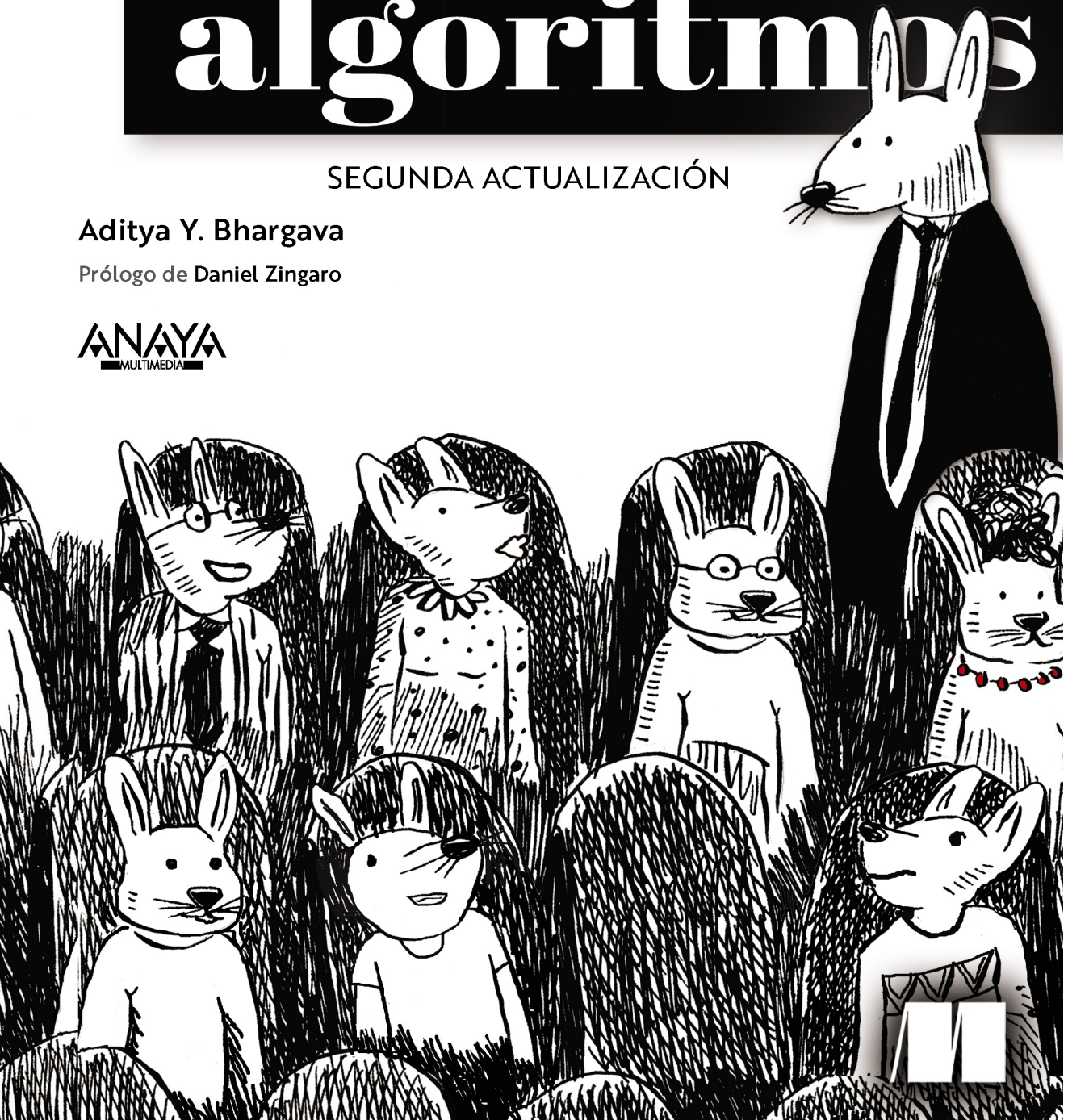
Comprender los algoritmos

SEGUNDA ACTUALIZACIÓN

Aditya Y. Bhargava

Prólogo de Daniel Zingaro

ANAYA MULTIMEDIA



Comprender
los algoritmos

Comprender los algoritmos

Segunda actualización

Aditya Y. Bhargava

Prólogo por Daniel Zingaro

ANAYA MULTIMEDIA

Título original: *Grokking Algorithms*

Primera edición en español: marzo de 2026
Diseño de cubierta: Leslie Haimes

© EDICIONES ANAYA MULTIMEDIA (GRUPO ANAYA, S.A.U.), Madrid 2026
Valentín Beato, 21
28037 Madrid.
www.anayamultimedia.es

Depósito legal: M-17564-2025
ISBN: 978-84-415-5252-4
Impreso en España - Printed in Spain

Para mis padres, Sangeeta y Yogesh

Agradecimientos

Gracias a Manning por la oportunidad de escribir este libro y por concederme una amplia libertad creativa. Gracias al editor Marjan Bace, a Mike Stephens por traerme a bordo, y a Ian Hough por ser un editor increíblemente atento y servicial. Gracias también a las personas del equipo de producción de Manning: Paul Wells, Debbie Holmgren y todas las personas que trabajaron entre bastidores. Además, quiero agradecer a todos aquellos que se leyeron el manuscrito y me ofrecieron recomendaciones: Daniel Zingaro, Ben Vinegar, Alexander Manning y Maggie Wenger. Gracias a David Eisenstat, mi revisor técnico, y a Tony Holdroyd, el corrector técnico de Manning, por detectar mis múltiples errores.

Gracias a todos los que me ayudaron a llegar hasta aquí: Bert Bates por enseñarme cómo escribir, a la gente del juego de mesa FlashKit, por enseñarme a programar; a los muchos amigos que revisaron capítulos, me dieron consejos y escucharon múltiples variantes de mis explicaciones —incluidos Ben Vinegar, Karl Puzon, Alex Manning, Esther Chan, Anish Bhatt, Michael Glass, Nikrad Mahdi, Charles Lee, Jared Friedman, Hema Manickavasagam, Hari Raja, Murali Gudipati, Srinivas Varadan y muchos otros—, y a Gerry Brady por enseñarme algoritmos. Otro gran agradecimiento a académicos como CLRS, Knuth y Strang. Gracias a todos estos gigantes y atlas que me sostienen.

Papá, mamá, Priyanka y el resto de la familia: gracias por vuestro apoyo constante. Y gracias sobre todo a mi esposa Maggie y a mi hijo Yogi. Nos esperan muchas aventuras por delante, y algunas de ellas no implican quedarse un viernes en la noche reescribiendo párrafos.

A todos los revisores: Abhishek Koserwal, Alex Lucas, Andres Sacco, Arun Saha, Becky Huett, Cesar Augusto Orozco Manotas, Christian Sutton, Diógines Goldoni, Dirk Gómez, Ed Bacher, Eder Andres Avila Niño, Frans Oilinki, Ganesh Swaminathan, Giampiero Granatella, Glen Yu, Greg Kreiter, Javid Asgarov, João Ferreira, Jobinesh Purushothaman, Joe Cuevas, Josh McAdams, Krishna Anipindi, Krzysztof Kamyczek, Kyrylo Kalinichenko, Lakshminarayanan AS, Laud Bentil, Matteo Battista, Mikael Byström, Nick Rakochy, Ninoslav Cerkez, Oliver Korten, Ooi Kuan San, Pablo Varela, Patrick Regan, Patrick Wanjau, Philipp Konrad, Piotr Pindel, Rajesh Mohanan, Ranjit Sahai, Rohini Uppuluri, Roman Levchenko, Sambaran Hazra, Seth MacPherson, Shankar Swamy, Srihari Sridharan, Tobias Kopf, Vivek Veerappan, William Jamir Silva y Xiangbo Mao: sus sugerencias mejoraron este libro.

