultado (la línea rayada inferior) establece un límite inferior a $T(n)$ para todos los n posteriores al punto de cruce de n_0 . Por tanto, $T(n) = \Omega(f(n))$.

2.4.2 Notación Big-Theta

La notación *Big-Theta* o, sencillamente, la notación *Theta*, es análoga a “igual que”. Decir que $T(n) = \Theta(f(n))$ significa que $T(n) = \Omega(f(n))$ y

$T(n) = O(f(n))$. De forma equivalente, $T(n)$ llega a encontrarse “emparedada” entre dos múltiplos constantes diferentes de $f(n)$.⁵

Notación Big-Theta (versión matemática)

$T(n) = \Theta(f(n))$ si, y solo si, existen las constantes positivas c_1, c_2 y n_0 tales que

$$c_1 \cdot f(n) \leq T(n) \leq c_2 \cdot f(n)$$

para todos $n \geq n_0$.

Una advertencia

Los diseñadores de algoritmos suelen utilizar la notación *Big-O*, incluso cuando *Big-Theta* sería más precisa. Este libro continuará con esa tradición. Por ejemplo, pensemos en un subrutina que explore un *array* de longitud n , realizando un número constante de operaciones por entrada (como la subrutina *MERGE* de la sección 1.4.5). El tiempo de ejecución de esa subrutina será, evidentemente, $\Theta(n)$, pero es muy común mencionar únicamente $O(n)$.⁶ ¿Por qué tanto descuido? Porque, como diseñadores de algoritmos, solemos estar obsesionados con las garantías de los límites superiores de los tiempos de ejecución de nuestros algoritmos.

El siguiente cuestionario verifica tu comprensión de las notaciones *Big-O*, *Big-Omega* y *Big-Theta*.

⁵Demostrar esta equivalencia nos lleva a establecer que una de las versiones de la definición queda satisfecha si, y solo si, la otra también lo está. Si $T(n) = \Theta(f(n))$, de acuerdo a la segunda definición, entonces las constantes c_2 y n_0 demuestran que $T(n) = O(f(n))$, mientras que las constantes c_1 y n_0 demuestran que $T(n) = \Omega(f(n))$. En otro sentido, supongamos que podemos demostrar que $T(n) = O(f(n))$, utilizando las constantes c_2 y n'_0 , y que $T(n) = \Omega(f(n))$, utilizando las constantes c_1 y n''_0 . Entonces, $T(n) = \Theta(f(n))$ de acuerdo a la segunda definición, con las constantes c_1, c_2 y $n_0 = \max\{n'_0, n''_0\}$.

⁶Igualmente, los límites de $O(n)$ en los cuestionarios 2.1–2.2, y de $O(n^2)$ en los cuestionarios 2.3–2.4, serían más precisos si se expresasen como $\Theta(n)$ y $\Theta(n^2)$, respectivamente.

Cuestionario 2.5

Digamos que $T(n) = \frac{1}{2}n^2 + 3n$. ¿Cuál de las siguientes afirmaciones es cierta (puede haber más de una respuesta correcta)?

- a) $T(n) = O(n)$
- b) $T(n) = \Omega(n)$
- c) $T(n) = \Theta(n^2)$
- d) $T(n) = O(n^3)$

Solución y aclaraciones en la sección [2.4.5](#)

2.4.3 Notación Little-o

Existe una última pieza para completar la notación asintótica, la “notación Little-o”, que encontraremos de vez en cuando. Si la notación Big-O es análoga a “menor o igual que”, la notación Little-o se aplica a “estrictamente menor que”. En otras palabras, $T(n) = o(f(n))$ significa que la función $T(n)$ crece estrictamente más despacio que $f(n)$.⁷

Notación Little-o (versión matemática)

$T(n) = o(f(n))$ si, y solo si, por cada constante positiva $c > 0$, existe un n_0 tal que

$$T(n) \leq c \cdot f(n) \quad (2.2)$$

para todos $n \geq n_0$.

Demostrar que una función es la Big-O de otra solo requiere dos constantes, c y n_0 , elegidas una sola vez y para siempre. Para demostrar que una función es la Little-o de otra, necesitamos una prueba más sólida, en la que, para *cada* constante c , independientemente de lo pequeña que sea, $T(n)$ llegará a estar limitado en su parte superior por el múltiplo constante $c \cdot f(n)$. Es importante el hecho de que, en la definición de la notación

⁷Igualmente, existe una notación “Little-omega”, que corresponde a “estrictamente mayor que”, pero no tendremos ocasión de utilizarla. No hay una notación “Little-theta”.

Little-o, la constante n_0 elegida para cuantificar el “llegará a estar” puede depender de c (pero no de n), dándose la situación de que cuanto más pequeñas sean las constantes c , normalmente serán necesarias constantes n_0 más grandes. Por ejemplo, para cada entero positivo k , $n^{k-1} = o(n^k)$.⁸

2.4.4 ¿De dónde viene la notación?

La notación asintótica no fue inventada por científicos de la computación, se ha venido utilizando en el campo de la teoría de números desde la llegada del siglo XX. Donald E. Knuth, decano del análisis formal de algoritmos, propuso usarla como lenguaje normalizado para tratar las tasas de crecimiento y, en particular, para los tiempos de ejecución de los algoritmos.

“En base a los asuntos aquí tratados, propongo que los miembros de SIGACT⁹ y los editores de publicaciones sobre ciencias de la computación y matemáticas, adopten las notaciones O , Ω y Θ tal y como han quedado definidas, salvo que sea posible encontrar una alternativa mejor dentro de un plazo razonable.”¹⁰

2.4.5 Solución al cuestionario 2.5

Respuestas correctas: (b), (c), (d). Las tres últimas opciones son correctas y esperamos que, con algo de intuición, esté claro el porqué. $T(n)$ es una función cuadrática. El término lineal $3n$ no influye cuando n es grande, por lo que debemos esperar que $T(n) = \Theta(n^2)$ (respuesta (c)). Esto implica, automáticamente, que $T(n) = \Omega(n^2)$ y, en consecuencia, también $T(n) = \Omega(n)$ (respuesta (b)). Podemos ver que $\Omega(n)$ no resulta ser un límite inferior particularmente asombroso para $T(n)$ pero, en cualquier caso, es un límite legítimo. Igualmente, $T(n) = \Theta(n^2)$ implica automáticamente que $T(n) = O(n^2)$ y, con ello, también que $T(n) = O(n^3)$ (respuesta (d))¹¹.

⁸Y aquí está la demostración. Establecemos una constante arbitraria $c > 0$. En respuesta, determinamos que n_0 sea $\frac{1}{c}$, redondeado al entero más cercano. Entonces, para todos $n \geq n_0$, $n_0 \cdot n^{k-1} \leq n^k$ y, por lo tanto, $n^{k-1} \leq \frac{1}{n_0} \cdot n^k \leq c \cdot n^k$, como se pide.

⁹SIGACT es el grupo de interés especial de la ACM (*Association for Computing Machinery*) que se ocupa de la teoría de las ciencias de la computación y, en concreto, del análisis de algoritmos.

¹⁰Donald E. Knuth, “Big Omicron and Big Omega and Big Theta”, *SIGACT News*, abril-junio de 1976, página 23. Reimpreso en *Selected Papers on Analysis of Algorithms* (Center for the Study of Language and Information, 2000).

¹¹El hecho de que $T(n)$ pueda ser tanto $O(n^2)$ como $O(n^3)$ no supone ninguna diferencia en el hecho de que “ $5 \leq 5$ ” y “ $5 \leq 10$ ” sean afirmaciones sobre enteros totalmente

Demostrar formalmente estas afirmaciones pasa por mostrar las constantes oportunas que satisfagan las definiciones. Por ejemplo, si utilizamos $n_0 = 1$ y $c = \frac{1}{2}$, demostramos (b). Con $n_0 = 1$ y $c = 4$, demostramos (d). La combinación de estas constantes ($n_0 = 1$, $c_1 = \frac{1}{2}$, $c_2 = 4$) demuestra (c). Se puede utilizar el argumento utilizado en la demostración de la proposición 2.2, para demostrar formalmente que (a) no es una respuesta correcta.

2.5 Ejemplos adicionales

Esta sección está dirigida a aquellos lectores que desean practicar un poco más la notación asintótica. Si no estás interesado en ello, puedes ignorar los tres ejemplos que vienen a continuación y pasar al capítulo 3.

2.5.1 Suma de una constante a un exponente

Comenzamos con otro ejemplo de demostración de que una función es la Big-O de otra.

Proposición 2.3 Si

$$T(n) = 2^{n+10},$$

entonces $T(n) = O(2^n)$.

Es decir, sumar una constante al exponente de una función exponencial no modifica su tasa de crecimiento asintótica.

Demostración de la proposición 2.3: Para satisfacer la definición matemática de la notación Big-O (sección 2.2.3), necesitamos mostrar un par válido de constantes positivas c y n_0 (cada una de ellas independiente de n), tales que $T(n)$ sea, como máximo, $c \cdot 2^n$ para todos $n \geq n_0$. En la demostración de la proposición 2.1, nos sacamos esas constantes de la manga. En esta ocasión, les aplicaremos ingeniería inversa.

Buscamos una deducción que comience con $T(n)$ en el lado izquierdo, seguido de una sucesión de números grandes y que culmine en un múltiplo constante de 2^n . ¿Por dónde empezamos? El “10” del exponente resulta

válidas. La notaciones Big-O y Big-Theta se suelen mezclar en conversaciones informales pero, técnicamente, la notación Big-O indica, únicamente, un límite superior en la tasa de crecimiento de una función, y un límite que no es necesariamente estricto.

molesto, por lo que lo natural será empezar por separarlo:

$$T(n) = 2^{n+10} = 2^{10} \cdot 2^n = 1024 \cdot 2^n.$$

Ahora ya tiene todo mejor aspecto. El lado derecho es una constante múltiplo de 2^n y la deducción sugiere que deberíamos tomar $c = 1024$. Dada esta elección de c , tenemos $T(n) \leq c \cdot 2^n$ para todos $n \geq 1$, por lo que podemos tomar $n_0 = 1$. Este par de constantes certifica que $T(n)$ es, de hecho, $O(2^n)$. QED

2.5.2 Multiplicación de un exponente por una constante

A continuación viene otro antiejemplo, que muestra que una función *no es* la Big-O de otra.

Proposición 2.4 Si

$$T(n) = 2^{10n},$$

entonces $T(n)$ no es $O(2^n)$.

Es decir, multiplicar el exponente de una función exponencial por una constante cambia su tasa de crecimiento asintótica.

Demostración de la proposición 2.4: Al igual que con la proposición 2.2, el método habitual de demostrar que una función no es la Big-O de otra es mediante la contradicción. En consecuencia, vamos a asumir la afirmación opuesta a la de la proposición: que $T(n)$ es, de hecho, $O(2^n)$. Según la definición de la notación Big-O, esto significa que existen constantes positivas c y n_0 tales que

$$2^{10n} \leq c \cdot 2^n$$

para todos $n \geq n_0$. Como 2^n es un número positivo, podemos cancelarlo en ambos lados de la desigualdad, para obtener

$$2^{9n} \leq c$$

para todos $n \geq n_0$. Pero esta desigualdad resulta evidentemente falsa: la parte derecha corresponde a una constante fija (independiente de n), mientras que la izquierda tiende a infinito al crecer n . Esto demuestra que nuestra asunción de que $T(n) = O(2^n)$ no puede ser correcta, y podemos concluir que 2^{10n} no es $O(2^n)$. QED

2.5.3 Máximo frente a suma

Nuestro último ejemplo utiliza la notación *Big-Theta* (sección 2.4.2), la versión asintótica de “igual que”. Se muestra que, al utilizar notación asintótica, no existe diferencia entre tomar el máximo puntual de dos funciones no negativas o tomar su suma.

Proposición 2.5 *Digamos que f y g indican funciones desde los enteros positivos hasta los números reales no negativos, y definimos*

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

para cada $n \geq 1$. Entonces, $T(n) = \Theta(f(n) + g(n))$.

Una consecuencia de la proposición 2.5 es que un algoritmo que realiza un número constante (independiente de n) de subrutinas en tiempo $O(f(n))$ se ejecuta, él mismo, en tiempo $O(f(n))$. La solución al cuestionario 2.2 se puede ver como un caso especial sencillo de este hecho.

Demostración de la proposición 2.5: Recordemos que $T(n) = \Theta(f(n))$ significa que $T(n)$ llegará a verse emparedado entre dos constantes, múltiplos de $f(n)$, diferentes. Para darle más precisión, necesitamos mostrar tres constantes: la n_0 habitual y las c_1 y c_2 , que corresponden a los múltiplos menor y mayor de $f(n)$. Vamos a aplicar ingeniería inversa a los valores de estas constantes.

Elegiremos un entero positivo n arbitrario. Tenemos que

$$\max\{f(n), g(n)\} \leq f(n) + g(n),$$

porque la parte derecha no es más que la parte izquierda más un número no negativo ($f(n)$ o $g(n)$, el que sea menor). Igualmente

$$2 \cdot \max\{f(n), g(n)\} \geq f(n) + g(n),$$

porque la parte izquierda es igual a dos copias del valor mayor entre $f(n)$ y $g(n)$ y la parte derecha solo incluye una copia de cada uno. Al juntar estas dos desigualdades, encontramos que

$$\frac{1}{2} (f(n) + g(n)) \leq \max\{f(n), g(n)\} \leq f(n) + g(n) \quad (2.3)$$

para todo $n \geq 1$. Por lo tanto, $\max\{f(n), g(n)\}$ está, de hecho, encajonado entre dos múltiplos diferentes de $f(n) + g(n)$. Formalmente, la elección de

$n_0 = 1$, $c_1 = \frac{1}{2}$ y $c_2 = 1$ muestra (según (2.3)) que $\max\{f(n), g(n)\} = \Theta(f(n) + g(n))$. QED

Conclusiones

- ★ El propósito de la notación asintótica es el de suprimir factores constantes (que dependen demasiado del entorno) y términos de orden bajo (que resultan irrelevantes frente a entradas grandes).
- ★ Se dice que una función $T(n)$ es la “Big-O de $f(n)$ ”, expresado como “ $T(n) = O(f(n))$ ”, si llega a estar (con un n lo suficientemente grande) limitada en su valor superior por un múltiplo constante de $f(n)$. Es decir, existen unas constantes positivas c y n_0 tales que $T(n) \leq c \cdot f(n)$ para todos $n \geq n_0$.
- ★ Una función $T(n)$ es la “Big-Omega de $f(n)$ ”, expresada como “ $T(n) = \Omega(f(n))$ ”, si llega a estar limitada en su valor inferior por un múltiplo constante de $f(n)$.
- ★ Una función $T(n)$ es la “Big-Theta de $f(n)$ ”, expresada como “ $T(n) = \Theta(f(n))$ ” si se verifican tanto $T(n) = O(f(n))$ como $T(n) = \Omega(f(n))$.
- ★ Una afirmación Big-O es análoga a “menor o igual que”, Big-Omega lo es a “mayor o igual que” y Big-Theta a “igual que”.

Comprueba que lo has entendido

Problema 2.1 (S) Digamos que f y g son dos funciones no decrecientes con valores reales, definidas sobre enteros positivos, donde $f(n)$ y $g(n)$ son, al menos, 2 para todos $n \geq 1$. Asumimos que $f(n) = O(g(n))$, y tomamos c como una constante positiva. ¿Es $f(n) \cdot \log_2(f(n)^c) = O(g(n) \cdot \log_2(g(n)))$?

- Sí, para cualquier f , g y c .
- Nunca, independientemente de cuáles sean f , g y c .

- c) A veces sí y a veces no, dependiendo de la constante c .
- d) A veces sí y a veces no, dependiendo de las funciones f y g .

Problema 2.2 (P) Volvemos a asumir dos funciones positivas no decrecientes f y g , tales que $f(n) = O(g(n))$. ¿Es $2^{f(n)} = O(2^{g(n)})$ (puede haber varias opciones correctas, elige todas las que lo sean)?

- a) Sí, para cualquier f y g .
- b) Nunca, independientemente de cuáles sean f y g .
- c) A veces sí y a veces no, dependiendo de las funciones f y g .
- d) Sí, siempre que $f(n) \leq g(n)$ para todos los n suficientemente grandes.

Problema 2.3 Ordena, de menor a mayor, las siguientes funciones según su tasa de crecimiento, con $g(n)$ después de $f(n)$ si, y solo si, $f(n) = O(g(n))$.

- a) \sqrt{n}
- b) 10^n
- c) $n^{1.5}$
- d) $2^{\sqrt{\log_2 n}}$
- e) $n^{5/3}$

Problema 2.4 (S) Ordena, de menor a mayor, las siguientes funciones según su tasa de crecimiento, con $g(n)$ después de $f(n)$ si, y solo si, $f(n) = O(g(n))$.

- a) $n^2 \log_2 n$
- b) 2^n
- c) 2^{2^n}
- d) $n^{\log_2 n}$
- e) n^2

Problema 2.5 (S) Ordena, de menor a mayor, las siguientes funciones según su tasa de crecimiento, con $g(n)$ después de $f(n)$ si, y solo si, $f(n) = O(g(n))$.

- a) $2^{\log_2 n}$
- b) $2^{2^{\log_2 n}}$
- c) $n^{5/2}$
- d) 2^{n^2}
- e) $n^2 \log_2 n$

Problema 2.6 (S) Digamos que $f(n), g(n)$ indican funciones de enteros positivos a números reales positivos. Demuestra que $f(n) = O(g(n))$ si, y solo si, $g(n) = \Omega(f(n))$.

Problemas más difíciles

Problema 2.7 (S) Digamos que $T(n), f(n)$ indican dos funciones de enteros positivos a número reales positivos. Recuerda las definiciones formales de las notaciones *Big-O* (sección 2.2.3) y *Little-o* (sección 2.4.3).

- (a) Supongamos que sustituimos la desigualdad (2.1) en la definición de la notación *Big-O* por una desigualdad estricta ($T(n) < c \cdot f(n)$) y dejamos el resto de la definición sin cambios. Demuestra que $T(n) = O(f(n))$, en base a esta definición alternativa si, y solo si, $T(n) = O(f(n))$ según la definición original.
- (b) Supongamos que sustituimos la desigualdad (2.2) en la definición de la notación *Little-o* por una desigualdad estricta ($T(n) < c \cdot f(n)$) y dejamos el resto de la definición sin cambios. Demuestra que $T(n) = o(f(n))$, en base a esta definición alternativa si, y solo si, $T(n) = o(f(n))$ según la definición original.
- (c) Supuestamente, las notaciones *Big-O* y *Little-o* se corresponden con los conceptos “menor o igual que” y “estRICTAMENTE menor que”, respectivamente. ¿Cómo podemos conciliar esta afirmación con los puntos (a) y (b), que parecen mostrar que, en la notación asintótica, no hay diferencia entre ellas?

Algoritmos de divide y vencerás

Este capítulo permite practicar con el paradigma de diseño de algoritmos divide y vencerás, a través de su aplicación a tres problemas básicos.

Nuestro primer ejemplo es un algoritmo para el conteo de inversiones en un *array* (sección 3.2). Está relacionado con la medida de la similitud entre dos listas de clasificación, lo que puede ser relevante para hacer una buena recomendación a alguien, en base a tu conocimiento de sus preferencias y de las de terceros (llamado “filtrado colaborativo”)¹. Nuestro segundo algoritmo de divide y vencerás es el espectacular algoritmo recursivo de Strassen para la multiplicación de matrices, que mejorará el método iterativo que ya hemos visto (sección 3.3). El tercer algoritmo, que supone un material opcional y más avanzado, resuelve un problema fundamental de la geometría computacional: el cálculo del par de puntos más cercanos en el plano (sección 3.4).

3.1 El paradigma divide y vencerás

Ya hemos visto el ejemplo canónico de un algoritmo de divide y vencerás, MERGESORT (sección 1.4). De forma más generalista, el paradigma de diseño de divide y vencerás cuenta con tres pasos conceptuales.

El paradigma divide y vencerás

1. *Dividir* la entrada en subproblemas más pequeños.
2. *Resolver* los subproblemas recursivamente.
3. *Combinar* los subproblemas en la solución al problema original.

¹La presentación utilizada en la sección 3.2 está inspirada en el libro *Algorithm Design*, de Jon Kleinberg y Éva Tardos (Pearson, 2005).

Por ejemplo, en MERGESORT, el paso “dividir” separa el *array* de entrada en sus mitades izquierda y derecha, el paso “resolver” utiliza dos llamadas recursivas para ordenar los *subarrays* izquierdo y derecho, y el paso “combinar” se implementa mediante la subrutina MERGE (sección 1.4.5). En MERGESORT, y en muchos otros algoritmos de divide y vencerás, es este último paso el que requiere más ingenio. También existen algoritmos de divide y vencerás en los que la habilidad debe aplicarse al primer paso (ver QUICKSORT en el capítulo 5) o a la especificación de las llamadas recursivas (ver la sección 3.2).

Al enfrentarte a un problema nuevo, ¿cómo podrías saber si existe un buen algoritmo de divide y vencerás que lo resuelva? En general, este paradigma de diseño es muy apropiado para aquellos problemas en los que existe una forma evidente de dividir la entrada en subproblemas más pequeños, que no interfieran entre ellos. La mayoría de aplicaciones están relacionadas con la división de un *array* en dos mitades (como en MERGESORT, sección 3.2 y problemas 3.3–3.4), pero también encontraremos aplicaciones que implican números (sección 1.3 y problema 3.1), matrices (sección 3.3 y problema 3.5) y puntos del plano (sección 3.4 y problema 3.7)².

3.2 Conteo de inversiones en tiempo $O(n \log n)$

3.2.1 El problema

Esta sección estudia el problema del cálculo del número de inversiones en un *array*. Una *inversión* en un *array* se forma cuando un par de elementos están “desordenados”, lo que implica que un elemento del *array* es mayor que otro que aparecerá después.

Problema: conteo de inversiones

Entrada: un *array* A de enteros distintos.

Salida: el número de inversiones que hay en A (el número de pares (i, j) de índices del *array* en los que $i < j$ y $A[i] > A[j]$).

²La realidad: existen muchos problemas importantes para los que el paradigma divide y vencerás no ofrece ninguna ayuda. Los siguientes libros de esta serie se ocupan de otros paradigmas de diseño de algoritmos, como los voraces o la programación dinámica, a los que podrás recurrir cuando la técnica divide y vencerás sea inútil.

Por ejemplo, un *array* A que se encuentre ordenado no tendrá inversiones. Deberías convencerte de que lo contrario también es cierto: todo *array* que no esté ordenado contendrá, al menos, una inversión.

3.2.2 Un ejemplo

Consideremos el siguiente *array* de longitud 6:

1	3	5	2	4	6
---	---	---	---	---	---

¿Cuántas inversiones hay en este *array*? Una muy evidente está formada por el 5 y 2 (correspondientes a $i = 3$ y $j = 4$). Existen, exactamente, otros dos pares desordenados: uno formado por 3 y 2, y otro por 5 y 4.

Cuestionario 3.1

¿Cuál es el máximo de inversiones que puede haber en un *array* de 6 elementos?

- a) 15
- b) 21
- c) 36
- d) 64

Solución y aclaraciones en la sección 3.2.13

3.2.3 Filtrado colaborativo

¿Qué interés tenemos en el número de inversiones de un *array*? Puede que nos interese calcular la similitud numérica que cuantifique la cercanía de dos listas clasificadorias. Por ejemplo, supongamos que os pido a ti y a un amigo tuyo que clasifiquéis, de más favorita a menos, diez películas que hayáis visto los dos. ¿Serán vuestros gustos “similares” o “diferentes”? Una forma de responder a esta pregunta, de forma cuantitativa, es mediante el siguiente *array* de 10 elementos: $A[1]$ contiene la clasificación dada por

tu amigo a tu película favorita, $A[2]$ la clasificación de tu amigo sobre tu segunda película favorita, ..., y $A[10]$ la clasificación de tu amigo sobre tu película menos favorita. Así, si tu película favorita es *Star Wars*, pero ocupa el quinto lugar en la lista de tu amigo, diremos que $A[1] = 5$. Si vuestras clasificaciones son idénticas, este *array* estará ordenado y no habrá inversiones. Cuantas más inversiones encontraremos en un *array*, habrá más pares de películas sobre las que no estéis de acuerdo y, en consecuencia, más diferentes serán vuestros gustos.

Un motivo por el que podría interesarte realizar una medición de similitud entre clasificaciones es la realización de un *filtrado colaborativo*, una técnica utilizada para generar recomendaciones. ¿Cómo llegan las páginas web a realizar sugerencias de productos, películas, canciones, artículos, etc.? Mediante el filtrado colaborativo, cuya idea consiste en identificar a otros usuarios que tengan gustos similares y recomendarte elementos que hayan sido populares entre ellos. Por lo tanto, los algoritmos de filtrado colaborativo necesitan de una noción formal de “similitud” entre usuarios y el problema del cálculo de inversiones captura parte de la esencia de este problemas.

3.2.4 Búsqueda exhaustiva

¿A qué velocidad podemos calcular el número de inversiones de un *array*? Si no estamos muy creativos, siempre estará la búsqueda exhaustiva.

Búsqueda exhaustiva para contar inversiones

Entrada: *array A* de n enteros distintos.
Salida: el número de inversiones de *A*.

```
numInv := 0
para i := 1 hasta n - 1 hacer
    para j := i + 1 hasta n hacer
        si A[i] > A[j] entonces
            numInv := numInv + 1
devolver numInv
```

Indudablemente este algoritmo es correcto. Pero, ¿qué hay de su tiempo de ejecución? Por la solución al cuestionario 3.1, sabemos que el número

de iteraciones del bucle crece de forma cuadrática en relación a la longitud n del *array* de entrada. Como el algoritmo realiza un número constante de operaciones en cada iteración del bucle, su tiempo de ejecución asintótico es $\Theta(n^2)$. Recordemos el mantra del diseñador de algoritmos experimentado: *¿podemos hacerlo mejor?*

3.2.5 Una solución de divide y vencerás

La respuesta es sí y la solución será un algoritmo de divide y vencerás que se ejecute en tiempo $O(n \log n)$, lo que supone una enorme mejora con respecto a la búsqueda exhaustiva. El paso “dividir” será exactamente igual al del algoritmo MERGESORT, con una llamada recursiva para la mitad izquierda del *array* y otra para la mitad derecha. Para entender el trabajo residual que debemos realizar fuera de las llamadas recursivas, clasificaremos en tres tipos las inversiones (i, j) de un *array* A de longitud n :

1. *inversión izquierda*: una inversión en la que i, j se encuentran en la primera mitad del *array* (es decir, $i, j \leq \frac{n}{2}$),
2. *inversión derecha*: una inversión en la que i, j se encuentran en la segunda mitad del *array* (es decir, $i, j > \frac{n}{2}$),
3. *inversión separada*: una inversión en la que i se encuentra en la primera mitad y j en la segunda (es decir, $i \leq \frac{n}{2} < j$).

Por ejemplo, en el *array* de seis elementos de la sección 3.2.2, las tres inversiones son inversiones separadas.

La primera llamada recursiva, en la primera mitad del *array* de entrada, cuenta recursivamente todas las inversiones izquierdas (y nada más). Igualmente, la segunda llamada recursiva cuenta todas las inversiones derechas. La tarea restante consiste en contar aquellas inversiones no computadas en ninguna de las dos llamadas: las inversiones divididas. Este es el paso “combinar” del algoritmo, para el que tendremos que implementar una subrutina de tiempo lineal específica, análoga a la subrutina MERGE del algoritmo MERGESORT.

3.2.6 Algoritmo de alto nivel

Nuestra técnica de divide y vencerás se traduce en el siguiente pseudocódigo, dejando la subrutina COUNTSPLITINV sin implementar por el momento.

COUNTINV

Entrada: array A de n enteros distintos.

Salida: el número de inversiones de A .

```
si  $n = 0$  o  $n = 1$  entonces          // casos base
    devolver 0
en otro caso
     $leftInv := COUNTINV(\text{primera mitad de } A)$ 
     $rightInv := COUNTINV(\text{segunda mitad de } A)$ 
     $splitInv := COUNTSPLITINV(A)$ 
    devolver  $leftInv + rightInv + splitInv$ 
```

Las primera y segunda llamadas recursivas cuentan el número de inversiones izquierdas y derechas. Suponiendo que la subrutina COUNTSPLITINV calcule correctamente el número de inversiones separadas, COUNTINV calculará correctamente el número total de inversiones.

3.2.7 Idea clave: a cuestas de MERGESORT

Contar el número de inversiones separadas de un *array* en tiempo lineal es un objetivo ambicioso. Puede haber muchas inversiones separadas: si A consta de los números $\frac{n}{2} + 1, \dots, n$ en orden, seguidos de los números $1, 2, \dots, \frac{n}{2}$ en orden, habrá $n^2/4$ inversiones separadas. ¿Cómo es posible contabilizar un número cuadrático de elementos empleando una cantidad lineal de trabajo?

La gran idea consiste en diseñar nuestro algoritmo recursivo de conteo de inversiones de forma que se sitúe a cuestas del algoritmo MERGESORT. Esto implica pedirle más a nuestras llamadas recursivas, con el objetivo de facilitar el conteo del número de inversiones separadas. Cada llamada recursiva no será responsable únicamente del conteo de las inversiones del *array* recibido, sino que también devolverá una versión ordenada del mismo. Ya sabemos (gracias al teorema 1.2) que la ordenación es una operación básica de coste cero (ver la página 31), con un tiempo de ejecución de $O(n \log n)$, por lo que si apuntamos a ese límite del tiempo de ejecución de $O(n \log n)$, no hay motivo para no ordenar. Y ahora veremos cómo la tarea de combinar dos *subarrays* ordenados parece hecha a medida para identificar todas las inversiones separadas de un *array*.

Esta es la versión revisada del pseudocódigo de la sección 3.2.6, que cuenta las inversiones al tiempo que ordena el *array* de entrada.

SORT-AND-COUNTINV

Entrada: *array A* de n enteros distintos.

Salida: *array ordenado B* con los mismos enteros y el número de inversiones de *A*.

```
si  $n = 0$  o  $n = 1$  entonces           // casos base
    devolver (A, 0)
en otro caso
    (C, leftInv) := SORT-AND-COUNTINV(primer mitad de A)
    (D, rightInv) := SORT-AND-COUNTINV(segunda mitad de A)
    (B, splitInv) := MERGE-AND-COUNTSPLITINV(C, D)
    devolver (B, leftInv + rightInv + splitInv)
```

Seguimos necesitando una implementación de la subrutina MERGE-AND-COUNTSPLITINV. Sabemos cómo combinar dos listas ordenadas en tiempo lineal pero, ¿cómo podemos apoyarnos en esta tarea para contar también el número de inversiones separadas?

3.2.8 MERGE revisitado

Para ver por qué la combinación de dos *subarrays* ordenados revela, de forma natural, las inversiones separadas, vamos a revisitar el pseudocódigo de la subrutina MERGE.

MERGE

Entrada: *arrays ordenados C* y *D* (ambos de longitud $n/2$).

Salida: *array ordenado B* (longitud n).

Asunción para simplificar: n es par.

```
 $i := 1, j := 1$ 
para  $k := 1$  hasta  $n$  hacer
    si  $C[i] < D[j]$  entonces
         $B[k] := C[i], i := i + 1$ 
    en otro caso
         $B[k] := D[j], j := j + 1$            //  $D[j] < C[i]$ 
```

Para refrescar la memoria, recordemos que la subrutina MERGE aumenta paralelamente en uno el índice de cada uno de los *subarrays* ordenados (i para C y j para D), poblando el *array* de salida (B) de izquierda a derecha y ya ordenado (utilizando el índice k). La subrutina identifica, en cada iteración del bucle, al elemento más pequeño que todavía no se haya copiado a B . Como C y D están ordenados y todos los elementos anteriores a $C[i]$ y $D[j]$ ya se han copiado a B , los dos únicos candidatos a ser el menor elemento restante son $C[i]$ y $D[j]$. La subrutina determina cuál de los dos es menor y los copia en la siguiente posición del *array* de salida.

¿Qué tiene que ver la subrutina MERGE con contar el número de inversiones separadas? Comencemos con el caso especial de un *array* A que no contenga ninguna inversión separada (todas las inversiones de A son izquierdas o derechas).

Cuestionario 3.2

Supongamos que el *array* de entrada A no contiene inversiones separadas. ¿Cuál es la relación entre los *subarrays* ordenados C y D ?

- a) C contiene el elemento más pequeño de A , D el segundo más pequeño, C el tercero, etc.
- b) Todos los elementos de C son menores que todos los elementos de D .
- c) Todos los elementos de C son mayores que todos los elementos de D .
- d) No hay suficiente información para contestar a esta pregunta.

Solución y aclaraciones en la sección 3.2.13

Después de resolver el cuestionario 3.2, podrás ver que, al operar sobre un *array* sin inversiones separadas, MERGE se ejecuta de forma particularmente aburrida. Como cada elemento de C es menor que cada elemento de D , el menor elemento restante siempre es C (hasta que C quede vacío). Por tanto, la subrutina MERGE se limita a concatenar C y D , copiando primero todo C y, después, todo D . Esto nos sugiere que, quizás, las inversiones separadas puedan tener alguna relación con el número de elementos restantes en C cuando se copia un elemento de D al *array* de salida.

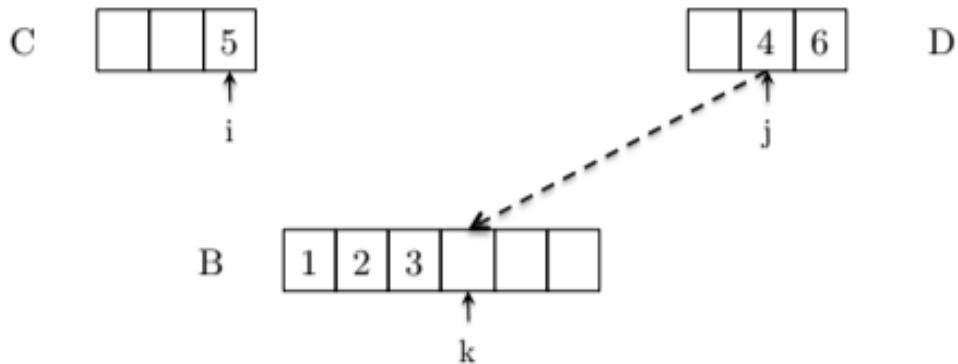


Figura 3.1: La cuarta iteración de la subrutina MERGE con los *subarrays* de entrada ordenados $\{1, 3, 5\}$ y $\{2, 4, 6\}$. Copiar el 4 desde D , con el 5 todavía en C , descubre la inversión separada que implica a estos dos elementos.

3.2.9 MERGE y las inversiones separadas

Para reforzar esta intuición, pensemos en cómo es la ejecución del algoritmo MERGESORT sobre el *array* de seis elementos $A = \{1, 3, 5, 2, 4, 6\}$ de la sección 3.2.2 (ver también la figura 3.1). Las mitades izquierda y derecha de este *array* están ordenadas, por lo que no habrá inversiones izquierdas ni derechas y las dos llamadas recursivas devolverán 0. En la primera iteración de la subrutina MERGE, se copiará a B el primer elemento de C (el 1). Esto no nos aporta ninguna indicación sobre inversiones separadas y, de hecho, no hay ninguna inversión separada que implique a este elemento. Sin embargo, en la segunda iteración, se copia el 2 al *array* de salida, incluso aunque C todavía contiene los elementos 3 y 5. Esto revela dos de las inversiones separadas de A , las que implican al 2. En la tercera iteración, se copia el 3 desde C y no hay más inversiones separadas que impliquen a este elemento. Cuando se copia el 4 desde D , el *array* C todavía contiene un 5, lo que pone de relevancia a la tercera y última inversión separada de A (que implica al 5 y al 4).

El siguiente lema afirma que se puede generalizar el patrón del ejemplo anterior: el número de inversiones separadas que implican a un elemento y del segundo *subarray* D es, precisamente, el número de elementos restantes en C en la iteración de la subrutina MERGE en la que se copia al *array* de salida.

Lema 3.1 (Identificación de inversiones separadas en MERGE) A es un *array* y C y D las versiones ordenadas de sus primera y segunda mitades. Un

elemento x de la primera mitad de A y un elemento y de la segunda mitad de A forman una inversión separada si, y solo si, en la subrutina `MERGE` con entradas C y D , y se copia al array de salida antes que x .

Demostración: Como el array de salida se rellena de izquierda a derecha de forma ordenada, primero se copia el elemento menor entre x e y . Como x está en la primera mitad de A e y en la segunda, x e y forman una inversión separada si, y solo si, $x > y$, lo que es cierto si, y solo si, y se copia al array de salida antes que x . \mathcal{QED}

3.2.10 MERGE-AND-COUNTSPLITINV

Con la información aportada por el lema 3.1, podemos extender la implementación de `MERGE` y convertirla en una implementación de `MERGE-AND-COUNTSPLITINV`. Mantenemos un contador para las inversiones separadas y , cada vez que se copie un elemento desde el segundo *subarray* D al array de salida B , aumentamos el contador en el número de elementos restantes en el primer *subarray* C .³

MERGE-AND-COUNTSPLITINV

Entrada: arrays ordenados C y D (ambos de longitud $n/2$).

Salida: array ordenado B (longitud n) y el número de inversiones separadas.

Asunción para simplificar: n es par.

```

i := 1, j := 1, splitInv := 0
para k := 1 hasta  $n$  hacer
    si  $C[i] < D[j]$  entonces
         $B[k] := C[i]$ , i := i + 1
    en otro caso                                //  $D[j] < C[i]$ 
         $B[k] := D[j]$ , j := j + 1
        splitInv := splitInv +  $\overbrace{(\frac{n}{2} - i + 1)}^{\# \text{ restantes en } C}$ 
devolver ( $B$ , splitInv)

```

³Al igual que en el caso de `MERGE`, la implementación completa debería mantener también un registro del momento en el que los recorridos de C o D superan el final del array.

3.2.11 Corrección

La corrección de `MERGE-AND-COUNTSPLITINV` proviene del lema 3.1. Cada inversión separada implica a, exactamente, un elemento y del segundo *subarray* y esta inversión se cuenta, exactamente, una vez, cuando se copia y al *array* de salida. La corrección del algoritmo `SORT-AND-COUNTINV` completo (sección 3.2.7) se verifica así: la primera llamada recursiva calcula correctamente el número de inversiones izquierdas, la segunda llamada recursiva hace lo propio con las inversiones derechas y `MERGE-AND-COUNTSPLITINV` se encarga del resto de inversiones (separadas).

3.2.12 Tiempo de ejecución

También podemos analizar el tiempo de ejecución del algoritmo `SORT-AND-COUNTINV` apoyándonos en el análisis que ya hicimos para el algoritmo `MERGESORT`. Consideremos, en primer lugar, el tiempo de ejecución de una única invocación de la subrutina `MERGE-AND-COUNTSPLITINV`, dados dos *subarrays* de longitud $\ell/2$ cada uno. Al igual que con la subrutina `MERGE`, por cada iteración del bucle se realiza un número constante de operaciones, más un número constante de operaciones adicionales, para llegar a un tiempo de ejecución de $O(\ell)$.

Si volvemos a consultar nuestro análisis del tiempo de ejecución del algoritmo `MERGESORT`, en la sección 1.5, podemos ver que había tres propiedades importantes del algoritmo que nos llevaron al límite del tiempo de ejecución de $O(n \log n)$. En primer lugar, cada invocación del algoritmo realiza dos llamadas recursivas. En segundo lugar, la longitud de la entrada se divide por la mitad en cada nivel de recursión. En tercer lugar, la cantidad de trabajo realizado durante una llamada recursiva, sin tener en cuenta el trabajo de las siguientes llamadas, es lineal en relación al tamaño de la entrada de esa llamada. Como el algoritmo `SORT-AND-COUNTINV` comparte estas tres propiedades, podemos conservar el análisis de la sección 1.5, llegando nuevamente a un límite del tiempo de ejecución de $O(n \log n)$.

Teorema 3.2 (Conteo de inversiones) *Por cada array de entrada A de longitud $n \geq 1$, el algoritmo `SORT-AND-COUNTINV` calcula correctamente el número de inversiones de A y se ejecuta en tiempo $O(n \log n)$.*

3.2.13 Soluciones a los cuestionarios 3.1–3.2

Solución al cuestionario 3.1

Respuesta correcta: (a). La respuesta correcta a esta pregunta es 15. El número máximo posible de inversiones es, como mucho, el número de formas distintas de elegir $i, j \in \{1, 2, \dots, 6\}$ con $i < j$. La segunda cantidad se expresa como $\binom{6}{2}$ o “6 sobre 2”. En general, $\binom{n}{2} = \frac{n(n-1)}{2}$, por lo que $\binom{6}{2} = 15$.⁴ En un *array* de seis elementos [6, 5, ..., 1], ordenado de forma inversa, todo par de elementos está desordenado, por lo que tal *array* llega a las 15 inversiones.

Solución al cuestionario 3.2

Respuesta correcta: (b). En un *array* sin inversiones separadas, toda la primera mitad es menor que toda la segunda. Si *había* un elemento $A[i]$ en la primera mitad (con $i \in \{1, 2, \dots, \frac{n}{2}\}$) mayor que un elemento $A[j]$ en la segunda (con $j \in \{\frac{n}{2} + 1, \frac{n}{2} + 2, \dots, n\}$), (i, j) constituiría una inversión separada.

3.3 Algoritmo de multiplicación de matrices de Strassen

Esta sección aplica el paradigma de diseño de algoritmos divide y vencerás al problema de la multiplicación de matrices, culminando con el impresionante algoritmo de multiplicación de matrices de tiempo subcúbico de Strassen. Este algoritmo es un ejemplo canónico de la magia y la potencia que se logran mediante el diseño inteligente de algoritmos, de cómo el ingenio algorítmico puede presentar mejoras sobre las soluciones directas, incluso en el caso de problemas fundamentales.

3.3.1 Multiplicación de matrices

Supongamos que X e Y son matrices de enteros de tamaño $n \times n$ (n^2 entradas en cada una). En el producto $Z = X \cdot Y$, la entrada z_{ij} de la i -ésima fila y j -ésima columna de Z se define como el producto escalar de la i -ésima

⁴Existen $n(n - 1)$ maneras de elegir (i, j) , tales que $i \neq j$ (n elecciones para i y, después, $n - 1$ para j). Por simetría, $i < j$ es, exactamente, la mitad de estos.

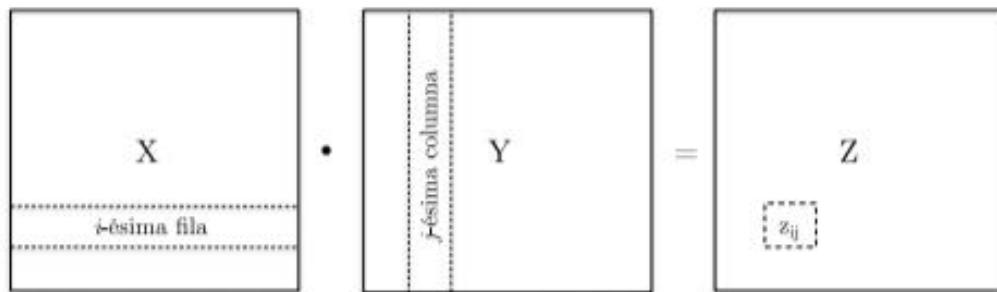


Figura 3.2: La entrada (i,j) del producto de matrices $\mathbf{X} \cdot \mathbf{Y}$ es el producto escalar de la fila i -ésima de \mathbf{X} y de la columna j -ésima de \mathbf{Y} .

columna de \mathbf{X} y la j -ésima columna de \mathbf{Y} (figura 3.2)⁵. Es decir,

$$z_{ij} = \sum_{k=1}^n x_{ik}y_{kj}. \quad (3.1)$$

3.3.2 Ejemplo ($n = 2$)

Vamos a desmenuzar el caso de $n = 2$. Podemos describir dos matrices de 2×2 utilizando ocho parámetros:

$$\underbrace{\begin{bmatrix} a & b \\ c & d \end{bmatrix}}_{\mathbf{X}} \quad \text{y} \quad \underbrace{\begin{bmatrix} e & f \\ g & h \end{bmatrix}}_{\mathbf{Y}}.$$

En el producto de matrices $\mathbf{X} \cdot \mathbf{Y}$, la entrada superior izquierda es el producto escalar de la primera fila de \mathbf{X} y la primera columna de \mathbf{Y} , o $ae + bg$. En general, para los \mathbf{X} e \mathbf{Y} anteriores,

$$\mathbf{X} \cdot \mathbf{Y} = \begin{bmatrix} ae + bg & af + bh \\ ce + dg & cf + dh \end{bmatrix}. \quad (3.2)$$

3.3.3 El algoritmo directo

Pensemos en algoritmos para el cálculo del producto de dos matrices.