Por último, muchísimas gracias a los lectores que se arriesgaron a comprar este volumen y a los lectores que me dieron opiniones en el foro del libro. En realidad, todos ayudaron a hacer de este un libro mejor.

Sobre el autor

Aditya Bhargava es un ingeniero de software. Posee un máster en Ingeniería Informática por la Universidad de Chicago. Además, administra un blog ilustrado de tecnología en adit.io.

Sobre el editor técnico

David Eisenstat es un investigador en ingeniería del software. Es doctor en Informática por la Universidad de Brown.

Reseñas de la primera edición

«Este libro consigue lo imposible:
¡hace que las matemáticas sean divertidas y fáciles!».
—Sander Rossel, COAS Software Systems

«¿Quieres darte el gusto de aprender algoritmos
de la misma manera que leerías tu novela favorita?
Si es así, ¡este es el libro que necesitas!».
—Sankar Ramanathan, IBM Analytics

«En el mundo de hoy, no hay ningún
aspecto de nuestras vidas que no esté
optimizado por algún algoritmo.
Deja que este sea el primer libro
que elijas si quieres una introducción
bien explicada al tema».
—Amit Lamba, Tech Overture, LLC

«¡Los algoritmos no son aburridos!
Este libro fue divertido y perspicaz
tanto para mis estudiantes como para mí».
—Christopher Haupt, Mobirobo, Inc

Índice abreviado

1	Introducción a los algoritmos	23
2	Ordenación por selección	43
3	Recursividad	63
4	Quicksort	77
5	Tablas hash	99
6	Búsqueda a lo ancho	123
7	Árboles	143
8	Árboles balanceados	163
9	El algoritmo de Dijkstra	187
10	Algoritmos voraces	213
11	Programación dinámica	225
12	K-vecinos más cercanos	251
13	¿Por dónde continuar?	269

Índice de contenidos

Prólogo 15
Prefacio 17
Acerca de este libro 19

1 Introducción a los algoritmos 23

Búsqueda binaria 25
Notación O grande 32

2 Ordenación por selección 43

Cómo funciona la memoria 44
Arrays y listas enlazadas 46
¿Qué se usa más: los arrays o las listas enlazadas? 53
Ordenación por selección 56
Ejemplo de código 60

3 Recursividad 63

Recursividad 64
Caso base y caso recursivo 66
La pila 68

4 Quicksort 77

Divide y vencerás 78
Quicksort 86
Notación O grande revisada 92

5 Tablas hash 99

Funciones hash 102
Casos de uso 107
Colisiones 113
Rendimiento 116

6 Búsqueda a lo ancho 123

Introducción a los grafos 124
¿Qué es un grafo? 126
Búsqueda a lo ancho 128
Implementar el grafo 133
Implementar el algoritmo 135

7 Árboles 143

Tu primer árbol 144
Una odisea en el espacio: búsqueda en profundidad 148
Árboles binarios 153
Codificación Huffman 155

8 Árboles balanceados 163

Un acto de equilibrio 164
Los árboles más bajos son más rápidos 169
Árboles AVL: un tipo de árbol balanceado 173
Árboles biselados 181
Árboles B 183

9 El algoritmo de Dijkstra 187

Cómo trabajar con el algoritmo de Dijkstra 188
Terminología 192
Intercambios para conseguir un piano 194
Aristas de peso negativo 200
Implementación 203

10 Algoritmos voraces 213

El problema de los horarios 214
El problema de la mochila 216
Problema del conjunto de cobertura 218

11 Programación dinámica 225

El problema de la mochila (revisado) 225
Preguntas frecuentes en el problema de la mochila 235
Mayor subcadena común 242

12 K-vecinos más cercanos 251

Clasificar de naranjas y pomelos 251
Construir un sistema de recomendación 253
Regresión 260
Introducción al aprendizaje automático 263
Información general de alto nivel sobre el entrenamiento de un modelo de ML 266

13 ¿Por dónde continuar? 269

Regresión lineal 269
Índices invertidos 271
La transformada de Fourier 272
Algoritmos paralelos 273
Map/reduce 274
Filtros Bloom y HyperLogLog 275
HTTPS y el intercambio de claves Diffie-Hellman 277
Hashing sensible al contexto 282
Montículos mínimos y colas de prioridad 283
Programación lineal 285
Epílogo 286

Apéndice A: Rendimientos de los árboles AVL 287

Apéndice B: Problemas NP-hard 289

Apéndice C: Respuestas a los ejercicios 299

Índice alfabético 313

Prólogo

Más gente que nunca necesita aprender a programar. Claro, algunas personas literalmente se dedican a hacer eso en su trabajo (ingenieros de software o desarrolladores web, por ejemplo). Pero muchos otros empleos, que históricamente no requerían programación, tienen un componente de programación ahora o lo tendrán en el futuro. La programación también ayuda a las personas a comprender el mundo tecnológico en el que viven.

Por desgracia, los beneficios de la programación no se distribuyen de manera equitativa. En los programas de informática (ciencias de la computación, CS) de Estados Unidos, por ejemplo, tenemos una participación muy baja de mujeres y de algunos grupos étnicos/raciales. Es fundamental que podamos hacer llegar la programación y la informática a un grupo más diverso.

La solución implicará avanzar en varios frentes, como la superación de los prejuicios, la formación de más docentes y la oferta de experiencias de aprendizaje más diversificadas. Tenemos que ayudar a «entrar» a más personas.

Estoy entusiasmado con el libro de Bhargava porque ofrece una nueva forma de adentrarse en los algoritmos, que son un componente clave de la programación efectiva. Algunas personas te dirán que solo hay una forma de aprender algoritmos: encontrar un libro matemático denso sobre algoritmos, leerlo y, por favor, entenderlo todo. Yo cambiaría esto por: pero eso privilegia a quienes pueden, disponen de tiempo y realmente necesitan aprender de esa manera. También supone que sabemos «por qué» alguien quiere aprender algoritmos, lo cual, seamos sinceros, no es una suposición justa.

Para ser claros, algunos de mis libros de informática favoritos son exactamente de ese tipo: libros de algoritmos orientados a las matemáticas. Y a mí me funcionan muy bien, como a muchos profesores de informática. Pero tal vez ese sea el problema: es demasiado fácil asumir que aprendemos de la misma manera que los demás. Lo que necesitamos es una gran diversidad de recursos de aprendizaje sobre todo tipo de temas de informática, cada uno diseñado para un público concreto.

El libro de Bhargava está diseñado intencionadamente para las personas que desean una introducción no matemática a los algoritmos. Lo que más me impresiona aquí no es lo que Bhargava eligió incluir, sino lo que no incluyó. No se puede incluir todo en un libro como este, eso sería abrumador y ese no es el objetivo.

La experiencia docente del autor le permite sacar muchas enseñanzas de pocas páginas. Al leer el capítulo «Programación dinámica», por ejemplo, me llamó la atención el cuidado con el que Bhargava responde a muchas preguntas anticipadas de los lectores que otros libros de algoritmos no responderían.

Espero que esta obra te ayude a aprender, tanto si es la primera vez que pruebas los algoritmos como si hasta ahora te ha costado encontrar el recurso adecuado. ¡Feliz viaje por el mundo de los algoritmos! ¡Que disfrutes dominando los algoritmos!

—Daniel Zingaro, Universidad de Toronto

Prefacio

Comencé a programar como un pasatiempo. El libro *Visual Basic 6 for Dummies* me enseñó los elementos básicos y de ahí en adelante continué leyendo libros para aprender más. No obstante, el tema de los algoritmos era indescifrable para mí. Recuerdo haber saboreado la tabla de contenidos de mi primer libro sobre algoritmos, mientras pensaba: «¡Por fin voy a entender estos temas!». Pero era un contenido muy denso y me rendí a las pocas semanas. No fue hasta que tuve mi primer buen profesor de algoritmos que comprendí cuán simples y elegantes eran estas ideas.

Escribí mi primera publicación ilustrada en un blog en 2012. Soy un aprendiz visual y me encantó el estilo ilustrado. Desde entonces he escrito algunas publicaciones ilustradas sobre programación funcional, Git, aprendizaje automático y concurrencia.

Por cierto, era un escritor mediocre cuando comencé. Explicar conceptos técnicos es difícil. Encontrar buenos ejemplos conlleva tiempo y explicar un concepto difícil también. Por ende, es fácil pasar por alto los temas complicados. Pensaba que estaba haciendo un buen trabajo, hasta que una de mis publicaciones se hizo popular y un compañero de trabajo se acercó y me dijo: «Leí tu publicación y todavía no entiendo esto». Aún tenía mucho que aprender sobre cómo escribir mejor.

En algún momento mientras escribía estas entradas en el blog, la editorial Manning se acercó y me preguntó si quería escribir un libro ilustrado. Bueno, resulta ser que los editores de Manning saben mucho sobre cómo explicar conceptos técnicos y me enseñaron a hacerlo. Escribí este libro para aliviar una picazón en particular: quería escribir un libro que explicara bien los conceptos técnicos complejos y que también resultara fácil de leer.

La primera edición de este libro se publicó en 2016. Desde entonces, más de 100 000 personas lo han leído. Estoy encantado de ver cuántas personas han conectado con el estilo de aprendizaje visual.

Con esta segunda edición, mi objetivo sigue siendo el mismo. En este libro, utilizo ilustraciones y ejemplos fáciles de recordar para que los conceptos se queden. El libro está diseñado para lectores que saben programar y quieren aprender más sobre algoritmos sin necesidad de ningún conocimiento matemático.

La segunda actualización rellena algunas lagunas de la primera. He oído de muchos lectores que quieren que explique los árboles. Ahora hay dos capítulos sobre árboles en este libro. También he ampliado la sección sobre los problemas NP-completos. Es un concepto muy abstracto y quería una explicación que lo hiciera más concreto. Si te sientes igual, espero que la nueva sección llene ese hueco para ti.

Mi escritura ha recorrido un largo camino desde esa primera publicación en el blog, así que espero que encuentres aquí una lectura fácil e informativa.

Acerca de este libro

Comprender los algoritmos está diseñado para ser fácil de seguir. Evita grandes saltos de razonamiento. Siempre que introduzco un concepto nuevo, lo explico de inmediato o te indico cuándo lo haré. Los conceptos fundamentales están reforzados con ejercicios y múltiples explicaciones para que puedas comprobar tus suposiciones y asegurarte de que lo estás comprendiendo.

Comienzo con ejemplos. En lugar de escribir una sopa de símbolos, mi objetivo es facilitar que visualices estos conceptos. Además, creo que aprendemos mejor cuando recalcamos algo que ya sabíamos y en ese sentido los ejemplos facilitan recordar. Por ejemplo, cuando intentes recordar la diferencia entre arrays y listas enlazadas (capítulo 2), puedes simplemente pensar en cómo te sientas en el cine para ver una película. A veces, aunque algo sea evidente, se aprende mejor de forma visual. Por eso, este libro está repleto de imágenes.

Los contenidos de este libro están cuidadosamente seleccionados. No hay necesidad de escribir un libro que cubra todos los algoritmos de ordenación, para eso tenemos la Wikipedia y la Khan Academy. Todos los algoritmos que he incluido son prácticos. Los he encontrado útiles en mi trabajo como ingeniero de software y proveen una base sólida para temas más complejos. ¡Buena lectura!

Cómo usar este libro

El orden y el contenido de este libro han sido cuidadosamente diseñados. Si te interesa un tema, no dudes en adelantarte. De lo contrario, lee los capítulos en orden, ya que se complementan entre sí.

Recomiendo encarecidamente ejecutar el código de los ejemplos tú mismo. No puedo enfatizar esta parte lo suficiente. Simplemente copia mis muestras de código tal cual (o descárgalas de https://www.manning.com/books/grokking-algorithms-second-edition o https://

github.com/egonschiele/grokking_algorithms)[1] y ejecútalas. Retendrás mucho más si lo haces.

También recomiendo hacer los ejercicios de este libro. Los ejercicios son cortos, por lo general de uno o dos minutos, a veces de 5 a 10 minutos. Te ayudarán a revisar tu pensamiento, para que sepas cuándo te has desviado antes de que hayas ido demasiado lejos.

¿Quién debería leer este libro?

Este libro está dirigido a cualquier persona que conozca los conceptos básicos de la programación y quiera comprender los algoritmos. Tal vez ya tengas un problema de programación y estés tratando de encontrar una solución algorítmica. O tal vez quieras entender para qué sirven los algoritmos. He aquí una lista corta e incompleta de perfiles que probablemente encontrarán útil este libro:

- Programadores aficionados.
- Estudiantes del campo de la programación.
- Graduados en informática que buscan un repaso.
- Graduados en física/matemáticas/otros que estén interesados en la programación.

Cómo está organizado este libro: una hoja de ruta

Los tres primeros capítulos de este libro sientan las bases:

- **Capítulo 1:** Aprenderás tu primer algoritmo práctico: búsqueda binaria. También estudiarás cómo analizar la velocidad de un algoritmo utilizando notación O grande. La notación O grande se utiliza a lo largo de todo el libro para analizar cuán rápido o lento es un algoritmo.
- **Capítulo 2:** Aprenderás sobre dos estructuras de datos fundamentales: arrays y listas enlazadas. Estas estructuras se utilizan a lo largo de todo el libro y se emplean para construir estructuras de datos más avanzadas como las tablas hash (capítulo 5).
- **Capítulo 3:** Conocerás la recursión, una técnica muy útil empleada por varios algoritmos (tales como quicksort, que veremos en el capítulo 4).

[1] En el repositorio https://github.com/DSRschool/grokking_algorithms_esp encontrarás la versión del código traducida al español, según se muestra en esta edición del libro.

En mi experiencia, la notación O grande y la recursión son temas desafiantes para los principiantes. Por lo tanto, he disminuido la velocidad y empleado tiempo adicional en estas secciones.

El resto del libro presenta algoritmos con amplias aplicaciones:

- **Técnicas de solución de problemas:** Contempladas en los capítulos 4, 10 y 11. Si te encuentras un problema y no sabes cómo solucionarlo eficientemente, intenta la técnica «divide y vencerás» (capítulo 4) o la programación dinámica (capítulo 11). O podrías darte cuenta de que no existe una solución eficiente y obtener una solución aproximada utilizando un algoritmo voraz en su lugar (capítulo 10).
- **Tablas hash:** Cubiertas en el capítulo 5. Las tablas hash son una estructura de datos muy útil. Contienen un conjunto de pares clave y valor, por ejemplo, el nombre de una persona y su dirección, o un nombre de usuario y su contraseña asociada. Es difícil pasar por alto la utilidad de las tablas hash. Cuando quiero resolver un problema, mi punto de partida son dos preguntas: ¿puedo utilizar una tabla hash?, y ¿puedo modelar esta situación como un grafo?
- **Algoritmos de grafos:** Contemplados en los capítulos 6, 7, 8 y 9. Los grafos son una manera de modelar una red: una red social, una red de caminos, neuronas o cualquier otro conjunto de conexiones. La búsqueda a lo ancho (capítulo 6) y el algoritmo de Dijkstra (capítulo 9) son formas de encontrar el camino más corto entre dos puntos de una red: puedes usar este enfoque para calcular los grados de separación entre dos personas o la ruta más corta hacia un destino. Los árboles son un tipo de grafo. Se utilizan en bases de datos (a menudo árboles B), en el navegador (el árbol DOM) o en el sistema de archivos.
- **K-vecinos más cercanos (KNN):** Tratados en el capítulo 12. Este es un sencillo algoritmo de aprendizaje de máquina. Puedes utilizar KNN para construir un sistema de recomendación, un motor de OCR, un sistema para predecir valores de la bolsa, cualquier cosa que involucre predecir un valor («Creemos que Adit evaluará esta película con cuatro estrellas») o clasificar un objeto («Esa letra es una Q»).
- **Próximos pasos:** El capítulo 13 aborda más algoritmos que proveen una buena lectura adicional.