⁵Para calcular el *producto escalar* de dos vectores de longitud n , $\mathbf{a} = (a_1, \dots, a_n)$ y $\mathbf{b} = (b_1, \dots, b_n)$, sumamos los resultados de la multiplicación por componentes. O, simbólicamente, $\mathbf{a} \cdot \mathbf{b} = \sum_{k=1}^n a_k b_k$.

Problema: multiplicación de matrices

Entrada: dos matrices enteras de $n \times n$, X e Y .⁶

Salida: el producto de matrices $X \cdot Y$.

El tamaño de la entrada es proporcional a n^2 , el número de entradas de X e Y . Como probablemente tendremos que leer la entrada y escribir la salida, nuestra mejor expectativa es un algoritmo con tiempo de ejecución $O(n^2)$, lineal en relación al tamaño de la entrada y cuadrático en relación a la dimensión. ¿Cuándo podremos acercarnos a esta situación del mejor caso?

La traducción de la definición matemática a código nos proporciona un algoritmo directo de multiplicación de matrices.

Multiplicación directa de matrices

Entrada: matrices enteras X e Y de $n \times n$.

Salida: $Z = X \cdot Y$.

```
// bucle por las filas de X
para i := 1 hasta n hacer
    // bucle por las columnas de Y
    para j := 1 hasta n hacer
        // calcular producto escalar
        Z[i][j] := 0
        para k := 1 hasta n hacer
            Z[i][j] := Z[i][j] + X[i][k] · Y[k][j]
devolver Z
```

¿Cuál es el tiempo de ejecución de este algoritmo?

⁶Los algoritmos que veremos se pueden adaptar para multiplicar matrices no cuadradas pero, por simplicidad, nosotros operaremos con las que sí lo son.

Cuestionario 3.3

¿Cuál es el tiempo de ejecución asintótico del algoritmo directo para la multiplicación de matrices, en función de la dimensión n de la matriz? Puedes asumir que la suma o la multiplicación de dos entradas de una matriz es una operación de tiempo constante.

- a) $\Theta(n \log n)$
- b) $\Theta(n^2)$
- c) $\Theta(n^3)$
- d) $\Theta(n^4)$

Solución y aclaraciones en la sección [3.3.7](#)

3.3.4 Una solución de divide y vencerás

La pregunta es, como siempre, *¿podemos hacerlo mejor?* La reacción más normal ante la cuestión será, seguramente, pensar en que la multiplicación de matrices, básicamente por definición, necesita de tiempo cúbico (es decir, $\Omega(n^3)$). Pero quizás nos envalentonemos gracias al éxito del algoritmo KARATSUBA para la multiplicación de enteros (sección [1.3](#)), un audaz algoritmo de divide y vencerás que mejora al algoritmo directo que aprendimos en la escuela (en realidad esto no ha quedado todavía demostrado, pero volveremos sobre ello en la sección [4.3](#)). ¿Seremos capaces de encontrar una técnica similar para la multiplicación de matrices?

Para poder aplicar el paradigma divide y vencerás (sección [3.1](#)), tenemos que establecer cómo dividir la entrada en subproblemas más pequeños y cómo combinar las soluciones a estos en una solución para el problema original. El método más sencillo para dividir una matriz cuadrada en submatrices cuadradas más pequeñas consiste en partirla por la mitad, tanto vertical como horizontalmente. En otras palabras, escribir

$$\mathbf{X} = \begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{C} & \mathbf{D} \end{bmatrix} \quad \text{y} \quad \mathbf{Y} = \begin{bmatrix} \mathbf{E} & \mathbf{F} \\ \mathbf{G} & \mathbf{H} \end{bmatrix}, \quad (3.3)$$

donde $\mathbf{A}, \mathbf{B}, \dots, \mathbf{H}$ son todas matrices $\frac{n}{2} \times \frac{n}{2}$.⁷

⁷Como en otras ocasiones, asumimos que n es par por comodidad. Y, como en otras ocasiones, tampoco tiene ninguna importancia.

Un detalle muy interesante de la multiplicación de matrices es que los bloques del mismo tamaño se comportan como si fuesen entradas individuales. Es decir, para X e Y , como acabamos de ver, tenemos

$$X \cdot Y = \begin{bmatrix} A \cdot E + B \cdot G & A \cdot F + B \cdot H \\ C \cdot E + D \cdot G & C \cdot F + D \cdot H \end{bmatrix}, \quad (3.4)$$

completamente análogo a la ecuación (3.2) para el caso de $n = 2$ (esto viene de la definición de la multiplicación de matrices, como deberías verificar). En la ecuación (3.4), la suma de dos matrices significa sumarlas a nivel de entradas: la entrada (i,j) de $K + L$ es la suma de las entradas (i,j) de K y L . La descomposición y el cálculo en (3.4) se traduce de forma natural a un algoritmo recursivo para la multiplicación de matrices, que llamaremos RECMATMULT.

RECMATMULT

Entrada: matrices de enteros $n \times n$ X e Y .

Salida: $Z = X \cdot Y$.

Asunción: n es una potencia de 2.

```

// caso base
si n = 1 entonces
    devolver la matriz 1 × 1 con entrada X[1][1] · Y[1][1]

// caso recursivo
A, B, C, D := submatrices de X como en (3.3)
E, F, G, H := submatrices de Y como en (3.3)

// ocho llamadas recursivas
calcular recursivamente los ocho productos de las matrices del lado
derecho de la ecuación (3.4)

// número constante de sumas de matrices
completar el cálculo del lado derecho de (3.4) y devolver el
resultado

```

El tiempo de ejecución del algoritmo RECMATMULT no resulta obvio a primera vista. Lo que está claro es que hay ocho llamadas recursivas, cada una de ellas sobre una entrada de la mitad de la dimensión. Aparte de estas llamadas recursivas, el único trabajo necesario es la suma de matrices de (3.4). Como una matriz $n \times n$ tiene n^2 entradas y el número de ope-

raciones necesario para sumar dos matrices es proporcional al número de entradas, una llamada recursiva a un par de matrices $\ell \times \ell$ ejecuta $\Theta(\ell^2)$ operaciones, sin tener en cuenta el trabajo realizado por sus propias llamadas recursivas.

Lamentablemente, este algoritmo recursivo resulta tener un tiempo de ejecución de $\Theta(n^3)$, idéntico al del algoritmo directo⁸. ¿Tanto trabajo y no nos sirve de nada? Recordemos que, en el problema de la multiplicación de enteros, la clave para vencer al algoritmo de la escuela estaba en el truco de Gauss, que reducía el número de llamadas recursivas de cuatro a tres (sección 1.3.3). ¿Existe un truco análogo al de Gauss para el caso de la multiplicación de matrices, que nos permita reducir el número de llamadas recursivas de ocho a siete?

3.3.5 Cómo ahorrar una llamada recursiva

El plan de alto nivel del algoritmo STRASSEN consiste en ahorrar una llamada recursiva relacionada con el algoritmo RECMATMULT, a cambio de un número constante adicional de sumas y restas de matrices.

STRASSEN (descripción de muy alto nivel)

Entrada: matrices de enteros $n \times n$ X e Y.

Salida: $Z = X \cdot Y$.

Asunción: n es una potencia de 2.

```
// caso base
si n = 1 entonces
    devolver la matriz 1 × 1 con entrada X[1][1] · Y[1][1]
// caso recursivo
A, B, C, D := submatrices de X como en (3.3)
E, F, G, H := submatrices de Y como en (3.3)
// siete llamadas recursivas
calcular recursivamente siete productos (bien elegidos) que
impliquen a A, B, ..., H
// número constante de sumas/restas de matrices
devolver las sumas y restas oportunas (bien elegidas) de las
matrices calculadas en el paso anterior
```

⁸Este hecho surge del “método maestro”, que explicaremos en el siguiente capítulo.

Ahorrar una de las ocho llamadas recursivas supone una gran victoria. No solo reduce el tiempo de ejecución del algoritmo en un 12,5%. La llamada recursiva se elimina una y otra vez, por lo que el ahorro se acumula y (latencia, *spoiler!*) esto resulta en un tiempo de ejecución superior asintóticamente. En la sección 4.3 veremos el límite exacto del tiempo de ejecución pero, por ahora, lo importante es saber que ese ahorro de una llamada recursiva nos llevará a un algoritmo con tiempo de ejecución subcúbico.

Con esto ponemos fin a todos los comentarios de alto nivel que puedan tener interés en relación al algoritmo de multiplicación de matrices de Strassen. ¿No te has creído que sea posible mejorar el algoritmo más evidente? ¿O, quizás, tienes curiosidad por saber cómo se eligen los productos, sumas y restas apropiados? Si la respuesta es afirmativa, la siguiente sección te irá como un guante.

3.3.6 Los detalles

Digamos que X e Y son las dos matrices $n \times n$ de entrada y definimos A, B, \dots, H como en (3.3). Estas son las siete multiplicaciones recursivas de matrices realizadas por el algoritmo de Strassen:

$$\begin{aligned} P_1 &= A \cdot (F - H) \\ P_2 &= (A + B) \cdot H \\ P_3 &= (C + D) \cdot E \\ P_4 &= D \cdot (G - E) \\ P_5 &= (A + D) \cdot (E + H) \\ P_6 &= (B - D) \cdot (G + H) \\ P_7 &= (A - C) \cdot (E + F). \end{aligned}$$

Después de consumir un tiempo $\Theta(n^2)$ realizando las sumas y restas de matrices necesarias, es posible calcular P_1, \dots, P_7 utilizando siete llamadas recursivas sobre pares de $\frac{n}{2} \times \frac{n}{2}$ matrices. Pero, ¿esta información es suficiente como para reconstruir el producto de matrices de X e Y en tiempo $\Theta(n^2)$? Esta asombrosa ecuación resulta en una respuesta afirmativa:

$$\begin{aligned} X \cdot Y &= \left[\begin{array}{c|c} A \cdot E + B \cdot G & A \cdot F + B \cdot H \\ \hline C \cdot E + D \cdot G & C \cdot F + D \cdot H \end{array} \right] \\ &= \left[\begin{array}{c|c} P_5 + P_4 - P_2 + P_6 & P_1 + P_2 \\ \hline P_3 + P_4 & P_1 + P_5 - P_3 - P_7 \end{array} \right]. \end{aligned}$$

La primera ecuación está copiada de (3.4). Para la segunda, debemos comprobar si se verifica la igualdad en cada uno de los cuatro cuadrantes. Para calmar tus dudas, observa las cancelaciones asombrosas del cuadrante superior izquierdo:

$$\begin{aligned} P_5 + P_4 - P_2 + P_6 &= (A + D) \cdot (E + H) + D \cdot (G - E) \\ &\quad - (A + B) \cdot H + (B - D) \cdot (G + H) \\ &= A \cdot E + A \cdot H + D \cdot E + D \cdot H + D \cdot G \\ &\quad - D \cdot E - A \cdot H - B \cdot H + B \cdot G \\ &\quad + B \cdot H - D \cdot G - D \cdot H \\ &= A \cdot E + B \cdot G. \end{aligned}$$

El cálculo del cuadrante inferior derecho es similar y la igualdad es muy evidente en los otros dos cuadrantes. Por lo tanto, el algoritmo STRASSEN puede, ciertamente, multiplicar matrices empleando únicamente siete llamadas recursivas y un trabajo adicional $\Theta(n^2)$.⁹

3.3.7 Solución al cuestionario 3.3

Respuesta correcta: (c). La respuesta correcta es $\Theta(n^3)$. Tenemos tres bucles anidados. Esto resulta en n^3 iteraciones del bucle interior (una por cada elección de $i, j, k \in \{1, 2, \dots, n\}$) y el algoritmo realiza un número constante de operaciones en cada una de esas iteraciones (una multiplicación y una suma). Alternativamente, por cada una de las n^2 entradas de Z, el algoritmo consume un tiempo $\Theta(n)$ durante la evaluación de (3.1).

*3.4 Un algoritmo en $O(n \log n)$ para el par más cercano

Nuestro último ejemplo de divide y vencerás es un algoritmo extraordinario para el problema del par más cercano, en el que recibes n puntos del plano

⁹Obviamente, comprobar que el algoritmo funciona es mucho más sencillo que diseñarlo. ¿Y cómo llegó Volker Strassen a esta conclusión en 1969? Esto es lo que me explicó (en una comunicación personal en 2017):

“Tal y como lo recuerdo, había observado que un algoritmo no conmutativo más rápido resultaba en un exponente mejor para algunos casos pequeños. Traté de demostrar que el algoritmo directo resulta óptimo para matrices 2×2 . Para simplificar las operaciones trabajé con módulo 2 y, entonces, descubrí el algoritmo más rápido mediante combinatoria.”

y quieras determinar qué par de puntos son más cercanos entre sí. Será nuestro primer contacto con la geometría computacional, un área de las ciencias de la computación que estudia algoritmos que manipulan objetos geométricos y razonan sobre ellos. La geometría computacional tiene aplicaciones en la robótica, la visión computarizada y los gráficos generados por ordenador¹⁰.

3.4.1 El problema

El problema del par más cercano se refiere a los puntos $(x, y) \in \mathbb{R}^2$ del plano. Para medir la distancia entre dos puntos $p_1 = (x_1, y_1)$ y $p_2 = (x_2, y_2)$, utilizaremos la distancia euclídea (en línea recta) habitual:

$$d(p_1, p_2) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}. \quad (3.5)$$

Problema: par más cercano

Entrada: $n \geq 2$ puntos $p_1 = (x_1, y_1), \dots, p_n = (x_n, y_n)$ del plano.

Salida: el par p_i, p_j de puntos distintos con menor distancia euclídea $d(p_i, p_j)$.

Asumiremos, por comodidad, que no habrá dos puntos que tengan la misma coordenada x ni la misma coordenada y . Tendrás que pensar en cómo modificar el algoritmo de esta sección para que se ocupe de los empates¹¹.

El problema del par más cercano se puede resolver en tiempo cuadrático mediante búsqueda exhaustiva, es decir, calculando la distancia de cada uno de los $\Theta(n^2)$ pares de puntos, uno a uno, y devolviendo aquel que resulte más cercano. Por ejemplo, si la entrada comprende los cuatro puntos $(1, 8)$, $(2, 5)$, $(4, 7)$ y $(6, 3)$, las distancias entre ellos son

¹⁰Las secciones destaca como esta, indicadas con un asterisco, son las más complejas. Puedes obviarlas en una primera lectura.

¹¹En una implementación para el mundo real, un algoritmo para el par más cercano no se molestará en calcular la raíz cuadrada incluida en (3.5). El par de puntos con la menor distancia euclídea es el mismo que aquel que tenga la menor distancia euclídea al cuadrado, y esta última es más sencilla de calcular.

Punto	(1, 8)	(2, 5)	(4, 7)	(6, 3)
(1, 8)	0	$\sqrt{10}$	$\sqrt{10}$	$\sqrt{50}$
(2, 5)	$\sqrt{10}$	0	$\sqrt{8}$	$\sqrt{20}$
(4, 7)	$\sqrt{10}$	$\sqrt{8}$	0	$\sqrt{20}$
(6, 3)	$\sqrt{50}$	$\sqrt{20}$	$\sqrt{20}$	0

y el par más cercano de puntos distintos resultará ser (2, 5) y (4, 7).

Para el problema del conteo de inversiones (sección 3.2), fuimos capaces de mejorar la búsqueda exhaustiva en tiempo cuadrático mediante un algoritmo de divide y vencerás. En este caso, ¿podemos también hacerlo mejor?

3.4.2 ¿Nos ayuda en algo la ordenación?

Si estamos satisfechos con un tiempo de ejecución de $O(n \log n)$ (y sí que lo estamos, porque resultaría mucho más rápido que la búsqueda exhaustiva), una de nuestras primeras intuiciones debería ser utilizar nuestra operación básica de coste cero favorita, la ordenación. Por ejemplo, la versión unidimensional del problema del par más cercano, en el que los puntos de la entrada se sitúan en una línea en vez de en el plano, se puede resolver utilizando una sola llamada a la subrutina de ordenación seguida de una única pasada sobre los puntos ordenados (ver también el problema 1.5).

Pero, ¿cómo podemos ordenar los puntos en el plano, cuando tienen dos coordenadas? Lo más evidente parece ser utilizar las coordenadas x e y . Por ejemplo, los cuatro puntos anteriores aparecerían ordenados como $[(1, 8), (2, 5), (4, 7), (6, 3)]$, si utilizásemos la coordenada x , y como $[(6, 3), (2, 5), (4, 7), (1, 8)]$, si utilizásemos la y .

Sin embargo, centrarnos en una sola coordenada cada vez puede llevar a confusión. En nuestro ejemplo, donde los puntos (1, 8) y (2, 5) son los más cercanos según la coordenada x y (4, 7) y (1, 8) lo son según la coordenada y , la realidad es que los puntos verdaderamente más cercanos son (2, 5) y (4, 7). ¿Cómo podríamos extraer la información contenida en las dos ordenaciones unidimensionales para identificar el par más cercano?

3.4.3 Una solución de divide y vencerás

Podemos superar a la búsqueda exhaustiva mediante una técnica de divide y vencerás. En primer lugar, durante un procesamiento previo, se ordenan los puntos de entrada (utilizando MERGESORT), primero por la coordenada x y, después, por la y . En nuestro ejemplo, esto creará dos

copias del conjunto de puntos, $P_x = [(1, 8), (2, 5), (4, 7), (6, 3)]$ y $P_y = [(6, 3), (2, 5), (4, 7), (1, 8)]$. El plan de alto nivel será entonces análogo al que nos resultó tan eficaz para contar inversiones (sección 3.2):

- (i) dividir los puntos de la entrada en “mitad izquierda” y “mitad derecha” y, entonces,
- (ii) buscar por separado pares de puntos en la mitad izquierda (con una llamada recursiva), pares de puntos en la mitad derecha (con una llamada recursiva) y pares de puntos separados en las mitades izquierda y derecha (con una subrutina hecha a medida).

Aquí, “mitad izquierda” y “mitad derecha” hacen referencia a las primera y segunda mitades de la lista de puntos P_x , ordenados por la coordenada x . Llamaremos a los tres tipos de pares de puntos *pares izquierdos*, *pares derechos* y *pares separados*. Así, con los puntos $[(1, 8), (2, 5), (4, 7), (6, 3)]$ ordenados por la coordenada x , $(1, 8)$ y $(2, 5)$ forman un par izquierdo, $(4, 7)$ y $(6, 3)$ un par derecho y $(2, 5)$ y $(4, 7)$ un par separado.

El siguiente pseudocódigo resume el plan de alto nivel. Dejamos sin implementar, de momento, la subrutina CLOSESTSPLITPAIR.

CLOSESTPAIR (versión preliminar)

Entrada: dos copias P_x y P_y de un conjunto P de $n \geq 2$ puntos del plano, la primera ordenada por la coordenada x y la segunda por la y .

Salida: el par p_i, p_j de puntos distintos con la menor distancia euclídea entre ellos.

```
// caso base de <= 3 puntos omitido
1  $L_x :=$  mitad izquierda de  $P$ , ordenada por  $x$ 
2  $L_y :=$  mitad izquierda de  $P$ , ordenada por  $y$ 
3  $R_x :=$  mitad derecha de  $P$ , ordenada por  $x$ 
4  $R_y :=$  mitad derecha de  $P$ , ordenada por  $y$ 

5  $(l_1, l_2) := \text{CLOSESTPAIR}(L_x, L_y)$  // mejor par izquierdo
6  $(r_1, r_2) := \text{CLOSESTPAIR}(R_x, R_y)$  // mejor par derecho
7  $(s_1, s_2) := \text{CLOSESTSPLITPAIR}(P_x, P_y)$  // mejor par separado

8 devolver el mejor de  $(l_1, l_2)$ ,  $(r_1, r_2)$ ,  $(s_1, s_2)$ 
```

En el caso base omitido, donde la entrada consta de dos o tres puntos, el algoritmo calcula el par más cercano mediante búsqueda exhaustiva. Con un número constante de puntos, la búsqueda exhaustiva necesita solamente un número constante de operaciones ($O(1)$).

Cada llamada recursiva espera como entrada dos copias del mismo conjunto de puntos, uno de ellos ordenado por la coordenada x y el otro por la y , motivo por el cual calculamos las cuatro listas L_x , L_y , R_x y R_y . Estas listas se pueden generar en tiempo lineal ($O(n)$) a partir de las dos copias ordenadas del conjunto de puntos de la entrada (P_x y P_y). Las listas L_x y R_x son, exactamente, las primera y segunda mitades de P_x . El último punto de L_x (o el primero de R_x) nos indican qué coordenadas x pertenecen a la mitad izquierda de P y cuáles a la mitad derecha. Para calcular L_y y R_y en tiempo lineal, el algoritmo puede realizar una única pasada sobre P_y , colocando cada punto al final de L_y o R_y , según cuál sea su coordenada x . Esta pasada partitiona correctamente el conjunto de puntos P en sus mitades izquierda y derecha, al tiempo que mantiene la ordenación por la coordenada y . En nuestro ejemplo de cuatro puntos, $L_x = [(1, 8), (2, 5)]$, $R_x = [(4, 7), (6, 3)]$, $L_y = [(2, 5), (1, 8)]$, y $R_y = [(6, 3), (4, 7)]$.

Dando por hecho que implementaremos la subrutina `CLOSESTSPLITPAIR` correctamente, tenemos la garantía de que el algoritmo calculará el par de puntos más cercano (las tres llamadas a las subrutinas en las líneas 5–7 cubren todas las posibilidades de dónde podría estar el par más cercano y la línea 8 calcula explícitamente y devuelve el mejor de los tres candidatos restantes).

Cuestionario 3.4

Si implementamos la subrutina `CLOSESTSPLITPAIR` en tiempo $O(n)$, ¿cuál será el tiempo de ejecución del algoritmo `CLOSESTPAIR` (elige el límite correcto más pequeño)?

- a) $O(n)$
- b) $O(n \log n)$
- c) $O(n(\log n)^2)$
- d) $O(n^2)$

Solución y aclaraciones en la sección 3.4.9

3.4.4 Un cambio sutil

La solución al cuestionario 3.4 deja claro nuestro objetivo: buscamos una implementación en $O(n)$ de la subrutina CLOSESTSPLITPAIR, que nos lleve a un límite del tiempo de ejecución de $O(n \log n)$.

Vamos a diseñar una subrutina ligeramente peor que resulte adecuada a nuestro objetivo. Esta es la observación clave: *dependemos de la subrutina CLOSESTSPLITPAIR para identificar correctamente el par separado más cercano únicamente cuando también es el par más cercano global*. Si el par más cercano resulta ser un par izquierdo o derecho, CLOSESTSPLITPAIR podría devolver información inútil. La línea 8 del pseudocódigo de la sección 3.4.3 ignorará, en cualquier caso, su par de puntos sugerido, en favor del par de puntos más cercano correcto, calculado por una de las llamadas recursivas. Nuestro algoritmo aprovechará totalmente el uso de este requisito relajado de corrección.

Para implementar esta idea, pasaremos a la subrutina CLOSESTSPLITPAIR la distancia δ del par más cercano izquierdo o derecho. Así, la subrutina sabrá que solo debe preocuparse de pares separados con una distancia entre puntos menor que δ . En otras palabras, sustituiremos las líneas 7–8 del pseudocódigo de la sección 3.4.3 con lo siguiente:

CLOSESTPAIR (añadido)

- 7 $\delta := \min\{d(l_1, l_2), d(r_1, r_2)\}$
- 8 $(s_1, s_2) := \text{CLOSESTSPLITPAIR}(P_x, P_y, \delta)$
- 9 devolver el mejor de (l_1, l_2) , (r_1, r_2) , (s_1, s_2)

3.4.5 CLOSESTSPLITPAIR

Ahora añadiremos una implementación de la subrutina CLOSESTSPLITPAIR que se ejecuta en tiempo lineal y calcula correctamente el par más cercano, cuando este es un par separado. Quizá te cueste creer que el siguiente pseudocódigo cumple con todos los requisitos, pero lo cierto es que así es. La idea de alto nivel consiste en realizar una búsqueda exhaustiva sobre un conjunto inteligentemente acotado de pares de puntos.

CLOSESTSPLITPAIR

Entrada: dos copias P_x y P_y de un conjunto P de $n \geq 2$ puntos del plano, la primera ordenada por la coordenada x y la segunda por la y , y un parámetro δ .

Salida: el par más cercano, siempre que sea un par separado.

```
// calcular la mediana de las coordenadas x
1  $\bar{x}$  := mayor coordenada  $x$  de la mitad izquierda
    // identificar puntos cerca de los límites
    // izquierdo/derecho
2  $S_y$  := lista de puntos  $q_1, q_2, \dots, q_\ell$  con coordenada  $x$ 
    entre  $\bar{x} - \delta$  y  $\bar{x} + \delta$ , ordenados por la coordenada  $y$ 
3  $closestDistance := \delta$                                 // distancia a mejorar
4  $closestSplitPair := \emptyset$ 
5 para  $i := 1$  hasta  $\ell - 1$  hacer // todo punto cerca del límite
6     para  $j := i + 1$  hasta  $\min\{i + 7, \ell\}$  hacer // comprobar los 7
        siguientes
7         si  $d(q_i, q_j) < closestDistance$  entonces
8              $closestDistance := d(q_i, q_j)$ 
9              $closestSplitPair := (q_i, q_j)$ 
10 devolver  $closestSplitPair$ 
```

La subrutina identifica, en la línea 1, el punto más a la derecha en la mitad izquierda del conjunto, que define \bar{x} , la “mediana de las coordenadas x ”. Un par de puntos es un par separado si, y solo si, uno de los puntos tiene una coordenada x de, como mucho, \bar{x} y el otro mayor que \bar{x} . Como \bar{x} es la coordenada x de la entrada $(n/2)$ -ésima de la lista P_x , podemos recuperarla en tiempo constante ($O(1)$). En la línea 2, la subrutina realiza un filtrado, descartando todos los puntos salvo aquellos que queden en la franja vertical de ancho 2δ centrada en \bar{x} (figura 3.3). El conjunto S_y se puede calcular en tiempo lineal explorando P_y y eliminando cualquier punto con una coordenada x que se encuentre fuera del rango de interés¹². Las líneas 5–9, realizan la búsqueda exhaustiva oportunamente restringida, examinando únicamente los pares de puntos en los que: (i)

¹²Este paso es el motivo por el que hemos ordenado el conjunto de puntos por la coordenada y , de una vez y para siempre, durante el procesamiento previo. Como estamos buscando una subrutina de tiempo lineal, ahora no tenemos tiempo para nuevas ordenaciones.

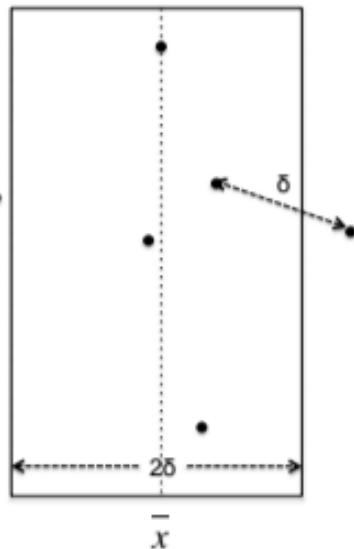


Figura 3.3: La subrutina CLOSESTSPLITPAIR. La lista S_y es el conjunto de puntos delimitado por la franja vertical, ordenado por la coordenada y . El parámetro δ indica la menor distancia entre un par izquierdo o un par derecho de puntos. Los pares separados tienen un punto a cada uno de los lados de la línea punteada.

ambos puntos quedan dentro de la franja vertical de ancho 2δ y (ii) entre tales puntos las coordenadas y son casi consecutivas (con un máximo de seis punto más entre ellos). La subrutina devuelve el par de puntos más cercano¹³.

La subrutina realiza un número constante de operaciones en cada iteración del bucle, junto con el trabajo $O(n)$ que se ejecuta fuera del doble bucle. Debido al “7” misterioso del bucle interno, el número de iteraciones tendrá un máximo de $7\ell \leq 7n$, y podemos concluir que la subrutina se ejecuta en tiempo $O(n)$, tal y como queríamos. Pero, ¿cómo es posible que encuentre el par más cercano? ¿Cómo podemos justificar el hecho de examinar solamente un número lineal de los muchos pares cuadráticos de puntos? ¿Cómo sabemos que la subrutina no ha pasado por alto al verdadero par más cercano?

¹³O, en caso de que no exista un par de puntos a una distancia inferior a δ , devuelve un conjunto vacío. En este caso, en CLOSESTPAIR, la comparación final de la línea 9 se realiza solo entre los pares de puntos devueltos por las dos llamadas recursivas.

3.4.6 Corrección

El siguiente lema, que puede resultar un tanto impactante, garantiza que cuando el par más cercano es un par separado, sus puntos aparecen de forma casi consecutiva en el conjunto filtrado S_y .

Lema 3.3 (CLOSESTSPLITPAIR examina todos los pares relevantes)

Supongamos que (p, q) es un par separado con $d(p, q) < \delta$, donde δ es la menor distancia entre un par izquierdo o derecho de puntos. Entonces, en la subrutina CLOSESTSPLITPAIR:

- (a) p y q estarán incluidos en el conjunto S_y ,
- (b) un máximo de seis puntos de S_y tendrán una coordenada y entre las de p y q .

El lema 3.3 no resulta especialmente evidente y lo demostraremos en las dos próximas secciones. De momento, podemos ver que este lema implica que la subrutina CLOSESTSPLITPAIR hace correctamente su trabajo.

Corolario 3.4 (Corrección de CLOSESTSPLITPAIR) Siempre que el par más cercano sea un par separado, la subrutina CLOSESTSPLITPAIR lo devolverá.

Demostración: Asumimos que el par más cercano (p, q) es un par separado, y que $d(p, q) < \delta$, donde δ es la distancia mínima entre un par izquierdo o derecho. Por tanto, el lema 3.3 asegura que tanto p como q pertenecen al conjunto S_y en la subrutina CLOSESTSPLITPAIR y que hay un máximo de seis puntos de S_y entre ellos en la coordenada y . Como la subrutina busca exhaustivamente todos los pares de puntos que satisfacen estas dos propiedades, calculará ese par más cercano, que debe ser el verdadero par más cercano (p, q) . QED

A la espera de la demostración del lema 3.3, ahora tenemos un algoritmo correcto y espectacularmente rápido para el problema del par más cercano.

Teorema 3.5 (Cálculo del par más cercano) Por cada conjunto P de $n \geq 2$ puntos del plano, el algoritmo CLOSESTPAIR calcula correctamente el par más cercano de P y se ejecuta en tiempo $O(n \log n)$.

Demostración: Ya hemos considerado el límite del tiempo de ejecución: el algoritmo emplea tiempo $O(n \log n)$ durante el procesamiento previo y el resto tiene el mismo tiempo de ejecución asintótico que MERGESORT (con dos llamadas recursivas para cada mitad de la entrada, además del trabajo lineal adicional), que también es $O(n \log n)$.

Para ser correcto, si el par más cercano es izquierdo, es devuelto por la primera llamada recursiva (línea 5 en la sección 3.4.3). Si es un par derecho, es devuelto por la segunda llamada recursiva (línea 6). Si es un par separado, entonces el corolario 3.4 garantiza que será devuelto por la subrutina CLOSESTSPLITPAIR. En consecuencia, el par más cercano siempre se encuentra entre los tres últimos candidatos examinados por el algoritmo (en la línea 9 de la sección 3.4.4) y será devuelto como respuesta final. \mathcal{QED}

3.4.7 Demostración del lema 3.3(a)

La parte (a) del lema 3.3 es la más sencilla de demostrar. Asumimos que hay un par separado (p, q) , con $d(p, q) < \delta$, donde p y q pertenecen, respectivamente, a las mitades izquierda y derecha del conjunto de puntos, y δ es la distancia mínima entre un par izquierdo o derecho. Escribimos $p = (x_1, y_1)$ y $q = (x_2, y_2)$, y utilizamos \bar{x} para indicar la coordenada x del punto más a la derecha de la mitad izquierda del conjunto. Como p y q se encuentran, respectivamente, en las mitades izquierda y derecha, tenemos

$$x_1 \leq \bar{x} < x_2.$$

Al mismo tiempo, como p y q son cercanos, x_1 y x_2 no pueden ser muy diferentes. Formalmente, dado que la distancia euclídea entre p y q es inferior a δ , cada una de sus coordenadas diferirá por menos que δ :

$$|x_1 - x_2|, |y_1 - y_2| < \delta. \quad (3.6)$$

¿Por qué es esto así? Porque si, digamos, $|x_1 - x_2| \geq \delta$, la distancia euclídea (3.5) entre p y q sería de, al menos, $\sqrt{(x_1 - x_2)^2} \geq \sqrt{\delta^2} = \delta$, lo que resulta en una contradicción.

Como $x_1 \leq \bar{x}$ y x_2 es, como mucho, δ más grande que x_1 , tenemos que $x_2 \leq \bar{x} + \delta$ (figura 3.4)¹⁴. Igualmente, como $x_2 \geq \bar{x}$ y x_1 es, como mucho,

¹⁴Imaginemos que p y q son personas atadas por la cintura por una cuerda de longitud δ . El punto p solo puede ir hacia la derecha hasta \bar{x} , lo que limita la capacidad de movimiento de q a $\bar{x} + \delta$.

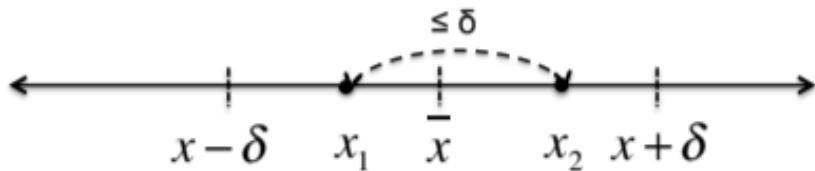


Figura 3.4: Demostración del lema 3.3(a). Tanto p como q tienen coordenadas x entre $\bar{x} - \delta$ y $\bar{x} + \delta$.

δ más pequeño que x_2 , tenemos que $x_1 \geq \bar{x} - \delta$. En particular, p y q tienen coordenadas x situadas entre $\bar{x} - \delta$ y $\bar{x} + \delta$. Todos estos puntos, incluyendo p y q , pertenecen a la lista S_y .

3.4.8 Demostración del lema 3.3(b)

Recordemos lo que hemos asumido: existe un par separado (p, q) , con $p = (x_1, y_1)$ en la mitad izquierda del conjunto de puntos y $q = (x_2, y_2)$ en la mitad derecha, tal que $d(p, q) < \delta$, donde δ es la distancia mínima entre un par izquierdo o derecho. El lema 3.3(b) afirma que p y q no solo están presentes en la lista S_y (como quedó demostrado en la parte (a)), sino que son casi consecutivos, con un máximo de seis puntos de S_y que tengan una coordenada y entre y_1 e y_2 .

Para demostrarlo, debemos dibujar en nuestra cabeza (y no en el algoritmo) ocho cajas en el plano, con un patrón 2×4 , donde cada una de ellas tenga un lado de longitud $\frac{\delta}{2}$ (figura 3.5). Hay dos columnas de cajas a cada lado de \bar{x} , la mediana de las coordenadas x . La parte inferior de las cajas está alineada con los puntos p y q más bajos, en la coordenada y mín{ y_1, y_2 }.

Gracias a la parte (a), sabemos que tanto p como q tienen sus coordenadas x entre $\bar{x} - \delta$ y $\bar{x} + \delta$. Para concretar, supongamos que q tiene la coordenada y más pequeña. El otro caso es análogo. Por tanto, q aparece en la parte inferior de una de las cajas de la fila de abajo (en la mitad derecha). Como la coordenada y de p solo puede ser δ más grande que la de q (ver (3.6)), p también aparece en una de las cajas (en la mitad izquierda). Cada punto de S_y con una coordenada y entre p y q tiene: (i) la coordenada x entre $\bar{x} - \delta$ y $\bar{x} + \delta$ (el requisito para pertenecer a S_y), y (ii) la coordenada y entre y_2 e $y_1 < y_2 + \delta$. En consecuencia, cada uno de esos puntos reside en una de las ocho cajas.

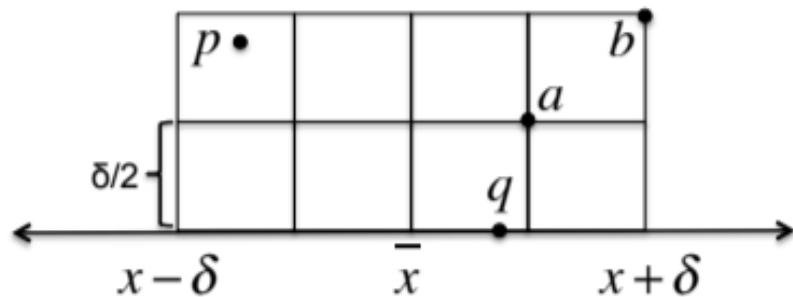


Figura 3.5: Demostración del lema 3.3(b). Los puntos p y q se encuentran en dos de estas ocho cajas y hay un máximo de un punto en cada una.

Podría preocuparnos que haya muchos puntos en las cajas que tengan una coordenada y entre y_1 e y_2 . Para mostrar que algo así no es posible, demostraremos que cada una de las cajas contiene, como mucho, un punto. Con ello, las ocho cajas sumarán un máximo de ocho puntos (incluyendo a p y q), y solo puede haber seis puntos de S_y en la coordenada y entre p y q .¹⁵

¿Por qué cada caja contiene un máximo de un punto? Esta es la parte de la explicación que utiliza la observación que realizamos en la sección 3.4.4 y el hecho de que δ sea la menor distancia entre un par izquierdo o un par derecho. Para deducir una contradicción, supongamos que alguna de las cajas contiene dos puntos, a y b (uno de los cuales podría ser p o q). Este par de puntos será o un par izquierdo (en caso de que los puntos se encuentren en las dos primeras columnas) o un par derecho (si están en las dos últimas). Lo más lejos que pueden estar a y b es en esquinas opuestas de su caja (figura 3.5), en cuyo caso, según el teorema de Pitágoras¹⁶, la distancia entre a y b será $\sqrt{2} \cdot \frac{\delta}{2} < \delta$. Pero esto contradice la asunción de que la distancia entre cada par de puntos izquierdo y derecho es de, al menos, δ . Esta contradicción implica que cada una de las ocho cajas de la figura 3.5 contiene un punto como mucho, lo que completa la demostración.

¹⁵Si un punto tiene su coordenada x exactamente en \tilde{x} , consideramos que forma parte de la caja a su izquierda. Podemos asignar arbitrariamente el resto de puntos que se encuentren en los límites de varias cajas.

¹⁶En un triángulo rectángulo, la suma de los cuadrados de las longitudes de los catetos es igual al cuadrado de la longitud de la hipotenusa.

3.4.9 Solución al cuestionario 3.4

Respuesta correcta: (b). La respuesta correcta es $O(n \log n)$. $O(n)$ no es correcta porque, entre otras razones, el algoritmo CLOSESTPAIR ya consume un tiempo $\Theta(n \log n)$ durante el procesamiento previo, al crear las listas ordenadas P_x y P_y . El límite superior de $O(n \log n)$ sigue exactamente el mismo razonamiento que en MERGESORT: el algoritmo CLOSESTPAIR realiza dos llamadas recursivas, cada una de ellas del tamaño de la mitad de la entrada, y ejecuta un trabajo $O(n)$ fuera de esas llamadas recursivas (recordemos que las líneas 1–4 y 8 se pueden implementar en tiempo $O(n)$) y para este cuestionario estamos asumiendo que CLOSESTSPLITPAIR también se ejecuta en tiempo lineal). Este patrón coincide perfectamente con el que ya analizamos para MERGESORT en la sección 1.5, por lo que sabemos que el número total de operaciones realizadas es $O(n \log n)$. Como el paso de procesamiento previo también se ejecuta en tiempo $O(n \log n)$, el límite de tiempo de ejecución final es $O(n \log n)$.

Conclusiones

- ★ Un algoritmo de divide y vencerás partitiona la entrada en subproblemas más pequeños, resuelve esos subproblemas recursivamente y combina sus soluciones para obtener la solución al problema original.
- ★ Calcular el número de inversiones en un *array* es relevante para medir la similitud entre dos listas clasificadorias. El algoritmo de búsqueda exhaustiva del problema se ejecuta en tiempo $\Theta(n^2)$ para *arrays* de longitud n .
- ★ Existe un algoritmo de divide y vencerás que se apoya en MERGESORT y calcula el número de inversiones en tiempo $O(n \log n)$.
- ★ El algoritmo de divide y vencerás de tiempo subcúbico de Strassen, para la multiplicación de matrices, es un ejemplo desconcertante de cómo el ingenio algorítmico puede mejorar a las soluciones directas. La idea clave consiste en ahorrar una llamada recursiva en comparación a un algoritmo de divide y vencerás

más sencillo, de forma análoga a la multiplicación de Karatsuba.

- En el problema del par más cercano, la entrada consta de n puntos del plano y el objetivo es calcular el par de puntos con menor distancia euclídea entre ellos. El algoritmo de búsqueda exhaustiva se ejecuta en tiempo $\Theta(n^2)$.
- Existe un algoritmo de divide y vencerás sofisticado que resuelve el problema del par más cercano en tiempo $O(n \log n)$.

Comprueba que lo has entendido

Problema 3.1 (P) Considera el siguiente pseudocódigo para calcular a^b , donde a y b son enteros positivos¹⁷:

FASTPOWER

Entrada: enteros positivos a y b .

Salida: a^b .

```
si  $b = 1$  entonces
    devolver  $a$ 
en otro caso
     $c := a \cdot a$ 
     $ans := \text{FASTPOWER}(c, \lfloor b/2 \rfloor)$ 
    si  $b$  es impar entonces
        devolver  $a \cdot ans$ 
    en otro caso
        devolver  $ans$ 
```

Para este problema, asumimos que cada multiplicación y división se puede realizar en tiempo constante. ¿Cuál es el tiempo de ejecución asintótico de este algoritmo, en relación a b ?

¹⁷La notación $\lfloor x \rfloor$ indica la función “suelo”, que redondea su argumento al entero inferior más cercano.

- a) $\Theta(\log b)$
- b) $\Theta(\sqrt{b})$
- c) $\Theta(b)$
- d) $\Theta(b \log b)$

Problema 3.2 (S) Considera utilizar el algoritmo de STRASSEN para multiplicar estas dos matrices:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \text{ y } \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}.$$

En este caso, ¿cuáles son las matrices P_1 – P_7 definidas en la sección [3.3.6](#)?

Problemas más difíciles

Problema 3.3 (P) Recibes un *array unimodal* de n elementos distintos, lo que significa que sus entradas aparecen en orden creciente hasta llegar a su elemento máximo, a partir del cual los elementos pasan a ser decrecientes. Presenta un algoritmo, de tiempo $O(\log n)$, que calcule el elemento máximo de un *array unimodal*.

Problema 3.4 (P) Recibes un *array A* ordenado (de menor a mayor) que contiene n enteros distintos que pueden ser positivos, negativos o cero. Diseña el algoritmo más rápido que te sea posible para determinar si existe un índice i tal que $A[i] = i$.

Problema 3.5 (P) En el problema de la multiplicación matriz–vector, la entrada consta de una matriz de enteros **A**, con dimensiones $n \times n$, y un vector de enteros **x** de longitud n . El objetivo es calcular su producto **A**·**x**, que es el vector de longitud n que contiene el producto escalar de **x** con cada una de las filas de **A**. Por ejemplo,

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \cdot \begin{bmatrix} 5 \\ 6 \end{bmatrix} = \begin{bmatrix} 1 \cdot 5 + 2 \cdot 6 \\ 3 \cdot 5 + 4 \cdot 6 \end{bmatrix} = \begin{bmatrix} 17 \\ 39 \end{bmatrix}.$$

La multiplicación matriz–vector necesita, normalmente, $\Omega(n^2)$ operaciones, pero existen multitud de ejemplos prácticos importantes de matrices **A** para las que se puede realizar mucho más rápido, mediante una técnica de divide y vencerás.

Estas son las tres primeras *matrices de Hadamard*:

$$\mathbf{H}_1 = [1], \quad \mathbf{H}_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}, \quad \mathbf{H}_3 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix}.$$

En general, la k -ésima matriz de Hadamard, \mathbf{H}_k , se deduce de la anterior, \mathbf{H}_{k-1} , de acuerdo a

$$\mathbf{H}_k = \left[\begin{array}{c|c} \mathbf{H}_{k-1} & \mathbf{H}_{k-1} \\ \hline \mathbf{H}_{k-1} & -\mathbf{H}_{k-1} \end{array} \right].$$

Presenta un algoritmo de tiempo $O(n \log n)$ que acepte como entrada un vector de enteros \mathbf{x} , de longitud n , siendo $n = 2^k$ una potencia de dos, y devuelva el producto matriz–vector $\mathbf{H}_k \cdot \mathbf{x}$. En este problema, puedes asumir que cada operación aritmética se puede realizar en tiempo constante¹⁸.

Problema 3.6 (P) (Difícil) Recibes una rejilla $n \times n$ de número diferentes. Un número es un *mínimo local* si es menor que todos sus vecinos (el *vecino* de un número es aquel que se encuentra inmediatamente encima, debajo, a la izquierda o a la derecha; la mayoría de los números tienen cuatro vecinos, salvo los de los lados, que tienen tres, y los de las esquinas, con dos). Utiliza el paradigma de diseño divide y vencerás para calcular un mínimo local realizando únicamente $O(n)$ comparaciones entre pares de números (nota: como la entrada consta de n^2 números, no puedes permitirte comprobarlos todos).

Problema 3.7 (P) (Difícil) En el problema *todos los vecinos más cercanos*, la entrada consta de un conjunto $P = \{p_1, p_2, \dots, p_n\}$ de $n \geq 2$ puntos del plano, y el objetivo es calcular, para cada punto $p_i \in P$, el punto $p_j \in P$ que minimice la distancia euclídea $d(p_i, p_j)$, definida en (3.5) (evidentemente, p_i y p_j son necesariamente distintos).

Presenta un algoritmo de tiempo $O(n \log n)$ que resuelva este problema. Asume que los puntos de entrada tienen distintas coordenadas x e y .

¹⁸La transformada rápida de Fourier (FFT) es un famosísimo algoritmo de tiempo $O(n \log n)$ que funciona exactamente de la misma forma, salvo para el caso de la “matriz de Fourier” \mathbf{F}_k , definida recursivamente de forma similar, haciendo las veces de la matriz de Hadamard \mathbf{H}_k . Por ejemplo, si \mathbf{x} representa 2^k muestras separadas uniformemente de una señal de audio a lo largo de un periodo de tiempo, la FFT $\mathbf{F}_k \cdot \mathbf{x}$ expresa dicha señal en la forma de la suma de sus frecuencias, lo que simplifica algunas manipulaciones de la señal, como la reducción de ruido.

Problemas de programación

Problema 3.8 Implementa, utilizando tu lenguaje de programación favorito, el algoritmo COUNTINV de la sección 3.2, para contar el número de inversiones de un *array* (puedes consultar casos de prueba y conjuntos de datos complicados en www.algorithmsilluminated.org).

El método maestro

Este capítulo nos presenta un método de “caja negra” para determinar el tiempo de ejecución de algoritmos recursivos. Configuramos algunas características clave del algoritmo y obtenemos el límite superior del tiempo de ejecución del mismo. Este “método maestro” es aplicable a la mayoría de algoritmos de divide y vencerás con los que te puedas encontrar, incluyendo los de multiplicación de enteros de Karatsuba (sección 1.3) y multiplicación de matrices de Strassen (sección 3.3)¹. Este capítulo, además, ilustra un asunto más generalista del estudio de los algoritmos: la evaluación adecuada de ideas algorítmicas novedosas requiere, en ocasiones, de análisis matemáticos poco evidentes.

Una vez hayamos tratado las recurrencias en la sección 4.1, daremos una definición formal del método maestro (sección 4.2) y veremos seis aplicaciones de ejemplo (sección 4.3). La sección 4.4 contiene la demostración del método maestro, con especial énfasis en el significado que hay detrás de sus tres famosos casos. La demostración encaja a la perfección con el análisis del algoritmo MERGESORT, que realizamos en la sección 1.5.

4.1 Multiplicación de enteros revisitada

Para motivar la discusión sobre el método maestro, vamos a revisar lo que ya sabemos sobre la multiplicación de enteros (secciones 1.2–1.3). El problema consiste en multiplicar dos números de n dígitos, donde las operaciones básicas son la suma o la multiplicación de dos números de un solo dígito. El algoritmo iterativo de la escuela necesita $\Theta(n^2)$ operaciones para multiplicar dos números de n dígitos. ¿Podemos hacerlo mejor con una técnica de divide y vencerás?

¹El método maestro también es conocido como “teorema maestro”.

4.1.1 El algoritmo RECINTMULT

El algoritmo RECINTMULT de la sección 1.3 genera subproblemas más pequeños al particionar en dos mitades los números de n dígitos x e y dados: $x = 10^{n/2} \cdot a + b$ e $y = 10^{n/2} \cdot c + d$, donde a, b, c, d son números de $n/2$ dígitos (asumiendo, por simplificar, que n es par). Por ejemplo, si $x = 1234$, entonces $a = 12$ y $b = 34$. Podemos expandir el producto $x \cdot y$ como

$$x \cdot y = 10^n \cdot (a \cdot c) + 10^{n/2} \cdot (a \cdot d + b \cdot c) + b \cdot d, \quad (4.1)$$

lo que muestra que la multiplicación de dos números de n dígitos se reduce a la multiplicación de cuatro pares de números de $n/2$ dígitos, más $O(n)$ operaciones adicionales (para añadir ceros, si fuese necesario, y la suma de la escuela).

La mejor forma de describir formalmente este patrón es mediante una *recurrencia*. Digamos que $T(n)$ indica el número máximo de operaciones utilizadas por este algoritmo recursivo para multiplicar dos números de n dígitos (esta es la cantidad para la que buscamos un límite superior). Una recurrencia expresa un límite de tiempo de ejecución de $T(n)$, en términos del número de operaciones realizadas por llamadas recursivas. La recurrencia del algoritmo RECINTMULT es

$$T(n) \leq \underbrace{4 \cdot T\left(\frac{n}{2}\right)}_{\text{trabajo de llamadas recursivas}} + \underbrace{O(n)}_{\text{trabajo fuera de llamadas recursivas}}.$$

Al igual que en un algoritmo recursivo, una recurrencia necesita uno o más casos base, que indican qué $T(n)$ es aplicable a aquellos valores de n demasiado pequeños como para provocar llamadas recursivas. Aquí, el caso base lo encontramos cuando $n = 1$, porque el algoritmo realiza una sola multiplicación, $T(1) = 1$.

4.1.2 El algoritmo KARATSUBA

El algoritmo recursivo de Karatsuba para la multiplicación de enteros, utiliza un truco aportado por Gauss para ahorrar una llamada recursiva. Este truco consiste en calcular recursivamente los productos de a y c , b y d y $a + b$ y $c + d$, y extraer el coeficiente central $ad + bc$ de (4.1) mediante el cálculo $(a + b)(c + d) - ac - bd$. Estas operaciones nos proporcionan información suficiente para poder calcular el lado derecho de (4.1) empleando $O(n)$ operaciones básicas adicionales.

Cuestionario 4.1

¿Qué recurrencia describe mejor el tiempo de ejecución del algoritmo KARATSUBA para la multiplicación de enteros?

- a) $T(n) \leq 2 \cdot T\left(\frac{n}{2}\right) + O(n^2)$
- b) $T(n) \leq 3 \cdot T\left(\frac{n}{2}\right) + O(n)$
- c) $T(n) \leq 3 \cdot T\left(\frac{n}{2}\right) + O(n^2)$
- d) $T(n) \leq 4 \cdot T\left(\frac{n}{2}\right) + O(n)$

Solución y aclaraciones a continuación

Respuesta correcta: (b). El único cambio en relación a RECINTMULT es que el número de llamadas recursivas se reduce en uno. Es cierto que la cantidad de trabajo realizada fuera de las llamadas recursivas es mayor en el algoritmo KARATSUBA, pero únicamente por un factor constante que queda eliminado en la notación *Big-O*. En consecuencia, la recurrencia apropiada para el algoritmo KARATSUBA es

$$T(n) \leq \underbrace{3 \cdot T\left(\frac{n}{2}\right)}_{\text{trabajo de llamadas recursivas}} + \underbrace{O(n)}_{\text{trabajo fuera de llamadas recursivas}},$$

nuevamente con un caso base $T(1) = 1$.²

4.1.3 Comparativa de las recurrencias

Por el momento desconocemos los tiempos de ejecución de RECINTMULT o KARATSUBA, pero una inspección de sus recurrencias sugiere que el segundo será, al menos, tan rápido como el primero. Mientras tanto, en la sección 1.5 vimos que el tiempo de ejecución de MERGESORT está determinado por una recurrencia similar, que sigue teniendo una llamada recursiva menos:

$$T(n) \leq \underbrace{2 \cdot T\left(\frac{n}{2}\right)}_{\text{trabajo de llamadas recursivas}} + \underbrace{O(n)}_{\text{trabajo fuera de llamadas recursivas}},$$

²Técnicamente, las llamadas recursivas sobre $a + b$ y $c + d$ podrían implicar números de $(\frac{n}{2} + 1)$ dígitos. Pero, al estar entre amigos, lo ignoramos, pues no afecta al análisis final.

donde n es la longitud del array a ordenar. Este hecho sugiere que los límites del tiempo de ejecución de RECINTMULT y KARATSUBA no pueden ser mejores que el límite de MERGESORT, que es $O(n \log n)$. Más allá de estas pistas, en realidad no tenemos ni idea de cuál puede ser el tiempo de ejecución de cada uno de los algoritmos. A continuación, dejaremos que el método maestro nos ilumine.

4.2 Declaración formal

El método maestro es, ni más ni menos, la herramienta soñada para el análisis de algoritmos recursivos. Toma la recurrencia del algoritmo como entrada y, como por arte de magia, devuelve como salida un límite superior del tiempo de ejecución del mismo.

4.2.1 Recurrencias estándar

Veremos una versión del método maestro que se ocupa de lo que llamaremos “recurrencias estándar”, que cuentan con tres parámetros libres y el siguiente formato³.

Formato de recurrencia estándar

Caso base: $T(n)$ es, como mucho, una constante para todos los n suficientemente pequeños⁴.

Caso general: para valores más grandes de n ,

$$T(n) \leq a \cdot T\left(\frac{n}{b}\right) + O(n^d).$$

Parámetros:

- a = número de llamadas recursivas
- b = factor de reducción del tamaño de la entrada
- d = exponente del tiempo de ejecución del “paso combinar”

³Esta presentación del método maestro está inspirada en el capítulo 2 de *Algorithms*, de Sanjoy Dasgupta, Christos Papadimitriou y Umesh Vazirani (McGraw-Hill, 2006).

El caso base de una recurrencia estándar establece que, una vez que el tamaño de la entrada es tan pequeño que ya no necesita de más llamadas recursivas, el problema se puede resolver en tiempo $O(1)$. Esta será la situación de todas las aplicaciones que analicemos. El caso general asume que el algoritmo realiza a llamadas recursivas, cada una de ellas sobre un subproblema con un tamaño reducido por un factor b en relación a su entrada, y ejecuta un trabajo $O(n^d)$ fuera de dichas llamadas. Por ejemplo, en el caso del algoritmo MERGESORT, hay dos llamadas recursivas ($a = 2$), cada una de ellas sobre un *array* de la mitad del tamaño de la entrada ($b = 2$), y un trabajo de $O(n)$ fuera de las llamadas ($d = 1$). En general, a puede ser cualquier entero positivo, b puede ser cualquier número real mayor que 1 (si $b \leq 1$, entonces el algoritmo no finalizará nunca) y d puede ser cualquier número real no negativo, donde $d = 0$ indica que solo se realiza trabajo constante ($O(1)$) aparte de las llamadas recursivas. Como es habitual, ignoramos el detalle de que n/b podría necesitar un redondeo a un entero (y, como es habitual, esto no influirá en nuestras conclusiones). No olvidemos nunca que a , b y d deben ser números *constantes* independientes del tamaño n de la entrada⁵. Los valores más típicos de estos parámetros son 0 (para d), 1 (para a y d), 2 y otros enteros pequeños (sobre todo para a). Si alguna vez te encuentras en la situación de “aplicar el método maestro con $a = n$ o $b = \frac{n}{n-1}$ ”, significa que no lo estás utilizando correctamente.

Una restricción de las recurrencias estándar la encontramos en que toda llamada recursiva se produce sobre un subproblema del mismo tamaño. Por ejemplo, un algoritmo que aplique una recursión al primer tercio del *array* de entrada y otra al resto, resultará en un recurrencia no estándar. La mayoría (aunque no todos) de los algoritmos naturales de divide y vencerás se dirigen a recurrencias estándar. Por ejemplo, en el algoritmo MERGESORT, ambas llamadas recursivas operan sobre problemas con un tamaño que resulta ser la mitad que el del *array* de entrada. En nuestros algoritmos recursivos para la multiplicación de enteros, las llamadas recursivas siempre reciben números con la mitad de dígitos⁶.

⁴Formalmente, existen enteros positivos n_0 y c , independientes de n , tales que $T(n) \leq c$ para todos $n \leq n_0$.

⁵También están las constantes eliminadas en el caso base y en el término “ $O(n^d)$ ”, pero las conclusiones del método maestro no dependen de sus valores.

⁶Existen versiones más generalistas del método maestro que se ajustan a un rango mayor de recurrencias, pero la versión sencilla que utilizamos nosotros es válida para casi todos los algoritmos de divide y vencerás que puedas encontrar.

4.2.2 Declaración y tratamiento del método maestro

Ahora estamos en condiciones de declarar el método maestro, que proporciona el límite superior de una recurrencia estándar como una función de los parámetros a , b y d .

Teorema 4.1 (Método maestro) *Si $T(n)$ está definido por una recurrencia estándar, con parámetros $a \geq 1$, $b > 1$ y $d \geq 0$, entonces*

$$T(n) = \begin{cases} O(n^d \log n) & \text{si } a = b^d \quad [\text{caso 1}] \\ O(n^d) & \text{si } a < b^d \quad [\text{caso 2}] \\ O(n^{\log_b a}) & \text{si } a > b^d \quad [\text{caso 3}]. \end{cases} \quad (4.2)$$

¿Qué pasa con los tres casos? ¿Y por qué son tan importantes los valores relativos de a y b^d ? En el segundo caso, ¿podría, de verdad, ser el tiempo global de ejecución del algoritmo de solo $O(n^d)$, cuando la llamada recursiva más exterior ya consume $O(n^d)$? ¿Y de dónde sale el exótico límite de tiempo de ejecución del tercer caso? Para cuando llegues al final de este capítulo, tendrás respuestas satisfactorias a todas estas preguntas, y la declaración del método maestro te parecerá lo más normal del mundo⁷.

Más sobre logaritmos

Otro aspecto desconcertante del teorema 4.1 tiene relación con el uso inconsistente de logaritmos. El tercer caso afirma cuidadosamente que el logaritmo en cuestión tiene base b , el número de veces que se puede dividir n por b antes de que el resultado sea 1 como máximo. Al mismo tiempo, el primer caso no indica absolutamente nada sobre la base del logaritmo. El motivo lo encontramos en que *dos funciones logarítmicas cualquiera solo se diferencian por un factor constante*. Por ejemplo, el logaritmo de base 2 siempre supera al logaritmo natural (es decir, al logaritmo de base e , donde $e = 2,718\dots$) por un factor de $1/\ln 2 \approx 1,44$. En el primer caso del método maestro, cambiar la base del logaritmo solo modifica el factor

⁷Los límites del teorema 4.1 tienen el formato $O(f(n))$, en vez de $\Theta(f(n))$, porque, en nuestra recurrencia, solo asumimos un límite superior de $T(n)$. Si sustituimos, en la definición de un recurrencia estándar, “≤” por “=” y $O(n^d)$ por $\Theta(n^d)$, se conservarán los límites del teorema 4.1 con $O(\cdot)$ sustituido por $\Theta(\cdot)$. Ver también el problema 4.6.

constante que será eliminado oportunamente en la notación *Big-O*. En el tercer caso, el logaritmo aparece en el exponente, donde distintos factores constantes se traducen en límites de tiempo de ejecución muy diferentes (del orden de n^2 frente a n^{100}).

4.3 Seis ejemplos

El método maestro (teorema 4.1) puede ser difícil de entender en un principio. Lo ilustraremos utilizando seis ejemplos distintos.

4.3.1 MERGESORT revisitado

Verificaremos la validez de esta técnica volviendo a analizar MERGESORT, un algoritmo cuyo tiempo de ejecución ya conocemos. Para aplicar el método maestro, lo único que tenemos que hacer es identificar los valores de los tres parámetros libres: a , el número de llamadas recursivas; b , el factor por el que se reduce el tamaño de la entrada antes de las llamadas recursivas; y d , el exponente en el límite de la cantidad de trabajo realizada fuera de las llamadas recursivas⁸. En MERGESORT tenemos dos llamadas recursivas, por lo que $a = 2$. Cada una de esas llamadas recibe la mitad del *array* de entrada, con lo que también $b = 2$. El trabajo realizado fuera de las llamadas recursivas viene determinado por la subrutina MERGE, que funciona en tiempo lineal (sección 1.5.1), así $d = 1$. Por tanto

$$a = 2 = 2^1 = b^d,$$

nos sitúa en el primer caso del método maestro. Al proporcionarle los parámetros, el teorema 4.1 nos dice que el tiempo de ejecución de MERGESORT es $O(n^d \log n) = O(n \log n)$, reproduciendo, en consecuencia, el análisis que hicimos en la sección 1.5.

4.3.2 Búsqueda binaria

Para nuestro segundo ejemplo, tomamos en consideración el problema de buscar un elemento determinado dentro de un *array* ordenado. Piensa, por

⁸Todas las recurrencias que evaluemos cuentan con un caso base en el mismo formato que las recurrencias estándar, caso que no trataremos en lo sucesivo.

ejemplo, en buscar tu nombre dentro de una lista alfabética muy larga (los lectores de una cierta edad quizás recuerden las guías telefónicas). Puedes realizar una búsqueda lineal, comenzando desde el principio pero, con ello, perderías la ventaja que ofrece la ordenación alfabética. Un sistema más optimizado consiste en comenzar consultando el centro de la lista y centrarse recursivamente en su primera mitad (si el nombre encontrado es posterior al tuyo) o en la segunda (en caso contrario). Este algoritmo, llevado al problema de la búsqueda en un *array* ordenado, se conoce como *búsqueda binaria*⁹.

¿Cuál es el tiempo de ejecución de la búsqueda binaria? Es una pregunta fácil de responder directamente, pero veamos cómo la resuelve el método maestro.

Cuestionario 4.2

¿Cuáles son los valores de a , b y d en el algoritmo de la búsqueda binaria?

- a) 1, 2, 0 [caso 1]
- b) 1, 2, 1 [caso 2]
- c) 2, 2, 0 [caso 3]
- d) 2, 2, 1 [caso 1]

Solución y aclaraciones en la sección [4.3.7](#)

4.3.3 Multiplicación recursiva de enteros

Ahora la cosa se empieza a poner interesante, con algoritmos de divide y vencerás de los que todavía no conocemos el límite del tiempo de ejecución. Comenzaremos con el algoritmo RECINTMULT para la multiplicación de enteros. En la sección [4.1](#) vimos que la recurrencia adecuada para este algoritmo es

$$T(n) \leq 4 \cdot T\left(\frac{n}{2}\right) + O(n),$$

⁹Si nunca has visto un código fuente que implemente este algoritmo, puedes consultar tu libro favorito de introducción a la programación o algún tutorial en internet.

por lo que $a = 4$, $b = 2$ y $d = 1$. Por tanto

$$a = 4 > 2 = 2^1 = b^d,$$

lo que nos sitúa en el tercer caso del método maestro. En esta circunstancia, obtenemos un límite de ejecución de aspecto un tanto exótico, $O(n^{\log_b a})$. Con los valores de nuestros parámetros, esto se traduce en $O(n^{\log_2 4}) = O(n^2)$. Por lo tanto, el algoritmo RECINTMULT iguala, pero no supera, al algoritmo de multiplicación de enteros de la escuela (que realiza $\Theta(n^2)$ operaciones).

4.3.4 Multiplicación de Karatsuba

Emplear una técnica de divide y vencerás para multiplicar enteros solo compensa tras utilizar el truco de Gauss para ahorrar una llamada recursiva. Como vimos en la sección 4.1, el tiempo de ejecución del algoritmo KARATSUBA viene determinado por la recurrencia

$$T(n) \leq 3 \cdot T\left(\frac{n}{2}\right) + O(n),$$

que se diferencia únicamente de la recurrencia anterior en el hecho de que a se reduce de 4 a 3 (b y d siguen siendo 2 y 1, respectivamente). Nuestra expectativa es que el tiempo de ejecución se ubique en algún punto entre $O(n \log n)$ (el límite cuando $a = 2$, como en MERGESORT) y $O(n^2)$ (el límite cuando $a = 4$, como en RECINTMULT). Si sientes que la duda te supera, el método maestro ofrece una solución rápida. Tenemos que

$$a = 3 > 2 = 2^1 = b^d,$$

por lo que seguimos estando en el tercer caso del método maestro, pero con un límite de tiempo de ejecución mejorado: $O(n^{\log_b a}) = O(n^{\log_2 3}) = O(n^{1.59})$. En consecuencia, ahorrar una llamada recursiva resulta en un tiempo de ejecución sustancialmente mejor y en que el algoritmo para la multiplicación de enteros que aprendiste en tu tercer año escolar no sea el más rápido que existe¹⁰.

4.3.5 Multiplicación de matrices

La sección 3.3 trataba sobre el problema de multiplicar dos matrices de tamaño $n \times n$. Al igual que con la multiplicación de enteros, tratamos tres

¹⁰Un dato curioso: en el lenguaje de programación Python, la subrutina integrada para la multiplicación de enteros utiliza el algoritmo de la escuela para enteros de hasta 70 dígitos, pasando a aplicar KARATSUBA en el resto de casos.

algoritmos (un algoritmo iterativo directo, el algoritmo recursivo directo RECMATMULT y el ingenioso algoritmo de STRASSEN). El algoritmo iterativo emplea $\Theta(n^3)$ operaciones (cuestionario 3.3). El algoritmo RECMMULT divide cada una de las dos matrices de entrada en cuatro matrices de tamaño $\frac{n}{2} \times \frac{n}{2}$ (una por cada cuadrante), realiza las ocho llamadas recursivas correspondientes sobre las matrices más pequeñas y combina los resultados adecuadamente (utilizando la suma de matrices directa). El algoritmo de STRASSEN es más inteligente, pues identifica siete pares de matrices $\frac{n}{2} \times \frac{n}{2}$, cuyos productos bastan para reconstruir el producto de las matrices de entrada originales.

Cuestionario 4.3

¿Cuáles son los límites del tiempo de ejecución respectivos, que aporta el método maestro para los algoritmos RECMMULT y STRASSEN?

- a) $O(n^3)$ y $O(n^2)$
- b) $O(n^3)$ y $O(n^{\log_2 7})$
- c) $O(n^3)$ y $O(n^3)$
- d) $O(n^3 \log n)$ y $O(n^3)$

Solución y aclaraciones en la sección 4.3.7

4.3.6 Una recurrencia ficticia

De los cinco ejemplos que hemos visto, dos de la recurrencias corresponden al primer caso del método maestro, mientras que el resto se asignan al tercero. También es posible encontrar recurrencias que se corresponden, de forma natural, con el segundo caso. Por ejemplo, supongamos que tenemos un algoritmo de divide y vencerás que funciona igual que MERGESORT, salvo por el hecho de que realiza más trabajo fuera de las llamadas recursivas, pasando este a ser cuadrático en vez de lineal. Es decir, pensemos en la recurrencia

$$T(n) \leq 2 \cdot T\left(\frac{n}{2}\right) + O(n^2).$$

En este caso, tenemos que

$$a = 2 < 4 = 2^2 = b^d,$$

lo que nos sitúa de cabeza en el segundo caso del método maestro, para un límite del tiempo de ejecución de $O(n^d) = O(n^2)$. Esto podría parecer contradictorio, dado que el algoritmo MERGESORT realiza trabajo lineal fuera de las dos llamadas recursivas y tiene un tiempo de ejecución de $O(n \log n)$, cabría esperar que un paso de combinación de tiempo cuadrático implicaría un tiempo de ejecución de $O(n^2 \log n)$. El método maestro afirma que esta estimación es excesiva y aporta el límite más apropiado de $O(n^2)$. Es interesante ver cómo esto implica que el tiempo de ejecución total del algoritmo viene dominado por el trabajo realizado en la llamada más exterior (y el resto de llamadas recursivas solo aumentan por un factor constante el resto de operaciones realizadas)¹¹.

4.3.7 Soluciones a los cuestionarios 4.2–4.3

Solución al cuestionario 4.2

Respuesta correcta: (a). La búsqueda binaria opera recursivamente en la mitad izquierda o en la mitad derecha del *array* de entrada (nunca en ambas), por lo que solo se produce una llamada recursiva ($a = 1$). Esta llamada opera sobre una mitad del *array* de entrada, por lo que b vuelve a ser igual a 2. Fuera de la llamada recursiva, lo único que hace la búsqueda binaria es una comparación sencilla (entre el elemento central del *array* y el elemento que se busca) para determinar si la siguiente recurrencia será sobre la mitad izquierda o la derecha. Esto se traduce en un trabajo $O(1)$ fuera de la llamada recursiva, por lo que $d = 0$. Como $a = 1 = 2^0 = b^d$, volvemos a encontrarnos con el primer caso del método maestro, y obtenemos un límite de tiempo de ejecución de $O(n^d \log n) = O(\log n)$.

Solución al cuestionario 4.3

Respuesta correcta: (b). Comencemos con el algoritmo RECMATMULT (sección 3.3.4). Digamos que $T(n)$ indica el número máximo de operaciones básicas que utiliza el algoritmo para multiplicar dos matrices $n \times n$. El número a de llamadas recursivas es 8. Cada una de estas llamadas se produce sobre un par de matrices de tamaño $\frac{n}{2} \times \frac{n}{2}$, por lo que $b = 2$. El trabajo realizado fuera de las llamadas recursivas implica un número constante de sumas de matrices, que necesitan un tiempo $O(n^2)$ (tiempo constante para

¹¹Veremos otro ejemplo del caso 2 del método maestro cuando estudiemos la selección en tiempo lineal del capítulo 6.

cada una de las n^2 entradas de la matriz). Por tanto, la recurrencia es

$$T(n) \leq 8 \cdot T\left(\frac{n}{2}\right) + O(n^2),$$

y como

$$a = 8 > 4 = 2^2 = b^d,$$

estamos ante el tercer caso del método maestro, que resulta en un límite del tiempo de ejecución de $O(n^{\log_b a}) = O(n^{\log_2 8}) = O(n^3)$.

La única diferencia entre la recurrencia del algoritmo STRASSEN y la que acabamos de ver, es que el número a de llamadas recursivas se reduce de 8 a 7. Es cierto que el algoritmo STRASSEN realiza más sumas y restas de matrices que RECMATMULT, pero solo por un factor constante y, en consecuencia, d sigue siendo igual a 2. Con ello

$$a = 7 > 4 = 2^2 = b^d.$$

Seguimos en el tercer caso del método maestro, pero con un límite del tiempo ejecución mejorado: $O(n^{\log_b a}) = O(n^{\log_2 7}) = O(n^{2.81})$. Así, el algoritmo STRASSEN es, en realidad, asintóticamente superior al algoritmo iterativo directo¹².

*4.4 Demostración del método maestro

Esta sección demuestra el método maestro (teorema 4.1): si $T(n)$ viene determinado por una recurrencia estándar, con la forma

$$T(n) \leq a \cdot T\left(\frac{n}{b}\right) + O(n^d),$$

entonces

$$T(n) = \begin{cases} O(n^d \log n) & \text{si } a = b^d \quad [\text{caso 1}] \\ O(n^d) & \text{si } a < b^d \quad [\text{caso 2}] \\ O(n^{\log_b a}) & \text{si } a > b^d \quad [\text{caso 3}]. \end{cases}$$

Es importante recordar el significado de los tres parámetros libres:

¹²Existe una dilatada lista de trabajos de investigación que plantean algoritmos para la multiplicación de matrices, cada vez más sofisticados y que van mejorando los tiempos de ejecución asintóticos para el peor caso (aunque con factores constantes cada vez más grandes, lo que dificulta realizar implementaciones realistas). En la actualidad, el record mundial se encuentra en un límite del tiempo de ejecución de, aproximadamente, $O(n^{2.3729})$ y, por lo que sabemos, se podría llegar a descubrir un algoritmo de tiempo $O(n^2)$.

Parámetro	Significado
a	número de llamadas recursivas
b	factor por el que se reduce la entrada en cada llamada recursiva
d	exponente del trabajo realizado fuera de las llamadas recursivas

4.4.1 Preámbulo

La demostración del método maestro no es solo interesante por una cuestión de formalismo, también supone la explicación fundamental de por qué las cosas son como son en cuestiones tales como el hecho de que cuente con tres posibles casos. Con esta idea en mente, debes distinguir entre dos tipos de contenido en relación a la demostración. En un par de ocasiones recurriremos a cálculos algebraicos para entender qué está pasando. Merece la pena estudiar estos cálculos al menos una vez en la vida, aunque no es particularmente importante recordarlos a largo plazo. Lo que sí es necesario recordar es el sentido conceptual de los tres casos del método maestro. La demostración utilizará la técnica del árbol de recursión, que tan útil nos resultó al analizar el algoritmo MERGESORT (sección 1.5), donde los tres casos se corresponden con tres tipos diferentes de árboles de recursión. Si eres capaz de recordar el significado de los tres casos, no tendrás ninguna necesidad de memorizar los tiempos de ejecución del método maestro, pues siempre podrás obtenerlos por ingeniería inversa a partir del conocimiento conceptual de los mismos.

De cara a realizar una demostración formal, debemos expresar explícitamente todos los factores constantes de la recurrencia:

Caso base: $T(1) \leq c$.

Caso general: para $n > 1$,

$$T(n) \leq a \cdot T\left(\frac{n}{b}\right) + cn^d. \quad (4.3)$$

Para simplificar el proceso, asumiremos que la constante n_0 que determina cuándo entra en juego el caso base, es 1. La demostración para una constante n_0 diferente es, básicamente, la misma. Podemos asumir también que las constantes eliminadas del caso base, y del término $O(n^d)$ del caso general, son iguales al mismo número c . Si fuesen constantes diferentes, bastaría con trabajar con la mayor de las dos. Por último, vamos a centrarnos en el caso en el que n es una potencia de b . La demostración del caso general es similar, sin más contenido conceptual, aunque más tediosa.

4.4.2 Árboles de recursión revisitados

El plan de alto nivel de la demostración es de una enorme sencillez: generalizar el argumento del árbol de recursión de MERGESORT (sección 1.5), de forma que se adapte a otros valores de los parámetros clave a , b y d . Recordemos que un árbol de recursión nos proporciona un sistema para mantener un registro de todo el trabajo realizado por un algoritmo recursivo, a lo largo de todas sus llamadas. Los nodos del árbol corresponden a las llamadas recursivas y los hijos de un nodo hacen referencia a las llamadas recursivas realizadas por ese nodo (figura 4.1). La raíz (nivel 0) del árbol de recursión corresponde a la llamada más externa del algoritmo, el nivel 1 cuenta con a nodos, que corresponden a sus llamadas recursivas y, así, sucesivamente. Esto hace que, en la parte inferior del árbol, se sitúen las llamadas recursivas que provocan que se active el caso base.

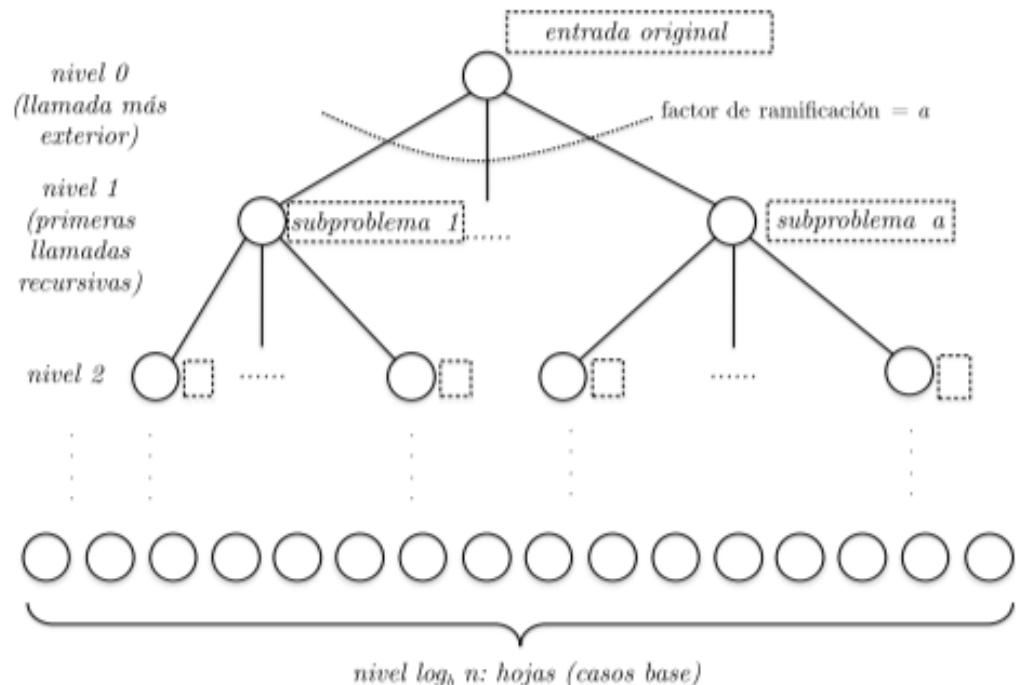


Figura 4.1: Árbol de recursión correspondiente a una recurrencia estándar. Los nodos corresponden a llamadas recursivas. El nivel 0 representa a la llamada más exterior, el nivel 1 a sus llamadas recursivas, etc.

Al igual que en nuestro análisis de MERGESORT, estamos interesados en conocer, nivel por nivel, el trabajo realizado por el algoritmo recursivo. Este plan requiere comprender dos cosas: el número de subproblemas dis-

tintos que se encuentran en un nivel de recursión j dado y la longitud de la entrada de cada uno de esos subproblemas.

Cuestionario 4.4

¿Cuál es el patrón? Completa los espacios en blanco de la siguiente afirmación: en cada nivel $j = 0, 1, 2, \dots$ del árbol de recursión, hay [espacio en blanco] subproblemas, cada uno de ellos operando sobre un *subarray* de longitud [espacio en blanco].

- a) a^j y n/a^j , respectivamente
- b) a^j y n/b^j , respectivamente
- c) b^j y n/a^j , respectivamente
- d) b^j y n/b^j , respectivamente

Solución y aclaraciones en la sección [4.4.10](#)

4.4.3 Trabajo realizado en un solo nivel

Inspirados por nuestro análisis de MERGESORT, el plan consiste en contar el número total de operaciones realizadas por los subproblemas del nivel j en un algoritmo de divide y vencerás y, después, sumar todos los niveles. Vamos a centrarnos en un nivel de recursión j . Según la solución al cuestionario 4.4, existen a^j subproblemas diferentes en el nivel j , cada uno de ellos con una entrada de tamaño n/b^j . El tamaño de un subproblema nos importa solo en cuanto a que determina la cantidad de trabajo que realiza la llamada recursiva. Nuestra recurrencia (4.3) afirma que el trabajo realizado por un subproblema del nivel j , sin tener en cuenta el correspondiente a sus llamadas recursivas, supone, como mucho, un número constante de repeticiones del tamaño de entrada elevado a la d -ésima potencia: $c(n/b^j)^d$. La suma de todos los a^j de los subproblemas del nivel j resulta en el límite superior del trabajo realizado en ese nivel j del árbol de recursión:

$$\text{trabajo en el nivel } j \leq \underbrace{\frac{a^j}{\# \text{ de subproblemas}}}_{\substack{\text{trabajo por subproblema} \\ \text{tamaño entrada}}} \cdot \underbrace{c \cdot \left[\frac{n}{b^j} \right]^d}_{\substack{\text{trabajo por subproblema} \\ \text{tamaño entrada}}}.$$

Vamos a simplificar esta expresión, separando las partes que dependen del nivel j de aquellas que no:

$$\text{trabajo en el nivel } j \leq cn^d \cdot \left[\frac{a}{b^d} \right]^j.$$

El lado derecho supone la entrada por la puerta grande de la razón crítica a/b^d . Dado que el valor de a frente a b^d es, exactamente, lo que determina el caso relevante del método maestro, no nos debería sorprender que esta razón aparezca explícitamente en el análisis.

4.4.4 Suma de todos los niveles

¿Cuántos niveles tenemos? El tamaño de la entrada es, inicialmente, n y se reduce por un factor de b en cada nivel. Como asumimos que n es una potencia de b y que el caso base se activa cuando el tamaño de la entrada es 1, el último nivel corresponde exactamente al número de veces que tendrás que dividir n entre b para llegar a 1, también conocido como $\log_b n$. Al acumular todos los niveles $j = 0, 1, 2, \dots, \log_b n$, obtenemos este misterioso límite superior del tiempo de ejecución (dando por hecho que cn^d es independiente de j y se puede poner al frente):

$$\text{trabajo total} \leq cn^d \cdot \sum_{j=0}^{\log_b n} \left[\frac{a}{b^d} \right]^j. \quad (4.4)$$

Lo creas o no, acabamos de alcanzar un hito importante en la demostración de método maestro. El lado derecho de la desigualdad (4.4) parecerá, seguramente, una sopa de letras pero, si lo interpretamos adecuadamente, contiene las claves para desbloquear un conocimiento profundo del método maestro.

4.4.5 El bien frente al mal: la necesidad de los tres casos

Ahora le añadiremos algo de semántica al límite del tiempo de ejecución de (4.4) y desarrollaremos cierta intuición sobre por qué los límites del tiempo de ejecución del método maestro son los que son.

¿Por qué es tan importante la razón entre a y b^d ? Básicamente, esta comparativa representa un tira y afloja entre las fuerzas del bien y del mal. El mal está representado por a , la *tasa de proliferación de subproblemas (RSP)*: en cada nivel de recursión, el número de subproblemas crece por un factor

de a , lo que puede resultar un tanto intimidante. El bien adopta la forma de b^d , la *tasa de reducción del trabajo* (*RWS*): la buena noticia es que, con cada nivel de recursión, la cantidad de trabajo por subproblema se reduce por un factor de b^d .¹³ La pregunta clave es, entonces: ¿quién gana, el bien o el mal? Los tres casos del método maestro se corresponden, precisamente, con cada uno de los tres posibles resultados de este tira y afloja: el empate ($RSP = RWS$), la victoria del bien ($RSP < RWS$) o la victoria del mal ($RSP > RWS$).

Para entender mejor este extremo, podemos meditar sobre la cantidad de trabajo realizada en cada nivel de un árbol de recursión (como se muestra en la figura 4.1). ¿Cuándo aumenta la cantidad de trabajo realizado con el nivel j del árbol de recursión? ¿Cuándo disminuye? ¿Alguna vez es igual en todos los niveles?

Cuestionario 4.5

¿Cuál de las siguientes afirmaciones es cierta (marca todas las que lo sean)?

- a) Si $RSP < RWS$, entonces la cantidad de trabajo realizada disminuye con el nivel de recursión j .
- b) Si $RSP > RWS$, entonces la cantidad de trabajo realizada aumenta con el nivel de recursión j .
- c) No se pueden obtener conclusiones sobre cómo varía la cantidad de trabajo con el nivel de recursión j salvo que $RSP = RWS$.
- d) Si $RSP = RWS$, entonces la cantidad de trabajo es la misma en todos los niveles de recursión.

Solución y aclaraciones en la sección 4.4.10

¹³¿Por qué b^d en lugar de b ? Porque b es la tasa por la que se reduce el *tamaño de la entrada*, que solo nos importa en tanto en cuanto determine la cantidad de trabajo realizado. Por ejemplo, en un algoritmo de divide y vencerás con un paso de combinación de tiempo cuadrático ($d = 2$), cuando el tamaño de la entrada se divide por la mitad ($b = 2$), solo será necesario el 25% del trabajo para resolver un subproblema más pequeño (porque $b^d = 4$).

4.4.6 Predicción de los límites del tiempo de ejecución

Ahora hemos entendido por qué el método maestro tiene tres posibles casos. Existen, básicamente, tres tipos de árboles de recursión (con un trabajo por nivel que se mantiene igual, que disminuye o que aumenta) y los tamaños relativos de a (la RSP) y b^d (la RWS) determinan el tipo de árbol de recursión de un algoritmo de divide y vencerás.

Incluso mejor, ahora hemos desarrollado una intuición lo suficientemente buena como para predecir con precisión los límites del tiempo de ejecución que aparecen en el método maestro. Consideremos el primer caso, cuando $a = b^d$ y el algoritmo realiza la misma cantidad de trabajo en cada nivel de su árbol de recursión. Sin duda conocemos la cantidad de trabajo que se realiza en la raíz (nivel 0), $O(n^d)$, pues está especificado de forma explícita en la recurrencia. Con un trabajo de $O(n^d)$ por nivel, y con $1 + \log_b n = O(\log n)$ niveles, deberíamos esperar, en este caso, un límite del tiempo de ejecución de $O(n^d \log n)$ (comparar con el caso 1 del teorema 4.1).

En el segundo caso, $a < b^d$ y resultan victoriosas las fuerzas del bien: la cantidad de trabajo realizado disminuye con el nivel. Por lo tanto, se realiza más trabajo en el nivel 0 que en cualquier otro. El resultado mejor y más sencillo que podemos esperar es que el trabajo realizado en la raíz determine el tiempo de ejecución del algoritmo. Como, en la raíz, se ejecuta un trabajo $O(n^d)$, estaríamos ante la situación del mejor caso posible, que se traduciría en un tiempo de ejecución global de $O(n^d)$ (comparar con el caso 2 del teorema 4.1).

En el tercer caso, cuando los subproblemas se reproducen más rápido que la reducción del trabajo por subproblema, la cantidad de trabajo aumenta con cada nivel de recursión, realizándose la mayor parte de la tarea en las hojas del árbol. Nuevamente, la situación más sencilla y mejor sería que el tiempo de ejecución viniese determinado por el trabajo realizado en las hojas. Una hoja corresponde a una llamada recursiva en la que se activa el caso base, de forma que el algoritmo realiza únicamente $O(1)$ operaciones por hoja. ¿Cuántas hojas tenemos? Según la solución al cuestionario 4.4, sabemos que hay a^j nodos en cada nivel j . Las hojas se encuentran en el último nivel $j = \log_b n$, por lo que hay $a^{\log_b n}$. Por tanto, el mejor caso se traduce en un límite del tiempo de ejecución de $O(a^{\log_b n})$.

Nos queda por resolver el misterio de la conexión entre nuestra predicción del límite del tiempo de ejecución para el tercer caso del método maestro ($O(a^{\log_b n})$) y el auténtico límite que indica el teorema 4.1 ($O(n^{\log_b a})$). Y

esa conexión es que... son exactamente el mismo. La identidad

$$\underbrace{a^{\log_b n}}_{\text{más intuitivo}} = \underbrace{n^{\log_b a}}_{\text{más fácil de aplicar}}$$

puede parecer un error de novato cometido por un estudiante de álgebra inexperto, pero la realidad es que es cierta¹⁴. Por tanto, el límite del tiempo de ejecución de $O(n^{\log_b a})$ se limita a indicar que el trabajo realizado en las hojas del árbol de recursión determina el cálculo, con el límite expresado en una forma adecuada para la inclusión de parámetros (al igual que con los algoritmos para multiplicación de enteros y matrices de la sección 4.3).

4.4.7 Los cálculos finales: caso 1

Todavía debemos verificar que la intuición que tuvimos en la sección anterior sea, de hecho, correcta, y la forma de lograrlo es a través de una demostración formal. Nuestros cálculos anteriores culminaron con el siguiente límite superior, un tanto intimidante, para el tiempo de ejecución de un algoritmo de divide y vencerás, en relación a a , b y d :

$$\text{trabajo total} \leq cn^d \cdot \sum_{j=0}^{\log_b n} \left[\frac{a}{b^d} \right]^j. \quad (4.5)$$

Obtuvimos este límite al centrarnos en un nivel j en concreto dentro del árbol de recursión (con sus a^j subproblemas y trabajo $c(n/b^j)^d$ por cada uno de ellos) y, después, acumulando los niveles.

Cuando las fuerzas del bien y del mal están en un equilibrio perfecto (es decir, $a = b^d$) y el algoritmo realiza la misma cantidad de trabajo en todos los niveles, el lado derecho de (4.5) queda notablemente simplificado:

$$cn^d \cdot \sum_{j=0}^{\log_b n} \underbrace{\left[\frac{a}{b^d} \right]^j}_{=1 \text{ por cada } j} = cn^d \cdot \underbrace{(1 + 1 + \dots + 1)}_{1 + \log_b n \text{ veces}},$$

que es $O(n^d \log n)$.¹⁵

¹⁴Para comprobarlo, basta con tomar el logaritmo en base b de ambos lados: $\log_b(a^{\log_b n}) = \log_b n \cdot \log_b a = \log_b a \cdot \log_b n = \log_b(n^{\log_b a})$. Como \log_b es una función estrictamente creciente, la única forma de que $\log_b x$ y $\log_b y$ sean iguales, es que x e y también lo sean.

¹⁵Recordemos que, como las distintas funciones logarítmicas solo se diferencian por un factor constante, no hay necesidad de especificar la base de este logaritmo.