Sobre el código

Todos los ejemplos de código de este libro utilizan Python 3. Todo el código del libro se presenta en una fuente de ancho fijo `como esta` para separarlo del texto ordinario. Las anotaciones de código acompañan a algunos de los fragmentos, destacando conceptos importantes.

Puedes obtener fragmentos ejecutables de código de la versión liveBook (en línea) de este libro en https://livebook.manning.com/book/grokking-algorithms-second-edition.

El código completo de los ejemplos del libro está disponible para su descarga en el sitio web de Manning en www.manning.com y en https://github.com/egonschiele/grokking_algorithms.[2]

Creo que se aprende mejor cuando realmente se disfruta aprendiendo, así que diviértete y ejecuta los ejemplos de código.

[2] En el repositorio https://github.com/DSRschool/grokking_algorithms_esp encontrarás la versión del código traducida al español, según se muestra en esta edición del libro.

Introducción a los algoritmos 1

En este capítulo:

- Obtendrás una base para el resto del libro.
- Escribirás tu primer algoritmo de búsqueda (búsqueda binaria).
- Aprenderás cómo hablar sobre el tiempo de ejecución de un algoritmo (notación O grande).

Un «algoritmo» es un conjunto de instrucciones orientadas a cumplir una tarea. Cada pedazo de código se puede llamar algoritmo, pero este libro aborda los más interesantes. Escogí incluir en este volumen algoritmos rápidos, que resuelven problemas interesantes o que hacen ambas cosas. Veamos un adelanto:

- El capítulo 1 habla sobre la búsqueda binaria y muestra cómo un algoritmo puede acelerar tu código. En uno de los ejemplos, ¡el número de pasos necesarios disminuye de cuatro mil millones a 32!
- Un dispositivo GPS utiliza algoritmos de grafos (como aprenderás en los capítulos 6 y 9) para calcular la ruta más corta a tu destino.
- Puedes utilizar programación dinámica (discutida en el capítulo 11) para escribir un algoritmo de IA que juegue a las damas.

En cada caso, describiré el algoritmo y te daré un ejemplo. Después hablaré sobre su tiempo de ejecución en notación O grande. Por último, exploraré qué otros tipos de problemas podrían solucionarse con el mismo algoritmo.

Qué aprenderás sobre rendimiento

La buena noticia es que probablemente encontrarás una implementación de cada algoritmo de este libro en tu lenguaje favorito, ¡así que no tendrás que escribirlos por tu cuenta! Pero esas implementaciones son inútiles si no comprendes las concesiones de cada una. En este volumen, aprenderás a comparar las diferentes concesiones realizadas entre distintos algoritmos: ¿Deberías usar ordenación por mezcla u ordenación rápida? ¿Deberías usar un array o una lista? Solamente el hecho de utilizar una estructura de datos distinta puede representar una gran diferencia.

Qué aprenderás sobre solución de problemas

Conocerás técnicas para solucionar problemas que podrían haber estado fuera de tu alcance hasta ahora. Por ejemplo:

- Si te gusta crear videojuegos, puedes escribir un sistema de IA que siga al usuario, utilizando algoritmos de grafos.
- Aprenderás a construir sistemas de recomendación con k-vecinos más cercanos.
- Algunos problemas no son solucionables en un tiempo prudencial. La parte de este libro que habla sobre los problemas NP-completos muestra cómo identificar dichos problemas y encontrar un algoritmo que provea una solución aproximada.

De forma general, hacia el final del libro, conocerás algunos de los algoritmos de uso más extendido. Podrás entonces emplear tu conocimiento para aprender de algoritmos más específicos para IA, bases de datos y mucho más. O aceptar mayores desafíos en el trabajo.

Qué necesitas saber

Necesitarás conocer álgebra básica antes de comenzar el libro. En particular, dada la función f(x) = x × 2, ¿qué se obtiene de f(5)? Si respondiste 10, estás listo.

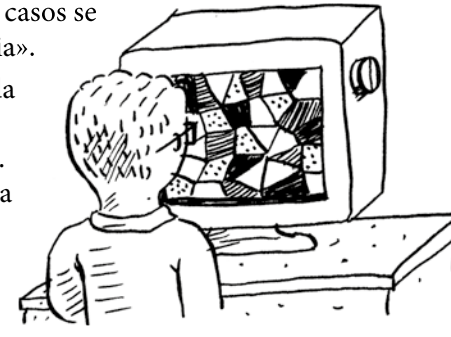
Adicionalmente, este capítulo (y este libro) será más fácil de entender si estás familiarizado con algún lenguaje de programación. Todos los ejemplos de este volumen se encuentran en Python. Si no conoces ningún lenguaje de programación y quieres aprender alguno, escoge Python; es genial para los principiantes. Si conoces algún otro lenguaje, por ejemplo, Ruby, te irá bien.

Búsqueda binaria

Supón que estás buscando a una persona en una guía telefónica (¡qué frase tan anticuada!). Su nombre comienza con K. Podrías comenzar por el principio y pasar páginas hasta que llegues a la K. Pero es más probable que comiences por una página del medio, porque sabes que la K estará cerca de la mitad de la guía telefónica.

O supón que estás buscando una palabra en un diccionario y comienza por O. También comenzarás cerca de la mitad.

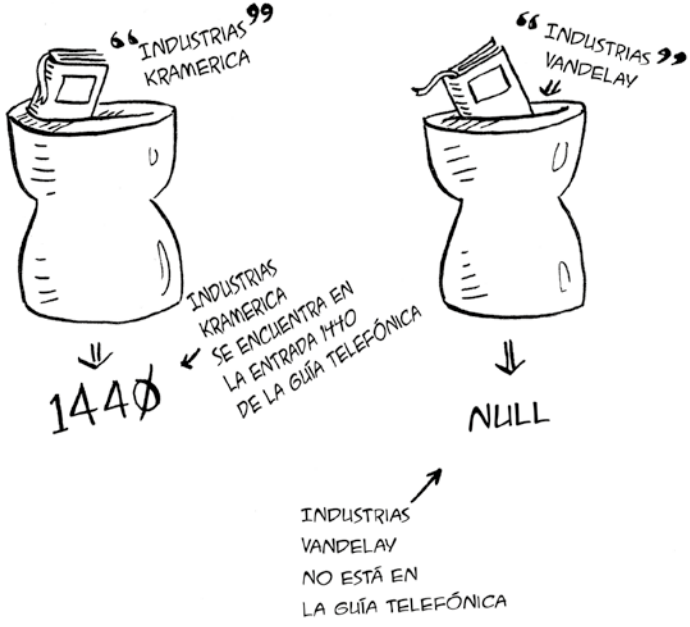
Ahora supón que accedes a Facebook. Al iniciar sesión, la plataforma debe verificar que tienes una cuenta. Para ello, necesita buscar tu nombre de usuario en su base de datos. Supón que tu nombre de usuario es karlmageddon. Facebook podría comenzar por la A y buscar tu nombre, pero tiene más sentido comenzar en algún lugar cerca de la mitad.



Este es un problema de búsqueda. En todos los casos se utiliza el mismo algoritmo: la «búsqueda binaria».

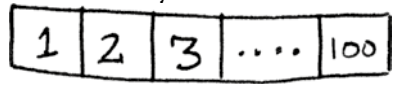
Búsqueda binaria es un algoritmo cuya entrada consiste en una lista ordenada de elementos (explicaré luego por qué debe estar ordenada). Si el elemento que estás buscando se encuentra en la lista, retorna la posición donde está localizado. De lo contrario, búsqueda binaria retorna `null`.

Aquí tienes un ejemplo:



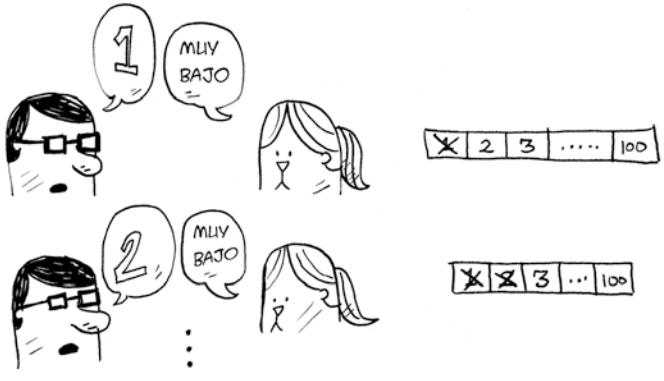
Buscando compañías en una guía telefónica mediante búsqueda binaria.

Ahora tienes un ejemplo de cómo funciona la búsqueda binaria. Estoy pensando en un número entre 1 y 100.



Tienes que intentar adivinar mi número en el menor número de intentos posible. En cada caso, te diré si tu suposición es demasiado baja, demasiado alta o correcta.

Supongamos que comienzas a adivinar así: 1, 2, 3, 4… Así es como iría.



Un mal enfoque para adivinar números.

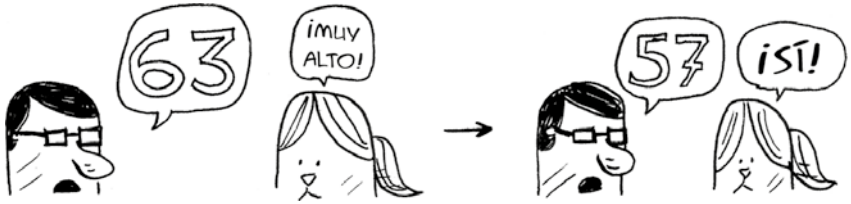
Esta es una «búsqueda simple» (tal vez una «búsqueda estúpida» sería un término mejor). En cada suposición solo eliminas un número. Si mi número fuera el 99, ¡tardarías 99 veces en llegar allí!

Una mejor manera de buscar

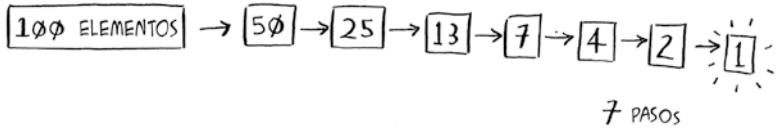
Veamos una técnica mejor. Empieza a partir de 50.

¡Demasiado bajo, pero acabas de eliminar «la mitad» de los números! Ahora ya sabes que del 1 al 50 son demasiado bajos. Siguiente conjetura: 75.

¡Demasiado alto, pero de nuevo has recortado la mitad de los números restantes! «Con la búsqueda binaria, empieza por el número del medio y elimina la mitad de los números restantes cada vez». El siguiente es el 63 (a medio camino entre el 50 y el 75).



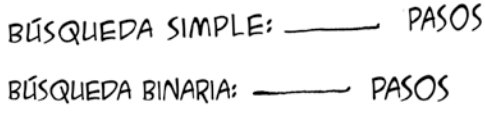
Esta es la búsqueda binaria. ¡Acabas de aprender tu primer algoritmo! Veamos la cantidad de números que puedes eliminar en cada intento.



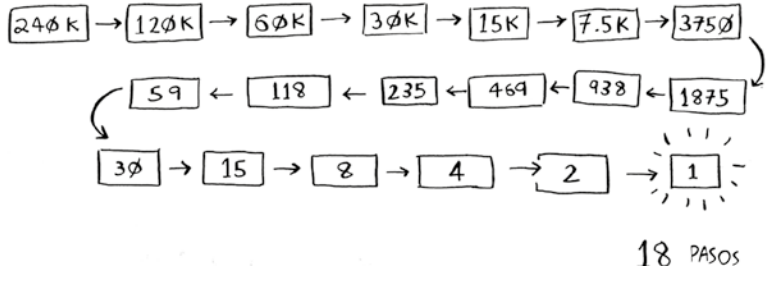
Elimina la mitad de los números cada vez con la búsqueda binaria.

Cualquiera que sea el número en el que yo esté pensando, podrías adivinarlo en un máximo de siete intentos, ¡porque eliminas muchos números en cada paso!

Supongamos que estás buscando una palabra en el diccionario. El diccionario tiene 240 000 palabras. «En el peor de los casos», ¿cuántos pasos crees que tomará cada búsqueda?



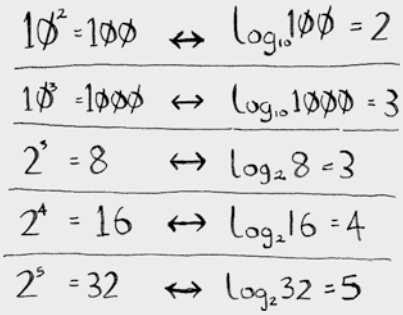
Una búsqueda simple puede necesitar 240 000 pasos si la palabra que estás buscando es la última. Con cada paso de la búsqueda binaria, el número de palabras se reduce a la mitad hasta que solo queda una.



Por lo tanto, la búsqueda binaria requerirá 18 pasos, ¡una gran diferencia! En general, para cualquier lista de tamaño *n*, la búsqueda binaria necesitará $\log_2 n$ pasos para ejecutarse en el peor de los casos, mientras que la búsqueda simple tomará *n* pasos.

Logaritmos

Es posible que no recuerdes qué son los logaritmos, pero probablemente sí sepas qué son los exponenciales. El $\log_{10} 100$ es como preguntar: «¿Cuántos 10 hay que multiplicar para obtener 100?». La respuesta es 2: 10×10. Así que $\log_{10} 100 = 2$. Los logaritmos son la inversa de los exponenciales.



Los logaritmos son la inversa de los exponenciales.

En este libro, cuando hablo de tiempo de ejecución en notación de O grande (que explicaré un poco más adelante), log siempre significa $\log_2$. Cuando se busca un elemento mediante la búsqueda simple, en el peor de los casos, es posible que haya que mirar cada uno de los elementos. Entonces, para una lista de ocho números, tendría que verificar ocho números como máximo. Para la búsqueda binaria, se deberían verificar los elementos log *n* en el peor de los casos. Para una lista de ocho elementos, log 8 == 3, porque $2^3 == 8$. Entonces, para una lista de ocho números, habría que comprobar tres números como máximo. Para una lista de 1024 elementos, log 1024 = 10, porque $2^{10} == 1024$. Entonces, para una lista de 1024 números, habría que verificar 10 números como máximo.

Nota

Hablaré mucho sobre el tiempo logarítmico en este libro, por lo que deberías comprender el concepto de los logaritmos. Si no, Khan Academy (https://khanacademy.org) tiene un bonito vídeo que lo deja claro.

Nota

La búsqueda binaria solo funciona cuando la lista está ordenada. Por ejemplo, los nombres de una guía telefónica se ordenan en orden alfabético, por lo que puede utilizar la búsqueda binaria para buscar un nombre. ¿Qué pasaría si los nombres no estuvieran ordenados?