4.4.8 Desvío: series geométricas

Nuestra esperanza es que, en los segundo y tercer tipos de árboles de recursión (con trabajo decreciente y creciente en cada nivel, respectivamente), el tiempo de ejecución global venga determinado por el trabajo realizado en el nivel más complejo (la raíz y las hojas, respectivamente). Convertir esta esperanza en una realidad implica la comprensión de las series geométricas, que son expresiones con la forma $1 + r + r^2 + \dots + r^k$, para un número real r y un entero no negativo k (en nuestro caso, r será la razón crítica a/b^d). Siempre que encuentres una expresión parametrizada como esta, es una buena idea tener en mente un par de valores canónicos para los parámetros. Por ejemplo, si $r = 2$, es la suma de las potencias positivas de 2: $1 + 2 + 4 + 8 + \dots + 2^k$. Cuando $r = \frac{1}{2}$, es la suma de las potencias negativas de 2: $1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \dots + \frac{1}{2^k}$.

Existe, para el caso de que $r \neq 1$, una fórmula cerrada muy útil para series geométricas¹⁶:

$$1 + r + r^2 + \dots + r^k = \frac{1 - r^{k+1}}{1 - r}. \quad (4.6)$$

Hay dos consecuencias de esta fórmula que son importantes para nosotros. En primer lugar, cuando r es positivo y menor que 1,

$$1 + r + r^2 + \dots + r^k \leq \frac{1}{1 - r} = \text{una constante (independiente de } k\text{)}.$$

Por tanto, *toda serie geométrica con $r \in (0, 1)$ está dominada por su primer término*: ese primer término es 1 y la suma es, solamente, $O(1)$. Por ejemplo, no tiene importancia el número de potencias de $\frac{1}{2}$ que puedas sumar, pues el resultado nunca será mayor que 2.

En segundo lugar, cuando $r > 1$,

$$1 + r + r^2 + \dots + r^k = \frac{r^{k+1} - 1}{r - 1} \leq \frac{r^{k+1}}{r - 1} = r^k \cdot \frac{r}{r - 1}.$$

Así, *toda serie geométrica con $r > 1$ está dominada por su último término*: ese último término es r^k , mientras que la suma resultará en, como mucho, un factor constante ($r/(r - 1)$) de veces el mismo. Por ejemplo, si sumas todas las potencias de 2 hasta 1024, el resultado será menor que 2048.

¹⁶Para verificar esta identidad, basta con multiplicar ambos lados por $1 - r$: $(1 - r)(1 + r + r^2 + \dots + r^k) = 1 - r + r - r^2 + r^2 - r^3 + r^3 - \dots - r^{k+1} = 1 - r^{k+1}$.

4.4.9 Los cálculos finales: casos 2 y 3

Volviendo a nuestro análisis de (4.5), supongamos que $a < b^d$. En este caso, la proliferación de subproblemas tiene menos peso que el ahorro de trabajo por subproblema, al tiempo que el número de operaciones realizadas disminuye según aumenta el nivel en el árbol de recursión. Establecemos $r = a/b^d$. Como a, b y d son constantes (independientes del tamaño n de la entrada), también lo es r . Como r es positivo y menor que 1, la serie geométrica de (4.5) alcanzará el valor máximo de la constante $1/(1 - r)$, y el límite de (4.5) será

$$cn^d \cdot \underbrace{\sum_{j=0}^{\log_b n} r^j}_{=O(1)} = O(n^d),$$

donde la notación *Big-O* del lado derecho eliminará el factor constante $c/(1 - r)$. Esto confirma nuestra esperanza de que, en el segundo tipo de árbol de recursión, la cantidad de trabajo total realizada queda determinada por las operaciones ejecutadas en la raíz.

Supongamos, para el último caso, que $a > b^d$, donde la proliferación de subproblemas vence a la reducción de trabajo que se produce en cada uno de ellos. Establecemos $r = a/b^d$. Como r ahora es mayor que 1, el último término de la serie geométrica es el dominante y el límite de (4.5) será

$$cn^d \cdot \underbrace{\sum_{j=0}^{\log_b n} r^j}_{=O(r^{\log_b n})} = O(n^d \cdot r^{\log_b n}) = O\left(n^d \cdot \left(\frac{a}{b^d}\right)^{\log_b n}\right). \quad (4.7)$$

El aspecto puede resultar un tanto confuso, hasta que detectamos algunas cancelaciones importantes. Como la exponenciación con base b y el logaritmo, también con base b , son operaciones inversas, podemos escribir

$$(b^{-d})^{\log_b n} = b^{-d \log_b n} = (b^{\log_b n})^{-d} = n^{-d}.$$

Por lo que el término $(1/b^d)^{\log_b n}$ de (4.7) cancela al término n^d , dejándonos con un límite superior de $O(a^{\log_b n})$. Esto confirma nuestra esperanza de que, en este caso, el tiempo de ejecución total viene dominado por el trabajo que se realiza en las hojas del árbol de recursión. Como $a^{\log_b n}$ es lo mismo que $n^{\log_b a}$, hemos finalizado la demostración del método maestro. *QED*

4.4.10 Soluciones a los cuestionarios 4.4–4.5

Solución al cuestionario 4.4

Respuesta correcta: (b). En primer lugar, por definición, el “factor de ramificación” del árbol de recursión es a : toda llamada recursiva que no active los casos base provoca a nuevas llamadas recursivas. Esto significa que el número de subproblemas diferentes se multiplica por a en cada nivel. Como hay un subproblema en el nivel 0, habrá a^j en el nivel j .

Para la segunda parte de la solución y, nuevamente, por definición, el tamaño del subproblema disminuye en cada nivel por un factor de b . Como el tamaño del problema es n en el nivel 0, todos los subproblemas del nivel j tienen un tamaño n/b^j .¹⁷

Solución al cuestionario 4.5

Respuestas correctas: (a), (b), (d). Supongamos inicialmente que $RSP < RWS$, por lo que las fuerzas del bien son más poderosas que las del mal: la reducción del trabajo realizado en cada subproblema cubre con creces el aumento del número de subproblemas. En este caso, el algoritmo realiza menos trabajo en cada nuevo nivel de recursión. Con ello, la primera afirmación es cierta (y la tercera es falsa). La segunda afirmación también es cierta por motivos similares: si los subproblemas crecen tan rápido que superan al ahorro por subproblema, entonces cada nivel de recursión realiza más trabajo que el anterior. En la última afirmación, cuando $RSP = RWS$, existe un equilibrio perfecto entre las fuerzas del bien y del mal. Los subproblemas se reproducen, pero el ahorro de trabajo por subproblema crece en la misma medida. Las dos fuerzas se cancelan entre sí, por lo que el trabajo realizado en cada nivel del árbol de recursión es el mismo.

¹⁷A diferencia del caso del análisis de MERGESORT, el hecho de que el número de subproblemas del nivel j sea a^j , no implica que el tamaño de cada subproblema sea n/a^j . En MERGESORT, las entradas de los subproblemas del nivel j forman una partición de la entrada original. Pero este caso no se reproduce en todos los algoritmos de divide y vencerás. Por ejemplo, en nuestros algoritmos recursivos para la multiplicación de enteros y matrices, hay partes de la entrada original que sea reutilizan en distintas llamadas recursivas.

Conclusiones

- ★ Una recurrencia expresa un límite del tiempo de ejecución $T(n)$ en términos del número de operaciones realizadas por las llamadas recursivas.
- ★ Un recurrencia estándar $T(n) \leq aT(\frac{n}{b}) + O(n^d)$ viene definida por tres parámetros: el número de llamadas recursivas a , el factor de reducción del tamaño de la entrada b y el exponente d del tiempo de ejecución del paso de combinación.
- ★ El método maestro aporta un límite superior asintótico a toda recurrencia estándar, en relación a a , b y d : $O(n^d \log n)$ si $a = b^d$, $O(n^d)$ si $a < b^d$ y $O(n^{\log_b a})$ si $a > b^d$.
- ★ Hay casos especiales que incluyen un límite de tiempo $O(n \log n)$ para MERGESORT, un límite de tiempo $O(n^{1,59})$ para KARATSUBA y un límite de tiempo $O(n^{2,81})$ para STRASSEN.
- ★ La demostración del método maestro generaliza el argumento del árbol de recursión utilizado para el análisis de MERGESORT.
- ★ Las cantidades a y b^d representan a las fuerzas del mal (la tasa de proliferación de subproblemas) y del bien (la tasa de reducción del trabajo).
- ★ Los tres casos del método maestro se corresponden con tres tipos diferentes de árboles de recursión: aquellos en los que el trabajo realizado en cada nivel es el mismo (una igualdad entre el bien y el mal), se reduce (cuando gana el bien) o aumenta (cuando gana el mal).
- ★ Las propiedades de las series geométricas implican que el trabajo realizado en la raíz del árbol de recursión (que es $O(n^d)$) domina el tiempo de ejecución global en el segundo caso, mientras que el trabajo realizado en las hojas (que es $O(a^{\log_b n}) = O(n^{\log_b a})$), hace lo propio en el tercero.

Comprueba que lo has entendido

Problema 4.1 (S) Recordemos el método maestro (teorema 4.1) y sus tres parámetros a , b y d . ¿Cuál de estas es la mejor interpretación de b^d ?

- a) La tasa por la que crece el trabajo total (por nivel de recursión).
- b) La tasa por la que crece el número de subproblemas (por nivel de recursión).
- c) La tasa por la que se reduce el tamaño de los subproblemas (por nivel de recursión).
- d) La tasa por la que se reduce el trabajo por subproblema (por nivel de recursión).

Problema 4.2 (S) ¿Cuáles de las siguientes afirmaciones sobre el método maestro son ciertas (marca todas las que correspondan)?

- a) Una recurrencia estándar con $a = 1$ siempre pertenece al caso 2 del método maestro.
- b) Una recurrencia estándar con $a = 1$ nunca pertenece al caso 3 del método maestro.
- c) Una recurrencia estándar con $a > b$ nunca pertenece al caso 2 del método maestro.
- d) Una recurrencia estándar con $a > b$ siempre pertenece al caso 3 del método maestro.

Problema 4.3 (S) Esta pregunta, junto a las dos siguientes, te permitirán practicar el método maestro un poco más. Supongamos que el tiempo de ejecución $T(n)$ de un algoritmo viene limitado por una recurrencia estándar con $T(n) \leq 7 \cdot T\left(\frac{n}{3}\right) + O(n^2)$. ¿Cuál de los siguientes es el límite superior correcto más pequeño del tiempo de ejecución asintótico del algoritmo?

- a) $O(n \log n)$
- b) $O(n^2)$
- c) $O(n^2 \log n)$
- d) $O(n^{2,81})$

Problema 4.4 Supongamos que el tiempo de ejecución $T(n)$ de un algoritmo viene limitado por una recurrencia estándar con $T(n) \leq 9 \cdot T\left(\frac{n}{3}\right) + O(n^2)$. ¿Cuál de los siguientes es el límite superior correcto más pequeño del tiempo de ejecución asintótico del algoritmo?

- a) $O(n \log n)$
- b) $O(n^2)$
- c) $O(n^2 \log n)$
- d) $O(n^{3,17})$

Problema 4.5 Supongamos que el tiempo de ejecución $T(n)$ de un algoritmo viene limitado por una recurrencia estándar con $T(n) \leq 5 \cdot T\left(\frac{n}{3}\right) + O(n)$. ¿Cuál de los siguientes es el límite superior correcto más pequeño del tiempo de ejecución asintótico del algoritmo?

- a) $O(n^{\log_5 3})$
- b) $O(n \log n)$
- c) $O(n^{\log_3 5})$
- d) $O(n^{5/3})$
- e) $O(n^2)$
- f) $O(n^{2,59})$

Problemas más difíciles

Problema 4.6 Digamos que $T(n)$ es una recurrencia que satisface $T(1) = 1$ y, para valores mayores de n ,

$$T(n) \geq a \cdot T\left(\frac{n}{b}\right) + \Omega(n^d),$$

donde, como es habitual, a es un entero positivo, $b > 1$ y $d \geq 0$ (para simplificar, puedes centrar tu atención solamente en valores de n que sean potencias de b). Demuestra que

$$T(n) = \begin{cases} \Omega(n^d \log n) & \text{si } a = b^d \\ \Omega(n^d) & \text{si } a < b^d \\ \Omega(n^{\log_b a}) & \text{si } a > b^d. \end{cases}$$

Problema 4.7 (S) Supongamos que el tiempo de ejecución $T(n)$ viene limitado por la recurrencia (no estándar) con $T(1) = 1$ y $T(n) \leq T(\lfloor \sqrt{n} \rfloor) + 1$ para $n > 1$.¹⁸ ¿Cuál de los siguientes es el límite superior correcto más pequeño del tiempo de ejecución asintótico del algoritmo (en este caso no se puede aplicar el método maestro)?

- a) $O(1)$
- b) $O(\log \log n)$
- c) $O(\log n)$
- d) $O(\sqrt{n})$

¹⁸En este caso, $\lfloor x \rfloor$ representa la función “suelo”, que redondea su argumento a la parte entera de x .

Ordenación *QuickSort*

Este capítulo trata sobre **QUICKSORT**, un algoritmo digno de formar parte del salón de la fama. Después de una introducción de alto nivel sobre su funcionamiento (sección 5.1), trataremos sobre cómo particionar en tiempo lineal un *array* en torno a un “elemento pivote” (sección 5.2) y de cómo seleccionar adecuadamente ese elemento pivote (sección 5.3). La sección 5.4 nos presentará el **QUICKSORT** aleatorizado, mientras que la sección 5.5 demostrará que su tiempo de ejecución asintótico medio es $O(n \log n)$ para *arrays* de n elementos. La sección 5.6 culminará nuestro conocimiento sobre la ordenación demostrando que ningún algoritmo de ordenación “basado en comparaciones” puede tener un tiempo de ejecución más rápido que $O(n \log n)$.

5.1 Introducción

Si le pides a cualquier científico de la computación o programador profesional que te haga una lista de sus 10 algoritmos favoritos, te encontrarás con que **QUICKSORT** aparece con mucha frecuencia (está en mi propia lista, sin ir más lejos). ¿Cuál es el motivo? Si ya conocemos un algoritmo de ordenación espectacularmente rápido (**MERGESORT**), ¿para qué necesitamos otro?

Desde un punto de vista práctico, **QUICKSORT** iguala y, en muchas ocasiones, supera a **MERGESORT**, lo que motiva que sea el algoritmo de ordenación predeterminado en muchas bibliotecas de programación. La gran victoria de **QUICKSORT** sobre **MERGESORT** la podemos encontrar en que se ejecuta *in situ*: opera sobre el *array* de entrada realizando intercambios de pares de elementos, razón por la cual necesita solo una minúscula cantidad de memoria adicional para los cálculos intermedios. Desde un punto

de vista estético, QUICKSORT es un algoritmo particularmente bello, con un análisis del tiempo de ejecución igualmente atractivo.

5.1.1 Ordenación

El algoritmo QUICKSORT resuelve el problema de ordenar un *array*, mismo problema que ya tratamos en la sección [1.4](#).

Problema: ordenación

Entrada: un *array* de n números, en orden arbitrario.

Salida: un *array* con los mismos números, ordenados de menor a mayor.

Por lo que, si el *array* de entrada es

3	8	2	5	1	4	7	6
---	---	---	---	---	---	---	---

la salida correcta será

1	2	3	4	5	6	7	8
---	---	---	---	---	---	---	---

Al igual que hicimos con MERGESORT, asumiremos, en aras de la simplicidad, que el *array* de entrada está formado por elementos distintos, sin duplicados¹.

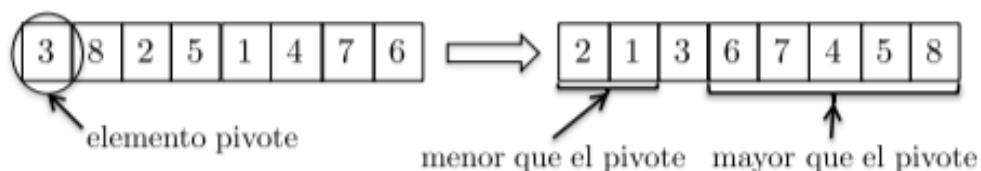
5.1.2 Particiones en torno a un pivote

QUICKSORT se construye alrededor de una subrutina rápida para la “ordenación parcial”, cuya responsabilidad es la de particionar un *array* en torno a un “elemento pivote”.

¹En el poco probable caso de que tuvieras que implementar una versión robusta de QUICKSORT, debes tener en cuenta que gestionar correcta y eficientemente los empates puede ser complicado, más que en MERGESORT. Puedes encontrar un tratamiento más detallado de la cuestión en la sección 2.3 de *Algorithms* (cuarta edición), de Robert Sedgewick y Kevin Wayne (Addison-Wesley, 2011).

Paso 1: elección del elemento pivote. En primer lugar, elegiremos un elemento del *array* para que actúe como *pivote*. La sección 5.3 insistirá hasta la obsesión en cómo realizar correctamente este paso. De momento, vamos a ser un poco ingenuos y elegiremos el primer elemento del *array* (en el caso anterior, el “3”).

Paso 2: preparar el array de entrada en torno al pivote. Dado el elemento pivote p , la siguiente tarea consiste en recolocar los elementos del *array* de forma que todo lo que esté situado antes de p sea menor que p y todo lo que esté situado después de p sea mayor. Siguiendo el ejemplo del *array* anterior, esta sería una forma válida de recolocar los elementos:



Este ejemplo deja claro que los elementos anteriores al pivote no tienen por qué aparecer en su orden relativo correcto (el “1” y el “2” están invertidos) y lo mismo es aplicable al resto de elementos posteriores al pivote. Esta subrutina de partición separa los elementos (distintos al pivote) en dos grupos, uno para los que son menores que el pivote y otro para los que son mayores.

Hay dos hechos clave en relación a esta subrutina de partición.

Es rápida. La subrutina de partición tiene una implementación espectacularmente rápida, pues se ejecuta en tiempo lineal ($O(n)$). Incluso mejor, y fundamental para la aplicación práctica de **QUICKSORT**, la subrutina se puede implementar *in situ*, sin emplear casi ninguna memoria adicional a la ocupada por el propio *array*². La sección 5.2 describe esta implementación en detalle.

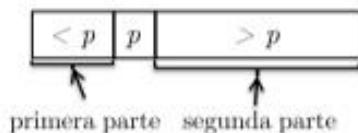
Supone un progreso significativo. La partición de un *array* en torno a un elemento pivote supone un progreso hacia la ordenación del mismo. En primer lugar, el elemento pivote encuentra su posición definitiva, es decir, la que ocupará cuando el *array* quede completamente ordenado (con todos los elementos menores situados antes y todos los mayores después).

²Esto supone un importante contraste con **MERGESORT** (sección 1.4), que copia repetidamente los elementos de un *array* a otro.

En segundo lugar, la partición reduce el problema de ordenación a dos problemas más pequeños: ordenar los elementos menores que el pivote (que, oportunamente, ocupan su propio *subarray*) y también los elementos mayores (que también tienen su propio *subarray*). Después de ordenar recursivamente los elementos de cada uno de los *subarrays*, el algoritmo habrá finalizado³.

5.1.3 Descripción de alto nivel

En la siguiente descripción de alto nivel del algoritmo **QUICKSORT**, nos referimos con “primera parte” y “segunda parte” del *array* a aquellos elementos que son, respectivamente, menores y mayores que el pivote:



QUICKSORT (descripción de alto nivel)

Entrada: *array A* de n enteros distintos.

Condición posterior: los elementos de *A* quedan ordenados de menor a mayor.

```
si  $n \leq 1$  entonces          // caso base ya ordenado
    devolver
elegir un elemento pivote  $p$       // pendiente de implementar
particionar A en torno a  $p$     // pendiente de implementar
ordenar recursivamente la primera parte de A
ordenar recursivamente la segunda parte de A
```

Aunque tanto **MERGESORT** como **QUICKSORT** son algoritmos de divide y vencerás, el orden de las operaciones es diferente. En **MERGESORT**, las llamadas recursivas se ejecutan primero, seguidas del paso de combinación **MERGE**. En **QUICKSORT**, las llamadas recursivas se producen tras la partición y no es necesario en absoluto combinar los resultados⁴.

³Uno de los subproblemas podría estar vacío, en caso de que el pivote resulte ser el elemento mínimo o máximo. Ante esta situación, se puede ignorar la llamada recursiva correspondiente.

⁴QUICKSORT fue inventado por Tony Hoare en 1959, cuando contaba con solo 25 años.

5.1.4 Seguimos avanzando

Todavía nos queda por resolver:

1. (Sección 5.2) ¿Cómo implementamos la subrutina de partición?
2. (Sección 5.3) ¿Cómo elegimos el elemento pivote?
3. (Secciones 5.4 y 5.5) ¿Cuál es el tiempo de ejecución de QUICKSORT?

Otra pregunta es: “¿de verdad estamos seguros de que QUICKSORT siempre ordena correctamente el *array* de entrada?”. Hasta ahora no le he prestado mucha atención a los argumentos de corrección formal, pues los estudiantes, normalmente, suelen tener una fuerte y precisa intuición sobre por qué los algoritmos de divide y vencerás son correctos (no hay que confundirlo con la compresión de los tiempos de ejecución de esos mismos algoritmos que, habitualmente, distan mucho ser obvios). No obstante, si te preocupa este aspecto, la verificación de la corrección de QUICKSORT es bastante sencilla mediante una demostración por inducción⁵.

Hoare realizó numerosas aportaciones fundamentales a los lenguajes de programación y fue galardonado con el Premio Turing de la ACM (el equivalente al Nobel de las ciencias de la computación) en 1980.

⁵Siguiendo la plantilla para demostraciones por inducción que incluimos en el apéndice A, digamos que $P(n)$ representa la afirmación “por cada *array* de entrada de longitud n , QUICKSORT lo ordena correctamente”. El caso base ($n = 1$) carece de interés: un *array* con un solo elemento está ordenado por definición, por lo que QUICKSORT es siempre correcto en un caso así. Para llevar a cabo el paso inductivo, establecemos un entero positivo fijo $n \geq 2$. Podemos asumir la hipótesis inductiva (es decir, $P(k)$ es cierto para todos $k < n$), lo que significa que QUICKSORT ordena correctamente cualquier *array* que contenga menos de n elementos.

Después del paso de partición, el elemento pivote p ocupa la misma posición que tendrá en la versión ordenada del *array* de entrada. Los elementos situados antes de p son exactamente los mismos que estarán antes de p en la versión ordenada del *array* de entrada (aunque, posiblemente, su orden relativo no sea el correcto) y lo mismo ocurrirá con los elementos después de p . Por tanto, las únicas tareas restantes serán la reorganización ordenada de los elementos anteriores a p e, igualmente, de los posteriores. Como ambas llamadas recursivas se producen sobre *subarrays* de longitud máxima $n - 1$ (p estará excluido en cualquier caso), la hipótesis inductiva implica que ambas llamadas ordenan correctamente sus *subarrays*. Esto pone fin al paso inductivo y a la demostración de corrección formal del algoritmo QUICKSORT.

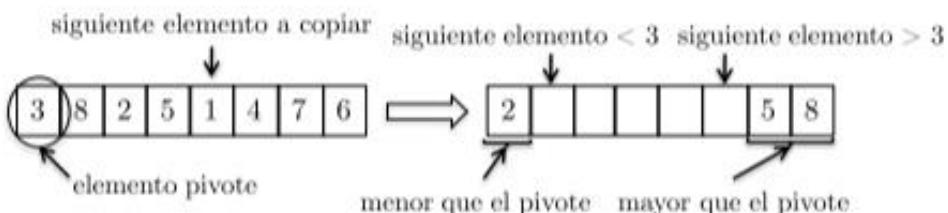
5.2 Partición en torno a un elemento pivote

A continuación conoceremos los detalles sobre cómo particionar un *array* en torno a un elemento pivote p , lo que implica reorganizar dicho *array* para que tenga este aspecto:



5.2.1 La vía rápida

Diseñar una subrutina de partición de tiempo lineal es muy sencillo, si no nos importa utilizar memoria adicional. Una forma de hacerlo consiste en realizar una única exploración sobre el *array* de entrada *A* y copiar sus elementos, con la excepción del pivote, de uno en uno, a un nuevo *array* *B* con la misma longitud, poblando *B* tanto desde el principio (para aquellos elementos menores que *p*) como desde el final (para los que son mayores que *p*). Se puede copiar el elemento pivote a la entrada restante de *B* una vez que se hayan procesado todas las demás. Esta es una imagen de la parte central de este cálculo, para nuestro *array* de entrada de ejemplo:



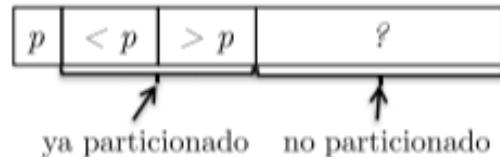
Como esta subrutina solo realiza $O(1)$ operaciones por cada uno de los n elementos del *array* de entrada, su tiempo de ejecución es de $O(n)$.

5.2.2 Implementación *in situ*: el plan de alto nivel

¿Cómo podemos particionar un *array* en torno a un elemento pivote sin consumir prácticamente nada de memoria adicional? Una técnica de alto nivel consistirá en explorar una sola vez el *array*, intercambiando los pares de elementos que resulte necesario, para que el *array* esté correctamente particionado al final de la pasada.

Asumimos que el elemento pivote es el primer elemento del *array*, lo que siempre se puede forzar (en tiempo $O(1)$) intercambiando el elemento pivote con el que esté primero en el *array*, en un paso de procesamiento.

previo. Según vamos explorando y transformando el *array* de entrada, nos aseguraremos de que mantiene la siguiente forma:

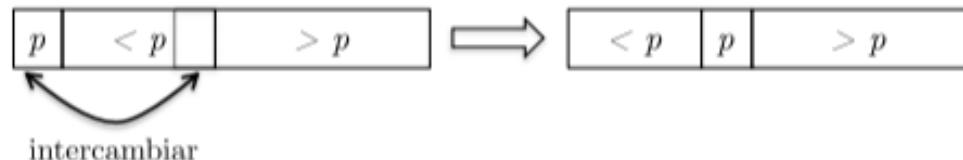


Esto es, la subrutina mantiene la siguiente invariante⁶: el primer elemento es el pivote; a continuación, se sitúan los elementos distintos al pivote que ya han sido procesados, con todos aquellos que sean menores que el pivote precediendo a los que sean mayores; seguidamente, encontramos los elementos distintos al pivote que todavía no han sido procesados, en orden arbitrario.

Si este plan tiene éxito, al finalizar la exploración lineal habremos transformado el *array* para que presente este aspecto:



Para completar la partición, podemos intercambiar el elemento pivote por el último elemento que sea menor que el mismo:



5.2.3 Ejemplo

Ahora, avanzaremos por la subrutina de la partición *in situ* mediante un ejemplo concreto. Puede que parezca extraño utilizar el ejemplo de un programa antes de haber visto su código, pero debes confiar en mí: esta es la forma más sencilla de entender la subrutina.

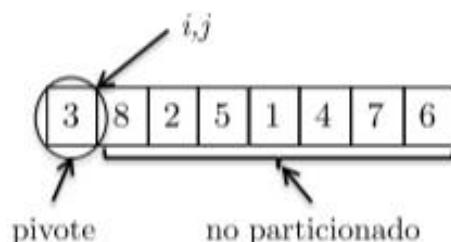
Según nuestro plan de alto nivel, tendremos que mantener un registro de dos límites: el que separa a los elementos distintos al pivote que ya hemos

⁶Una *invariante* de un algoritmo es una propiedad que siempre se mantiene cierta en puntos determinados de su ejecución (como al final de cada iteración del bucle).

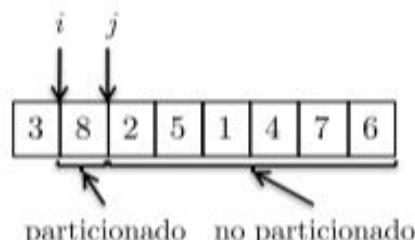
procesado de los que todavía están por procesar y, dentro del primer grupo, el que separa los elementos menores que el pivote de aquellos que son mayores. Utilizaremos, respectivamente, los índices j e i para mantener el registro de estos límites. La invariante buscada se puede replantear así:

Invariante: todos los elementos entre el pivote e i , son menores que el pivote, y todos los elementos entre i y j son mayores que el pivote.

Tanto i como j se inicializan al límite existente entre el elemento pivote y el resto. Por tanto, no habrá elementos entre el pivote y j , y la invariante se mantendrá vacía:

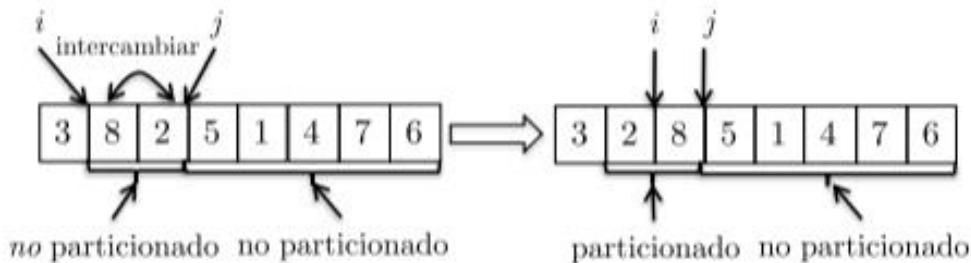


Con cada iteración, la subrutina comprueba un nuevo elemento e incrementa j . Cabe la posibilidad de que haya que realizar trabajo adicional para mantener la invariante. La primera vez que se incrementa j en nuestro ejemplo, obtenemos:

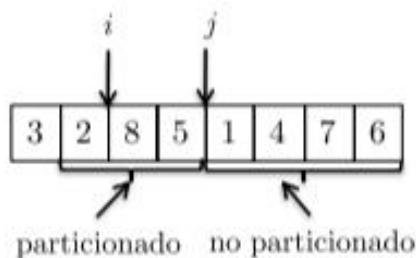


No hay elementos entre el pivote e i , y el único que se encuentra entre i y j (el “8”) es mayor que el pivote, por lo que la invariante se mantiene.

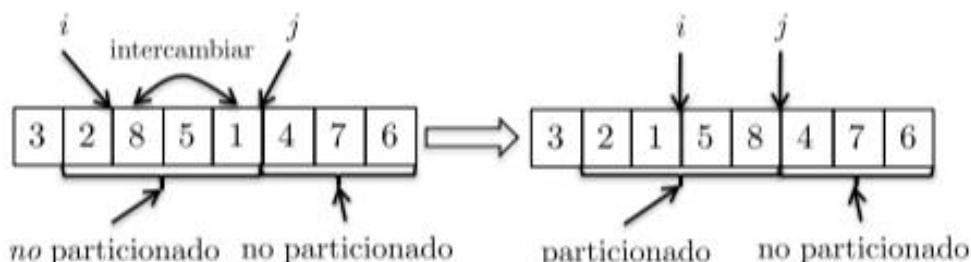
Ahora el guión se vuelve más denso. Después de incrementar j una segunda vez, existe un elemento entre i y j que es menor que el pivote (el “2”), lo que rompe la invariante. Para restaurarla, intercambiamos el “8” por el “2” y, además, incrementamos i , para que quede emparejado entre el “2” y el “8” y, nuevamente, marque el límite entre los elementos procesados menores y mayores que el pivote:



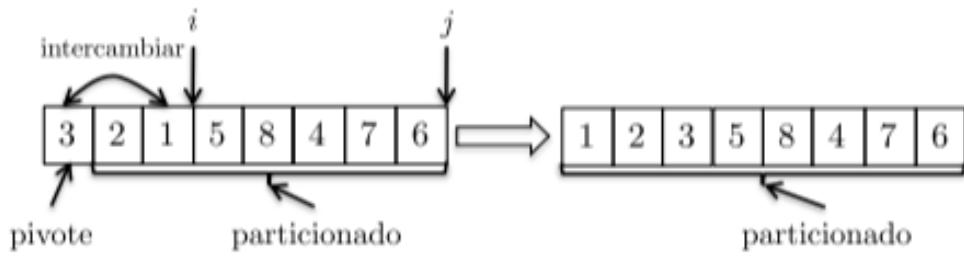
La tercera iteración es similar a la primera. Procesamos el siguiente elemento (el “5”) e incrementamos j . Como el nuevo elemento es mayor que el pivote, la invariante se mantiene y no tenemos que hacer nada más:



La cuarta iteración es similar a la segunda. El incremento de j hace que se deslice un elemento menor que el pivote (el “1”) entre i y j , lo que rompe la invariante. Pero restaurarla es sumamente sencillo: basta con intercambiar el “1” con el primer elemento mayor que el pivote (el “8”) e incrementar i para reflejar el nuevo límite entre los elementos procesados mayores y menores que el pivote:



Las tres últimas iteraciones procesan elementos que son mayores que el pivote, por lo que no es necesario hacer nada más que incrementar j . Una vez procesados todos los elementos y con todo lo que viene después del pivote ya particionado, terminamos con un intercambio final del pivote con el último elemento menor que el mismo:



Como se pedía, en el *array* definitivo todos los elementos menores que el *pivote* están situados antes del mismo, mientras que todos los elementos mayores lo están después. El hecho de que el “1” y el “2” aparezcan en su orden correcto, es mera coincidencia. Es evidente que los elementos posteriores al *pivote* no están ordenados.

5.2.4 Pseudocódigo de PARTITION

El pseudocódigo para la subrutina PARTITION es, exactamente, el que cabría esperar una vez visto el ejemplo⁷:

PARTITION

Entrada: *array A* de n enteros distintos, con extremos izquierdo y derecho $\ell, r \in \{1, 2, \dots, n\}$ con $\ell \leq r$.

Condición posterior: los elementos del *subarray*

$A[\ell], A[\ell + 1], \dots, A[r]$ quedan *particionados* en torno a $A[\ell]$.

Salida: posición final del elemento *pivote*.

```

 $p := A[\ell]$ 
 $i := \ell + 1$ 
para  $j := \ell + 1$  hasta  $r$  hacer
    si  $A[j] < p$  entonces          // si  $A[j] > p$  no hacer nada
        intercambiar  $A[j]$  y  $A[i]$ 
         $i := i + 1$                   // restaura la invariante
    intercambiar  $A[\ell]$  y  $A[i - 1]$   // situar pivote correctamente
    devolver  $i - 1$                 // posición final del pivote

```

⁷Si consultas otros libros de texto o páginas web, encontrarás variantes de esta subrutina con detalles diferentes (existe, incluso, una versión interpretada por bailarines populares húngaros: <https://www.youtube.com/watch?v=ywWBBy6J5gz8>). Estas variantes serían igualmente válidas para nuestro propósito.

La subrutina `PARTITION` toma como entrada un *array* A , pero solo opera sobre el *subarray* con elementos $A[\ell], \dots, A[r]$, donde ℓ y r son parámetros dados. Desde una visión más general, cada llamada recursiva a `QUICKSORT` será responsable de un *subarray* específico del *array* de entrada original, y los parámetros ℓ y r especifican sus extremos correspondientes.

Al igual que en el ejemplo, el índice j mantiene el registro de los elementos que ya han sido procesados, mientras que i hace lo propio con el límite entre los elementos ya procesados que son menores y mayores que el pivote (siendo $A[i]$ el elemento procesado más a la izquierda mayor que el pivote, si es que está presente). Cada iteración del bucle procesa un nuevo elemento. Al igual que en el ejemplo, cuando el nuevo elemento $A[j]$ sea mayor que el pivote, la invariante se mantiene de forma automática y no hay nada más que hacer. En caso contrario, la subrutina restaura la invariante intercambiando $A[j]$, el nuevo elemento, con $A[i]$, el elemento más a la izquierda mayor que el pivote, e incrementando i para actualizar el límite entre los elementos menores y mayores que el pivote^{8,9}. El último paso, como ya anunciábamos antes, intercambia el elemento pivote con el elemento más a la derecha que sea menor que el mismo, para colocarlo en su posición correcta. La subrutina `PARTITION` finaliza informando de esta posición a la invocación de `QUICKSORT` que hizo la llamada.

Esta implementación es espectacularmente rápida. Únicamente realiza un número constante de operaciones por cada elemento $A[\ell], A[\ell+1], \dots, A[r]$ del *subarray* relevante, por lo que se ejecuta en tiempo lineal en relación a la longitud de este *subarray*. Es importante recordar que la subrutina opera *in situ* sobre el *subarray*, sin consumir memoria adicional más allá de un espacio $O(1)$, necesario para mantener el registro de variables como i y j .

⁸No es necesario realizar ningún intercambio si no se han encontrado todavía elementos mayores que el pivote: el *subarray* de elementos procesados estará particionado de forma trivial. Pero ese intercambio adicional sería inocuo (como deberás comprobar), por lo que mantendremos la sencillez de nuestro pseudocódigo.

⁹¿Por qué este intercambio e incremento restauran siempre la invariante? La invariante se mantenía antes del incremento más reciente de j (por inducción, si quieras verlo desde un punto de vista formal). Esto significa que todos los elementos $A[\ell+1], \dots, A[i-1]$ son menores que el pivote y que todos los elementos $A[i], \dots, A[j-1]$ son mayores. El único problema es que $A[j]$ es menor que el pivote. Después de intercambiar $A[i]$ y $A[j]$, los elementos $A[\ell+1], \dots, A[i]$ y $A[i+1], \dots, A[j]$ son menores y mayores que el pivote, respectivamente. Después de incrementar i , $A[\ell+1], \dots, A[i-1]$ y $A[i], \dots, A[j]$ son menores y mayores que el pivote, respectivamente, lo que restaura la invariante.

5.2.5 Pseudocódigo de QUICKSORT

Ahora tenemos una descripción completa del algoritmo QUICKSORT, a falta de la subrutina CHOOSEPIVOT, que selecciona el elemento pivote.

QUICKSORT

Entrada: array A de n enteros distintos, con extremos izquierdo y derecho $\ell, r \in \{1, 2, \dots, n\}$.

Condición posterior: los elementos del *subarray* $A[\ell], A[\ell + 1], \dots, A[r]$ quedan ordenados de menor a mayor.

```
si  $\ell \geq r$  entonces           // subarray de 0 o 1 elementos
    devolver
     $i := \text{CHOOSEPIVOT}(A, \ell, r)$       // pendiente de implementar
    intercambiar  $A[\ell]$  y  $A[i]$             // procesar primero el pivote
     $j := \text{PARTITION}(A, \ell, r)$         //  $j =$  nueva posición del pivote
     $\text{QUICKSORT}(A, \ell, j - 1)$           // recursión en la primera parte
     $\text{QUICKSORT}(A, j + 1, r)$           // recursión en la segunda parte
```

La llamada a la función $\text{QUICKSORT}(A, 1, n)$ se puede utilizar para ordenar un *array* A de n elementos¹⁰.

5.3 La importancia de los buenos pivotes

¿Es QUICKSORT un algoritmo rápido? El listón está alto: los algoritmos de ordenación sencillos, como INSERTIONSORT, funcionan en tiempo cuadrático ($O(n^2)$), y ya conocemos un algoritmo de ordenación (MERGESORT) que se ejecuta en tiempo $O(n \log n)$. La respuesta a esta pregunta depende de cómo implementemos la subrutina CHOOSEPIVOT, que selecciona un elemento de un *subarray* designado. Para que QUICKSORT sea rápido, es importante elegir “buenos” elementos pivote, que resulten en dos subproblemas de tamaños similares.

¹⁰El *array* A siempre se pasa por referencia, lo que significa que todas las llamadas a la función operan directamente sobre la copia original del *array* de entrada.

5.3.1 Implementación ingenua de CHOOSEPIVOT

En la introducción a QUICKSORT mencionamos una implementación ingenua, que elige siempre el primer elemento.

CHOOSEPIVOT (implementación ingenua)

Entrada: array A de n enteros distintos, con extremos izquierdo y derecho $\ell, r \in \{1, 2, \dots, n\}$.

Salida: un índice $i \in \{\ell, \ell + 1, \dots, r\}$.

devolver ℓ

¿Será esta implementación nativa lo suficientemente buena?

Cuestionario 5.1

¿Cuál es el tiempo de ejecución del algoritmo QUICKSORT, utilizando la implementación ingenua de CHOOSEPIVOT, cuando el array de entrada de n elementos ya está ordenado?

- a) $\Theta(n)$
- b) $\Theta(n \log n)$
- c) $\Theta(n^2)$
- d) $\Theta(n^3)$

Solución y aclaraciones en la sección 5.3.3

5.3.2 Implementación excesiva de CHOOSEPIVOT

El cuestionario 5.1 plantea una situación del peor caso que se puede producir en QUICKSORT, donde se elimina un solo elemento por cada llamada recursiva. ¿Cuál sería el mejor caso? La división con un equilibrio perfecto se logra utilizando el elemento que resulte ser la *mediana* del array, es decir, aquél elemento que tenga, a su vez, el mismo número de elementos

a su izquierda y derecha¹¹. Así, si queremos buscar intensamente nuestro elemento pivot, podemos calcular la mediana del *subarray* dado.

CHOOSEPIVOT (implementación excesiva)

Entrada: array A de n enteros distintos, con extremos izquierdo y derecho $\ell, r \in \{1, 2, \dots, n\}$.

Salida: un índice $i \in \{\ell, \ell + 1, \dots, r\}$.

devolver la posición de la mediana de $\{A[\ell], \dots, A[r]\}$

En el próximo capítulo veremos cómo se puede calcular la mediana de un *array* en tiempo lineal, en relación a la longitud del mismo. De momento, de cara al siguiente cuestionario, creeremos ciegamente en este hecho¹². ¿Tiene alguna recompensa el trabajo intenso para calcular el elemento pivot ideal?

Cuestionario 5.2

¿Cuál es el tiempo de ejecución del algoritmo QUICKSORT, utilizando la implementación excesiva de CHOOSEPIVOT, sobre un *array* de entrada arbitrario de n elementos? Asume que CHOOSEPIVOT se ejecuta en tiempo $\Theta(n)$.

- a) Faltan datos para poder responder
- b) $\Theta(n)$
- c) $\Theta(n \log n)$
- d) $\Theta(n^2)$

Solución y aclaraciones en la sección 5.3.3

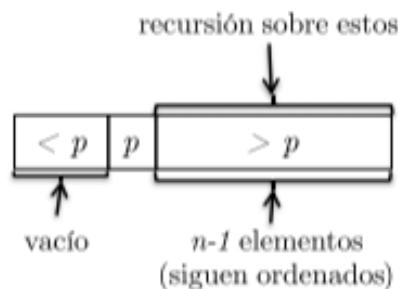
¹¹Por ejemplo, la mediana de un *array* que contenga $\{1, 2, 3, \dots, 9\}$, es el 5. En el caso de un *array* de longitud par, existen dos elecciones de mediana válidas y cualquiera de ellas sirve a nuestro propósito. Por tanto, en un *array* formado por $\{1, 2, 3, \dots, 10\}$, la mediana corresponderá tanto al 5 como al 6.

¹²De hecho, ya conoces un algoritmo de tiempo $O(n \log n)$ para calcular la mediana de un *array* (pista: ordenación).

5.3.3 Soluciones a los cuestionarios 5.1–5.2

Solución al cuestionario 5.1

Respuesta correcta: (c). La combinación de pivotes elegidos de forma ingenua y un *array* de entrada ya ordenado, provoca que **QUICKSORT** se ejecute en tiempo $\Theta(n^2)$, lo que es mucho peor que **MERGESORT** y tampoco mejora a los algoritmos sencillos como **INSERTIONSORT**. ¿Qué ha fallado? La subrutina **PARTITION** de la llamada más externa de **QUICKSORT**, con el primer elemento (más pequeño) como pivote, no hace nada: recorre el *array* y, como solo encuentra elementos mayores que el pivote, nunca intercambia ningún par de elementos. Una vez que esta llamada a **PARTITION** finaliza, la imagen es:



En la llamada recursiva que no está vacía, el patrón se repite: el *subarray* ya está ordenado, el primer elemento (más pequeño) es elegido como pivote y tendremos una llamada recursiva vacía y otra llamada recursiva que recibe un *subarray* de $n - 2$ elementos. Y, así, sucesivamente.

Al final, la subrutina **PARTITION** habrá recibido llamadas sobre *subarrays* de longitudes $n, n-1, n-2, \dots, 2$. Como el trabajo realizado en una llamada a **PARTITION** es proporcional a la longitud del *subarray* de esa llamada, la cantidad de trabajo total realizada por **QUICKSORT** es, en este caso, proporcional a

$$\underbrace{n + (n - 1) + (n - 2) + \dots + 1}_{=\Theta(n^2)}$$

y, en consecuencia, es cuadrática en relación a la longitud n de la entrada¹³.

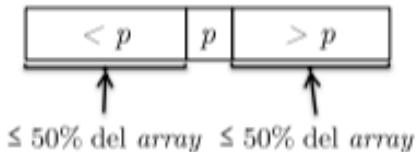
¹³Una forma rápida de ver que $n + (n - 1) + (n - 2) + \dots + 1 = \Theta(n^2)$, es fijarse en el hecho de que alcanza un máximo de n^2 (cada uno de los n términos es, como mucho, n) y, como mínimo, será $n^2/4$ (cada uno de los primeros $n/2$ términos es, como mínimo, $n/2$).