Veamos cómo escribir una búsqueda binaria en Python. En el ejemplo de código se usan arrays. Si no sabes cómo funcionan los arrays, no te preocupes, los trataremos en el siguiente capítulo. Solo necesitas saber que es posible almacenar una secuencia de elementos en una fila de cubos consecutivos llamada array.

Los espacios se numeran empezando por 0: el primer hueco está en la posición 0, el segundo en la 1, el tercero en la 2 y así sucesivamente.

Nota

Me verás usar los términos lista y array indistintamente en el código. Esto se debe a que en Python, los arrays se llaman listas.

La función `busqueda_binaria` toma una lista ordenada y un elemento. Si el elemento está en la lista, la función devuelve su posición. Realizará un seguimiento de la parte de la lista en la que tiene que buscar. Al principio, esta es la lista completa:

```
menor = 0
mayor = len(lista) - 1
```

Cada vez, se comprueba el elemento del medio:

```
medio = (menor + mayor) // 2              ❶
estimado = lista[medio]
```

❶ Python redondea «medio» hacia abajo automáticamente si (menor + mayor) no es un número par.

Si la estimación es demasiado baja, se actualiza `menor` en consecuencia:

```
if estimado < elemento:
    menor = medio + 1
```

Si por el contrario, el número es muy alto, se actualiza `mayor`. Aquí tienes el código completo:

```
def busqueda_binaria(lista, elemento):
    menor = 0                                          ❶
    mayor = len(lista) - 1                             ❶

    while menor <= mayor:                              ❷
        medio = (menor + mayor) // 2                   ❸
        estimado = lista[medio]
        if estimado == elemento:                       ❹
            return medio
        elif estimado > elemento:                      ❺
            mayor = medio - 1
        else:                                          ❻
            menor = medio + 1
    return None                                        ❼

mi_lista = [1, 3, 5, 7, 9]                             ❽

print(busqueda_binaria(mi_lista, 3))  # => 1           ❾
print(busqueda_binaria(mi_lista, -1)) # => None        ❿
```

❶ «menor» y «mayor» indican en qué parte de la lista buscarás.
❷ Mientras no hayas reducido la búsqueda a un solo elemento…
❸ … comprueba el elemento central.
❹ Elemento encontrado.
❺ El número era muy alto.
❻ El número era muy bajo.
❼ El elemento no existe.
❽ ¡Probemos la búsqueda!
❾ Recuerda, la lista comienza en la posición 0. El segundo elemento tiene índice 1.
❿ «None» significa nulo en Python. Indica que el elemento no fue encontrado.

EJERCICIOS

1.1 Supón que tienes una lista ordenada de 128 nombres y estás realizando una búsqueda binaria. ¿Cuál es el máximo número de pasos que necesitarás para completarla?

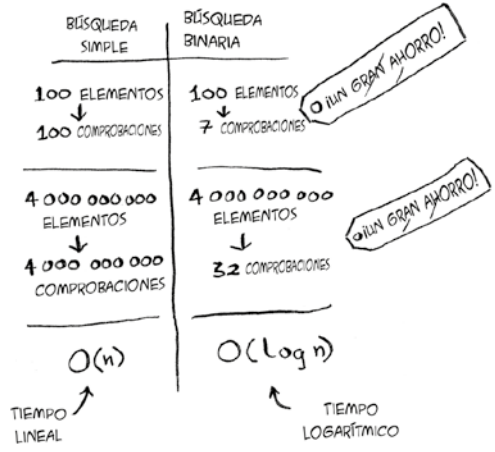
1.2 Supón que duplicas el tamaño de la lista. ¿Cuál es el máximo número de pasos ahora?

Tiempo de ejecución

En todo momento, cuando hablo sobre un algoritmo, discuto su tiempo de ejecución. Por lo general quieres escoger el algoritmo más eficiente, ya sea porque intentas optimizar el tiempo de ejecución o el espacio en memoria.

Volviendo a la búsqueda binaria. ¿Cuánto tiempo ahorras al utilizarla? Bueno, el primer enfoque consistía en analizar cada número, uno a la vez. Si la lista es de cien números, necesitarás hasta cien comprobaciones. Si la lista es de cuatro mil millones de números, necesitarás hasta cuatro mil millones de comprobaciones. Entonces, el máximo número de chequeos es igual al tamaño de la lista. Esto se conoce como «tiempo lineal».

La búsqueda binaria es diferente. Si la lista tiene tamaño 100, se necesitan a lo sumo siete chequeos. Si la lista contiene cuatro mil millones de elementos, serían a lo sumo 32 comprobaciones. Potente, ¿verdad? La búsqueda binaria se ejecuta en «tiempo logarítmico» (o log time, como le llaman los nativos). He aquí una tabla que resume nuestros hallazgos de hoy.



Tiempo de ejecución para algoritmos de búsqueda.

Notación O grande

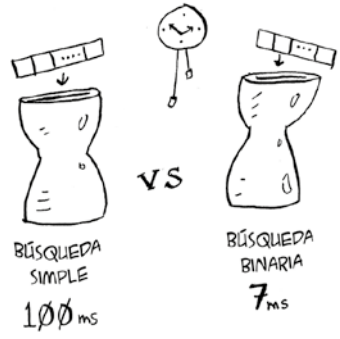
La notación «O grande» es una notación especial que indica cuán rápido es un algoritmo. ¿A quién le preocupa? Resulta que usarás con frecuencia algoritmos de otras personas y, cuando lo hagas, viene bien entender lo rápidos o lentos que son. Explicaré qué es la notación O grande y daré una lista de los tiempos de ejecución más comunes para algunos algoritmos mediante esta notación.

Los tiempos de ejecución de los algoritmos crecen a diferentes ritmos

Teo está escribiendo un algoritmo de búsqueda para la NASA. Su algoritmo comenzará a trabajar en cuanto un cohete se disponga a aterrizar en la luna y ayudará a calcular dónde aterrizar.

Este es un ejemplo de cómo el tiempo de ejecución de dos algoritmos puede aumentar a diferente ritmo. Teo intenta decidir entre búsqueda simple y búsqueda binaria. El algoritmo necesita ser rápido y correcto a la vez. Por una parte, la búsqueda binaria es más rápida. Teo solo dispone de diez segundos para encontrar dónde aterrizar; de lo contrario, el cohete saldrá de su trayectoria. Por otro lado, la búsqueda simple es más sencilla de implementar y hay menos probabilidad de introducir errores. ¡Y Teo no quiere «en ningún caso» introducir errores en el código que hará aterrizar un cohete! Para ser extracuidadoso, decide medir el tiempo que le toma a ambos algoritmos analizar una lista de 100 elementos.

Asumamos que es necesario 1 milisegundo para comprobar un elemento. Con la búsqueda simple, Teo debe analizar 100 elementos, así que la búsqueda tomará 100 ms. Por otra parte, solo tiene que comprobar siete elementos con la búsqueda binaria ($\log_2$ 100 es aproximadamente 7). Pero en realidad la lista tendrá unos mil millones de elementos. En ese caso, ¿cuánto tiempo le tomará al algoritmo de búsqueda simple? ¿Y cuánto al de búsqueda binaria? Asegúrate de tener una respuesta antes de continuar leyendo.



Tiempo de ejecución de la búsqueda simple y de la búsqueda binaria en una lista de 100 elementos.

Teo ejecuta una búsqueda binaria con mil millones de elementos y necesita 30 ms ($\log_2$ 1 000 000 000 es aproximadamente 30). Piensa: «¡Treinta milisegundos!». «La búsqueda binaria es alrededor de 15 veces más rápida que la búsqueda simple, ya que la búsqueda simple necesitó 100 ms con cien elementos y búsqueda binaria necesitó 7 ms. Entonces la búsqueda simple tardará 30 × 15 = 450 ms, ¿cierto? Bien por debajo de mi límite de 10 segundos». Teo decide utilizar búsqueda simple. ¿Es esa la decisión correcta?