Solución al cuestionario 5.2

Respuesta correcta: (c). En esta situación de mejor caso, QUICKSORT se ejecuta en tiempo $\Theta(n \log n)$. El motivo es que su tiempo de ejecución viene determinado por la misma recurrencia que determina el tiempo de ejecución de MERGESORT. Es decir, si $T(n)$ indica el tiempo de ejecución de esta implementación de QUICKSORT sobre arrays de longitud n , entonces

$$T(n) = \underbrace{2 \cdot T\left(\frac{n}{2}\right)}_{\text{porque pivote} = \text{mediana}} + \underbrace{\Theta(n)}_{\text{CHOOSEPIVOT y PARTITION}}.$$

El principal trabajo realizado por una llamada a QUICKSORT fuera de sus llamadas recursivas, se produce en las subrutinas CHOOSEPIVOT y PARTITION. Asumimos que la primera es $\Theta(n)$, y la sección 5.2 demuestra que la segunda también es $\Theta(n)$. Como estamos utilizando la mediana como elemento pivote, obtenemos una división perfecta del array de entrada y cada llamada recursiva obtiene un array con un máximo de $\frac{n}{2}$ elementos:



Aplicar el método maestro (teorema 4.1) con $a = b = 2$ y $d = 1$ resulta, por tanto, en $T(n) = \Theta(n \log n)$.¹⁴

5.4 QUICKSORT aleatorizado

Elegir como pivote el primer elemento de un subarray solo emplea un tiempo $O(1)$, pero puede provocar que QUICKSORT se ejecute en tiempo $\Theta(n^2)$. Elegir la mediana como pivote garantiza una ejecución global en $\Theta(n \log n)$, pero consume mucho más tiempo (si es que sigue siendo lineal). ¿No podríamos tener lo mejor de los dos mundos? ¿Existe alguna forma sencilla y ligera de elegir un elemento pivote que nos lleve a una división más o menos equilibrada del array? La respuesta es sí, y la clave la encontramos en el uso de la *aleatorización*.

¹⁴Técnicamente, en esta ocasión estamos utilizando la variante del método maestro que funciona con notación Θ , en vez de con notación O , pero, por lo demás, el teorema 4.1 se aplica exactamente igual.

5.4.1 Implementación aleatorizada de CHOOSEPIVOT

Un *algoritmo aleatorizado* es aquel que “lanza monedas al aire” según avanza, y puede tomar decisiones en base a los resultados de esos lanzamientos de monedas. Si ejecutas un algoritmo aleatorizado, una y otra vez, sobre la misma entrada, observarás distintos comportamientos en las diferentes ejecuciones. Todos los lenguajes de programación importantes cuentan con bibliotecas que facilitan la obtención de números aleatorios y la aleatorización es una herramienta que debería estar muy presente para todo diseñador de algoritmos serio.

¿Por qué habría de interesarnos incluir comportamientos aleatorios en un algoritmo? ¿No es un algoritmo el elemento más determinista que pueda existir? Pues resulta que existen docenas de problemas de computación para los que la aleatorización es más rápida, más eficaz y más fácil de programar que si utilizásemos una versión determinista¹⁵.

El método más sencillo para incorporar la aleatorización a **QUICKSORT**, que resulta ser extraordinariamente eficaz, consiste en elegir siempre elementos pivote *de forma totalmente aleatoria*.

CHOOSEPIVOT (implementación aleatorizada)

Entrada: array A de n enteros distintos, con extremos izquierdo y derecho $\ell, r \in \{1, 2, \dots, n\}$.

Salida: un índice $i \in \{\ell, \ell + 1, \dots, r\}$.

devolver un elemento de $\{\ell, \ell + 1, \dots, r\}$, elegido de forma totalmente aleatoria

Por ejemplo, si $\ell = 41$ y $r = 50$, entonces cada uno de los 10 elementos $A[41], \dots, A[50]$ tiene un 10% de posibilidades de ser elegido pivote¹⁶.

¹⁵Los científicos de la computación tardaron en darse cuenta de esto, y no fue hasta mediados de la década de 1970 cuando se abrieron las compuertas gracias a la aparición de algoritmos aleatorizados rápidos para determinar si un entero es un número primo.

¹⁶Una forma igualmente efectiva de aplicar la aleatorización consiste en mezclar aleatoriamente el array de entrada, durante un paso de procesamiento previo, y, a continuación, ejecutar la implementación ingenua de **QUICKSORT**.

5.4.2 Tiempo de ejecución del QUICKSORT aleatorizado

El tiempo de ejecución del QUICKSORT aleatorizado, cuyos elementos pivotes se eligen al azar, no es siempre el mismo. Claro que existe alguna posibilidad, aunque remota, de que el algoritmo seleccione siempre como pivote el elemento mínimo del *subarray* restante, lo que llevaría al tiempo de ejecución de $\Theta(n^2)$ visto en el cuestionario 5.2. Así, el tiempo de ejecución del algoritmo puede fluctuar entre $\Theta(n \log n)$ y $\Theta(n^2)$. ¿Qué caso se dará con más frecuencia, el mejor o el peor? Sorprendentemente, el rendimiento de QUICKSORT se acerca casi siempre a su desempeño en el mejor caso.

Teorema 5.1 (Tiempo de ejecución del QUICKSORT aleatorizado) *Por cada array de entrada de longitud $n \geq 1$, el tiempo de ejecución medio del QUICKSORT aleatorizado es $O(n \log n)$.*

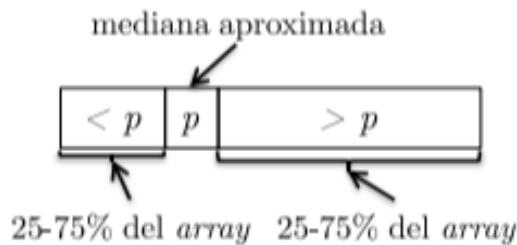
La palabra “medio” del teorema hace referencia a la aleatoriedad del propio algoritmo QUICKSORT. El teorema 5.1 *no asume* que el *array* de entrada sea aleatorio. El QUICKSORT aleatorizado es un algoritmo de uso general (comparar con la sección 1.6.1): es irrelevante cuál sea el *array* de entrada, si ejecutas el algoritmo una y otra vez, el tiempo de ejecución medio será de $O(n \log n)$, lo que resulta suficiente como para obtener la consideración de operación básica de coste cero. En teoría, el QUICKSORT aleatorizado podría llegar a ejecutarse en tiempo $\Theta(n^2)$ pero, en la práctica, nos encontraremos casi siempre con un tiempo de $O(n \log n)$. Dos particularidades más: la constante oculta en la notación Big-O en el teorema 5.1 es razonablemente pequeña (como lo era en MERGESORT) y el algoritmo no pierde tiempo gestionando memoria adicional (a diferencia de MERGESORT).

5.4.3 Intuición: ¿por qué son buenos los pivotes aleatorios?

La única oportunidad real de entender por qué QUICKSORT es tan rápido, pasa por estudiar la demostración del teorema 5.1 que se explica en la sección 5.5. En preparación para esa demostración y, también, como premio de consolación para el lector que no pueda dedicar el tiempo necesario a asimilar la sección 5.5, vamos a elaborar un punto de vista intuitivo sobre por qué el teorema 5.1 debe ser cierto.

El primer aspecto es que, para lograr un tiempo de ejecución de $O(n \log n)$, como el del mejor caso del cuestionario 5.2, resulta exagerado utilizar la mediana como elemento pivote. Supongamos que, en su lugar, utilizamos

una “mediana aproximada”, refiriéndonos con ello a un elemento que resulte en una división 25%-75% o mejor. Dicho de otra forma, hablamos de un elemento que sea mayor que, al menos, el 25% del resto de elementos y, además, menor que el 25% de los mismos. La imagen de una partición en torno a un pivote de este tipo queda así:



Si cada llamada recursiva selecciona un elemento pivote que sea, en este sentido, una mediana aproximada, el tiempo de ejecución de QUICKSORT seguirá siendo de $O(n \log n)$. No podemos deducir este hecho directamente a partir del método maestro (teorema 4.1), porque utilizar un valor que no sea estrictamente la mediana resultará en subproblemas de tamaños desiguales. Pero no resulta difícil generalizar el análisis de MERGESORT (sección 1.5) de forma que también sea aplicable a este caso¹⁷.

El segundo aspecto es que, mientras que necesitas una suerte extraordinaria para elegir la mediana en el QUICKSORT aleatorizado (solo una posibilidad entre n), necesitas mucha menos fortuna para dar con una mediana aproximada. Por ejemplo, pensemos en un *array* de longitud 100 que contenga los elementos $\{1, 2, 3, \dots, 100\}$, en orden arbitrario. Cualquier número entre 26 y 75, ambos inclusive, es una mediana aproximada, con al menos 25 elementos menores y al menos 25 elementos mayores. Esto supone el 50% de los números del *array*. Por lo tanto, QUICKSORT tiene una posibilidad 50-50 de elegir aleatoriamente una mediana aproximada, com-

¹⁷Dibuja el árbol de recursión del algoritmo. Siempre que QUICKSORT se llama a sí misma, de forma recursiva, sobre dos subproblemas, dichos subproblemas implican a elementos distintos (los que son menores y los que son mayores que el pivote). Esto significa que, para cada nivel de recursión j , no habrá superposiciones entre los *subarrays* de los subproblemas de ese nivel j , por lo que la suma de las longitudes de los *subarrays* de los subproblemas del nivel j será, como mucho, de n . El trabajo total realizado en este nivel (mediante las llamadas a PARTITION) es lineal en relación a la suma de las longitudes de los *subarrays*. Por tanto, al igual que en MERGESORT, el algoritmo realiza un trabajo $O(n)$ por cada nivel de recursión. ¿Cuántos niveles hay? Con elementos pivote que sean medianas aproximadas pasarán, a una misma llamada recursiva, un máximo del 75% de los elementos, por lo que el tamaño del subproblema quedará dividido por, al menos, $4/3$ en cada nivel. Esto significa que hay un máximo de $\log_{4/3} n = O(\log n)$ niveles en el árbol de recursión y, en consecuencia, el trabajo total realizado es $O(n \log n)$.

parable a las opciones que tendríamos lanzando una moneda al aire. Esto significa que nuestra expectativa está en que, aproximadamente, la mitad de las llamadas a `QUICKSORT` utilicen medianas aproximadas y podremos esperar que el análisis del tiempo de ejecución de $O(n \log n)$, mencionado en el párrafo anterior, se mantenga, quizá con el doble de niveles.

No nos equivoquemos: esto no se puede considerar una demostración formal, más bien es un argumento heurístico de que el teorema 5.1 podría considerarse razonablemente cierto. Si yo estuviese en tu lugar, dada la figura nuclear que supone `QUICKSORT` en el diseño y análisis de algoritmos, exigiría una prueba irrefutable de que el teorema 5.1 es absolutamente correcto.

*5.5 Análisis del `QUICKSORT` aleatorizado

El `QUICKSORT` aleatorizado parece ser una gran idea pero, ¿cómo podemos estar seguros de que funcionará correctamente? Planteado de forma más general, cuando descubres un nuevo algoritmo, ¿cómo puedes saber si es brillante o es una basura? Una técnica útil, aunque *ad hoc*, consiste en programarlo y ejecutarlo contra una buena cantidad de entradas distintas. Otra técnica está en desarrollar una intuición de por qué el algoritmo debería de funcionar bien, como hemos hecho en la sección 5.4.3 con el `QUICKSORT` aleatorizado. Pero la comprensión profunda de qué hace que un algoritmo sea bueno o malo, necesita normalmente un análisis matemático. Esta sección te aportará esa comprensión de por qué `QUICKSORT` es tan rápido.

Asumimos un conocimiento de los conceptos de probabilidad discreta que se mencionan en el apéndice B: espacios de muestreo, eventos, variables aleatorias, expectativa y linealidad de la expectativa.

5.5.1 Preliminares

El teorema 5.1 afirma que, para cada *array* de entrada de longitud $n \geq 1$, el tiempo medio de ejecución del `QUICKSORT` aleatorizado (en el que los elementos pivote se eligen de forma aleatoria y uniforme) es de $O(n \log n)$. Comencemos por traducir esta afirmación a una declaración formal en términos de probabilidad discreta.

Establecemos, para el resto del análisis, un *array* de entrada arbitrario A de longitud n . Recordemos que un espacio de muestreo es el conjunto de

todos los resultados posibles de un proceso aleatorio. En el `QUICKSORT` aleatorizado, toda la aleatoriedad se encuentra en las elecciones de los elementos pivote en las diferentes llamadas recursivas. Por tanto, tomaremos el espacio de muestreo Ω como el conjunto de todos los posibles resultados de las elecciones aleatorias de `QUICKSORT` (es decir, todas las sucesiones de pivotes).

Recordemos que una variable aleatoria es una medición numérica del resultado de un proceso aleatorio: una función con valor real definida sobre Ω . La variable aleatoria que nos importa es el número RT de operaciones básicas (es decir, líneas de código) realizadas por el `QUICKSORT` aleatorizado. Esta variable aleatoria está bien definida, porque, siempre que las elecciones del pivote estén predeterminadas (es decir, $\omega \in \Omega$ es fijo), `QUICKSORT` tendrá un tiempo de ejecución estable de $RT(\omega)$. Al recorrer todas las posibles sucesiones ω del pivote, $RT(\omega)$ tendrá un valor entre $\Theta(n \log n)$ y $\Theta(n^2)$ (ver la sección 5.3).

Nos sirve el análisis de una variable aleatoria más sencilla, que solo tenga en cuenta las comparaciones e ignore los otros tipos de operaciones básicas realizadas. Digamos que C representa a la variable aleatoria igual al número de comparaciones entre pares de elementos de entrada realizadas por `QUICKSORT`, con una sucesión dada de elecciones de pivotes. Si volvemos a consultar el pseudocódigo, veremos que estas comparaciones se producen en un lugar concreto: la línea “si $A[j] < p$ ” de la subrutina `PARTITION` (sección 5.2.4), que compara el elemento pivote actual con algún otro elemento del *subarray* de entrada.

El siguiente lema demuestra que las comparaciones determinan el tiempo de ejecución global de `QUICKSORT`, lo que significa que la última es mayor que la primera solo por un factor constante. Esto implica que, para demostrar un límite superior de $O(n \log n)$ en el tiempo de ejecución esperado de `QUICKSORT`, solo necesitamos demostrar un límite superior de $O(n \log n)$ sobre el número esperado de comparaciones realizadas.

Lema 5.2 (Las comparaciones determinan el trabajo de `QUICKSORT`)
Existe una constante $a > 0$ tal que, para todo array A de entrada, de longitud mínima 2, y cada sucesión de pivotes ω ,

$$RT(\omega) \leq a \cdot C(\omega).$$

Incluimos la demostración para los escépticos, pero puedes ignorarla si piensas que el lema 5.2 es obvio por intuición.

Demostración del lema 5.2: En primer lugar, en cada llamada a PARTITION, el elemento pivote se compara, exactamente, una vez con cada uno de los otros elementos del subarray dado. Por tanto, el número de comparaciones de la llamada es lineal en relación a la longitud del subarray y, según el análisis del pseudocódigo de la sección 5.2.4, el número total de operaciones es, como mucho, un factor constante del mismo. Según la inspección del pseudocódigo de la sección 5.2.5, el Quicksort aleatorizado realiza solamente un número constante de operaciones en cada llamada recursiva exterior a la subrutina PARTITION¹⁸. En todo Quicksort hay un máximo de n llamadas recursivas, pues cada elemento del array puede ser elegido una sola vez como pivote antes de quedar excluido de las llamadas recursivas futuras, por lo que el trabajo total realizado fuera de las llamadas a PARTITION es de $O(n)$. Al acumular todas las llamadas recursivas, el número total de operaciones $RT(\omega)$ será, como mucho, un factor constante del número de comparaciones $C(\omega)$, más $O(n)$. Como $C(\omega)$ siempre es, al menos, proporcional a n , el trabajo $O(n)$ adicional se puede absorber en el factor constante a de la declaración del lema, lo que completa la demostración. *QED*

El resto de esta sección se concentra en delimitar el número esperado de comparaciones.

Teorema 5.3 (Comparaciones en el Quicksort aleatorizado) *Por cada array de entrada de longitud $n \geq 1$, el número esperado de comparaciones entre elementos de ese array de entrada tiene un máximo, en el Quicksort aleatorizado, de $2(n - 1) \ln n = O(n \log n)$.*

Según el lema 5.2, el teorema 5.3 implica al teorema 5.1, con un factor constante diferente y oculto en la notación Big-O.

5.5.2 Un esquema de descomposición

El método maestro (teorema 4.1) ha resuelto el tiempo de ejecución de todos los algoritmos de divide y vencerás vistos hasta ahora, pero existen dos razones por las que no es de aplicación al Quicksort aleatorizado. En primer lugar, el tiempo de ejecución del algoritmo corresponde a una recurrencia aleatoria o a un árbol de recursión aleatorio, y el método maestro

¹⁸Esta afirmación asume que elegir un elemento pivote aleatorio supone una operación básica. La demostración sigue siendo válida incluso aunque la elección del pivote aleatorio consuma $\Theta(\log n)$ operaciones básicas (como deberías comprobar), lo que cubre las implementaciones más habituales de generadores de números aleatorios.

funciona solo con recurrencias deterministas. En segundo lugar, los dos subproblemas que se resuelven recursivamente (elementos menores y mayores que el pivote) no suelen tener el mismo tamaño. Necesitamos otra idea¹⁹.

Para demostrar el teorema 5.3, seguiremos un esquema de descomposición que resulte útil para analizar la expectativa de variables aleatorias complejas. El primer paso consiste en identificar la variable aleatoria Y (posiblemente compleja) de la que debemos ocuparnos. Para nosotros, dicha variable se corresponde con el número C de comparaciones entre los elementos del *array* de entrada, realizadas por el QUICKSORT aleatorizado, como en el teorema 5.3. El segundo paso es expresar Y como la suma de variables aleatorias más sencillas, idealmente las variables binarias aleatorias (es decir, 0-1) X_1, \dots, X_m ,

$$Y = \sum_{\ell=1}^m X_\ell.$$

Ahora dependemos de la linealidad de la expectativa, que afirma que la expectativa de la suma de variables aleatorias es igual a la suma de sus propias expectativas (teorema B.1). El tercer paso del esquema utiliza esta propiedad para reducir el cálculo de la expectativa de Y al de las variables aleatorias sencillas:

$$\mathbb{E}[Y] = \mathbb{E}\left[\sum_{\ell=1}^m X_\ell\right] = \sum_{\ell=1}^m \mathbb{E}[X_\ell].$$

Cuando los X_ℓ son variables binarias aleatorias, sus expectativas son particularmente fáciles de calcular, mediante la definición (B.1):

$$\mathbb{E}[X_\ell] = \underbrace{0 \cdot \Pr[X_\ell = 0]}_{=0} + 1 \cdot \Pr[X_\ell = 1] = \Pr[X_\ell = 1].$$

El último paso calcula las expectativas de las variables aleatorias sencillas y acumula los resultados²⁰.

¹⁹Existen generalizaciones del método maestro que abordan estas dos cuestiones, pero son complejas y quedan fuera del ámbito de este libro.

²⁰El análisis del balanceo de carga aleatorizado de la sección B.6 es un ejemplo sencillo de este esquema en acción.

Un esquema de descomposición

1. Identificar la variable aleatoria Y que te interesa.
2. Expresar Y como una suma de las variables binarias (es decir, 0-1) aleatorias X_1, \dots, X_m :

$$Y = \sum_{\ell=1}^m X_\ell.$$

3. Aplicar la linealidad de la expectativa:

$$\mathbb{E}[Y] = \sum_{\ell=1}^m \Pr[X_\ell = 1].$$

4. Calcular cada una de las $\Pr[X_\ell = 1]$ y sumar los resultados para obtener $\mathbb{E}[Y]$.

5.5.3 Aplicación del esquema

Para aplicar el esquema de descomposición al análisis del `QUICKSORT` aleatorizado, necesitamos descomponer la variable C que verdaderamente nos importa en variables aleatorias más sencillas (idealmente con valores 0-1). La idea clave consiste en fragmentar el número total de comparaciones según los pares del *array* de entrada que vayan a ser comparados.

Para ser más precisos, digamos que z_i indica el i -ésimo elemento más pequeño del *array* de entrada, también conocido como el *i-ésimo estadístico de orden*. Por ejemplo, en el *array*

6	8	9	2
---	---	---	---

z_1 se refiere al “2”, z_2 al “6”, z_3 al “8” y z_4 al “9”. Vemos que z_i no hace referencia al elemento que ocupa posición i -ésima del *array* (no ordenado) de entrada, sino al elemento que ocupa esa misma posición en la versión ordenada del *array* de entrada.

Por cada par de índices del *array* $i, j \in \{1, 2, \dots, n\}$, con $i < j$, definimos una variable aleatoria X_{ij} de la siguiente manera:

por cada elección fija de pivotes ω , $X_{ij}(\omega)$ es el número de veces que los elementos z_i y z_j resultan comparados en QUICK-SORT cuando los pivotes vienen especificados por ω .

Por ejemplo, en el *array* de entrada anterior, $X_{1,3}$ es el número de veces que el algoritmo QUICKSORT compara el “2” con el “8”. No nos preocupan los X_{ij} *per se*, salvo por el hecho de que su suma resulta en la variable aleatoria C , que sí nos interesa.

El sentido de esta definición es el de implementar el segundo paso del esquema de descomposición. Como cada comparación implica, exactamente, un par de elementos del *array* de entrada,

$$C(\omega) = \sum_{i=1}^{n-1} \sum_{j=i+1}^n X_{ij}(\omega)$$

para todos $\omega \in \Omega$. El doble sumatorio del lado derecho, que tan buen aspecto presenta, solo indica la iteración sobre todos los pares (i,j) con $i < j$, y todo lo que dice la ecuación es que los X_{ij} suponen todas las comparaciones realizadas por el algoritmo QUICKSORT.

Cuestionario 5.3

Establece dos elementos distintos del *array* de entrada, digamos que z_i y z_j . ¿Cuántas veces podrían compararse z_i y z_j entre ellos durante la ejecución de QUICKSORT?

- a) exactamente una
- b) 0 o 1 veces
- c) 0, 1 o 2 veces
- d) cualquier número entre 0 y $n - 1$ es posible

Solución y aclaraciones en la sección 5.5.6

La solución al cuestionario 5.3 muestra que todos los X_{ij} son valores binarios aleatorios. Por tanto, podemos aplicar el tercer paso de nuestro esquema

de descomposición para obtener

$$E[C] = \sum_{i=1}^{n-1} \sum_{j=i+1}^n E[X_{ij}] = \sum_{i=1}^{n-1} \sum_{j=i+1}^n \Pr[X_{ij} = 1]. \quad (5.1)$$

Para calcular lo que de verdad nos importa, el número esperado de comparaciones $E[C]$, lo único que tenemos que hacer es entender los $\Pr[X_{ij} = 1]$. Cada uno de estos números indica la probabilidad de que, en algún momento, se comparen un z_i con un z_j durante el `QUICKSORT` aleatorizado, y la siguiente tarea consiste en garantizar estos números²¹.

5.5.4 Cálculo de las probabilidades de comparación

Existe una fórmula que satisface la probabilidad de que dos elementos del array sean comparados en `QUICKSORT`.

Lema 5.4 (Probabilidad de comparación) *Si z_i y z_j indican los elementos menores i -ésimo y j -ésimo del array de entrada, con $i < j$, entonces*

$$\Pr[z_i, z_j \text{ se comparan en el } \text{QUICKSORT aleatorizado}] = \frac{2}{j - i + 1}.$$

Por ejemplo, si z_i y z_j son el elemento mínimo y el elemento máximo ($i = 1$ y $j = n$), entonces serán comparados con una probabilidad de solo $\frac{2}{n}$. Si no hay elementos cuyo valor esté entre z_i y z_j ($j = i + 1$), entonces z_i y z_j siempre serán comparados entre ellos.

Establecemos z_i y z_j con $i < j$ y tomamos en consideración el pivote z_k , elegido en la primera llamada a `QUICKSORT`. ¿Con qué opciones nos encontramos?

²¹La sección B.5 le da una gran importancia al hecho de que la linealidad de la expectativa es aplicable incluso a variables aleatorias que no son independientes (es decir, cuando el conocimiento de una variable aleatoria ofrece información sobre las otras). Este hecho nos resulta crucial, porque los X_{ij} no son independientes. Por ejemplo, si te digo que $X_{1n} = 1$, sabrás que z_1 o z_n ha sido elegido como elemento pivote en la llamada más exterior de `QUICKSORT` (¿por qué?) y esto, a su vez, hace mucho más probable que una variable aleatoria con la forma X_{ij} o X_{jn} sea también igual a 1.

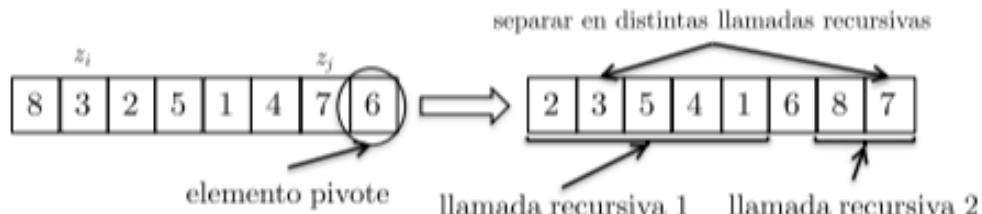
Cuatro situaciones de QUICKSORT

1. El pivote elegido es menor tanto de z_i como de z_j ($k < i$). Tanto z_i como z_j pasan a la segunda llamada recursiva.
2. El pivote elegido es mayor tanto de z_i como de z_j ($k > j$). Tanto z_i como z_j pasan a la primera llamada recursiva.
3. El pivote elegido se encuentra entre z_i y z_j ($i < k < j$). Los elementos z_i y z_j pasan, respectivamente, a la primera y segunda llamadas recursivas.
4. El pivote elegido es z_i o z_j ($k \in \{i, j\}$). El pivote queda excluido de ambas llamadas recursivas y el otro elemento pasa la primera (si $k = j$) o a la segunda (si $k = i$) llamadas recursivas.

Aquí están pasando dos cosas. En primer lugar, recuerda que cada comparación implica al elemento pivote actual. Por tanto, z_i y z_j se comparan en la llamada más exterior a QUICKSORT si, y solo si, uno de ellos es elegido como elemento pivote (situación 4). En segundo lugar, en la situación 3, no solo z_i y z_j no se compararán ahora, sino que no volverán a aparecer juntos en la misma llamada recursiva y, en consecuencia, no podrán ser comparados en el futuro. Por ejemplo, en el array

8	3	2	5	1	4	7	6
---	---	---	---	---	---	---	---

con $z_i = 3$ y $z_j = 7$, si cualquiera de los elementos $\{4, 5, 6\}$ son elegidos como elementos pivote, entonces z_i y z_j se envían a llamadas recursivas distintas, para nunca volverse a encontrar. Así, si se elige el “6”, la imagen será



Las situaciones 1 y 2 se comportan como un patrón de espera: z_i y z_j todavía no han sido comparados, pero es posible que lo sean en el futuro.

Durante este patrón de espera, z_i y z_j , y todos los elementos z_{i+1}, \dots, z_{j-1} con valores entre z_i y z_j , viven vidas paralelas y van pasando a la misma llamada recursiva. En algún momento, su viaje en compañía se verá interrumpido por una llamada recursiva a QUICKSORT en la que uno de los elementos $z_i, z_{i+1}, \dots, z_{j-1}, z_j$ será elegido como elemento pivote, activando las situaciones 3 o 4.²²

Si damos un salto adelante en esta llamada recursiva, para ver dónde está la acción, la situación 4 (y una comparación entre z_i y z_j) se activará si z_i o z_j son el pivote elegido, mientras que la situación 3 (sin que ocurra jamás la comparación) se activará si cualquiera de z_{i+1}, \dots, z_{j-1} es elegido como pivote. Así que hay dos casos incómodos (z_i y z_j) de entre las $j - i + 1$ opciones ($z_i, z_{i+1}, \dots, z_{j-1}, z_j$). Como el QUICKSORT aleatorizado siempre elige uniformemente los elementos pivote aleatorios, por simetría, *cada elemento de $\{z_i, z_{i+1}, \dots, z_{j-1}, z_j\}$ tiene la misma probabilidad de ser el primer elemento pivote seleccionado*. Si combinamos toda esta información,

$$\Pr[z_i, z_j \text{ se comparan en algún momento del QUICKSORT aleatorizado}]$$

es lo mismo que decir

$$\Pr[z_i \text{ o } z_j \text{ serán los pivotes antes que cualquiera de } z_{i+1}, \dots, z_{j-1}],$$

lo que es

$$\frac{\text{número de casos incómodos}}{\text{número total de opciones}} = \frac{2}{j - i + 1}.$$

Esto pone fin a la demostración del lema 5.4. *QED*

Volviendo a nuestra fórmula (5.1) para el número esperado de comparaciones realizadas por el QUICKSORT aleatorizado, obtenemos una expresión sorprendentemente exacta:

$$E[C] = \sum_{i=1}^{n-1} \sum_{j=i+1}^n \Pr[X_{ij} = 1] = \sum_{i=1}^{n-1} \sum_{j=i+1}^n \frac{2}{j - i + 1}. \quad (5.2)$$

Lo único que nos queda por comprobar, para demostrar el teorema 5.3, es que el lado derecho de (5.2) es, de hecho, $O(n \log n)$.

²²A falta de otra cosa, la llamadas recursivas anteriores terminarán por cercenar el subarray para dejarlo como $\{z_i, z_{i+1}, \dots, z_{j-1}, z_j\}$.

5.5.5 Cálculos finales

Demostrar un límite superior de $O(n^2)$ en el lado derecho de (5.2) es bastante sencillo: hay un máximo de n^2 términos en el sumatorio doble y cada uno de ellos tiene un valor máximo de $\frac{1}{2}$ (al que se llega cuando $j = i + 1$). Pero nosotros tenemos la ambición de lograr un límite superior de $O(n \log n)$, mucho más atractivo, y tendremos que ser muy hábiles para lograrlo, aprovechando el hecho de que la mayoría de los varios términos cuadráticos son mucho menores que $\frac{1}{2}$.

Pensemos en una de las sumas interiores del (5.2), con un valor fijo de i :

$$\sum_{j=i+1}^n \frac{2}{j-i+1} = 2 \cdot \underbrace{\left(\frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n-i+1} \right)}_{n-i \text{ términos}}.$$

Podemos limitar cada uno de estos sumatorios por el mayor de ellos, que se produce cuando $i = 1$:

$$\sum_{i=1}^{n-1} \sum_{j=i+1}^n \frac{2}{j-i+1} \leq \sum_{i=1}^{n-1} \underbrace{\sum_{j=2}^n \frac{2}{j}}_{\text{independiente de } i} = 2(n-1) \cdot \sum_{j=2}^n \frac{1}{j}. \quad (5.3)$$

¿Qué tamaño tiene $\sum_{j=2}^n \frac{1}{j}$? Vamos a verlo en una imagen.

Al observar los términos del sumatorio $\sum_{j=2}^n \frac{1}{j}$ como rectángulos en el plano, como hemos hecho con la figura 5.1, observamos que podemos limitar dicho sumatorio con el área bajo la curva $f(x) = \frac{1}{x}$, entre los puntos 1 y n , también conocida como la integral $\int_1^n \frac{dx}{x}$. Si recuerdas algo sobre cálculo, reconocerás que la solución a esta integral es el logaritmo natural $\ln x$ (es decir, $\ln x$ es la función cuya derivada es $\frac{1}{x}$):

$$\sum_{j=2}^n \frac{1}{j} \leq \int_1^n \frac{1}{x} dx = \ln x \Big|_1^n = \ln n - \underbrace{\ln 1}_{=0} = \ln n. \quad (5.4)$$

Si combinamos las ecuaciones y desigualdades de (5.2)–(5.4), tendremos

$$E[C] = \sum_{i=1}^{n-1} \sum_{j=i+1}^n \frac{2}{j-i+1} \leq 2(n-1) \cdot \sum_{j=2}^n \frac{1}{j} \leq 2(n-1) \ln n.$$

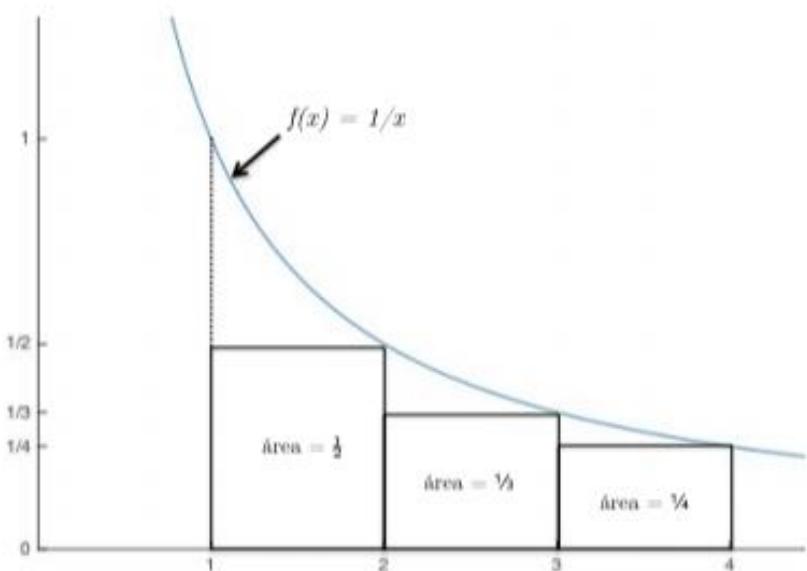


Figura 5.1: Cada término del sumatorio $\sum_{j=2}^n 1/j$ se puede identificar con un rectángulo de ancho 1 (entre las coordenadas x_{j-1} y x_j) y alto $1/j$ (entre las coordenadas y_0 y $1/j$). El grafo de la función $f(x) = 1/x$ roza la esquina superior derecha de cada uno de los rectángulos, por lo que el área bajo la curva (es decir, la integral) es un límite superior al área de los rectángulos.

Por tanto, el número esperado de comparaciones realizadas por el `QUICK-SORT` aleatorizado (y también la expectativa de su tiempo de ejecución, según el lema 5.2) es, en realidad, $O(n \log n)$. \mathcal{QED}

5.5.6 Solución al cuestionario 5.3

Respuesta correcta: (b). Si z_i o z_j son elegidos como elemento pivote en la llamada más exterior de `QUICKSORT`, entonces z_i y z_j serán comparados en la primera llamada a `PARTITION` (recuerda que el elemento pivote es comparado con todos los demás elementos del *subarray*). Si i y j difieren por más de 1, también es posible que z_i y z_j nunca lleguen a ser comparados (ver también la sección 5.5.4). Por ejemplo, los elementos mínimo y máximo no serán comparados entre sí salvo que uno de ellos sea elegido como elemento pivote en la llamada recursiva más exterior (*¿puedes ver por qué?*).

Por último, como cabría esperar de un buen algoritmo de ordenación, z_i y z_j nunca serán comparados entre sí más de una vez (pues sería redundante).

Cada comparación implica al elemento pivote actual, por lo que la primera vez que z_i y z_j son comparados en alguna llamada (si es que llegan a serlo), uno de ellos debe ser el elemento pivote. Como el elemento pivote está excluido de todas las llamadas recursivas futuras, z_i y z_j nunca volverán a aparecer juntos en la misma llamada recursiva (y mucho menos volverán a ser comparados).

*5.6 La ordenación necesita $\Omega(n \log n)$ comparaciones

¿Existirá un algoritmo de ordenación más rápido que MERGESORT y QUICKSORT, con un tiempo de ejecución mejor que $\Theta(n \log n)$? La intuición nos dice que el algoritmo deberá consultar cada elemento de la entrada, al menos, una vez, pero esto solo implica un límite inferior lineal de $\Omega(n)$. Esta sección opcional demuestra que, en el campo de la ordenación, *no podemos* hacerlo mejor: los algoritmos MERGESORT y QUICKSORT logran los mejores tiempos de ejecución asintóticos que es posible.

5.6.1 Algoritmos de ordenación basados en comparaciones

Esta es la declaración formal del límite inferior $\Omega(n \log n)$ para la ordenación de uso general.

Teorema 5.5 (Límite inferior para la ordenación) *Existe una constante $c > 0$ tal que, para cada $n \geq 1$, todo algoritmo de ordenación basado en la comparación realiza, al menos, $c \cdot n \log_2 n$ operaciones sobre un array de entrada de longitud n .*

Con “algoritmo de ordenación basado en la comparación”, nos referimos a un algoritmo que accede al *array* de entrada únicamente mediante comparaciones entre pares de elementos y nunca lo hace directamente al valor de un elemento. Los algoritmos de ordenación basados en la comparación tienen un uso general, en el sentido de que no asumen ningún aspecto de los elementos de entrada, más allá de que pertenecen a un conjunto totalmente ordenado. Puedes ver un algoritmo de ordenación basado en la comparación como una interacción con el *array* de entrada a través de una API que solo permite realizar una operación: dados dos índices i y j (ambos entre 1 y la longitud n del *array*), la operación informa de si el elemento i -ésimo es menor, igual o mayor que el elemento j -ésimo²³.

²³Por ejemplo, la rutina de ordenación predeterminada en el sistema operativo Unix, funciona así. El único requisito es una función definida por el usuario que compare los pares de elementos del *array* de entrada.

Por ejemplo, el algoritmo MERGESORT es un algoritmo de ordenación basado en la comparación, no le importa si está ordenando enteros o frutas (asumiendo que estemos de acuerdo en cuál es el orden correcto de todas las frutas posibles como, por ejemplo, el alfabético)²⁴. También lo son SELECTIONSORT, INSERTIONSORT, BUBBLESORT y QUICKSORT.

5.6.2 Ordenación más rápida bajo condiciones estrictas

La mejor forma de entender la ordenación basada en la comparación es utilizar antiejemplos. Estos son los tres famosos algoritmos de ordenación que asumen información sobre la entrada pero, a cambio, logran derrotar al límite inferior de $\Omega(n \log n)$ del teorema 5.5.

BUCKETSORT. El algoritmo BUCKETSORT resulta útil con datos numéricos, especialmente cuando están distribuidos uniformemente sobre un rango conocido. Por ejemplo, supongamos que el *array* de entrada tiene n elementos entre 0 y 1 que están, más o menos, distribuidos equitativamente. Dividimos mentalmente el intervalo [0, 1] en n “casilleros”, el primero reservado a los elementos de la entrada entre 0 y $\frac{1}{n}$, el segundo para los elementos entre $\frac{1}{n}$ y $\frac{2}{n}$ y, así, sucesivamente. El primer paso del algoritmo BUCKETSORT realiza una única pasada en tiempo lineal sobre el *array* de entrada, y coloca cada elemento en su casillero correspondiente. *No se trata de una pasada de comparación*: el algoritmo BUCKETSORT consulta el valor concreto de cada elemento de la entrada para identificar a qué casillero pertenece. Tiene importancia que el valor de un elemento de la entrada sea 0,17 o 0,27, aunque mantengamos inamovible la ordenación relativa de los elementos.

Si los elementos están más o menos bien distribuidos, la población de cada casillero será pequeña. El segundo paso del algoritmo ordena los elementos dentro de cada casillero (utilizando, por ejemplo, INSERTIONSORT). Como cada casillero contendrá pocos elementos, este paso también se ejecuta en tiempo lineal (con un número constante de operaciones realizadas en cada casillero). Por último, se concatenan las listas ordenadas de los diferentes casilleros, del primero al último. Este paso también se ejecuta en tiempo lineal. De esta forma, llegamos a la conclusión de que es posible realizar

²⁴Como analogía, podemos comparar los rompecabezas Sudoku y KenKen. Los rompecabezas Sudoku solo necesitan una noción de igualdad entre objetos diferentes, y seguirían teniendo sentido aunque sustituysésemos los dígitos 1–9 por nueve frutas diferentes. Los rompecabezas KenKen implican operaciones aritméticas y, por ello, necesitan números: ¿cuál sería la suma entre una ciruela y un mangostino?

ordenación en tiempo lineal, bajo condiciones muy estrictas de los datos de entrada.

COUNTINGSORT. El algoritmo COUNTINGSORT es una variante de la misma idea. En este caso, asumimos que solo habrá k valores diferentes posibles de cada elemento de la entrada (los conocemos por adelantado), como los enteros $\{1, 2, \dots, k\}$. El algoritmo prepara k casilleros, uno para cada valor posible y , en una sola pasada del *array* de entrada, coloca cada elemento en su casillero correspondiente. El *array* de salida es, sencillamente, la concatenación de estos casilleros (en orden). COUNTINGSORT se ejecuta en tiempo lineal cuando $k = O(n)$, donde n es la longitud del *array* de entrada. Tampoco es un algoritmo basado en la comparación, por las mismas razones que en BUCKETSORT.

RADIXSORT. El algoritmo RADIXSORT es una extensión de COUNTINGSORT, capaz de gestionar con facilidad *arrays* de entrada de n elementos enteros, con números razonablemente grandes representados en su forma binaria (cadenas de ceros y unos, o “bits”). El primer paso de RADIXSORT solo tiene en consideración el bloque de los $\log_2 n$ bits menos significativos de los números de entrada y los ordena adecuadamente. Como $\log_2 n$ bits solo pueden codificar n valores diferentes (correspondientes a los números $0, 1, 2, \dots, n - 1$, expresados en binario), se puede utilizar el algoritmo COUNTINGSORT para implementar este paso en tiempo lineal. Entonces, el algoritmo RADIXSORT reordena todos los elementos utilizando el bloque de los siguientes $\log_2 n$ bits menos significativos, y continúa repitiendo el mismo método hasta haber procesado todos los bits de la entrada. Para que este algoritmo se comporte correctamente, es importante implementar la subrutina COUNTINGSORT de forma que sea *estable*, es decir, que conserve el orden relativo de los diferentes elementos que tengan el mismo valor²⁵. El algoritmo RADIXSORT se ejecuta en tiempo lineal, siempre y cuando el *array* de entrada contenga solo enteros entre 0 y n^k , para una k constante.

Estos tres algoritmos de ordenación demuestran cómo ciertas condiciones adicionales aplicadas a los datos de entrada (como que no sean enteros muy grandes) permite utilizar técnicas que van más allá de la comparación (como el uso de casilleros) y crear algoritmos más rápidos que $\Theta(n \log n)$. El teorema 5.5 afirma que tales mejoras son imposibles de aplicar a algoritmos de ordenación generalistas basados en la comparación. Veamos el porqué.

²⁵No todos los algoritmos de ordenación son estables. Por ejemplo, QUICKSORT no es un algoritmo de ordenación estable (¿podrías decir por qué?).

5.6.3 Demostración del teorema 5.5

Establecemos arbitrariamente un algoritmo de ordenación determinista y basado en la comparación²⁶.

Cero comparaciones. Imaginemos, en primer lugar, un algoritmo que realice *cero* comparaciones. Como el algoritmo solo puede obtener información de la entrada a través de las comparaciones, en este caso no sabrá absolutamente nada. Por ejemplo, tal algoritmo ignora por completo si la entrada es

1	2	3	4
o			
4	3	2	1

Para comportarse correctamente en el primer caso, el *array* de salida del algoritmo debe ser igual al de entrada. Como el algoritmo no sabe en qué caso se encuentra, el *array* de salida también será el mismo que el de la entrada (desordenado) en el segundo caso. Este argumento muestra que cualquier algoritmo de ordenación basado en comparaciones que no realice ninguna comparación se comportará incorrectamente en, al menos, uno de estos dos casos.

Una comparación. Pensemos ahora en un algoritmo que realice *una* comparación, digamos que entre los primer y segundo elementos del *array* de entrada. Este algoritmo es capaz de distinguir entre los dos casos anteriores y comportarse de forma distinta según la situación. Pero seguirá sin tener ni idea de si la entrada es

1	2	3	4
o			
1	2	4	3

Como el algoritmo se ejecuta de forma idéntica en estos dos casos, el resultado será incorrecto en, al menos, uno de ellos. Si su *array* de salida no es igual al de entrada en estos casos, entonces será erróneo en la primera situación. En caso contrario, lo será en la segunda.

Dos comparaciones. Un algoritmo que realice *dos* comparaciones es capaz de obtener algo de información de cada elemento de un *array* de longitud 4, por ejemplo al comparar los primer y segundo elementos y, después,

²⁶Se puede aplicar una argumentación similar a los algoritmos de ordenación basados en la comparación y aleatorizados, donde ninguno de ellos tiene una expectativa de tiempo de ejecución mejor que $\Theta(n \log n)$. Ver también los problemas 5.6–5.7.

los tercero y cuarto. Sigue sin ser suficiente, pues el algoritmo no distinguirá si la entrada es

1	2	3	4
o			
1	3	2	4

Si el algoritmo pudiese acceder a los *valores* de los elementos del *array* de entrada, podrían distinguir los dos casos. Pero, como se trata de un algoritmo basado en la comparación, no es capaz de extraer información más allá del hecho de que el primer elemento es menor que el segundo y que el tercero es menor que el cuarto. El algoritmo se comporta de forma idéntica en ambos casos y solo puede ser correcto en uno de ellos.

k comparaciones. Consideremos el caso general de un algoritmo que siempre realiza, como mucho, k comparaciones. El patrón general es el siguiente: si dos *arrays* de entrada diferentes conducen a las mismas respuestas en las k comparaciones, entonces el algoritmo no puede distinguirlos y se ejecutará de forma idéntica la dos veces (una de ellas será incorrecta). Como existen 2^k conjuntos de respuestas posibles a las k comparaciones del algoritmos, este es capaz de distinguir entre un máximo de 2^k entradas diferentes y ejecutarse de, como mucho, 2^k formas distintas (figura 5.2).

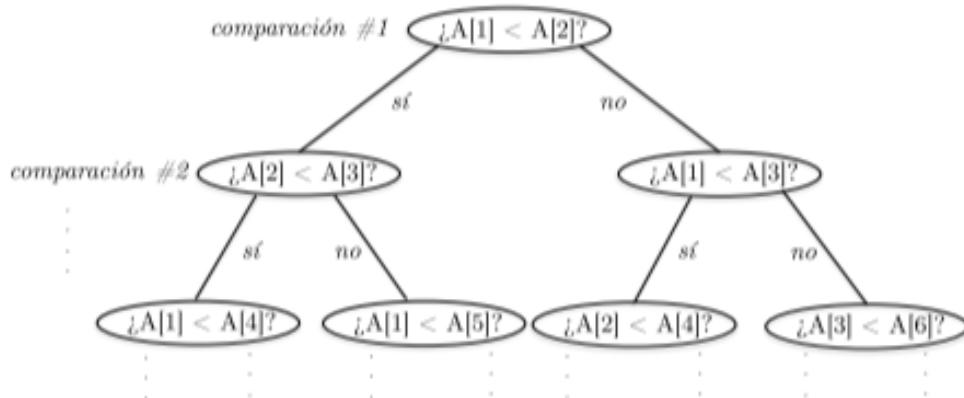


Figura 5.2: Un algoritmo basado en la comparación que realiza un máximo de k comparaciones se puede ejecutar de, como mucho, 2^k formas distintas.

Existen $n! = n \cdot (n-1) \cdots 2 \cdot 1$ *arrays* de entrada diferentes de longitud n , que contienen los números $\{1, 2, 3, \dots, n\}$, y un algoritmo de ordenación correcto

debe distinguirlas todas²⁷. Por tanto, el número máximo de k comparaciones realizadas por un algoritmo de ordenación basado en la comparación que sea correcto, debe satisfacer

$$2^k \geq \underbrace{n!}_{n \cdot (n-1) \cdots 2 \cdot 1} \geq \left(\frac{n}{2}\right)^{n/2},$$

donde aprovecharemos el hecho de que los primeros $n/2$ términos de $n \cdot (n-1) \cdots 2 \cdot 1$ son todos, al menos, $\frac{n}{2}$. Teniendo en cuenta el logaritmo en base 2 de ambos lados, se demuestra que

$$k \geq \frac{n}{2} \log_2 \left(\frac{n}{2}\right) = \Omega(n \log n).$$

Con esto finaliza la demostración del teorema 5.5. *QED*

Conclusiones

- ★ El famoso algoritmo **QUICKSORT** cuenta con tres pasos de alto nivel: primero, selecciona un elemento p de la entrada para que opere como “elemento pivote”; en segundo lugar, su subrutina **PARTITION** reorganiza el *array* para que los elementos menores y mayores que p se coloquen, respectivamente, antes y después de este; por último, ordena recursivamente los dos *subarrays* situados a ambos lados del pivote.
- ★ La subrutina **PARTITION** se puede implementar con una ejecución *in situ* y en tiempo lineal, lo que significa que necesitará muy poca memoria adicional. En consecuencia, **QUICKSORT** también se ejecuta *in situ*.
- ★ La corrección del algoritmo **QUICKSORT** no depende de cómo se elijan los elementos pivote, pero su tiempo de ejecución sí.
- ★ El peor caso es un tiempo de ejecución de $\Theta(n^2)$, donde n es la longitud del *array* de entrada. Esto sucede cuando el *array* de entrada ya está ordenado y el primer elemento se utiliza siempre como pivote.
- ★ El mejor caso es un tiempo de ejecución de $\Theta(n \log n)$.

²⁷Hay n posiciones posibles para el “1”, $n - 1$ posiciones restantes para el “2”, etc.

Esto sucede cuando siempre se utiliza el elemento correspondiente a la mediana como pivote.

- ★ En el **QUICKSORT** aleatorizado, el elemento pivote se elige siempre de forma aleatoria y equitativa. Su tiempo de ejecución puede ser cualquiera entre $\Theta(n \log n)$ y $\Theta(n^2)$, dependiendo del resultado de las operaciones aleatorias.
- ★ El tiempo medio de ejecución del **QUICKSORT** aleatorizado es de $\Theta(n \log n)$, lo que supone solo un pequeño factor constante peor que el tiempo de ejecución del mejor caso.
- ★ Por intuición sabemos que elegir un pivote aleatorio es una buena idea, porque hay una probabilidad del 50% de obtener una partición 25%-75% o mejor del *array* de entrada.
- ★ El análisis formal utiliza un esquema de descomposición que expresa una variable aleatoria compleja como la suma de variables aleatorias 0-1 y, después, aplica la linealidad de la expectativa.
- ★ La clave es que, en **QUICKSORT**, los elementos i -ésimo y j -ésimo más pequeños del *array* de entrada se comparan si, y solo si, uno de ellos es elegido como pivote antes que un elemento con un valor que se encuentre, estrictamente, entre ellos.
- ★ Un algoritmo de ordenación basado en la comparación es un algoritmo de uso generalista que únicamente accede al *array* de entrada para comparar pares de elementos y nunca utiliza directamente los valores de esos elementos.
- ★ Los algoritmos de ordenación no basados en la comparación tienen un peor caso de tiempo de ejecución asintótica mejor que $O(n \log n)$.

Comprueba que lo has entendido

Problema 5.1 (S) ¿Cuál de las siguientes afirmaciones es cierta (elige todas las que lo sean)?

- a) En la práctica, siempre deberíamos elegir QUICKSORT sobre MERGESORT.
- b) En la práctica, siempre deberíamos elegir MERGESORT sobre QUICKSORT.
- c) El límite inferior de ordenación $\Omega(n \log n)$ para arrays de longitud n se puede superar utilizando la aleatorización.
- d) El límite inferior de ordenación $\Omega(n \log n)$ se puede superar cuando se sabe que todos los elementos del array son enteros entre 1 y n^{10} .

Problema 5.2 (S) Recuerda la subrutina PARTITION utilizada por QUICKSORT (sección 5.2). Sabes que el siguiente array ha sido particionado en torno a un elemento pivote:

3	1	2	4	5	8	7	6	9
---	---	---	---	---	---	---	---	---

¿Cuál de los elementos podría ser ese pivote (cabe la posibilidad de que haya más de una opción)?

Problema 5.3 (S) Digamos que α es una constante, independiente de la longitud n del array de entrada, y con un valor situado estrictamente entre 0 y $\frac{1}{2}$. ¿Cuál es la probabilidad de que, con un pivote elegido aleatoriamente, la subrutina PARTITION genere una separación en la que el tamaño de los dos subproblemas resultantes sea, al menos, α veces el tamaño del array original?

- a) α
- b) $1 - \alpha$
- c) $1 - 2\alpha$
- d) $2 - 2\alpha$

Problema 5.4 (P) Digamos que α es una constante, independiente de la longitud n del array de entrada, y con un valor situado estrictamente entre 0 y $\frac{1}{2}$. Asume que, en cada llamada recursiva, has logrado las particiones más o menos equilibradas del problema anterior (es decir, cuando una llamada recursiva recibe un array de longitud k , cada una de sus dos llamadas recursivas reciben un array con longitud entre αk y $(1 - \alpha)k$). ¿Cuántas llamadas recursivas sucesivas se pueden producir antes de que se active el caso base? Igualmente, ¿qué niveles del árbol de recursión del algoritmo pueden tener hojas? Expresa tu respuesta como un rango d de números posibles, del mínimo al máximo número de llamadas recursivas que podrían ser necesarias.

- a) $0 \leq d \leq -\frac{\ln n}{\ln \alpha}$
- b) $-\frac{\ln n}{\ln \alpha} \leq d \leq -\frac{\ln n}{\ln(1-\alpha)}$
- c) $-\frac{\ln n}{\ln(1-\alpha)} \leq d \leq -\frac{\ln n}{\ln \alpha}$
- d) $-\frac{\ln n}{\ln(1-2\alpha)} \leq d \leq -\frac{\ln n}{\ln(1-\alpha)}$

Problema 5.5 (S) Definamos el nivel de recursión de QUICKSORT como el número máximo de llamadas recursivas sucesivas que realiza antes de llegar al caso base o, dicho de otra manera, el nivel más profundo de su árbol de recursión. En el QUICKSORT aleatorizado, la profundidad de recursión es una variable aleatoria, en función de los pivotes elegidos. ¿Cuáles son las profundidades de recursión mínima y máxima posibles del QUICKSORT aleatorizado?

- a) mínimo: $\Theta(1)$, máximo: $\Theta(n)$
- b) mínimo: $\Theta(\log n)$, máximo: $\Theta(n)$
- c) mínimo: $\Theta(\log n)$, máximo: $\Theta(n \log n)$
- d) mínimo: $\Theta(\sqrt{n})$, máximo: $\Theta(n)$

Problemas más difíciles

Problema 5.6 (P) Considera los $n!$ arrays diferentes que contienen los números $\{1, 2, \dots, n\}$ con distintas ordenaciones. Demuestra que todo algoritmo de ordenación determinista basado en la comparación tiene un tiempo de ejecución medio de $\Omega(n \log n)$ sobre estas $n!$ entradas.

Problema 5.7 (P) Demuestra que, para cada algoritmo de ordenación autorizado basado en la comparación, existe un *array* de longitud n cuyo tiempo de ejecución esperado es de $\Omega(n \log n)$ (de hecho, este límite inferior se mantiene incluso en la media de las $n!$ entradas evaluadas en el problema anterior).

Problemas de programación

Problema 5.8 Implementa el algoritmo `QUICKSORT` utilizando tu lenguaje de programación preferido. Experimenta con los rendimientos que ofrecen los distintos métodos de elección del pivote.

Una técnica consiste en mantener un registro del número de comparaciones realizadas por `QUICKSORT` entre los elementos del *array* de entrada²⁸. Determina, con diferentes *arrays* de entrada, el número de comparaciones realizadas con las siguientes implementaciones de `CHOOSEPIVOT`:

1. Utilizar siempre el primer elemento como pivote.
2. Utilizar siempre el último elemento como pivote.
3. Utilizar un elemento aleatorio como pivote (en este caso, deberías ejecutar el algoritmo 10 veces sobre un mismo *array* y obtener la media de los resultados).
4. Utilizar la *mediana de tres* como pivote. El objetivo de esta regla es la de realizar algo de trabajo extra a cambio de obtener un rendimiento mucho mejor en *arrays* de entrada que estén casi ordenados u ordenados inversamente.

Entrando en más detalle, esta implementación de `CHOOSEPIVOT` tiene en consideración los elementos primero, central y último del *array* dado (en el caso de un *array* de longitud par $2k$, utilizar el k -ésimo elemento como “central”). Después, identifica cuál de estos tres elementos es la mediana (es decir, aquel cuyo valor se encuentre entre los otros dos), y lo devuelve como pivote²⁹.

²⁸No es necesario contar las comparaciones una a una. Cuando hay una llamada recursiva sobre un *subarray* de longitud m , basta con sumar $m - 1$ al número total de comparaciones actual (recuerda que el elemento pivote se compara con cada uno de los otros $m - 1$ elementos del *subarray* en esta llamada recursiva).

²⁹Un análisis cuidadoso mantendría un registro de las comparaciones realizadas para identificar la mediana de los tres elementos candidatos, además de las comparaciones realizadas por las llamadas a `PARTITION`.

Por ejemplo, con el *array* de entrada

8	3	2	5	1	4	7	6
---	---	---	---	---	---	---	---

la subrutina tendría en consideración los elementos primero (8), central (5) y último (6). Devolvería 6, la mediana del conjunto $\{5, 6, 8\}$, como elemento pivote.

Puedes visitar www.algorithmsilluminated.org para encontrar casos de prueba y conjuntos de datos complicados.

Selección en tiempo lineal

Este capítulo estudia el *problema de selección*, donde el objetivo está en identificar el elemento i -ésimo más pequeño de un *array* desordenado. Es fácil resolver el problema en tiempo $O(n \log n)$ utilizando la ordenación, pero podemos hacerlo mejor. La sección 6.1 describe un algoritmo aleatorizado extraordinariamente práctico, de espíritu muy similar al `QUICKSORT` aleatorizado, que tiene, en promedio, un tiempo de ejecución *lineal*. La sección 6.2 aporta un elegante análisis a este algoritmo: existe una forma muy atractiva de plantear el progreso del algoritmo en términos de un experimento sencillo de lanzamiento de monedas y, después, la linealidad de la expectativa (sí, ha vuelto...) pone el broche final.

Los lectores más teóricos se podrían preguntar si el problema de selección se puede resolver en tiempo lineal sin recurrir a la aleatorización. La sección 6.3 describe un famoso algoritmo determinista para este problema, algoritmo que cuenta entre sus autores con más galardonados con el Premio Turing que cualquier otro que conozca. Es determinista (no hay lugar para la aleatorización) y basado en un ingenioso concepto de “mediana de medianas”, que garantiza buenas elecciones de los pivotes. La sección 6.4 demuestra el límite de tiempo de ejecución lineal, lo que no es tan fácil.

Este capítulo da por hecho que recuerdas la subrutina `PARTITION` de la sección 5.2, que partitiona un *array* en torno a un elemento pivote en tiempo real, así como el aspecto intuitivo de determinar si un pivote es apropiado o no (sección 5.3).

6.1 El algoritmo RSELECT

6.1.1 El problema de selección

En el *problema de selección*, la entrada es igual a la del problema de ordenación (un *array* de n números) unida a un entero $i \in \{1, 2, \dots, n\}$. El objetivo consiste en identificar el i -ésimo *estadístico de orden* (la i -ésima entrada más pequeña del *array*).

Problema: selección

Entrada: un *array* de n números, en orden arbitrario, y un entero $i \in \{1, 2, \dots, n\}$.

Salida: el i -ésimo elemento más pequeño de A .

Como viene siendo habitual, y por simplicidad, asumimos que los elementos del *array* de entrada son distintos, sin duplicados.

Por ejemplo, si el *array* de entrada es

6	8	9	2
---	---	---	---

y el valor de i es 2, la salida correcta es 6. Si i fuese 3, la salida correcta sería 8 y, así, sucesivamente.

Cuando $i = 1$, el problema de selección no es más que el problema de calcular el elemento mínimo de un *array*. Esto es fácil de realizar en tiempo lineal, pues basta con una pasada por el *array* recordando el elemento más pequeño que se ha encontrado. Igualmente, el caso de hallar el elemento máximo ($i = n$) es sencillo. Pero, ¿qué ocurre con valores de i que se encuentren entre el mínimo y el máximo? Por ejemplo, ¿qué hacemos si queremos calcular el elemento central (la *mediana*) de un *array*?

Para ser precisos, en el caso de un *array* de longitud n impar, la mediana es el i -ésimo estadístico de orden con $i = (n + 1)/2$. En el caso de que el *array* tenga una longitud n par, acordaremos definir la mediana como el menor valor de las dos posibilidades, que corresponde a $i = n/2$.¹

¹¿Por qué nos puede interesar calcular la mediana de un *array*? Después de todo, la

6.1.2 Reducción a ordenación

Ya conocemos un algoritmo rápido para el problema de selección, que se apoya en nuestro veloz algoritmo de ordenación.

Reducción de selección a ordenación

Entrada: array A de n números distintos y un entero
 $i \in \{1, 2, \dots, n\}$.

Salida: el i -ésimo estadístico de orden de A .

$B := \text{MERGESORT}(A)$
devolver $B[i]$

Una vez ordenado el *array* de entrada, es evidente que sabremos dónde encontrar el i -ésimo elemento más pequeño: estará esperándonos en la posición i -ésima del *array* ordenado. Como MERGESORT se ejecuta en tiempo $O(n \log n)$ (teorema 1.2), también lo hace este algoritmo de dos pasos².

Pero recordemos el mantra de cualquier diseñador de algoritmos que merezca tal título: *¿podemos hacerlo mejor?* ¿Podemos diseñar un algoritmo que resuelva el problema de selección en un tiempo inferior a $O(n \log n)$? Nuestra mejor expectativa es el tiempo lineal ($O(n)$) pues, si no consultamos cada elemento del *array*, no tendremos ninguna esperanza de identificar, digamos, su elemento mínimo. También sabemos, gracias al teorema 5.5, que cualquier algoritmo que utilice una subrutina de ordenación se verá atrapado por un tiempo de ejecución, en el peor caso, de $\Omega(n \log n)$.³ Por tanto, si *somos capaces* de lograr un tiempo ejecución mejor que $O(n \log n)$ para el problema de selección, habremos demostrado que *la selección es, en*

media (o promedio) es muy fácil de calcular en tiempo lineal, pues basta con sumar todos los elementos del *array*, en una sola pasada, y dividir el resultado entre n . Una razón es la de calcular una síntesis estadística de un *array* que resulte más robusta que la media. Por ejemplo, un elemento incorrecto, como podría ser un error en la entrada de datos, alteraría completamente la media un de *array*, pero tendría poco impacto en la mediana.

²Un científico de la computación llamaría a esto *reducción* del problema de selección al problema de ordenación. Una reducción te evita la necesidad de construir un nuevo algoritmo desde cero y, en su lugar, te permite apoyarte en otro algoritmo ya existente. Además de su utilidad práctica, las reducciones suponen un concepto fundamental en las ciencias de la computación y las trataremos extensamente en la cuarta parte.

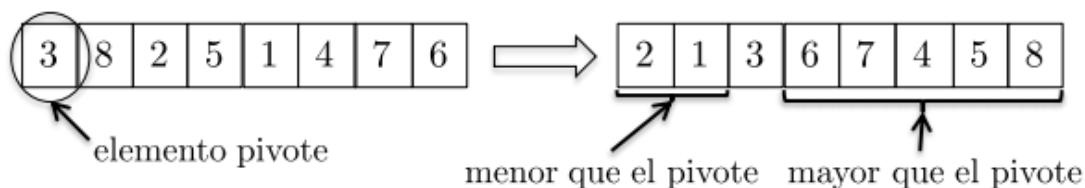
³Asumiendo que nos limitemos a algoritmos de ordenación basados en la comparación, como en la sección 5.6.

esencia, más sencilla que la ordenación. Lograrlo requerirá una buena dosis de ingenio y apoyarnos en nuestros algoritmos de ordenación no será suficiente.

6.1.3 Un método de divide y vencerás

El algoritmo de selección en tiempo lineal RSELECT sigue el patrón que tanto éxito nos proporcionó con el QSORT aleatorizado: elegir un elemento pivote aleatorio, particionar el *array* de entrada en torno a ese pivote y operar recursivamente en consecuencia. Por tanto, nuestra siguiente labor consiste en entender cuál puede ser la recursión más adecuada para el problema de selección.

Recordemos lo que hace la subrutina PARTITION de la sección 5.2: dados un *array* y una elección del elemento pivote, reorganiza los elementos de dicho *array* de forma que todo lo que sea menor y mayor aparezca, respectivamente, antes y después del pivote.



Por tanto, el elemento pivote termina colocado en su posición correcta, después de todos los elementos menores y antes de todos los mayores que el mismo.

QSORT ordena recursivamente el *subarray* de elementos menores que el pivote y, también, los que son mayores. ¿Cuál podría ser la analogía en el problema de selección?

Cuestionario 6.1

Supongamos que estamos buscando el quinto estadístico de orden en un *array* de entrada de 10 elementos. Supongamos que, después de particionar el *array*, el elemento pivote termina en la tercera posición. ¿Sobre qué lado del elemento pivote deberíamos ejecutar una nueva recursión y qué estadístico de orden deberíamos buscar?

- a) El tercer estadístico de orden a la izquierda del pivote.

- b) El segundo estadístico de orden a la derecha del pivote.
- c) El quinto estadístico de orden a la derecha del pivote.
- d) Puede que tengamos que ejecutar recursiones a ambos lados del pivote.

Solución y aclaraciones en la sección [6.1.6](#)

6.1.4 Pseudocódigo de RSELECT

Nuestro pseudocódigo para el algoritmo RSELECT sigue la descripción de alto nivel que utilizamos para QUICKSORT en la sección [5.1](#), con dos cambios. Primero, utilizamos siempre elementos pivote aleatorios, en vez que incorporar una subrutina CHOOSEPIVOT genérica. Segundo, RSELECT solo realiza una llamada recursiva, mientras que, en el caso de QUICKSORT, eran dos. Esta diferencia es la razón principal por la que esperamos que RSELECT pueda ser incluso más rápido que el QUICKSORT aleatorizado.

RSELECT

Entrada: array A de $n \geq 1$ enteros distintos y un entero

$$i \in \{1, 2, \dots, n\}.$$

Salida: el i -ésimo estadístico de orden de A .

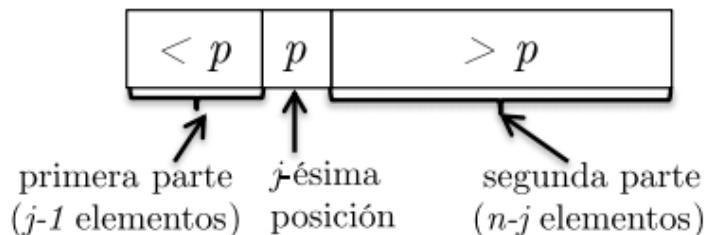
```

si  $n = 1$  entonces                                // caso base
    devolver  $A[1]$ 

seleccionar en  $A$  un elemento pivote  $p$  uniformemente aleatorio
particionar  $A$  en torno a  $p$ 
 $j :=$  posición de  $p$  en el array particionado
si  $j = i$  entonces                            // has tenido suerte
    devolver  $p$ 
si no, si  $j > i$  entonces
    devolver RSELECT(primeras partes de  $A$ ,  $i$ )
en otro caso                                //  $j < i$ 
    devolver RSELECT(segundas partes de  $A$ ,  $i - j$ )

```

La partición del *array* de entrada en torno al elemento pivote p separa dicho *array* en tres partes, lo que lleva a los tres casos del algoritmo RSELECT:



Como el elemento pivote p asume su posición correcta en el *array* particionado, si ocupa la posición j -ésima debe ser el estadístico de orden j -ésimo. Si, por pura suerte, el algoritmo estaba buscando el estadístico de orden j -ésimo (es decir, $i = j$), habremos terminado. Si el algoritmo busca un número más pequeño (es decir, $i < j$), este debe pertenecer a la primera parte del *array* particionado. En este caso, la recursión solo descartará los elementos mayores que el j -ésimo estadístico de orden (y, por tanto, del i -ésimo), por lo que el algoritmo seguirá buscando el i -ésimo elemento más pequeño entre los del primer *subarray*. En el último caso ($i > j$), el algoritmo busca un número mayor que el elemento pivote, y la recursión imita la solución al cuestionario 6.1. El algoritmo realiza recursiones sobre la segunda parte del *array* particionado, descartando el elemento pivote, así como los $j-1$ elementos menores que este, que no volverán a ser tenidos en cuenta. Debido a que el algoritmo buscaba, inicialmente, el i -ésimo elemento más pequeño, ahora buscará el $(i-j)$ -ésimo elemento más pequeño de entre los que quedan.

El algoritmo RSELECT se puede implementar para que trabaje *in situ*, sin necesidad de un consumo de memoria adicional significativo. La implementación *in situ* utiliza los extremos izquierdo y derecho para mantener un registro del *subarray* actual, al igual que ocurría en el pseudocódigo de QUICKSORT en la sección 5.2.5. Puedes consultar también el problema de programación 6.7.

6.1.5 Tiempo de ejecución de RSELECT

Al igual que el QUICKSORT aleatorizado, el tiempo de ejecución del algoritmo RSELECT depende de la elección de los pivotes. ¿Qué es lo peor que podría ocurrir?

Cuestionario 6.2

¿Cuál es el tiempo de ejecución del algoritmo RSELECT si la elección de los elementos pivote siempre es la peor posible?

- a) $\Theta(n)$
- b) $\Theta(n \log n)$
- c) $\Theta(n^2)$
- d) $\Theta(2^n)$

Solución y aclaraciones en la sección [6.1.6](#)

Ahora sabemos que el algoritmo RSELECT no se ejecuta en tiempo lineal para todas las posibles elecciones de elementos pivote pero, ¿podría ejecutarse en tiempo lineal promedio, gracias a la elección aleatoria de los pivotes? Vamos a empezar con un objetivo más modesto: ¿existe *alguna* elección de pivotes que haga que RSELECT se ejecute en tiempo lineal?

¿Qué hace bueno a un pivote? La respuesta es la misma que en el caso de QUICKSORT (ver la sección [5.3](#)): los buenos pivotes garantizan que las llamadas recursivas reciban subproblemas significativamente más pequeños. La mejor situación es aquella en la que el pivote ofrece la separación más equilibrada posible, con dos *subarrays* de la misma longitud⁴. Esta situación se produce cuando el pivote elegido se corresponde con la mediana del *array*. Puede parecer que analizar esta situación es entrar en un bucle, porque podríamos haber empezado calculando la mediana. Pero sigue siendo un experimento interesante el intentar entender cuál es el mejor tiempo de ejecución que podría lograr RSELECT (y más vale que sea lineal).

Digamos que $T(n)$ indica el tiempo de ejecución de RSELECT en *arrays* de longitud n . Si el algoritmo, por arte de magia, elige siempre la mediana del *subarray* actual, toda llamada recursiva realizará trabajo lineal en su *subarray* (principalmente en la subrutina PARTITION) y realizará una llamada

⁴Ignoramos el caso en el que, por cuestión de suerte, el pivote elegido se corresponde exactamente con el estadístico de orden que estamos buscando, pues es bastante difícil que ocurra antes de las últimas llamadas recursivas del algoritmo.

recursiva sobre un *array* de la mitad de su tamaño:

$$T(n) \leq \underbrace{T\left(\frac{n}{2}\right)}_{\text{porque pivote = mediana}} + \underbrace{O(n)}_{\text{PARTITION, etc.}}$$

Esta recurrencia entra de lleno en el ámbito del método maestro (teorema 4.1); como hay una llamada recursiva ($a = 1$), el tamaño del subproblema se reduce por un factor de 2 ($b = 2$) y el trabajo realizado fuera de la llamada recursiva es lineal ($d = 1$), $1 = a < b^d = 2$, y el segundo caso del método maestro nos dice que $T(n) = O(n)$. Esta comprobación nos indica que vamos por el buen camino: si RSELECT tiene la suficiente suerte, se ejecuta en tiempo lineal.

Entonces, ¿el tiempo de ejecución de RSELECT es típicamente más cercano a su mejor rendimiento de $\Theta(n)$ o a su peor rendimiento de $\Theta(n^2)$? Gracias al éxito que obtuvimos con el QUICKSORT aleatorizado, podemos esperar que las ejecuciones habituales de RSELECT tengan rendimientos más cercanos al mejor caso. Y, de hecho, aunque teóricamente RSELECT podría tener un tiempo de ejecución de $\Theta(n^2)$, en la práctica casi siempre es de $O(n)$.

Teorema 6.1 (Tiempo de ejecución de RSELECT) *Por cada array de entrada de longitud $n \geq 1$, el tiempo de ejecución medio de RSELECT es $O(n)$.*

La sección 6.2 ofrece la demostración del teorema 6.1.

Por increíble que parezca, el tiempo de ejecución medio de RSELECT solo es un factor constante más grande que el tiempo necesario para leer la entrada. Como la ordenación necesita tiempo $\Omega(n \log n)$ (sección 5.6), el teorema 6.1 demuestra que la selección es, fundamentalmente, más sencilla que la ordenación.

Podemos aplicar aquí los mismos comentarios sobre el tiempo medio de ejecución que vimos para el QUICKSORT aleatorizado (teorema 5.1). El algoritmo RSELECT tiene un uso generalista debido a que el límite del tiempo de ejecución es aplicable a todas las entradas posibles y ese tiempo “medio” solo hace referencia a los elementos pivote aleatorios elegidos por el algoritmo. Al igual que con QUICKSORT, la constante oculta en la notación Big-O del teorema 6.1 es razonablemente pequeña y el algoritmo se puede implementar de forma que trabaje *in situ*.

6.1.6 Soluciones a los cuestionarios 6.1–6.2

Solución al cuestionario 6.1

Respuesta correcta: (b). Después de particionar el *array*, sabemos que el pivote se encontrará en su posición correcta, con los números menores situados antes y los mayores después del mismo. Como el elemento pivote ha terminado en la tercera posición del *array*, resulta ser el tercer elemento más pequeño. Estamos buscando el quinto elemento más pequeño, que es mayor. Por tanto, tenemos la seguridad de que el quinto estadístico de orden se encuentra en el segundo *subarray*, por lo que solo habrá que ejecutar una recursión. ¿Qué estadístico de orden buscaremos en esa llamada recursiva? Inicialmente buscábamos el quinto número más pequeño, pero hemos descartado el pivote y los dos números anteriores. Como $5 - 3 = 2$, ahora buscamos el segundo elemento más pequeño de los que han pasado a la llamada recursiva.

Solución al cuestionario 6.2

Respuesta correcta: (c). El peor caso del tiempo de ejecución de RSELECT es el mismo que el del QUICKSORT aleatorizado. El ejemplo más perjudicial es el mismo que el del cuestionario 5.1: supongamos que el *array* de entrada ya está ordenado y que el algoritmo elige reiteradamente el primer elemento como pivote. En cada llamada recursiva, la primera parte del *subarray* estará vacía, mientras que la segunda incluirá todos los elementos menos el pivote actual. Por tanto, la longitud del *subarray* de cada llamada recursiva solo será de una unidad menos que en la llamada anterior. El trabajo realizado en cada llamada recursiva (sobre todo por la subrutina PARTITION) es lineal en relación a la longitud del *subarray*. Al calcular la mediana, habrá $\approx n/2$ llamadas recursivas, cada una con un *subarray* de longitud mínima de $n/2$, por lo que el tiempo global será de $\Theta(n^2)$.

*6.2 Análisis de RSELECT

Una forma de demostrar el límite del tiempo de ejecución lineal esperado del algoritmo RSELECT (teorema 6.1) consiste en seguir el mismo esquema de descomposición que tan bien nos funcionó en el análisis del QUICKSORT aleatorizado (sección 5.5), con variables binarias aleatorias que registran las comparaciones. En el caso de RSELECT, también podemos solucionarlo

con una versión más sencilla del esquema de descomposición, que formalice la intuición de la sección 5.4.3: (i) los pivotes aleatorios son, probablemente, una buena elección y (ii) los buenos pivotes hacen que el algoritmo progrese muy rápido.

6.2.1 Seguimiento del progreso en fases

Ya nos hemos dado cuenta de que una llamada a RSELECT realiza un trabajo $O(n)$ fuera de su llamada recursiva, principalmente en la llamada a PARTITION. Es decir, existe un constante $c > 0$ tal que

(*) por cada *array* de entrada de longitud n , RSELECT realiza un máximo de cn operaciones fuera de su llamada recursiva.

Como RSELECT siempre realiza una sola llamada recursiva, podemos seguir su progreso gracias a la longitud del *subarray* sobre el que esté trabajando en cada momento, que se va reduciendo con el tiempo. Para simplificar, utilizaremos una versión más burda de esta medición del progreso⁵. Supongamos que la llamada más exterior a RSELECT recibe un *array* de longitud n . Para un entero $j \geq 0$, decimos que una llamada recursiva a RSELECT se encuentra en la *fase* j si la longitud de su *subarray* está entre

$$\left(\frac{3}{4}\right)^{j+1} \cdot n \text{ y } \left(\frac{3}{4}\right)^j \cdot n.$$

Por ejemplo, la llamada más exterior a RSELECT pertenece siempre a la fase 0, como lo está cualquier llamada recursiva posterior que opere sobre, al menos, el 75% del *array* de entrada original. Las llamadas recursivas sobre *subarrays* que contengan entre el $(\frac{3}{4})^2 \approx 56\%$ y el 75% de los elementos originales pertenecen a la fase 1 y, así, sucesivamente. Al llegar a la fase $j \approx \log_{4/3} n$, el *subarray* tendrá un tamaño máximo de 1 y no quedarán llamadas recursivas por hacer.

Digamos que X_j indica, para cada entero $j \geq 0$, la variable aleatoria igual al número de llamadas recursivas en la fase j . X_j puede tener un valor de hasta 0, porque una fase podría ser ignorada por completo y, desde luego, no puede ser mayor que n , el número máximo de llamadas recursivas

⁵Se puede realizar un análisis más refinado para demostrar un factor constante menor en el límite del tiempo de ejecución.

realizadas por RSELECT. Segundo (*), RSELECT realiza un máximo de

$$c \cdot \underbrace{\left(\frac{3}{4}\right)^j \cdot n}_{\substack{\text{longitud máxima del subarray} \\ (\text{fase } j)}}$$

operaciones en cada llamada recursiva de la fase j . Podemos descomponer el tiempo de ejecución de RSELECT a lo largo de las diferentes fases:

$$\begin{aligned} \text{tiempo de ejecución de RSELECT} &\leq \sum_{j \geq 0} \underbrace{X_j}_{\substack{\# \text{ de llamadas} \\ (\text{fase } j)}} \cdot \underbrace{c \left(\frac{3}{4}\right)^j n}_{\substack{\text{trabajo por llamada} \\ (\text{fase } j)}} \\ &= cn \sum_{j \geq 0} \left(\frac{3}{4}\right)^j X_j. \end{aligned}$$

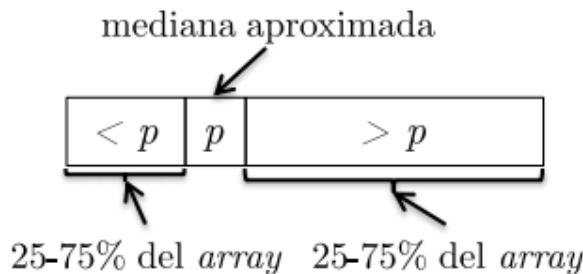
Este límite superior del tiempo de ejecución de RSELECT es una variable aleatoria compleja, pero también una suma ponderada de variables aleatorias más sencillas (las X_j). Tu respuesta automática, en este punto, debería ser la aplicación de la linealidad de la expectativa (teorema B.1), para reducir el cálculo de la variable aleatoria compleja al de las más sencillas:

$$E[\text{tiempo de ejecución de RSELECT}] \leq cn \sum_{j \geq 0} \left(\frac{3}{4}\right)^j E[X_j]. \quad (6.1)$$

Entonces, ¿qué es $E[X_j]$?

6.2.2 Reducción al lanzamiento de monedas

Para establecer el límite del número esperado $E[X_j]$ de llamadas recursivas en la fase j , tenemos que ocuparnos de dos cuestiones. En primer lugar, siempre que elegimos un buen pivote, pasamos a una fase posterior. Al igual que en la sección 5.4.3, definimos una *mediana aproximada* de un *subarray* como un elemento que es mayor que, al menos, el 25% del resto de elementos del *subarray* y también menor que el 25% de los otros elementos. La imagen queda así, después de realizar la partición en torno a ese elemento pivote:



Independientemente del caso que se active en RSELECT, la llamada recursiva recibe un *subarray* de longitud máxima $\frac{3}{4}$ del de la llamada anterior y, en consecuencia, pertenece a una fase posterior. Este argumento demuestra la siguiente proposición.

Proposición 6.2 (Las medianas aproximadas hacen progresos) *Si una llamada recursiva de la fase j elige una mediana aproximada como elemento pivote, entonces la siguiente llamada recursiva pertenece a la fase $j + 1$ o posterior.*

En segundo lugar, como se demostró en la sección 5.4.3, una llamada recursiva tiene bastantes probabilidades de elegir una mediana aproximada.

Proposición 6.3 (Hay abundancia de medianas aproximadas) *Una llamada a RSELECT elige una mediana aproximada como pivote con una probabilidad de, al menos, el 50%.*

Por ejemplo, en un *array* que contenga $\{1, 2, \dots, 100\}$, cada uno de los cincuenta elementos entre 26 y 75, ambos inclusive, son una mediana aproximada.

Las proposiciones 6.2 y 6.3 nos permiten sustituir un experimento sencillo de lanzamiento de monedas por el número de llamadas recursivas en la fase j . Supongamos que tenemos una moneda perfectamente equilibrada, con las mismas posibilidades de obtener cara o cruz. Lanzamos la moneda repetidamente, deteniéndonos la primera vez que obtengamos “cara”, y establecemos N como el número de lanzamientos realizados (incluyendo el último). Piensa en la “cara” como un equivalente de la elección de una mediana aproximada (lo que finaliza el experimento).

Proposición 6.4 (Reducción al lanzamiento de monedas) *Por cada fase $j = 0, 1, 2, \dots$, $E[X_j] \leq E[N]$.*

Demostración: Existen dos diferencias entre las variables aleatorias X_j y N , y ambas hacen que el valor esperado de la primera sea menor que el de

la segunda. En primer lugar, podría no haber llamadas recursivas en la fase j (en caso de que la fase sea ignorada por completo), mientras que siempre habrá un lanzamiento de monedas (el primero). En segundo lugar, cada lanzamiento de la moneda tiene una probabilidad exacta del 50% de dar por terminado el experimento (si sale cara). Mientras tanto, las proposiciones 6.2 y 6.3 implican que una llamada recursiva en la fase j finaliza la fase con una probabilidad del 50% o más: la elección de una mediana aproximada como pivote garantiza el final de la fase y, en muchos casos, otros elementos pivote también funcionarán correctamente. \mathcal{QED}

La variable aleatoria N se denomina *variable aleatoria geométrica con parámetro $\frac{1}{2}$* . Si buscamos su expectativa en un libro de texto o en internet, encontraremos que $E[N] = 2$. Además, podemos comprobarlo escribiendo el valor esperado de N en términos de sí mismo. La idea clave consiste en aprovecharse del hecho de que el experimento aleatorio no tiene memoria: si el primer lanzamiento de la moneda resulta en “cruz”, el resto del experimento es una copia del original. Según la matemática, sea cual sea el valor esperado de N , debe satisfacer la relación

$$E[N] = \underbrace{1}_{\text{primer lanzamiento}} + \underbrace{\frac{1}{2} \cdot E[N]}_{\Pr[\text{cruz}] \text{ más lanzamientos}}.$$

El único valor de $E[N]$ que satisface esta ecuación es 2 .⁶

La proposición 6.4 implica que este valor es el límite superior que nos interesa, el número esperado de llamadas recursivas en la fase j .

Corolario 6.5 (Dos llamadas por fase) *Para cada j , $E[X_j] \leq 2$.*

6.2.3 Lo juntamos todo

Ahora podemos utilizar el límite superior del corolario 6.5 sobre la $E[X_j]$, para simplificar nuestro límite superior (6.1) sobre el tiempo de ejecución esperado de RSELECT:

$$E[\text{tiempo de ejecución de RSELECT}] \leq cn \sum_{j \geq 0} \left(\frac{3}{4}\right)^j E[X_j] \leq 2cn \sum_{j \geq 0} \left(\frac{3}{4}\right)^j.$$

⁶Si queremos ser estrictos, también debemos descartar la posibilidad de que $E[N] = +\infty$ (lo que no es difícil).

El sumatorio $\sum_{j \geq 0} \left(\frac{3}{4}\right)^j$ puede parecer confuso, pero es una bestia que ya tenemos domesticada. Al demostrar el método maestro (sección 4.4), tomamos un desvío para tratar series geométricas (sección 4.4.8) y deducir la fórmula exacta (4.6):

$$1 + r + r^2 + \dots + r^k = \frac{1 - r^{k+1}}{1 - r}$$

para todo número real $r \neq 1$ y entero no negativo k . Cuando r es positivo y menor que 1, esta cantidad es, como máximo, de $\frac{1}{1-r}$, independientemente del tamaño de k . Si lo incorporamos a $r = \frac{3}{4}$, obtenemos

$$\sum_{j \geq 0} \left(\frac{3}{4}\right)^j \leq \frac{1}{1 - \frac{3}{4}} = 4,$$

y, por tanto,

$$E[\text{tiempo de ejecución de RSELECT}] \leq 8cn = O(n).$$

Así finaliza el análisis de RSELECT y la demostración del teorema 6.1. *QED*

*6.3 El algoritmo DSELECT

El algoritmo RSELECT se ejecuta con una expectativa de tiempo lineal en todos los casos, expectativa que surge de las elecciones aleatorias realizadas. ¿Es la aleatorización un requisito indispensable para la selección en tiempo lineal?⁷ Esta sección y la siguiente resuelven la cuestión, presentando un algoritmo determinista de tiempo lineal para el problema de selección.

En el problema de ordenación, el tiempo medio $O(n \log n)$ del QUICKSORT aleatorizado queda igualado por el algoritmo determinista MERGESORT, y tanto QUICKSORT como MERGESORT resultan muy útiles en la práctica. En contraste, mientras que el algoritmo de selección determinista de tiempo lineal de esta sección funciona correctamente, no es competitivo con RSELECT. Las dos razones que lo explican vienen dadas por los grandes factores constantes aplicados al tiempo de ejecución y por el trabajo realizado por DSELECT para reservar y gestionar memoria adicional. Aun así, las ideas que presenta el algoritmo son tan interesantes que no puedo evitar hablarte de ellas.

⁷La comprensión del poder de la aleatorización en el mundo de la computación es una cuestión compleja y, a día de hoy, sigue siendo un tema activo de investigación dentro de las ciencias de la computación teóricas.

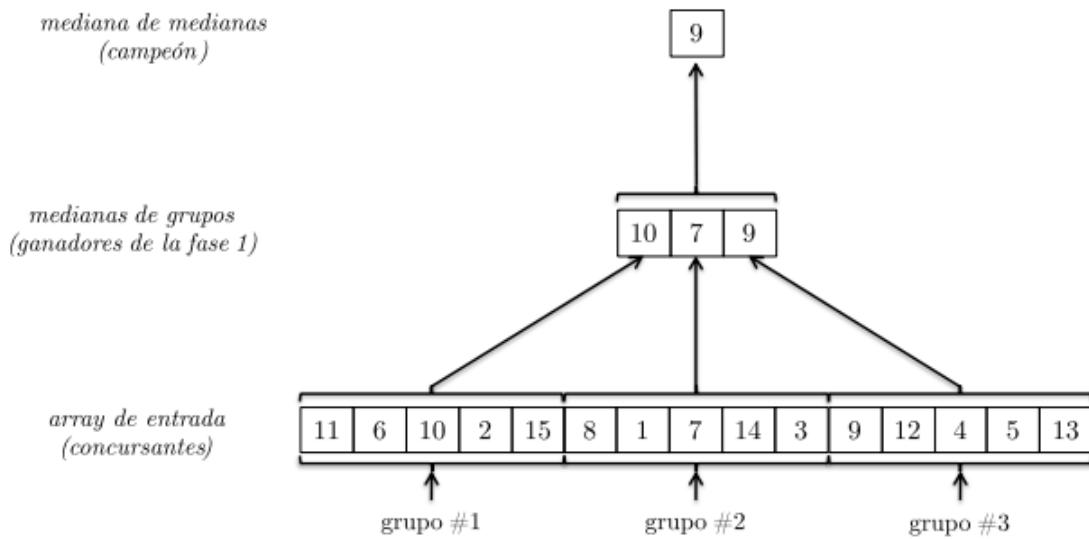


Figura 6.1: Cálculo de un elemento pivote con una eliminatoria a dos rondas. En este ejemplo, el pivote elegido no es la mediana del *array* de entrada, pero se le acerca mucho.

6.3.1 La gran idea: mediana de medianas

El algoritmo RSELECT es rápido porque los elementos pivote aleatorios tienden a ser bastante buenos, generando particiones más o menos equilibradas del *array* de entrada, y porque los buenos pivotes suponen un progreso rápido de las operaciones. Pero supongamos que no podemos utilizar la aleatorización. ¿Cómo podríamos calcular un buen elemento pivote sin que suponga mucho trabajo?

La gran idea de la selección determinista en tiempo lineal consiste en utilizar una “mediana de medianas”, como paso intermedio para llegar a la verdadera mediana. Este algoritmo trata los elementos del *array* de entrada como si fuesen equipos deportivos que jugasen una eliminatoria a dos rondas, siendo el campeón el elemento pivote. Ver también la figura 6.1.