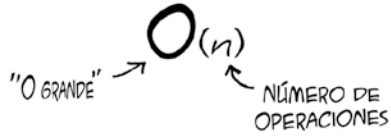
No. Resulta ser que Teo está equivocado. Completamente equivocado. El tiempo de ejecución para búsqueda simple con mil millones de elementos será de mil millones de ms, ¡que equivalen a 11 días! El problema es que los tiempos de ejecución con la búsqueda simple y la binaria «no aumentan al mismo ritmo».

¡Los tiempos de ejecución crecen a velocidades muy diferentes!

Es decir, a medida que el número de elementos se incrementa, la búsqueda binaria utiliza un poco más de tiempo para ejecutarse. Pero la búsqueda simple utiliza «mucho» más tiempo. Así que a medida que la lista de números se hace más grande, la búsqueda binaria de repente se vuelve «mucho» más rápida que la simple. Teo pensó que la búsqueda binaria era 15 veces más rápida que la simple, pero eso no es correcto. Si la lista tiene mil millones de elementos, será alrededor de 33 millones de veces más rápida. Por eso no es suficiente conocer cuánto tarda en ejecutarse un algoritmo, hay que saber cómo se incrementa el tiempo de ejecución a medida que aumenta el tamaño de la lista. Ahí es donde la notación O grande entra en juego.

La notación O grande te muestra cuán rápido es un algoritmo. Por ejemplo, supón que tienes una lista de tamaño n. La búsqueda simple necesita comprobar cada elemento, así que utiliza n operaciones. El tiempo de ejecución en notación O grande es $O(n)$. ¿Dónde están los segundos? No los hay. O grande no define la velocidad en segundos. «La notación O grande permite comparar el número de operaciones». Te dice cuán rápido crece el algoritmo.

La búsqueda binaria necesita log *n* operaciones para analizar una lista de tamaño *n*. ¿Cuál es el tiempo de ejecución en notación O grande? Es O(log *n*). En general, O grande se escribe como sigue.



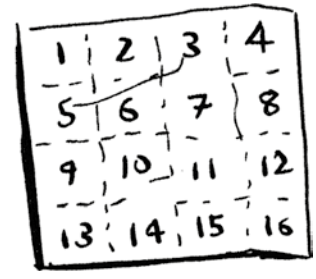
Cómo se ve la notación O grande.

Esta notación indica el número de operaciones que un algoritmo realizará. Se le llama notación O grande porque pones una «O grande o mayúscula» delante del número de operaciones (parece una broma, pero ¡es cierto!).

Ahora veamos algunos ejemplos. Intenta descifrar el tiempo de ejecución de los siguientes algoritmos.

Visualización de diferentes tiempos de ejecución en notación O grande

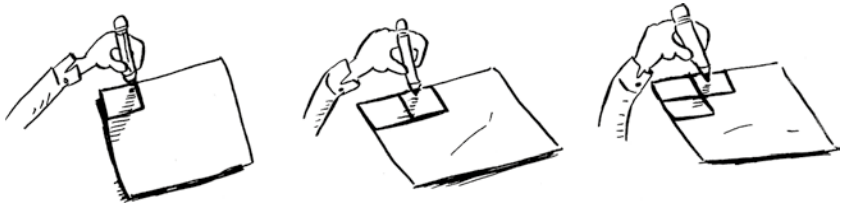
Aquí tienes un ejemplo práctico que puedes seguir en casa con algunas hojas de papel y un bolígrafo. Supón que debes dibujar una matriz de 16 celdas.



¿Cuál sería un buen algoritmo para dibujar esta matriz?

Algoritmo 1

Una manera es dibujar 16 celdas, una tras otra. Recuerda, la notación O grande cuenta el número de operaciones. En este ejemplo, dibujar una celda es una operación. Tienes que dibujar 16. ¿Cuántas operaciones tomará, dibujando una celda cada vez?



Dibujar una celda de la matriz cada vez.

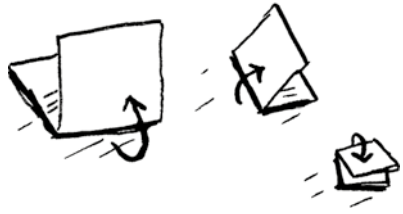
Se necesitan 16 pasos para dibujar 16 celdas. ¿Cuál es el tiempo de ejecución para este algoritmo?

Algoritmo 2

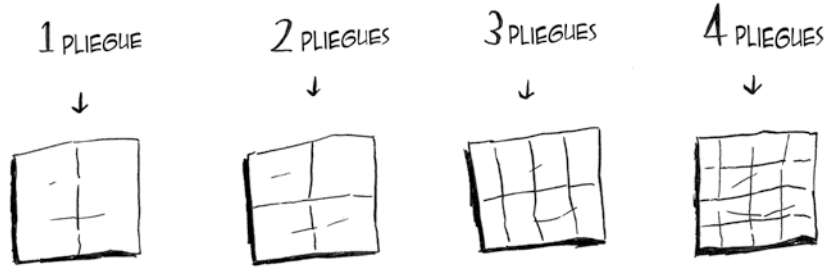
Intenta este algoritmo en su lugar. Dobla el papel.

En este ejemplo, doblar el papel una vez cuenta como una operación. Acabas de crear dos celdas con esa operación.

Dobla el papel otra vez, y otra, y otra.



¡Desdóblalo después de cuatro pliegues y tendrás una hermosa matriz! Cada pliegue duplica el número de celdas. ¡Has hecho 16 celdas en cuatro operaciones!



Dibujar una matriz en cuatro pliegues.

Puedes «dibujar» el doble de celdas con cada pliegue, así que puedes dibujar 16 celdas en cuatro pasos. ¿Cuál es el tiempo de ejecución de este algoritmo? Intenta definir los tiempos de ejecución de ambos algoritmos antes de continuar leyendo.

Respuestas: El algoritmo 1 tardará un tiempo de ejecución O(n) y el algoritmo 2 necesita O(log n).

La notación O grande establece el tiempo de ejecución en el peor caso

Supón que utilizas la búsqueda simple para localizar a una persona en la agenda telefónica. Sabes que la búsqueda simple necesita un tiempo O(n) para ejecutarse, lo que significa que en el peor caso tendrás que buscar en cada una de las entradas de tu agenda telefónica. En este caso, buscas a Adit, que es la primera entrada de tu agenda. Así que no necesitaste mirar cada una de las entradas, lo encontraste en el primer intento. ¿Acaso este algoritmo toma un tiempo O(n)? ¿O toma un tiempo O(1) porque encontraste a la persona en el primer intento?

La búsqueda simple sigue tomando tiempo O(n). En este caso encontraste lo que estabas buscando en un instante. Ese es el escenario del mejor caso. Pero la notación O grande se ocupa del peor escenario. Entonces podrías decir que, en el peor caso, tendrías que buscar en cada una de las entradas de la agenda al menos una vez. Eso corresponde a un tiempo O(n). Es tranquilizador porque sabes que la búsqueda simple nunca será más lenta que O(n).

> **Nota**
>
> Junto al tiempo de ejecución del peor caso, también es importante analizar el tiempo del caso promedio. En el capítulo 4 se comparan el peor caso y el caso promedio.