La primera ronda es la fase de grupos, con los elementos de las posiciones 1–5 del *array* de entrada en el primer grupo, los elementos 6–10 en el segundo, etc. El ganador de la primera ronda de un grupo de 5 quedará definido como la mediana de ese grupo (el tercer elemento más pequeño). Como hay $\approx n/5$ grupos de 5, habrá $\approx n/5$ ganadores de la primera ronda (como es habitual, ignoraremos las fracciones para simplificar el proceso). El campeón del torneo será, entonces, la mediana de los ganadores de la primera ronda.

6.3.2 Pseudocódigo de DSELECT

¿Cómo calculamos la mediana de medianas? La implementación de la primera fase del torneo es sencilla, porque el cálculo de cada mediana solo implica a 5 elementos. Se puede, por ejemplo, realizar ese cálculo de forma ingenua (comprobando si cada uno de los 5 elementos es el central) o utilizando nuestra reducción a ordenación (sección 6.1.2). Para implementar la segunda fase, calculamos la mediana de los $\approx n/5$ ganadores de la primera ronda de forma recursiva.

DSELECT

Entrada: array A de $n \geq 1$ enteros distintos y un entero $i \in \{1, 2, \dots, n\}$.
Salida: el i -ésimo estadístico de orden de A .

```
1 si  $n = 1$  entonces                                // caso base
2     devolver  $A[1]$ 
3 para  $h := 1$  hasta  $n/5$  hacer    // ganadores de la primera fase
4      $C[h] :=$  elemento central del  $h$ -ésimo grupo de 5
5  $p :=$  DSELECT( $C, n/10$ )                      // mediana de medianas
6 particionar  $A$  en torno a  $p$ 
7  $j :=$  posición de  $p$  en el array particionado
8 si  $j = i$  entonces                            // has tenido suerte
9     devolver  $p$ 
10 si no, si  $j > i$  entonces
11     devolver DSELECT(primer parte de  $A, i$ )
12 en otro caso                                //  $j < i$ 
13     devolver DSELECT(segunda parte de  $A, i - j$ )
```

Las líneas 1–2 y 6–13 son idénticas a las de RSELECT. Las líneas 3–5 son las únicas nuevas en el algoritmo. Calculan la mediana de medianas del array de entrada, sustituyendo la línea de RSELECT que elige un pivote de forma aleatoria.

Las líneas 3 y 4 calculan los ganadores de la primera fase del torneo, determinando el elemento central de cada grupo de 5 de forma ingenua o

utilizando un algoritmo de ordenación, y los copia a un nuevo array C .⁸ La línea 5 determina el campeón del torneo calculando recursivamente la mediana de C . Como C tiene una longitud de (aproximadamente) $n/5$, este es el $(n/10)$ -ésimo estadístico de orden de C . No se utiliza la aleatorización en ningún paso del algoritmo.

6.3.3 Entender DSELECT

Puede parecer peligroso llamar recursivamente a DSELECT mientras se está calculando el elemento pivote. Para entender lo que ocurre, vamos a aclarar primero cuál es el número total de llamadas recursivas.

Cuestionario 6.3

¿Cuántas llamadas recursivas causará, normalmente, una sola llamada a DSELECT?

- a) 0
- b) 1
- c) 2
- d) 3

Solución y aclaraciones a continuación

Respuesta correcta: (c). Dejando a un lado el caso base y la situación excepcional en la que el elemento pivote resulte ser el estadístico de orden deseado, el algoritmo DSELECT realiza dos llamadas recursivas. No le des muchas vueltas para saber por qué, basta inspeccionar el pseudocódigo de DSELECT línea por línea. Hay una llamada recursiva en la línea 5 y otras más en las líneas 11 o 13.

Existen dos puntos de confusión habituales en estas dos llamadas recursivas. En primer lugar, ¿no es el hecho de que RSELECT realice una sola llamada recursiva el motivo de que sea más rápido que nuestros algoritmos

⁸Este array auxiliar es el motivo por el que DSELECT, a diferencia de RSELECT, no se puede ejecutar *in situ*.

de ordenación? ¿No está DSELECT abandonando esta mejora al hacer dos llamadas? La sección 6.4 indica que, como la llamada recursiva adicional de la línea 5 solo necesita resolver un subproblema relativamente pequeño (con un 20% de los elementos del *array* original), todavía podemos conservar el tiempo lineal.

En segundo lugar, las dos llamadas recursivas juegan papeles fundamentalmente distintos. El objetivo de la llamada recursiva de la línea 5 es el de identificar un buen elemento pivote para la llamada recursiva actual. El objetivo de la llamada en las líneas 11 o 13 es el normal, resolver recursivamente el problema residual más pequeño que nos ha dejado la llamada recursiva actual. En cualquier caso, la estructura recursiva de DSELECT entra de lleno en la tradición del resto de algoritmos de divide y vencerás que hemos estudiado: cada llamada recursiva realiza un pequeño número de llamadas recursivas adicionales sobre subproblemas estrictamente más pequeños, además de cierto trabajo adicional. Si no nos preocupa que algoritmos como MERGESORT o QUICKSORT entren en bucles infinitos, tampoco debería hacerlo DSELECT.

6.3.4 Tiempo de ejecución de DSELECT

El algoritmo DSELECT no es únicamente un programa bien definido que se ejecuta en un tiempo finito: se ejecuta en tiempo *lineal*, realizando un trabajo solamente un factor constante más grande que el necesario para leer la entrada.

Teorema 6.6 (Tiempo de ejecución de DSELECT) *Para todo array de entrada de longitud $n \geq 1$, el tiempo de ejecución de DSELECT es $O(n)$.*

A diferencia del tiempo de ejecución de RSELECT que, en principio, podría llegar a ser tan péjimo como $\Theta(n^2)$, el tiempo de ejecución de DSELECT siempre es $O(n)$. Aun así, en la práctica normalmente será más apropiado utilizar RSELECT que DSELECT, debido a que el primero se ejecuta *in situ* y la constante oculta en el tiempo medio de ejecución “ $O(n)$ ” del teorema 6.1 es menor que la constante oculta en el teorema 6.6.

Un superequipo de las ciencias de la computación

Uno de los objetivos de esta serie de libros es el de lograr que los algoritmos famosos se muestren tan sencillos

(al menos, en apariencia) que tengas la sensación de que podrías haberlos descubierto tú mismo, si hubieses estado en el lugar y época oportunos. Casi nadie tiene esa sensación al fijarse en DSELECT, que fue diseñado por un superequipo de las ciencias de la computación compuesto de cinco investigadores, cuatro de los cuales han sido galardonados con el Premio Turing de la ACM (y todos por logros diferentes), el equivalente al Premio Nobel de las ciencias de la computación⁹. A la vista de esto, no te inquietes si te resulta difícil imaginar un algoritmo como DSELECT, incluso en tus días más creativos: también es difícil imaginarse vencer a Roger Federer (no digamos ya a cinco copias de él) en una pista de tenis.

*6.4 Análisis de DSELECT

¿De verdad el algoritmo DSELECT se puede ejecutar en tiempo lineal? Aparentemente, realiza una cantidad de trabajo extravagante, con dos llamadas recursivas y un trabajo significativo fuera de ellas. Cualquier otro algoritmo con dos o más llamadas recursivas que haya visto tiene un tiempo de ejecución de $\Theta(n \log n)$ o peor.

6.4.1 Trabajo fuera de las llamadas recursivas

Comenzaremos tratando de entender el número de operaciones realizadas por una llamada a DSELECT fuera de sus llamadas recursivas. Los dos pasos que necesitan un trabajo significativo son el cálculo de los ganadores de la

⁹El algoritmo y su análisis fueron publicados en el artículo “*Time Bounds for Selection*”, de Manuel Blum, Robert W. Floyd, Vaughan Pratt, Ronald L. Rivest y Robert E. Tarjan (*Journal of Computer and System Sciences*, 1973). En aquella época era muy poco habitual encontrar artículos firmados por cinco autores. En orden cronológico: Floyd obtuvo el Premio Turing en 1978 por sus aportaciones a la algoritmia y a los lenguajes de programación y compiladores, Tarjan fue reconocido en 1986 (junto a John E. Hopcroft) por su trabajo sobre algoritmos y estructuras de datos, Blum fue galardonado en 1995, principalmente por sus aportaciones a la criptografía, y Rivest, a quien quizás conozcas por ser la “R” en el sistema criptográfico RSA, lo obtuvo en 2002 (junto a Leonard Adleman y Adi Shamir) por su trabajo sobre la criptografía de clave pública. Al mismo tiempo, Pratt es famoso por un abanico de logros que se extiende desde algoritmos de prueba de primalidad hasta ser el cofundador de *Sun Microsystems*.

primera fase (líneas 3 y 4) y la partición del *array* de entrada en torno a la mediana de medianas (línea 6). Al igual que en **QUICKSORT** o **RSELECT**, el segundo paso se ejecuta en tiempo lineal. Pero, ¿qué ocurre con el primero?

Nos centraremos en un grupo concreto de 5 elementos. Como se trata de un número constante de elementos (con independencia de la longitud n del *array* de entrada), el tiempo de cálculo de la mediana también es constante. Por ejemplo, vamos a suponer que realizamos este cálculo mediante una reducción a ordenación (sección 6.1.2), utilizando, por ejemplo, **MERGESORT**. A estas alturas ya tenemos muy claro el trabajo que realiza **MERGESORT** (teorema 1.2): un máximo de

$$6m(\log_2 m + 1)$$

operaciones para ordenar un *array* de longitud m . Quizá te preocupe el hecho de que **MERGESORT** no se ejecuta en tiempo lineal. Pero lo estamos utilizando únicamente con *subarrays* de tamaño constante ($m = 5$) y, en consecuencia, realiza un número constante de operaciones (un máximo de $6 \cdot 5 \cdot (\log_2 5 + 1) \leq 120$) por *subarray*. Si sumamos los $n/5$ grupos de 5 que necesitamos ordenar, obtenemos un máximo de $120 \cdot \frac{n}{5} = 24n = O(n)$ operaciones en total. En conclusión, fuera de sus llamadas recursivas, el algoritmo **DSELECT** solo realiza una cantidad de trabajo lineal.

6.4.2 Una recurrencia un poco tosca

En el capítulo 4 analizamos los algoritmos de divide y vencerás utilizando recurrencias, lo que expresaba un límite del tiempo de ejecución de $T(n)$ en términos del número de operaciones realizadas por las llamadas recursivas. Vamos a intentar repetir el proceso, siendo $T(n)$ el número máximo de operaciones que realiza el algoritmo **DSELECT** sobre un *array* de entrada de longitud n . Cuando $n = 1$, el algoritmo **DSELECT** se limita a devolver un *array* de un único elemento, por lo que $T(1) = 1$. Para n más grandes, el algoritmo **DSELECT** realiza una llamada recursiva en la línea 5, otra en las líneas 11 o 13 y un trabajo adicional de $O(n)$ (para particionar, calcular y copiar los ganadores de la primera fase). Esto se traduce a una recurrencia de la forma

$$T(n) \leq \underbrace{T(\text{tamaño del subproblema 1})}_{=n/5} + \underbrace{T(\text{tamaño del subproblema 2})}_{=?} + O(n).$$

Para evaluar el tiempo de ejecución de **DSELECT**, debemos entender los tamaños de los subproblemas resueltos por sus dos llamadas recursivas. El

tamaño del primer subproblema (línea 5) es $n/5$, el número de ganadores de la primera fase. No conocemos el tamaño del segundo subproblema, pues depende de qué elemento sea el pivote y de si el estadístico de orden que buscamos es mayor o menor que ese pivote. El no poder determinar el tamaño de este subproblema es la razón por la que no utilizamos recurrencias para analizar los algoritmos **QUICKSORT** y **RSELECT**.

En el caso especial de que el elemento elegido como pivote sea la verdadera mediana del *array* de entrada, tendremos la garantía de que el segundo subproblema está formado por un máximo de $n/2$ elementos. En general, la mediana de medianas no suele ser la verdadera mediana (figura 6.1). Pero, ¿se aproxima lo suficiente como para asegurar una división más o menos equilibrada del *array* de entrada y, en consecuencia, un subproblema no demasiado grande en las líneas 11 o 13?

6.4.3 El lema del 30-70

El siguiente lema es el alma del análisis de **DSELECT**, pues cuantifica la recompensa por el duro trabajo realizado para calcular la mediana de medianas: este elemento pivote garantiza una partición del *array* de entrada del 30%-70% o mejor.

Lema 6.7 (Lema del 30-70) *Por cada array de entrada de longitud $n \geq 2$, el subarray que recibe la llamada recursiva de las líneas 11 o 13 de **DSELECT** tiene una longitud máxima de $\frac{7}{10}n$.¹⁰*

El lema del 30-70 nos permite sustituir “ $\frac{7}{10}n$ ” por “?” en la recurrencia un tanto tosca que hemos visto antes: para cada $n \geq 2$,

$$T(n) \leq T\left(\frac{1}{5} \cdot n\right) + T\left(\frac{7}{10} \cdot n\right) + O(n). \quad (6.2)$$

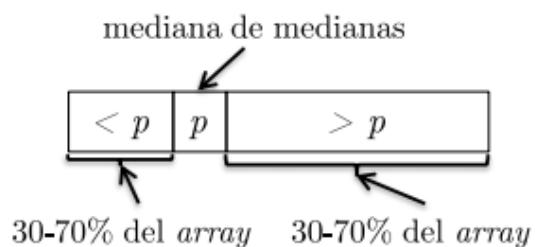
Demostraremos primero el lema del 30-70 y, a continuación, haremos lo propio con el hecho de que la recurrencia (6.2) implica que **DSELECT** es un algoritmo de tiempo lineal.

Demostración del lema 6.7: Digamos que $k = n/5$ indica el número de grupos de 5 y, con ello, el número de ganadores de la primera fase. Definimos

¹⁰Siendo estrictos, debido a que uno de los “grupos de 5” podría tener menos de cinco elementos (si n no es múltiplo de 5), $\frac{7}{10}n$ debería ser $\frac{7}{10}n + 2$, redondeado al entero más cercano. Ignoraremos el “+2” por el mismo motivo que ignoramos las fracciones: es un detalle que complica el análisis de forma poco constructiva y, en el fondo, no tiene un impacto real.

x_i como el ganador i -ésimo más pequeño de la primera fase. Igualmente, $x_1, x_2 \dots, x_k$ indican los ganadores de la primera fase, indexados en orden creciente. El campeón del torneo, la mediana de medianas, es $x_{k/2}$ (o $x_{\lceil k/2 \rceil}$, si k es impar¹¹).

El plan consiste en considerar que $x_{k/2}$ es mayor que, al menos, el 60% de los elementos en, al menos, el 50% de los grupos de 5, y que es menor que, al menos, el 60% de los elementos en, al menos, el 50% de los grupos. Entonces, al menos el $60\% \cdot 50\% = 30\%$ de los elementos del *array* de entrada serán menores que la mediana de medianas y, al menos, el 30% serán mayores:

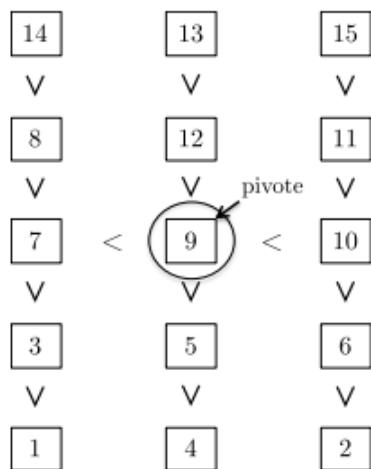


Para implementar este plan, podemos llevar a cabo un ejercicio mental. Coloquemos en nuestra imaginación (y no en el algoritmo) todos los elementos del *array* de entrada en una rejilla bidimensional. Hay cinco filas y cada una de las $n/5$ columnas corresponde a uno de los grupos de 5. Dentro de cada columna, colocamos los 5 elementos ordenados de abajo a arriba. Por último, disponemos las columnas de forma que los ganadores de la primera fase (es decir, los elementos de la fila central) estén ordenados de izquierda a derecha. Por ejemplo, si el *array* de entrada es

11	6	10	2	15	8	1	7	14	3	9	12	4	5	13
----	---	----	---	----	---	---	---	----	---	---	----	---	---	----

la rejilla correspondiente será

¹¹La notación $[x]$ indica la función “techo”, que redondea su argumento al entero superior más cercano.



con el pivote, la mediana de medianas, en la posición central.

Observación importante

Como la fila central está ordenada de izquierda a derecha y cada columna de abajo a arriba, *todos los elementos a la izquierda y por debajo del pivote son menores que el mismo, mientras que todos los elementos a la derecha y por encima son mayores*.

En nuestro ejemplo, el pivote es el “9”, los elementos a la izquierda y debajo son $\{1, 3, 4, 5, 7\}$ y los elementos a la derecha y encima son $\{10, 11, 12, 13, 15\}$.¹² Por tanto habrá, al menos, 6 elementos excluidos del subarray que pasará a la siguiente llamada recursiva: el elemento pivote 9 y bien $\{10, 11, 12, 13, 15\}$ (en la línea 11), o bien $\{1, 3, 4, 5, 7\}$ (en la línea 13). En cualquier caso, la siguiente llamada recursiva recibirá un máximo de 9 elementos, y 9 es menos que el 70% de 15.

El argumento es el mismo en el caso general. La figura 6.2 describe el aspecto de la rejilla con un *array* de entrada arbitrario. Como el elemento pivote es la mediana de los elementos de la fila central, al menos el 50% de las columnas están a la izquierda de la que contiene al pivote (contando la propia columna del pivote). En cada una de estas columnas, al menos el 60% de los elementos (los tres más pequeños de los cinco) no son mayores que la mediana de la columna y, por tanto, no son mayores que la mediana de medianas. En consecuencia, al menos el 30% de los elementos del *array*

¹²Vemos que los elementos a la izquierda y encima o a la derecha y debajo pueden ser menores o mayores que el pivote.

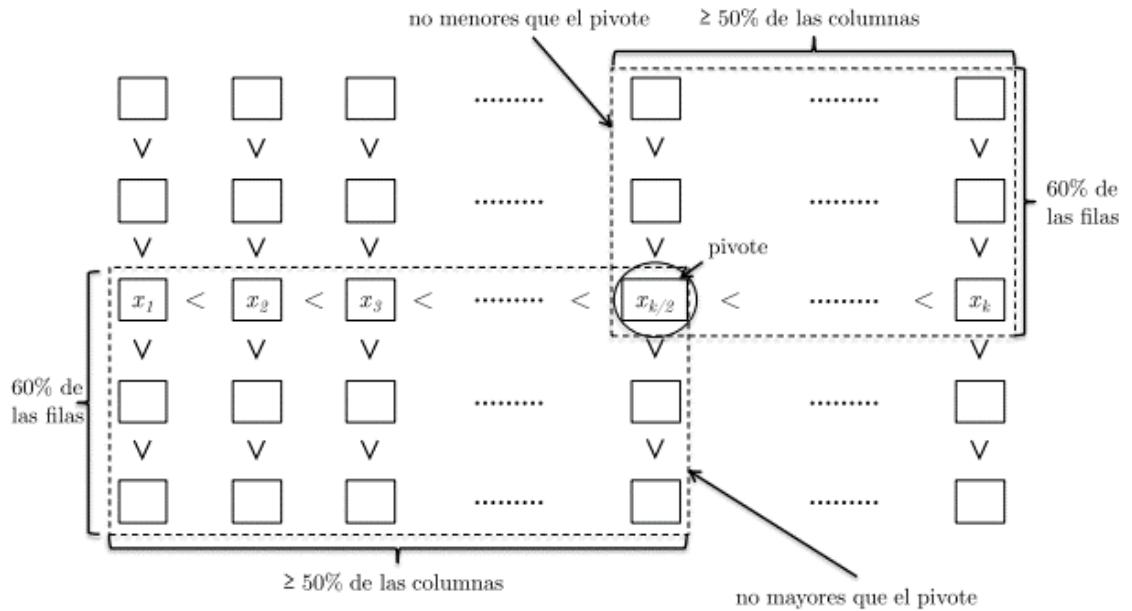


Figura 6.2: Demostración del lema del 30-70. Imaginemos los elementos del *array* de entrada distribuidos como una rejilla. Cada columna corresponde a un grupo de 5, ordenados de abajo a arriba. Las columnas están ordenadas según sus elementos centrales. La imagen asume que k es par. En caso de que k fuese impar, " $x_{k/2}$ " sería sustituido por " $x_{\lceil k/2 \rceil}$ ". Los elementos al suroeste de la mediana de medianas son menores que esta y los que están al noreste son mayores. Como resultado, al menos el $60\% \cdot 50\% = 30\%$ de los elementos quedan excluidos de cada una de las dos posibles llamadas recursivas.

de entrada no son mayores que el elemento pivote (contando al mismo pivote) y todos ellos serán excluidos en la llamada recursiva de la línea 13. Igualmente, al menos el 30% de los elementos no son menores que el pivote (contando también al pivote) y serán excluidos de la llamada recursiva de la línea 11. Con esto finaliza la demostración del lema del 30-70. \mathcal{QED}

6.4.4 Solución de la recurrencia

El lema del 30-70 implica que el tamaño de entrada se reduce por un factor constante en cada llamada recursiva de DSELECT, lo que casa bien con un tiempo de ejecución lineal. Pero, ¿estamos ante una victoria pírrica? ¿Resulta que el coste de calcular la mediana de medianas supera a los beneficios de realizar las particiones en torno a un buen pivote? La respuesta a estas preguntas, así como la finalización de la demostración del teorema 6.6, requiere obtener la solución a la recurrencia de (6.2).

Como el algoritmo DSELECT realiza una cantidad de trabajo lineal ($O(n)$) fuera de sus llamadas recursivas (cálculo de los ganadores de la primera fase, partición del *array*, etc.), existe una constante $c > 0$ tal que, para todos $n \geq 2$,

$$T(n) \leq T\left(\frac{1}{5} \cdot n\right) + T\left(\frac{7}{10} \cdot n\right) + cn, \quad (6.3)$$

donde $T(n)$ es el límite superior del tiempo de ejecución de DSELECT sobre *arrays* de longitud n . Podemos asumir que $c \geq 1$ (ya que incrementar c no puede invalidar la desigualdad (6.3)). Además, $T(1) = 1$. Como veremos, la propiedad más importante de esta recurrencia es que $\frac{1}{5} + \frac{7}{10} < 1$.

En el método maestro (capítulo 4) nos inclinamos por evaluar todas las recurrencias encontradas hasta el momento. En MERGESORT, KARATSUBA, STRASSEN y otros, nos limitamos a introducir los tres parámetros relevantes (a , b y d) y obtener un límite superior para el tiempo de ejecución del algoritmo. Por desgracia, las dos llamadas recursivas de DSELECT cuentan con tamaños de entrada diferentes, lo que descarta la aplicación del teorema 4.1. Aunque sería posible generalizar el argumento del árbol de recrusión del teorema 4.1, para admitir la recurrencia de (6.3). Para ampliar la variedad de herramientas a nuestra disposición, utilizaremos un método diferente¹³.

¹³Como argumento heurístico, puedes pensar en el primer par de llamadas recursivas a DSELECT: los dos nodos del nivel 1 del árbol de recrusión del algoritmo. Uno contiene un 20% de los elementos del *array* de entrada, el otro un máximo del 70% y el trabajo realizado en este nivel es lineal en relación a la suma de los tamaños de los dos subproblemas. Por

6.4.5 El método probar–comprobar

El método *probar–comprobar* para evaluar recurrencias es tan *ad hoc* como podría deducirse de su nombre, pero también resulta extraordinariamente flexible y se puede aplicar a recurrencias disparatadamente arbitrarias.

Paso 1: probar. Prueba a utilizar una función $f(n)$ que, presumiblemente, satisface $T(n) = O(f(n))$.

Paso 2: comprobar. Demuestra por inducción sobre n que $T(n)$ es, en realidad, $O(f(n))$.

En general, el paso de prueba resulta ser un arte un tanto oscuro. En nuestro caso, como estamos intentando demostrar un límite de tiempo de ejecución lineal, plantearemos que $T(n) = O(n)$.¹⁴ Es decir, planteamos que existe una constante $\ell > 0$ (independiente de n) tal que

$$T(n) \leq \ell \cdot n \quad (6.4)$$

para cada posible entero positivo de n . Si es cierto y como ℓ es una constante, esto implicaría nuestro deseo de que $T(n) = O(n)$.

Al verificar (6.4), tendremos la libertad de elegir ℓ de la forma que más nos guste, siempre que sea independiente de n . Al igual que en la demostración con notación asintótica, el método más normal para determinar cuál puede ser la constante apropiada pasa por la ingeniería inversa (comparar con la sección 2.5). En este caso, tomaremos $\ell = 10c$, donde c es el factor constante de la recurrencia (6.3) (porque c es una constante, al igual que ℓ). ¿De dónde viene este número? Es la constante más pequeña con la que la desigualdad (6.5), que veremos más adelante, es válida.

Demostramos (6.4) por inducción. Según la terminología del apéndice A, $P(n)$ es la afirmación de que $T(n) \leq \ell \cdot n = 10c \cdot n$. Para el caso base, necesitamos demostrar directamente que $P(1)$ es cierto, lo que significa que $T(1) \leq 10c$. La recurrencia indica explícitamente que $T(1) = 1$ y $c \geq 1$, por lo que tenemos la certeza de que $T(1) \leq 10c$.

tanto, la cantidad de trabajo realizado en el nivel 1 será, como mucho, del 90% del realizado en el nivel 0, y el mismo principio se aplica al resto de niveles. Esto guarda un parecido con el segundo caso del método maestro, en el que la cantidad de trabajo se reduce por un factor constante en cada nivel. La analogía sugiere que el trabajo $O(n)$ realizado en la raíz debería determinar el tiempo de ejecución (comparar con la sección 4.4.6).

¹⁴“Rezar y comprobar” podría ser una descripción más apropiada en este caso.

Para el paso inductivo, establecemos un entero positivo arbitrario $n \geq 2$. Necesitamos demostrar que $T(n) \leq \ell \cdot n$. La hipótesis inductiva plantea que $P(1), P(2), \dots, P(n-1)$ son ciertos, lo que significa que $T(k) \leq \ell \cdot k$ para todos los $k < n$. Sigue tu instinto para demostrar $P(n)$.

En primer lugar, la recurrencia (6.3) descompone $T(n)$ en tres términos:

$$T(n) \leq \underbrace{T\left(\frac{1}{5} \cdot n\right)}_{\leq \ell \cdot \frac{n}{5} \text{ (hip. ind.)}} + \underbrace{T\left(\frac{7}{10} \cdot n\right)}_{\leq \ell \cdot \frac{7n}{10} \text{ (hip. ind.)}} + cn.$$

No podemos manipular directamente ninguno de estos términos, pero podemos aplicar la hipótesis inductiva, una vez con $k = n/5$ y otra con $k = 7n/10$:

$$T(n) \leq \ell \cdot \frac{n}{5} + \ell \cdot \frac{7n}{10} + cn.$$

Agrupando los términos,

$$T(n) \leq n \underbrace{\left(\frac{9}{10}\ell + c\right)}_{=\ell \text{ (como } \ell=10c\text{)}} = \ell \cdot n. \quad (6.5)$$

Esto demuestra el paso inductivo, que verifica que $T(n) = O(n)$ y completa la demostración de que el ingenioso algoritmo DSELECT se ejecuta en tiempo lineal (teorema 6.6). \mathcal{QED}

Conclusiones

- ★ El problema de selección busca calcular el elemento i -ésimo más pequeño de un *array* desordenado.
- ★ El problema de selección se puede resolver en tiempo $O(n \log n)$, donde n es la longitud del *array* de entrada, ordenando el *array* y devolviendo el elemento i -ésimo.
- ★ El problema también se puede resolver particionando el *array* de entrada en torno a un elemento pivote, al igual que en QUICKSORT, y ejecutando una recurrencia sobre el lado relevante. El algoritmo RSELECT elige el elemento pivote aleatoriamente.

- ★ El tiempo de ejecución de RSELECT varía entre $\Theta(n)$ y $\Theta(n^2)$, dependiendo de los pivotes elegidos.
- ★ El tiempo de ejecución promedio de RSELECT es $\Theta(n)$. La demostración utiliza una reducción a un experimento de lanzamiento de monedas.
- ★ La gran idea del algoritmo determinista DSELECT es la utilización de la “mediana de medianas” como elemento pivote: se divide el *array* de entrada en grupos de 5, se calcula directamente la mediana de cada uno de esos grupos y, de forma recursiva, se calcula la mediana de los $n/5$ ganadores de la primera fase.
- ★ El lema del 30-70 muestra que la mediana de medianas garantiza una partición del 30%-70%, o mejor, del *array* de entrada.
- ★ El análisis de DSELECT muestra que el trabajo empleado en la llamada recursiva que calcula la mediana de medianas es superado por el beneficio de una división 30%-70%, lo que resulta en un tiempo lineal.

Comprueba que lo has entendido

Problema 6.1 (S) ¿Cuál de las siguientes afirmaciones es cierta (elige todas las que lo sean)?

- Si el problema de ordenación se puede resolver en tiempo $T(n)$, entonces el problema de selección se puede resolver en tiempo $O(T(n))$.
- Si el problema de selección se puede resolver en tiempo $T(n)$, entonces el problema de ordenación se puede resolver en tiempo $O(T(n))$.
- El uso de la aleatorización no mejora el peor caso de tiempo de ejecución asintótico de los algoritmos para el problema de selección.
- El uso de la aleatorización no mejora la utilidad de los algoritmos para el problema de selección.

Problema 6.2 (S) Digamos que α es una constante, independiente de la longitud n del *array* de entrada, y con un valor estrictamente entre $\frac{1}{2}$ y 1. Supón que estás utilizando el algoritmo RSELECT para calcular el elemento correspondiente a la mediana de un *array* de longitud n . ¿Cuál es la probabilidad de que la primera llamada recursiva reciba un *subarray* con una longitud máxima de αn ?

- a) $1 - \alpha$
- b) $\alpha - \frac{1}{2}$
- c) $1 - \frac{\alpha}{2}$
- d) $2\alpha - 1$

Problema 6.3 Digamos que α es una constante, independiente de la longitud n del *array* de entrada, y con un valor estrictamente entre $\frac{1}{2}$ y 1. Asumimos que cada llamada recursiva a RSELECT realiza progresos, igual que en el problema anterior, de forma que siempre que una llamada recursiva recibe un *array* de longitud k , su propia llamada recursiva recibirá un *subarray* con una longitud máxima de αk . ¿Cuál es el número máximo de llamadas recursivas sucesivas que se pueden producir antes de que se active el caso base?

- a) $-\frac{\ln n}{\ln \alpha}$
- b) $-\frac{\ln n}{\alpha}$
- c) $-\frac{\ln n}{\ln(1-\alpha)}$
- d) $-\frac{\ln n}{\ln(\frac{1}{2} + \alpha)}$

Problemas más difíciles

Problema 6.4 (P) Supón que modificamos el algoritmo DSELECT separando los elementos en grupos de 7, en vez de grupos de 5 (utiliza la mediana de medianas como elemento pivote, igual que hacíamos antes). ¿Se ejecutará este algoritmo modificado también en tiempo $O(n)$? ¿Y si usamos grupos de 3?

Problema 6.5 (*P*) Recibes como entrada un *array* desordenado de n enteros, en el que puede haber entradas duplicadas. Proporciona un algoritmo determinista de tiempo $O(n)$ que identifique todo valor que aparezca más de $n/4$ en ese *array* (si es que hay alguno). Tanto en este problema como en el siguiente, puedes utilizar como subrutina cualquiera de los algoritmos que hemos visto en el capítulo.

Problema 6.6 (*S*) En este problema, la entrada es un *array* desordenado de n elementos diferentes x_1, x_2, \dots, x_n con pesos positivos w_1, w_2, \dots, w_n . Digamos que W indica la suma $\sum_{i=1}^n w_i$ de los pesos. Define una *mediana ponderada* como un elemento x_k para el que el peso total de todos los elementos con valores menores que x_k (es decir, $\sum_{x_i < x_k} w_i$) sea, como máximo, de $W/2$ y, además, el peso total de los elementos con valores mayores que x_k (es decir, $\sum_{x_i > x_k} w_i$) sea, como máximo, de $W/2$. Observa que habrá una o dos medianas ponderadas. Proporciona un algoritmo determinista de tiempo lineal para calcular la mediana ponderada del *array* de entrada.

Problemas de programación

Problema 6.7 Implementa, utilizando tu lenguaje de programación favorito, el algoritmo RSELECT de la sección 6.1. La implementación debe operar *in situ*, utilizando una implementación también *in situ* de PARTITION (que quizás ya hayas desarrollado para el problema 5.8), y debe pasar los índices a las llamadas recursivas, para mantener un registro de la porción del *array* de entrada original que sigue siendo relevante (encontrarás casos de prueba y conjuntos de datos en www.algorithmsilluminated.org).

Un vistazo a las demostraciones por inducción

Las demostraciones por inducción aparecen *constantemente* en el campo de las ciencias de la computación. Por ejemplo, en la sección 5.1, utilizamos una demostración por inducción para afirmar que el algoritmo `QUICKSORT` ordena siempre correctamente su *array* de entrada. En la sección 6.4, utilizamos la inducción para demostrar que el algoritmo `DSELECT` funciona en tiempo lineal.

Las demostraciones por inducción pueden resultar poco intuitivas, al menos a primera vista. La buena noticia es que siguen al pie de la letra una plantilla bastante estricta y, con un poco de práctica, son casi automáticas. Este apéndice explica esa plantilla y proporciona dos ejemplos breves. Si nunca has tenido contacto con las demostraciones por inducción, deberías complementar este apéndice con algún otro material que plantee más ejemplos¹.

A.1 Plantilla para demostraciones por inducción

Aunque no lo sepas, probablemente ya estás bastante familiarizado con las demostraciones por inducción. Por ejemplo, si entiendes las recursiones significa que también entiendes la inducción. O, ¿alguna vez has finalizado una explicación con la frase “y, así, sucesivamente” para indicar la repetición infinita de un patrón? En esencia, la inducción no es más que una traducción a las matemáticas de este recurso retórico.

A nuestros efectos, una demostración por inducción establece una afirmación $P(n)$ para cada entero positivo n . Por ejemplo, al demostrar la co-

¹Por ejemplo, el libro gratuito *Mathematics for Computer Science*, de Eric Lehman, F. Thomson Leighton y Albert R. Meyer.

rrección del algoritmo `QUICKSORT` en la sección 5.1, podemos definir, para cada $n \geq 1$, la afirmación $P(n)$ como:

“`QUICKSORT` ordena correctamente todos los *arrays* de entrada de longitud n .”

Al analizar el tiempo de ejecución del algoritmo `DSELECT`, en la sección 6.4, podemos definir la afirmación $P(n)$, para todos los $n \geq 1$, como:

“para cada *array* de entrada de longitud n , `DSELECT` se detiene después de realizar un máximo de $100n$ operaciones.”

La inducción nos permite demostrar una propiedad de un algoritmo, como la corrección o un tiempo límite de ejecución, determinando, a cambio, la propiedad para cada longitud de entrada.

De forma análoga a un algoritmo recursivo, una demostración por inducción consta de dos partes: un *caso base* y un *paso inductivo*. El caso base demuestra que $P(n)$ es cierto para todos los valores de n lo suficientemente pequeños (habitualmente solo para $n = 1$). En el paso inductivo, por cada valor mayor de n , podemos asumir que $P(1), P(2), \dots, P(n - 1)$ son verdaderos y demostrar que $P(n)$ es, en consecuencia, también verdadero.

Caso base: demostrar directamente que $P(1)$ es verdadero.

Paso inductivo: demostrar que, para cada entero $n \geq 2$,

si $\underbrace{P(1), P(2), \dots, P(n - 1)}$ son verdaderos entonces $P(n)$ es verdadero.
hipótesis inductiva

Debes *asumir*, en el paso inductivo, que $P(k)$ ya ha sido determinado para todos los valores de k menores que n (esto es lo que se denomina *hipótesis inductiva*), y utilizar esta asunción para establecer $P(n)$.

Si demuestras tanto el caso base como el paso inductivo, entonces $P(n)$ es, de hecho, verdadero para todo entero positivo $n \geq 1$: $P(1)$ es cierto por el caso base y, como un dominó que cae, la aplicación del paso inductivo una y otra vez demuestra que $P(n)$ es verdadero para valores arbitrariamente grandes de n .

A.2 Ejemplo: una fórmula cerrada

Podemos utilizar la inducción para deducir una fórmula cerrada para la suma de los primeros n enteros positivos. Digamos que $P(n)$ indica la afirmación de que

$$1 + 2 + 3 + \dots + n = \frac{(n + 1)n}{2}.$$

Cuando $n = 1$, el lado izquierdo es 1 y el derecho es $\frac{2 \cdot 1}{2} = 1$. Esto demuestra que $P(1)$ es verdadero y completa el caso base. Para el caso inductivo, elegimos un entero $n \geq 2$ arbitrario y damos por hecho que $P(1), P(2), \dots, P(n - 1)$ son verdaderos. En concreto, podemos asumir que $P(n - 1)$, que corresponde a la afirmación

$$1 + 2 + 3 + \dots + (n - 1) = \frac{n(n - 1)}{2}.$$

Ahora podemos sumar n a ambos lados para deducir

$$1 + 2 + 3 + \dots + n = \frac{n(n - 1)}{2} + n = \frac{n^2 - n + 2n}{2} = \frac{(n + 1)n}{2},$$

lo que demuestra $P(n)$. Como hemos establecido tanto el caso base como el paso inductivo, podemos concluir que $P(n)$ es verdadero para cualquier entero positivo n .

A.3 Ejemplo: el tamaño de un árbol binario completo

A continuación, vamos a contar el número de nodos de un árbol binario completo con n niveles. Podemos ver, en la figura A.1, que con $n = 4$ niveles, el número de nodos es $15 = 2^4 - 1$. ¿Podremos aplicar este patrón de forma general?

Para cada entero positivo n , digamos que $P(n)$ es la afirmación:

“un árbol binario completo con n niveles tiene $2^n - 1$ nodos.”

En el caso base, nos damos cuenta de que un árbol binario completo de 1 nivel tiene, exactamente, un nodo. Como $2^1 - 1 = 1$, esto demuestra que $P(1)$ es verdadero. Para el paso inductivo, establecemos un entero positivo $n \geq 2$ y asumimos que $P(1), P(2), \dots, P(n - 1)$ son verdaderos. Los nodos del árbol binario completo con n niveles se pueden dividir en tres grupos: (i) la

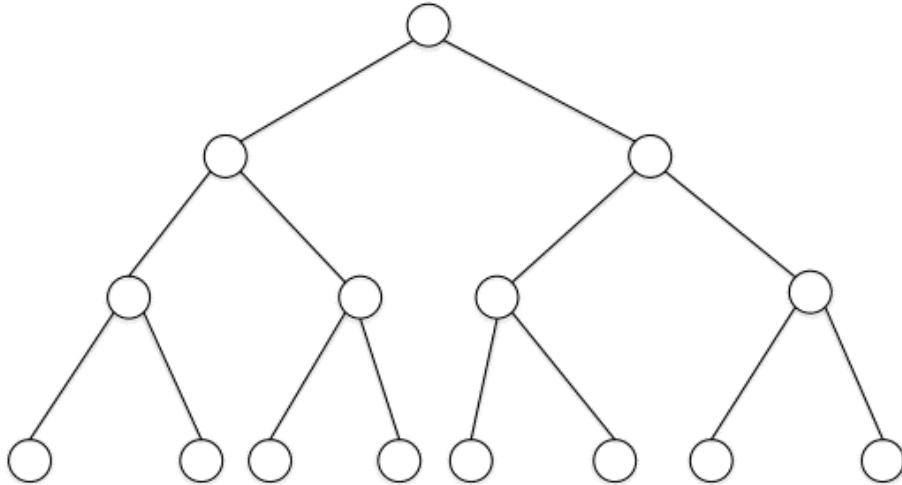


Figura A.1: Un árbol binario completo con 4 niveles y $2^4 - 1 = 15$ nodos.

raíz, (ii) los nodos en el subárbol izquierdo de la raíz y (iii) los nodos en el subárbol derecho de la raíz. Los subárboles izquierdo y derecho de la raíz son, en sí mismos, árboles binarios completos, cada uno con $n - 1$ niveles. Como asumimos que $P(n-1)$ es verdadero, hay exactamente $2^{n-1} - 1$ nodos en cada uno de los subárboles izquierdo y derecho. Sumando los nodos de los tres grupos, obtenemos un total de

$$\underbrace{1}_{\text{raíz}} + \underbrace{2^{n-1} - 1}_{\text{subárbol izquierdo}} + \underbrace{2^{n-1} - 1}_{\text{subárbol derecho}} = 2^n - 1$$

nodos en el árbol. Esto demuestra la afirmación $P(n)$ y, como $n \geq 2$ era arbitrario, completa el paso inductivo. Podemos concluir que $P(n)$ es verdadero para todo entero positivo n .

Un vistazo a la probabilidad discreta

Este apéndice revisa cinco conceptos de la probabilidad discreta que resultan necesarios para nuestro análisis del `QUICKSORT` aleatorizado (teorema 5.1 y sección 5.5) y `RSELECT` (teorema 6.1 y sección 6.2): espacios de muestra, eventos, variables aleatorias, expectativa y linealidad de la expectativa. La sección B.6 pone punto y final con un ejemplo de balanceo de carga que relaciona todos estos conceptos entre sí.

Si es la primera vez que entras en contacto con esta materia, quizá te convenga complementar este apéndice con un conocimiento más profundo¹. Si ya tienes un conocimiento previo, no te sientas obligado a leer este apéndice de arriba a abajo, basta con que le eches un vistazo siempre que necesites refrescar algún concepto.

B.1 Espacios de muestra

Nos interesan los procesos aleatorios, en los que puede ocurrir cualquier cosa. El *espacio de muestra* es el conjunto Ω de todas las situaciones diferentes con las que nos podríamos encontrar: el universo en el que vamos a asignar probabilidades, tomar valores promedio, etc². Por ejemplo, si nuestro proceso aleatorio corresponde al lanzamiento de un dado de seis caras, entonces $\Omega = \{1, 2, 3, 4, 5, 6\}$. Por suerte, en el análisis de algoritmos aleatorizados, Ω es casi siempre un conjunto finito que nos permite trabajar con probabilidad discreta, que es mucho más sencilla que la teoría de la probabilidad general.

¹Existe, por ejemplo, un *wikilibro* gratuito sobre probabilidad discreta (https://en.wikibooks.org/wiki/High_School_Mathematics_Extensions/Discrete_Probability).

²Además de su uso en la notación *Big-Omega* (sección 2.4.1), el símbolo Ω resulta ser, lamentablemente, el más utilizado para indicar un espacio de muestra.

Cada elemento i de un espacio de muestra Ω viene acompañado de una *probabilidad* no negativa $p(i)$, que podemos ver como la frecuencia con la que el resultado del proceso aleatorio es i . Por ejemplo, si un dado de seis caras está correctamente equilibrado, entonces $p(i)$ es $1/6$ para cada $i = 1, 2, 3, 4, 5, 6$. En general, como Ω captura cualquier situación que se pueda producir, las probabilidades deberían sumar 1:

$$\sum_{i \in \Omega} p(i) = 1.$$

Un caso especial bastante habitual se produce cuando cada elemento de Ω tiene la misma probabilidad de aparecer (lo que se conoce como *distribución uniforme*), en cuyo caso $p(i) = 1/|\Omega|$ para todos $i \in \Omega$.³

Los espacios de muestra pueden parecer un concepto bastante abstracto, así que presentaremos dos ejemplos concretos. En el primero, el proceso aleatorio consiste en el lanzamiento de dos dados normales (de seis caras). El espacio de muestra es el conjunto de las 36 situaciones que se podrían producir:

$$\Omega = \underbrace{\{(1, 1), (2, 1), (3, 1), \dots, (5, 6), (6, 6)\}}_{36 \text{ pares ordenados}}.$$

Asumiendo que los dados estén bien equilibrados, cada uno de estos resultados aparecerá con la misma probabilidad: $p(i) = 1/36$ para cada $i \in \Omega$.

El segundo ejemplo, más relacionado con los algoritmos, es la elección del elemento pivote en la llamada más exterior al `QUICKSORT` aleatorizado (sección 5.4). Cualquier elemento del *array* de entrada puede ser elegido como pivote, por tanto

$$\Omega = \underbrace{\{1, 2, 3, \dots, n\}}_{\text{posibles posiciones del elemento pivote}},$$

donde n es la longitud del *array* de entrada. Por definición, todo elemento en el `QUICKSORT` aleatorizado tiene la misma probabilidad de ser elegido como elemento pivote y, por tanto, $p(i) = 1/n$ para cada $i \in \Omega$.

B.2 Eventos

Un *evento* es un subconjunto $S \subseteq \Omega$ del espacio de muestra: una colección de resultados posibles de un proceso aleatorio. La probabilidad $\Pr[S]$ de

³Para un conjunto finito S , $|S|$ indica el número de elementos que pertenecen a S .

un evento S se define, tal y como cabe esperar, como la probabilidad de que se produzca uno de los resultados de S :

$$\Pr[S] = \sum_{i \in S} p(i).$$

Vamos a practicar con este concepto utilizando nuestros dos ejemplos.

Cuestionario B.1

Digamos que S representa al conjunto de resultados para los que la suma de dos dados normales es igual a 7. ¿Cuál es la probabilidad de que se produzca el evento S ?⁴

- a) $\frac{1}{36}$
- b) $\frac{1}{12}$
- c) $\frac{1}{6}$
- d) $\frac{1}{2}$

Solución y aclaraciones en la sección B.2.1

El segundo cuestionario guarda relación con la elección del elemento pivote aleatorio de la llamada más exterior a QUICKSORT. Decimos que un elemento pivote es una “mediana aproximada” si al menos el 25% de los elementos del *array* son menores que el pivote y al menos el 25% de los elementos son mayores.

Cuestionario B.2

Digamos que S indica el evento en el que el elemento pivote elegido en la llamada más exterior a QUICKSORT es una mediana aproximada. ¿Cuál es la probabilidad de que se produzca el evento S ?

- a) $\frac{1}{n}$, donde n es la longitud del *array*
- b) $\frac{1}{4}$

⁴Un dato útil cuando juguemos a los dados...

c) $\frac{1}{2}$

d) $\frac{3}{4}$

Solución y aclaraciones en la sección B.2.2

B.2.1 Solución al cuestionario B.1

Respuesta correcta: (c). Existen seis resultados en los que la suma de los dos dados es igual a 7:

$$S = \{(6, 1), (5, 2), (4, 3), (3, 4), (2, 5), (1, 6)\}.$$

Como cada resultado de Ω tiene la misma probabilidad de aparecer, $p(i) = 1/36$ para cada $i \in S$ y, por tanto

$$\Pr[S] = |S| \cdot \frac{1}{36} = \frac{6}{36} = \frac{1}{6}.$$

B.2.2 Solución al cuestionario B.2

Respuesta correcta: (c). Imagina, en un ejercicio mental, que dividimos los elementos del *array* de entrada en cuatro grupos: los $n/4$ elementos más pequeños, los siguientes $n/4$ elementos más pequeños, los siguientes $n/4$ elementos más pequeños y, por último, los $n/4$ elementos más grandes (como siempre, evitamos utilizar fracciones por comodidad). Cada elemento de los segundo y tercer grupos constituyen una mediana aproximada: todos los $n/4$ elementos de los primer y último grupos son menores y mayores que el pivote, respectivamente. Por contra, si el algoritmo selecciona un elemento pivote de los primer o último grupos, este no será una mediana aproximada. Por lo tanto, el evento S corresponde a los $n/2$ elementos de los segundo y tercer grupos. Como cada elemento tiene la misma probabilidad de ser elegido como pivote,

$$\Pr[S] = |S| \cdot \frac{1}{n} = \frac{n}{2} \cdot \frac{1}{n} = \frac{1}{2}.$$

B.3 Variables aleatorias

Una *variable aleatoria* es la medida numérica del resultado de un proceso aleatorio. Formalmente, es la función de valor real $X : \Omega \rightarrow \mathbb{R}$ definida sobre

el espacio de muestra Ω : la entrada $i \in \Omega$ hasta X es el resultado del proceso aleatorio y la salida $X(i)$ es un valor numérico.

En nuestro primer ejemplo, podemos definir una variable aleatoria que sea la suma de los dos dados. Esta variable aleatoria relaciona cada resultado (i, j) con $i, j \in \{1, 2, \dots, 6\}$ al número real $i + j$. En nuestro segundo ejemplo, podemos definir una variable aleatoria que contenga la longitud del *subarray* que pasa a la primera llamada recursiva de QUICKSORT. Esta variable aleatoria relaciona cada resultado (es decir, cada elección de un elemento pivote) al número de elementos del *array* menores que ese pivote (un entero entre 0 y $n - 1$, donde n es la longitud del *array* de entrada).

La sección 5.5 estudia la variable aleatoria X que representa al tiempo de ejecución del QUICKSORT aleatorizado sobre un *array* de entrada dado. En este caso, el estado-espacio Ω corresponde a todas las secuencias posibles de elementos pivote que podría elegir el algoritmo, y $X(i)$ es el número de operaciones realizadas por el algoritmo para una sucesión concreta $i \in \Omega$ de elecciones de pivotes⁵.

B.4 Expectativa

La *expectativa* o *valor esperado* $E[X]$ de una variable aleatoria X es el valor promedio de todo lo que podría llegar a ocurrir, ponderado adecuadamente con las probabilidades de los diferentes resultados. La intuición nos dice que, si un proceso aleatorio se repite una y otra vez, $E[X]$ corresponde al valor promedio que tendrá, a la larga, la variable aleatoria X . Por ejemplo, si X es el valor de un dado de seis caras equilibrado, entonces $E[X] = 3,5$.

En matemáticas, si $X: \Omega \rightarrow \mathbb{R}$ es una variable aleatoria y $p(i)$ indica la probabilidad del resultado $i \in \Omega$,

$$E[X] = \sum_{i \in \Omega} p(i) \cdot X(i). \quad (\text{B.1})$$

Los dos siguientes comentarios te piden que calcules la expectativa de las dos variables aleatorias definidas en la sección anterior.

⁵Como el único elemento aleatorio en el QUICKSORT aleatorizado es la elección de los elementos pivote, una vez que hayamos determinado la misma, QUICKSORT cuenta con un tiempo de ejecución bien definido.

Cuestionario B.3

¿Cuál es la expectativa de la suma de dos dados?

- a) 6.5
- b) 7
- c) 7.5
- d) 8

Solución y aclaraciones en la sección [B.4.1](#)

Volviendo al QUICKSORT aleatorizado, ¿qué tamaño tiene, en promedio, la longitud del *subarray* que recibe la primera llamada recursiva? Igualmente, ¿cuántos elementos son, en promedio, menores que un pivote elegido aleatoriamente?

Cuestionario B.4

¿Cuál de los siguientes valores es el más cercano a la expectativa del tamaño del *subarray* que recibe la primera llamada recursiva en QUICKSORT?

- a) $\frac{n}{4}$
- b) $\frac{n}{3}$
- c) $\frac{n}{2}$
- d) $\frac{3n}{4}$

Solución y aclaraciones en la sección [B.4.2](#)

B.4.1 Solución al cuestionario B.3

Respuesta correcta: (b). Existen varias maneras de determinar que la expectativa es 7. La primera es calcularlo de forma ingenua, utilizando la ecuación (B.1). Con 36 resultados posibles, resulta factible aunque es te-

dioso. Un método más eficaz consiste en emparejar todos los valores posibles de la suma y utilizar la simetría. La suma tiene la misma probabilidad de ser 2 que 12, o 3 que 11, etc. En todos estos pares, el valor promedio es 7, por lo que también será el promedio global. La tercera forma, y la mejor, consiste en emplear la linealidad de la expectativa, como veremos en la siguiente sección.

B.4.2 Solución al cuestionario B.4

Respuesta correcta: (c). El valor exacto de la expectativa es $(n - 1)/2$. Existe una probabilidad de $1/n$ de que el *subarray* tenga longitud 0 (si el pivote resulta ser el elemento más pequeño), una probabilidad de $1/n$ de que tenga longitud 1 (si el pivote es el segundo elemento más pequeño) y, así, sucesivamente, hasta llegar a la probabilidad de $1/n$ de que tenga longitud $n - 1$ (si el pivote es el elemento más grande). Según la definición (B.1) de la expectativa, y recordando la identidad⁶

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

tenemos que

$$\begin{aligned} E[X] &= \frac{1}{n} \cdot 0 + \frac{1}{n} \cdot 1 + \dots + \frac{1}{n} \cdot (n-1) = \frac{1}{n} \cdot \underbrace{(1 + 2 + \dots + (n-1))}_{=\frac{n(n-1)}{2}} \\ &= \frac{n-1}{2}. \end{aligned}$$

B.5 Linealidad de la expectativa

B.5.1 Declaración formal y casos de uso

Nuestro último concepto es una propiedad matemática y no una definición. La *linealidad de la expectativa* es la propiedad por la que la expectativa de

⁶Una forma de ver que $1 + 2 + \dots + (n-1) = \frac{n(n-1)}{2}$ es utilizando la inducción sobre n (ver la sección A.2). Para una demostración más audaz, tomamos dos copias del lado izquierdo y emparejamos el “1” de la primera copia con el “ $n-1$ ” de la segunda, el “2” de la primera con el “ $n-2$ ” de la segunda, etc. Esto resulta en $n-1$ pares, cada uno de ellos de valor n . Como duplicar la suma resulta en $n(n-1)$, la suma original es igual a $\frac{n(n-1)}{2}$.

una suma de variables aleatorias es igual a la suma de sus expectativas individuales. Resulta increíblemente útil para calcular la expectativa de una variable aleatoria compleja, como el tiempo de ejecución de QUICKSORT, cuando esa variable aleatoria se puede expresar como la suma ponderada de variables aleatorias más sencillas.

Teorema B.1 (Linealidad de la expectativa) *Digamos que X_1, X_2, \dots, X_n son variables aleatorias definidas en el mismo espacio de muestra Ω , y que a_1, a_2, \dots, a_n son números reales. Entonces*

$$E\left[\sum_{j=1}^n a_j \cdot X_j\right] = \sum_{j=1}^n a_j \cdot E[X_j]. \quad (\text{B.2})$$

Es decir, puedes tomar la suma y la expectativa en cualquier orden y el resultado será el mismo. El caso de uso más común se produce cuando $\sum_{j=1}^n a_j X_j$ es una variable aleatoria compleja (como el tiempo de ejecución del QUICKSORT aleatorizado) y los X_j son variables aleatorias sencillas (por ejemplo, binarias)⁷.

Por ejemplo, digamos que X es la suma de dos dados normales. Podemos expresar X como la suma de dos variables aleatorias X_1 y X_2 , que son los valores de los primer y segundo dados, respectivamente. La expectativa de X_1 o X_2 es fácil de calcular utilizando la definición (B.1) como $\frac{1}{6}(1 + 2 + 3 + 4 + 5 + 6) = 3,5$. En consecuencia, la linealidad de la expectativa resulta en

$$E[X] = E[X_1 + X_2] = E[X_1] + E[X_2] = 3,5 + 3,5 = 7,$$

lo que reproduce nuestra respuesta de la sección B.4.1, pero con menos trabajo.

Un aspecto extremadamente importante es que *la linealidad de la expectativa se mantiene incluso para variables aleatorias que no son independientes*. En este libro no es necesario realizar una definición formal de la independencia, pero es muy probable que la intuición ya esté funcionando: saber algo sobre el valor de una variable aleatoria no indica nada nuevo sobre los valores de las otras. Por ejemplo, las variables aleatorias X_1 y X_2 que hemos visto, son independientes porque se supone que los dados se lanzan de forma independiente.

⁷Cuando impartí este curso en Stanford, tras más de diez semanas de charlas delante de una pizarra, terminé dibujando un recuadro alrededor de, exactamente, una identidad matemática: la linealidad de la expectativa.

Para ver un ejemplo de variables aleatorias dependientes, pensemos en un par de dados unidos magnéticamente, donde el segundo dado siempre proporciona un valor mayor en una unidad que el del primero (o 1, si el resultado del primer dado es 6). Así, conocer el valor de cualquiera de los dados revela exactamente el valor del otro. Pero todavía podemos escribir la suma X de los dos dados como $X_1 + X_2$, donde X_1 y X_2 son los valores de los dados. Sigue siendo cierto que X_1 , visto de forma aislada, tiene la misma probabilidad de obtener cualquiera de los valores $\{1, 2, 3, 4, 5, 6\}$ y lo mismo es cierto para X_2 . Por lo tanto, seguimos teniendo $E[X_1] = E[X_2] = 3,5$ y, según la linealidad de la expectativa, $E[X]$ sigue siendo 7.

¿Por qué habría de sorprendernos? A primera vista, la identidad de (B.2) puede parecer una tautología. Pero, si pasamos de sumas a *productos* de variables aleatorias, la analogía del teorema B.1 ya no se verifica para variables aleatorias dependientes⁸. Por lo tanto, la linealidad de la expectativa es una propiedad especial para las sumas de variables aleatorias.

B.5.2 La demostración

La utilidad de la linealidad de la expectativa solo se ve igualada por la sencillez de su demostración⁹.

Demostración del teorema B.1: Comenzando con el lado derecho de (B.2) y expandiendo según la definición (B.1) de la expectativa, obtenemos

$$\begin{aligned}\sum_{j=1}^n a_j \cdot E[X_j] &= \sum_{j=1}^n a_j \cdot \left(\sum_{i \in \Omega} p(i) \cdot X_j(i) \right) \\ &= \sum_{j=1}^n \left(\sum_{i \in \Omega} a_j \cdot p(i) \cdot X_j(i) \right).\end{aligned}$$

Al invertir el orden del sumatorio, tenemos

$$\sum_{j=1}^n \left(\sum_{i \in \Omega} a_j \cdot p(i) \cdot X_j(i) \right) = \sum_{i \in \Omega} \left(\sum_{j=1}^n a_j \cdot p(i) \cdot X_j(i) \right). \quad (\text{B.3})$$

⁸Los dados unidos magnéticamente nos proporcionan un contraejemplo. Si queremos otro contraejemplo más sencillo, podemos suponer que X_1 y X_2 son iguales a 0 y 1 o a 1 y 0, respectivamente, con una probabilidad del 50% para cada resultado. En ese caso, $E[X_1 \cdot X_2] = 0$, a pesar de que $E[X_1] \cdot E[X_2] = \frac{1}{4}$.

⁹La primera vez que leas esta demostración, deberás asumir, por razones de simplicidad, que $a_1 = a_2 = \dots = a_n = 1$.

Como $p(i)$ es independiente de $j = 1, 2, \dots, n$, podemos extraerlo de la suma interior:

$$\sum_{i \in \Omega} \left(\sum_{j=1}^n a_j \cdot p(i) \cdot X_j(i) \right) = \sum_{i \in \Omega} p(i) \cdot \left(\sum_{j=1}^n a_j \cdot X_j(i) \right).$$

Por último, si volvemos a utilizar la definición (B.1) de la expectativa, obtenemos la parte izquierda de (B.2):

$$\sum_{i \in \Omega} p(i) \cdot \left(\sum_{j=1}^n a_j \cdot X_j(i) \right) = E \left[\sum_{j=1}^n a_j \cdot X_j \right].$$

Y eso es todo. La linealidad de la expectativa no es más que una inversión de un sumatorio doble. QED

Hablando de sumatorios dobles, la ecuación (B.3) podría parecer un tanto opaca si estás un poco desentrenado con este tipo de manipulaciones algebraicas. Para verlo de forma más pragmática, distribuimos los $a_j p(i) X_j(i)$ en una rejilla, con las filas indexadas por $i \in \Omega$, las columnas indexadas por $j \in \{1, 2, \dots, n\}$ y el número $a_j \cdot p(i) \cdot X_j(i)$ colocado en la casilla correspondiente a la i -ésima fila y a la j -ésima columna:

↑	{	$\begin{matrix} \cdot & \cdot & \vdots & \cdot & \cdot \\ \cdot & \cdot & \vdots & \cdot & \cdot \\ \cdots & \cdots & a_j p(i) X_j(i) & \cdots & \cdots \\ \cdot & \cdot & \vdots & \cdot & \cdot \\ \cdot & \cdot & \vdots & \cdot & \cdot \end{matrix}$	}
↓		$\longleftrightarrow j \longrightarrow$	

La parte izquierda de (B.3) comienza sumando cada una de las columnas y, después, suma a su vez esos resultados. La parte derecha comienza sumando las filas y, después, suma los resultados de las mismas. En cualquier caso, terminas obteniendo la suma de todas las entradas de la rejilla.

B.6 Ejemplo: balanceo de carga

Para enlazar todos los conceptos vistos hasta ahora, vamos a estudiar un ejemplo sobre balanceo de carga. Supongamos que necesitamos un algoritmo que asigne procesos a servidores, pero nos sentimos especialmente

perezosos. Una solución sencilla es, simplemente, asignar cada proceso a un servidor aleatorio, con la misma probabilidad para todos ellos. ¿Qué tal funcionaría esto?¹⁰

Por concretar un poco, asumimos que hay n procesos y también n servidores, donde n es un entero positivo. En primer lugar, vamos a concretar el espacio de muestra: el conjunto Ω corresponde a todas las n^n formas posibles de asignar procesos a los servidores, con n elecciones para cada uno de los n procesos. Según la definición de nuestro algoritmo perezoso, cada uno de estos n^n resultados tiene la misma probabilidad de aparecer.

Ahora que ya tenemos el espacio de muestra, estamos en posición de definir variables aleatorias. Una cifra interesante es la de la carga del servidor, con lo que vamos a definir Y como la variable aleatoria igual al número de procesos que se asignan al primer servidor (la historia se repetirá para todos los servidores por simetría, así que vamos a centrarnos en el primero). ¿Cuál es la expectativa de Y ?

En principio, podemos calcular $E[Y]$ mediante la evaluación directa de la ecuación (B.1), pero esto solo resultará práctico para los valores más pequeños de n . Por suerte, como Y se puede expresar como la suma de variables aleatorias sencillas, la linealidad de la expectativa puede venir al rescate. Formalmente, para $j = 1, 2, \dots, n$, definimos

$$X_j = \begin{cases} 1 & \text{si el proceso } j\text{-ésimo se asigna al primer servidor} \\ 0 & \text{en caso contrario.} \end{cases}$$

Las variables aleatorias que solo pueden tener los valores 0 o 1 se denominan, normalmente, variables aleatorias *binarias*, porque se limitan a indicar si se ha producido algún evento (como el caso de que el proceso j sea asignado al primer servidor).

Partiendo de las definiciones, podemos expresar Y como la suma de todos los X_j :

$$Y = \sum_{j=1}^n X_j.$$

Según la linealidad de la expectativa (teorema B.1), la expectativa de Y será entonces la suma de las expectativas de los X_j :

$$E[Y] = E\left[\sum_{j=1}^n X_j\right] = \sum_{j=1}^n E[X_j].$$

¹⁰Este ejemplo también será relevante en el tema de *hashing* de la segunda parte.

Como cada variable aleatoria X_j es tan sencilla, resulta fácil calcular directamente su expectativa:

$$\mathbf{E}[X_j] = \underbrace{0 \cdot \mathbf{Pr}[X_j = 0]}_{=0} + 1 \cdot \mathbf{Pr}[X_j = 1] = \mathbf{Pr}[X_j = 1].$$

Como el proceso j -ésimo tiene la misma probabilidad de ser asignado a cualquiera de los n servidores, $\mathbf{Pr}[X_j = 1] = 1/n$. Si lo juntamos todo, tenemos que

$$\mathbf{E}[Y] = \sum_{j=1}^n \mathbf{E}[X_j] = n \cdot \frac{1}{n} = 1.$$

Por tanto, si solo nos preocupan las cargas promedio de los servidores, nuestro algoritmo superperezoso funcionará perfectamente. Este ejemplo y el QUICKSORT aleatorizado son característicos del papel que juega la aleatorización en el diseño de algoritmos: normalmente podemos salir adelante utilizando algoritmos realmente sencillos en los que se producen elecciones aleatorias.

Cuestionario B.5

Consideremos un grupo de k personas. Asumimos que la fecha del cumpleaños de todas ellas está distribuida uniformemente y de forma aleatoria en las 365 posibilidades (con lo que ignoramos los años bisiestos). ¿Cuál es el valor más pequeño de k tal que el número esperado de parejas de personas distintas que comparten fecha de cumpleaños sea de, al menos, uno?

[Pista: define una variable aleatoria binaria para cada par de personas. Utiliza la linealidad de la expectativa.]

- a) 20
- b) 23
- c) 27
- d) 28
- e) 366

Solución y aclaraciones a continuación

Respuesta correcta: (d). Establecemos un entero positivo k e indicamos el conjunto de personas mediante $\{1, 2, \dots, k\}$. Digamos que Y representa al número de parejas de personas que comparten fecha de cumpleaños. Como sugiere la pista, definimos una variable aleatoria X_{ij} para cada elección $i, j \in \{1, 2, \dots, k\}$ de personas con $i < j$. Definimos X_{ij} como 1 si i y j comparten cumpleaños y como 0 en caso contrario. Por tanto, los X_{ij} son variables aleatorias binarias, e

$$Y = \sum_{i=1}^{k-1} \sum_{j=i+1}^k X_{ij}.$$

Según la linealidad de la expectativa (teorema B.1),

$$\mathbb{E}[Y] = \mathbb{E}\left[\sum_{i=1}^{k-1} \sum_{j=i+1}^k X_{ij}\right] = \sum_{i=1}^{k-1} \sum_{j=i+1}^k \mathbb{E}[X_{ij}]. \quad (\text{B.4})$$

Como X_{ij} es una variable aleatoria binaria, $\mathbb{E}[X_{ij}] = \Pr[X_{ij} = 1]$. Existen $(365)^2$ posibilidades de cumpleaños de las personas i y j , y en 365 de esas posibilidades i y j coincidirán en la fecha. Asumiendo que todas las combinaciones de cumpleaños tienen la misma probabilidad de aparecer,

$$\Pr[X_{ij} = 1] = \frac{365}{(365)^2} = \frac{1}{365}.$$

Si introducimos esto en (B.4), tenemos que

$$\mathbb{E}[Y] = \sum_{i=1}^{k-1} \sum_{j=i+1}^k \frac{1}{365} = \frac{1}{365} \cdot \binom{k}{2} = \frac{k(k-1)}{730},$$

donde $\binom{k}{2}$ indica el coeficiente binomial “ k sobre 2” (igual que en la solución al cuestionario 3.1). El valor más pequeño de k para el que

$$\frac{k(k-1)}{730} \geq 1$$

es 28.

Pistas y soluciones a problemas seleccionados

Solución al problema 1.1: El “2” ocupa la séptima posición.

Pista para el problema 1.2: La subrutina MERGE todavía se puede implementar de forma que el número de operaciones sea únicamente lineal en relación a la suma de las longitudes del *array* de entrada.

Pista para el problema 1.3: Utiliza el hecho de que $1 + 2 + \dots + k = \frac{(k+1)k}{2}$.

Solución al problema 1.4: (c). Hay $\approx \log_2 k$ iteraciones y cada una de ellas toma un tiempo proporcional a nk .

Solución al problema 1.5: (a), (b), (c), (d), (e).

Pista para el problema 1.6: La respuesta depende de k . Para $k = 1$, se requieren, al menos, n operaciones. Para un k lo suficientemente grande, el siguiente algoritmo utiliza menos de kn operaciones: (i) ordenar el *array* de entrada A ; (ii) para cada $i = 1, 2, \dots, k$, utilizar búsqueda binaria para comprobar si A contiene a t_i .

Pista para el problema 1.7: Considera calcular el número más grande utilizando una competición por eliminatorias. ¿Contra quién perderá el segundo número más grande?

Solución al problema 2.1: (a). Como la constante c del exponente se encuentra dentro de un logaritmo, forma parte de la constante principal y queda suprimida en la notación *Big-O*.

Pista para el problema 2.2: Para (a), recuerda la proposición 2.4.

Solución al problema 2.4: (e), (a), (d), (b), (c).

Solución al problema 2.5: La exponenciación con base 2 y los logaritmos con base 2 son operaciones inversas, por tanto $2^{\log_2 n}$ es lo mismo que n y $2^{2^{\log_2 n}}$ es lo mismo que 2^n . En consecuencia, el orden correcto es (a), (e), (c), (b), (d).

Solución al problema 2.6: Como $f(n) \leq c \cdot g(n)$ si, y solo si, $g(n) \geq f(n)/c$, siempre que dos constantes $c, n_0 > 0$ satisfacen que $f(n) = O(g(n))$, las constantes $\frac{1}{c}$ y n_0 satisfacen que $g(n) = \Omega(f(n))$. Por otro lado, siempre que dos constantes $c, n_0 > 0$ demuestran que $g(n) = \Omega(f(n))$, las constantes $\frac{1}{c}$ y n_0 demuestran que $f(n) = O(g(n))$.

Solución al problema 2.7: En la parte (a), dos constantes cualesquiera c y n_0 que muestren que $T(n) = O(f(n))$ bajo la definición alternativa, funcionan correctamente con la definición original. Por otro lado, si las constantes c y n_0 muestran que $T(n) = O(f(n))$ bajo la definición original, las constantes $2c$ y n_0 funcionan con la definición alternativa. La argumentación para la parte (b) es similar. En la parte (c), las definiciones de las notaciones *Big-O* y *Little-o* no implementan los diferentes conceptos de “menor o igual que” y “estrictamente menor que” mediante la elección (irrelevante) de desigualdades estrictas o laxas en (2.1)–(2.2), sino a través de sus diferentes órdenes de cuantificadores (“existen $c, n_0 > 0$ tales que...” frente a “para todos $c > 0$ existe $n_0 > 0$ tal que...”).

Pista para el problema 3.1: El algoritmo realiza un número constante de operaciones por dígito en la expansión binaria de b .

Solución al problema 3.2: Hay matrices de tamaño 1×1 con los valores $-2, 24, 35, 8, 65, -30$ y -22 , respectivamente.

Pista para el problema 3.3: El algoritmo es similar a la búsqueda binaria (ver también la sección 4.3.2). Si el elemento central del *array* de entrada es mayor que sus dos vecinos, has terminado. En caso contrario, realiza una recursión sobre la mitad del *array* que contenga al vecino más grande.

Pista para el problema 3.4: Si el elemento central del *array* de entrada tiene un valor igual a su índice, has terminado. En caso contrario, realiza una recursión sobre la mitad adecuada.

Pista para el problema 3.5: Divide \mathbf{x} en su primera mitad \mathbf{x}_1 y su segunda mitad \mathbf{x}_2 , calcula recursivamente $H_{k-1} \cdot \mathbf{x}_1$ y $H_{k-1} \cdot \mathbf{x}_2$ y combina los resultados adecuadamente.

Pista para el problema 3.6: Descubre cómo realizar una recursión sobre una rejilla de $\frac{n}{2} \times \frac{n}{2}$ después de realizar únicamente $O(n)$ comparaciones.

Pista para el problema 3.7: La técnica de alto nivel del algoritmo CLOSESTPAIR también funciona aquí, aunque solo después de realizar cambios significativos al planteamiento y la demostración del lema 3.3. Pensemos en la primera llamada recursiva, que calcula, para cada punto p de la mitad izquierda del conjunto, el punto más cercano q_p de la mitad derecha. Denominamos a un punto $q = (x, y)$ de la mitad derecha como *candidato* para ese punto p , si el punto $\bar{q} = (\bar{x}, y)$ está más cerca de p de lo que está q_p (como es habitual, \bar{x} indica la coordenada x más grande de entre los puntos de la mitad izquierda). Demuestra que cada punto q de la mitad derecha sirve como candidato para únicamente $O(1)$ de la mitad izquierda. En analogía a la subrutina CLOSESTSPLITPAIR, busca exhaustivamente entre todos los pares de punto-candidato.

Solución al problema 4.1: (d). Consulta la página 115.

Solución al problema 4.2: (b). Para mostrar (a) como falso, tomamos $a = 1, b = 2$ y $d = 0$ (como en la búsqueda binaria). Para mostrar (c) y (d) como falsos, tomamos $a = 4, b = 2$ y $d = 2$. La respuesta (b) es correcta, porque b^d no puede ser menor que 1 (ya que $b > 1$ y $d \geq 0$).

Solución al problema 4.3: (b). Siguiendo la notación del método maestro, $a = 7, b = 3$ y $d = 2$. Usando el caso 2, $T(n) = O(n^2)$.

Solución al problema 4.7: (b). Escribe n como $2^{\log_2 n}$. Cada aplicación de T divide el exponente por la mitad. El exponente cae de $\log_2 n$ a 1 en $\approx \log_2 \log_2 n$ iteraciones, por tanto $T(n) = O(\log \log n)$.

Solución al problema 5.1: (d). Las respuestas (a) y (b) no son correctas porque QUICKSORT y MERGESORT tienen fortalezas y debilidades que no son comparables. Por ejemplo, únicamente el primero trabaja *in situ*, mientras que solo el segundo es estable (página 157). La respuesta (c) no es correcta. Puedes consultar los problemas 5.6–5.7. RADIXSORT nos dice que (d) es correcta.

Solución al problema 5.2: Todo número que sea mayor que todos los que tiene a su izquierda y menor que los que están a su derecha: 4, 5 y 9.

Solución al problema 5.3: (c). La misma probabilidad de que un elemento pivote uniformemente aleatorio no se encuentra entre la fracción α de los elementos más pequeños ni entre la fracción α de los más grandes.

Pista para el problema 5.4: La fórmula que relaciona las funciones logarítmicas con bases diferentes es $\log_b n = \frac{\ln n}{\ln b}$.

Solución al problema 5.5: (b). En el mejor caso, el algoritmo siempre seleccionará la mediana como elemento pivote. En este caso, el árbol de recursión es, en esencia, idéntico al de MERGESORT, que tiene profundidad logarítmica. En el peor caso, el algoritmo siempre seleccionará los elementos mínimo o máximo como pivote, donde la profundidad de la recursión será lineal.

Pista para el problema 5.6: Siguiendo la demostración del teorema 5.5, demuestra que tal algoritmo puede ordenar correctamente un máximo de la mitad de los $n!$ arrays empleando $(\log_2 n!) - 1$ comparaciones o menos.

Pista para el problema 5.7: Considera ejecutar un algoritmo de ordenación aleatorizado basado en comparaciones sobre una de las $n!$ entradas del problema anterior, elegido aleatoriamente. Puedes ver el algoritmo aleatorizado como una distribución de probabilidad sobre algoritmos deterministas. Según el problema anterior, obteniendo el promedio de las elecciones aleatorias de la entrada, cada uno de los algoritmos deterministas tiene un tiempo de ejecución esperado de $\Omega(n \log n)$. Si volvemos a obtener el promedio sobre la elección aleatoria de un algoritmo determinista, se demuestra que el tiempo esperado de ejecución del algoritmo aleatorizado sobre una entrada aleatoria también es $\Omega(n \log n)$.

Solución al problema 6.1: (a), (c). La selección se reduce inmediatamente a la ordenación (sección 6.1.2), pero no al contrario (teoremas 5.5 y 6.6), por tanto, (a) es correcta y (b) no lo es. La respuesta (c) es correcta porque el algoritmo DSELECT se ejecuta en tiempo lineal y ningún algoritmo puede ser mejor por más que un factor constante. Incluso un algoritmo aleatorizado que siempre sea correcto debe examinar cada elemento del array de entrada. La respuesta (d) es incorrecta, ya que RSELECT es, en la práctica, superior a DSELECT.

Solución al problema 6.2: (d). La misma probabilidad que el caso en el que un elemento pivote aleatorio no se encuentre en la fracción $1 - \alpha$ de

los elementos más pequeños, ni en la fracción $1 - \alpha$ de los elementos más grandes, que es $1 - ((1 - \alpha) + (1 - \alpha)) = 2\alpha - 1$.

Pista para el problema 6.4: Con grupos de 7, seguir el planteamiento de la sección 6.4 lleva a la recurrencia $T(n) \leq T(n/7) + T(5n/7) + O(n)$, que se convierte en $T(n) = O(n)$. Con grupos de 3, seguir ese planteamiento lleva a la recurrencia $T(n) \leq T(n/3) + T(2n/3) + O(n)$ que, por un argumento de árbol de recursión, se convierte en $T(n) = O(n \log n)$.¹¹

Pista para el problema 6.5: Utiliza el algoritmo DSELECT para calcular el i -ésimo estadístico de orden para todos los valores de i que sean múltiplos de $n/8$. Comprueba los elementos repetidos. Realiza una pasada final sobre el *array* para contar el número de apariciones de cada uno de los valores candidatos restantes.

Solución al problema 6.6: Calcula el peso total W (en tiempo lineal). Inicializa D_L y D_R a 0. Estas variables llevarán el registro del peso total de los elementos descartados que son menores o mayores, respectivamente, que los elementos del *subarray* actual. En la llamada recursiva más exterior, utiliza DSELECT para calcular la mediana (no ponderada) y PARTITION para particionar el *array* en torno a ella (todo en tiempo lineal). Si el peso total a cada lado de la mediana es, como mucho, $W/2$, devuélvelo. En caso contrario, realiza una recursión sobre la (única) mitad del *array* cuyo peso total sea mayor que $W/2$, actualizando D_L o D_R , para llevar el registro del peso total de los elementos descartados.

En una llamada recursiva general, utiliza DSELECT para calcular la mediana (no ponderada) del *subarray* restante, aplica PARTITION al *subarray* en torno a esta mediana y calcula el peso total W_L y W_R de los elementos a cada lado del *subarray*. Si tanto $W_L + D_L$ como $W_R + D_R$ son, como máximo, $W/2$, devuelve esta mediana (es una mediana ponderada del *array* de entrada original, como deberías comprobar). En caso contrario, realiza una recursión sobre la (única) mitad del *subarray* con $W_L + D_L$ o $W_R + D_R$ mayores que $W/2$, actualizando D_L o D_R como corresponda.

El tiempo de ejecución del algoritmo viene dominado por la recurrencia $T(n) \leq T(n/2) + O(n)$. Según el caso 2 del método maestro, $T(n) = O(n)$ y, por tanto, el algoritmo se ejecuta en tiempo lineal.

¹¹¿Existe un análisis más refinado que demuestre un límite de tiempo lineal? Para profundizar en esta cuestión, puedes consultar el artículo “Select with Groups of 3 or 4 Takes Linear Time”, de Ke Chen y Adrian Dumitrescu (arXiv:1409.3600, 2014).

Índice alfabético

- $\binom{n}{2}$ (coeficiente binomial), 45, 74
 $n!$ (factorial), 160
 Ω , 51, 201
 $|x|$ (suelo), 94
 $|S|$ (tamaño del conjunto), 202
 $[x]$ (techo), 188
 $|x|$ (valor absoluto), 49
= frente a \coloneqq , 9
- ACM, 55
Adleman, Leonard, 185
Aho, Alfred V., 6
algoritmo, 1
 - espectacular, 63
 - paradigmas de diseño, 64
 - rápido, 31
 - tiempo constante, 39
 - tiempo cuadrático, 39
 - tiempo lineal, 31algoritmo *in situ*, 126
algoritmos aleatorizados, 141, 180, 212
algoritmos voraces, 64
análisis asintótico, 29
análisis del caso medio, 27
análisis del peor caso, 26
análisis global, 27
aplicación, 2
árbol de recursión, 22, 112
bit, 157
- Blum, Manuel, 185
BUBBLE SORT, 15, 156
BUCKET SORT, 156
búsqueda binaria, 105
búsqueda exhaustiva
 - para contar inversiones, 66
 - para el par más cercano, 82caso base (inducción), 198
caso base (recursividad), 8
casos de prueba, xvii
charla motivacional, 46
Chen, Ke, 219
CHOOSEPIVOT
 - implementación
 - aleatorizada, 141
 - implementación de la
 - mediana de tres, 164
 - implementación excesiva, 138, 140
 - implementación ingenua, 137, 139
- coeficiente intelectual, 3
conocimientos matemáticos, xvi, 197–212
constantes, 48, 103
 - ingeniería inversa, 50, 192conteo de inversiones
 - búsqueda exhaustiva, 66
 - corrección, 68, 73

- definición del problema, 65
implementación, 97
inversiones separadas, 72
pseudocódigo, 67, 69
tiempo de ejecución, 73
corolario, 19
COUNTINGSORT, 157
 implementación estable, 157
Coursera, xvii
cuestionarios, xvii
- Dasgupta, Sanjoy, 102
demostraciones, xvi
 de corrección, 129
 por contradicción, 51
 por inducción, 129, 192,
 197, 198
distancia euclídea, 82
distribución uniforme, 202
divide y vencerás, 11, 12, 63, 64
 cuándo utilizarlo, 64
demostraciones de
 corrección, 129
para contar inversiones, 67
para el par más cercano, 83
para multiplicación de
 matrices, 77
para ordenar, 64
- DSELECT**
 análisis del tiempo de
 ejecución, 185–193
 análisis heurístico, 191
 como una competición
 deportiva, 181
 con grupos de 3 o 7, 195
 frente a RSELECT, 180, 184
 historia, 184
 lema del 30-70, 187–191
 no se ejecuta *in situ*, 183
 pseudocódigo, 182
 tiempo de ejecución, 184
- Dumitrescu, Adrian, 219
duplicados, eliminación de, 35
- edX, xvii
el bien frente al mal, 114
elemento pivote, 127
entre amigos, 14, 19, 101
error de novato, 117
espacio de muestra, 201
espectacularmente rápido, xiv,
 xv, 32, 89, 125, 127,
 135
esquema de descomposición,
 146, 177
estadístico de orden, 148, 168
evento, 202
expectativa, 205
 linealidad de la, 207
- factores constantes, 28, 38, 39
Federer, Roger, 185
fiesta de informáticos, xv
filtrado colaborativo, 65, 66
Floyd, Robert W., 185
foco computacional, 3
foro de discusión, xvii
Fourier, matriz de, 96
fuerza bruta, búsqueda por, véase
 búsqueda exhaustiva
- Gauss, Carl Friedrich, 10
geometría computacional, 82
Google, 2
googol, 39
- Hadamard, matriz de, 96
hipótesis inductiva, 198
Hoare, Tony, 128
Hopcroft, John E., 6, 185
- inducción, véase demostraciones,
 por inducción

INSERTIONSORT, 15, 29, 156
invariante, 131
inversión, 65
 izquierda frente a derecha
 frente a separada, 67

KARATSUBA, 11
 implementación, 36
 recurrencia, 100
 tiempo de ejecución, 107

Karatsuba, Anatoly, 6

Kleinberg, Jon, 63

Knuth, Donald E., 55

lanzamiento de monedas, 178

Lehman, Eric, xvi, 197

Leighton, F. Thomson, xvi, 197

lema, 19

linealidad de la expectativa, 207
 no necesita independencia, 150, 208

$\ln x$, 104

logaritmos, 20, 104

mangostino, 156

mantra, 6

mediana (de un *array*), 35, 137, 168
 aproximada, 142, 177, 203
 frente a media, 169
 ponderada, 196

mediana de medianas, véase DSELECT

MERGE, 17, 18
 para contar inversiones, 69
 tiempo de ejecución, 18, 19

MERGESORT, 12–26
 análisis, 21–25
 basado en la comparación, 156
 como algoritmo de divide y vencerás, 64

implementación, 97
motivación, 12
no se ejecuta *in situ*, 126
pseudocódigo, 16
recurrencia, 101
tiempo de ejecución, 20, 105

método maestro
 a, b y d , 103, 110
 aplicado a MERGESORT, 105–107, 109
 aplicado a búsqueda binaria, 105, 109
 Big-Theta frente a Big-O, 104
 declaración formal, 104
 demostración, 112–119
 no es aplicable, 143, 146, 191
 significado de los tres casos, 114–117
 versiones más generalistas, 103

método probar–comprobar, 192

Meyer, Albert R., xvi, 197

moda (de un *array*), 35

Moore, ley de, 3, 30

multiplicación de enteros, 4–11, 99–102
 algoritmo de Karatsuba, 11
 algoritmo de la escuela, 4
 algoritmo recursivo sencillo, 9

multiplicación de Karatsuba, 6–11
 en Python, 107

multiplicación de matrices
 algoritmo de Strassen, 79–81
 algoritmo iterativo, 76
 algoritmo recursivo sencillo, 78

- definición, 74
el caso de 2×2 , 75
exponente, 110
multiplicación matriz-vector, 95, 96
- $n \log n$ frente a n^2 , 21, 29
notación Θ , véase notación *Big-Theta*
notación asintótica, 37–59
 Big-O frente a *Big-Theta*, 53
 como punto óptimo, 37
 en ocho palabras, 38
 historia, 55
 notación *Big-O*, véase
 notación *Big-O*
 notación *Big-Omega*, 51
 notación *Big-Theta*, 52
 notación *Little-o*, 54
notación *Big-O*, 46–49
 como un juego, 48
 definición en lenguaje
 convencional, 46
 definición matemática, 47
 definición visual, 46
 idea del alto nivel, 39
notación *Big-Omega*, 51
notación *Big-Theta*, 52
notación *Little-o*, 54
número primo, 141
- $o(f(n))$, véase notación *Little-o*
 $O(f(n))$, véase notación *Big-O*
 $\Omega(f(n))$, véase notación
 Big-Omega
operación básica, 4, 18, 25, 28
operación básica de coste cero, 142
ordenación
 aleatorizada, 141, 158
 algoritmos sencillos, 14, 15
- aplicaciones, 13, 35
basada en la comparación, 155
con bailarines populares húngaros, 134
con duplicados, 14
definición del problema, 14
en Unix, 155
estable, 157
MERGESORT frente a
 QUICKSORT, 126
in situ, 126
información asociada, 14
límite inferior, 155, 158–160
MERGESORT, véase
 MERGESORT
no basada en la
 comparación, 156, 157
por clave, 14
QUICKSORT, véase
 QUICKSORT
- Papadimitriou, Christos, 102
par clave-valor, 14
par más cercano
 búsqueda exhaustiva, 82
 caso unidimensional, 83
 corrección, 90–92
 definición del problema, 82
 pseudocódigo, 84, 86
 tiempo de ejecución, 85
paradoja del cumpleaños, 212
PARTITION, 134
 demostración de corrección, 135
 se ejecuta *in situ*, 135
paso inductivo, 198
patrones de diseño, xv
Pitágoras, teorema de, 92
¿podemos hacerlo mejor?, 6
¿por qué molestarte?, xiv, 1

- Pratt, Vaughan, 185
preguntas en entrevistas, xv
Premio Nobel, véase Premio Turing
Premio Turing, 129, 185
principios rectores, 26–31
probabilidad, 202
 de un evento, 203
problemas de programación, xvii
problemas frente a soluciones, 3
programación, xvi, 11
programación dinámica, 64
proposición, 19
pseudocódigo, 11, 17
- QED* (q.e.d.), 25
- QUICKSORT
- aleatorizado, 140
 - basado en la comparación, 156
 - demostración de corrección, 129
 - descripción de alto nivel, 128
 - elemento pivote, 127
 - gestión de empates, 126
 - historia, 129
 - implementación, 164
 - mediana de tres, 164
 - mejor caso, 137
 - mezcla aleatoria, 141
 - no es estable, 157
 - partición en torno a un pivote, 127, 130–135
 - peor caso, 137
 - pseudocódigo, 136
 - se ejecuta *in situ*, 126
 - tiempo de ejecución, 142
 - tiempo de ejecución (demostración), 144–154
- tiempo de ejecución (intuición), 142–144
- RADIXSORT, 157
- rápido, algoritmo, 31
- RECINTMULT, 9
- recurrencia, 100
 - tiempo de ejecución, 106
- RECMATMULT, 78
- recurrencia, 100
 - estándar, 102
- recursividad, 8
- reducción, 169
- resultado, xiv
- Rivest, Ronald L., 185
- RSELECT
- análisis del tiempo de ejecución, 176–180
 - implementación, 196
 - mejor caso, 173
 - peor caso, 172
 - pseudocódigo, 171
 - se ejecuta *in situ*, 172
 - tiempo de ejecución esperado, 174
- RSP (tasa de proliferación de subproblemas), véase método maestro, significado de los tres casos
- RWS (tasa de reducción del trabajo), véase método maestro, significado de los tres casos
- salón de la fama, 125
- secciones destacadas, xiv, 82
- Sedgewick, Robert, 126
- selección
- definición del problema, 168
- DSELECT, véase DSELECT

- reducción a ordenación, 169
RSELECT, véase RSELECT
SELECTIONSORT, 15, 156
series geométricas, 118, 180
Shamir, Adi, 185
SIGACT, 55
sistema de recomendaciones, 65,
 66
soluciones, xvii, 215–219
STRASSEN, 79–81
 tiempo de ejecución, 107,
 109
Strassen, Volker, 81
Sudoku frente a KenKen, 156
sumatorio doble, 210
superequipo, 184
Tardos, Éva, 63
Tarjan, Robert E., 185
tasa de crecimiento, véase
 análisis asintótico
técnica de coste cero, xiv, 31
teorema, 19
teorema maestro, véase método
 maestro
terminos de orden bajo, 38

 $\Theta(f(n))$, véase notación *Big-Theta*
tiempo de ejecución, 18, 25, 39
tiempo lineal, algoritmo de, 31
tira y afloja, 114
trabajo, véase tiempo de
 ejecución
transformada rápida de Fourier,
 96
truco de Gauss, 10, 79
Ullman, Jeffrey D., 6
valor esperado, 205
variable aleatoria, 204
 binaria, 211
 geométrica, 179
 independiente, 208
Vazirani, Umesh, 102
vecinos más cercanos, 96
victoria pírrica, 191
vídeos, xvii
von Neumann, John, 12
Wayne, Kevin, 126
YouTube, xvii