Algunos tiempos de ejecución O grande comunes

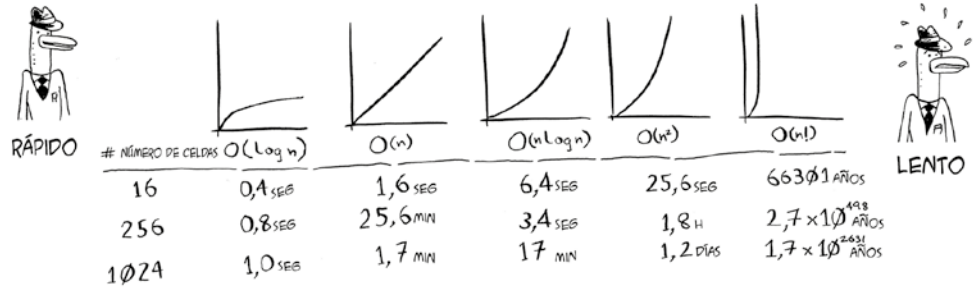
Aquí tienes cinco tiempos O grande que encontrarás con frecuencia, ordenados de más rápido a más lento:

- O(log n), también conocido como «tiempo logarítmico». Por ejemplo: la búsqueda binaria.
- O(n), también conocido como «tiempo lineal». Por ejemplo: la búsqueda simple.
- O(n * log n). Por ejemplo: un algoritmo de ordenación eficiente, como quicksort (se tratará en el capítulo 4).
- O(n^2). Por ejemplo: un algoritmo de ordenación lento, como ordenación por selección (se tratará en el capítulo 2).
- O(n!). Por ejemplo: un algoritmo muy lento, como el del vendedor ambulante (¡justo a continuación!).

Supón que otra vez debes dibujar una matriz de 16 celdas y puedes escoger entre cinco algoritmos diferentes. Si utilizas el primer algoritmo, demorarás un tiempo O(log *n*) para dibujar la matriz. Puedes realizar diez operaciones por segundo. Con un tiempo O(log *n*), necesitarás cuatro operaciones para dibujar una matriz de 16 celdas (log 16 = 4). Entonces te llevará 0,4 segundos construir la matriz. ¿Qué pasaría si tuvieras que dibujar 1024 celdas? Necesitarías log 1024 = 10 operaciones, o sea, un segundo para dibujar una matriz de 1024 celdas. Estos números se corresponden con ese primer algoritmo.

El segundo algoritmo es más lento: necesita un tiempo O(*n*). Se necesitarían 16 operaciones para dibujar 16 celdas y 1024 operaciones para 1024 celdas. ¿Cuánto tiempo es eso en segundos?

A continuación puedes ver cuánto tardarías en dibujar la matriz con el resto de los algoritmos, ordenados del más rápido al más lento:



Existen otros tiempos de ejecución, pero estos cinco son los más comunes.

Esta explicación es una simplificación. En realidad, no puedes convertir un tiempo de ejecución O grande a un número de operaciones de forma tan precisa, pero, por ahora, es una aproximación suficientemente buena. Volveremos al tema de la notación O grande en el capítulo 4, después de que hayas aprendido más algoritmos. Mientras tanto, las principales conclusiones son las siguientes:

- La velocidad de un algoritmo no se mide en segundos, sino en el crecimiento del número de operaciones necesarias.
- En lugar de hablar de segundos, hablamos de cuán rápido se incrementa el tiempo de ejecución de un algoritmo a medida que el tamaño de su entrada aumenta.
- El tiempo de ejecución de los algoritmos se expresa en la notación O grande.
- O(log *n*) es más rápido que O(*n*) y se acelera aún más a medida que crece la lista de elementos de la búsqueda.

EJERCICIOS

Define el tiempo de ejecución de los siguientes escenarios en términos de O grande:

1.3 Tienes un nombre y quieres encontrar el número de teléfono de la persona en la agenda telefónica.

1.4 Tienes un número de teléfono y quieres encontrar el nombre de la persona en la agenda telefónica (pista: tendrás que buscar en toda la agenda).

1.5 Quieres leer los números de cada persona en la agenda telefónica.

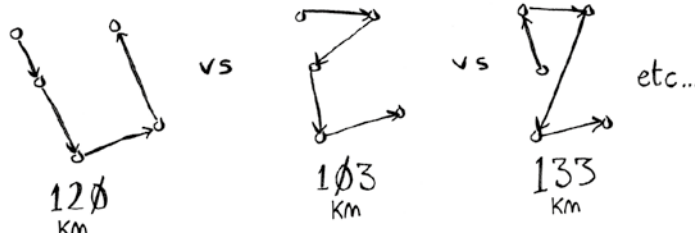
1.6 Solo quieres leer los números de las personas cuyo nombre comienza con A (¡este es un caso engañoso! Involucra conceptos que se tratarán en detalle en el capítulo 4. Lee la respuesta, ¡podrías sorprenderte!).

El vendedor ambulante

Tal vez mientras leías la última sección pensaste: «No hay manera de que alguna vez encuentre un algoritmo que necesite tiempo O(n!)». Bueno, ¡intentemos probar que te equivocas! Aquí tienes un ejemplo de un algoritmo con un tiempo de ejecución realmente malo. Se trata de un problema famoso en informática, porque su crecimiento es aterrador y algunas personas muy inteligentes piensan que no se puede mejorar. Se conoce como el problema del «vendedor ambulante» (*travelling salesman*, en inglés).

Tienes un vendedor. El vendedor tiene que visitar cinco ciudades.

Este vendedor, al que llamaré Óscar, quiere llegar a las cinco ciudades recorriendo la menor distancia posible. Una posible forma de hacerlo es la siguiente: buscar cada una de las posibles combinaciones de orden para visitar todas las ciudades.



El vendedor suma la distancia total y selecciona el recorrido con la menor distancia. Existen 120 permutaciones de las cinco ciudades, así que necesitará 120 operaciones para resolver el problema con cinco ciudades. Para seis ciudades utilizará 720 operaciones (hay 720 permutaciones).

Para siete ciudades, necesitaría 5040 operaciones.

CIUDADES	OPERACIONES
6	72Ø
7	5Ø4Ø
8	4Ø32Ø
...	...
15	13Ø7674368ØØØ
...	...
3Ø	265252859812191058636308480000000

El número de operaciones aumenta drásticamente.

En general, para *n* elementos, se necesitarán *n*! (*n* factorial) operaciones para calcular el resultado. Por tanto, hablamos de tiempo O(*n*!) o «tiempo factorial». Conlleva muchísimas operaciones para todos los casos, excepto para los números más pequeños. Una vez que lo intentas con más de 100 ciudades, es imposible calcular la respuesta en tiempo; el sol colapsaría antes.

¡Este es un algoritmo terrible! Óscar debería utilizar uno diferente, ¿verdad? Pero no puede. Este es uno de los problemas sin solucionar en informática. No se conoce ningún algoritmo rápido para resolverlo y hay personas muy inteligentes que piensan que es «imposible» encontrar un algoritmo inteligente para este problema. Lo mejor que podemos hacer es obtener una solución aproximada. Lee el capítulo 10 si deseas ahondar en el tema.

Recapitulación

- La búsqueda binaria es mucho más rápida que la búsqueda simple a medida que crece el tamaño del array.
- O(log *n*) es más rápido que O(*n*) y se acelera aún más a medida que crece la lista de elementos de la búsqueda.
- La velocidad de un algoritmo no se mide en segundos.
- El tiempo de ejecución de un algoritmo se mide en términos de «crecimiento» del algoritmo.
- Los tiempos de ejecución se escriben en notación O grande.

Comprender los algoritmos

SEGUNDA ACTUALIZACIÓN

Aditya Y. Bhargava

Los algoritmos que utilizas con más frecuencia ya han sido descubiertos, probados y comprobados. La segunda actualización de *Comprender los algoritmos* hace que sea muy fácil aprenderlos, comprenderlos y usarlos. Con explicaciones maravillosamente sencillas, más de 400 ilustraciones divertidas y decenas de ejemplos significativos, aprenderás a desbloquear el poder de los algoritmos en tu trabajo diario y a prepararte para tu próxima entrevista técnica de programación, ¡sin necesidad de saber nada de matemáticas!

Este libro te enseña los algoritmos más importantes para acelerar programas, simplificar el código y resolver problemas comunes de programación. Comenzarás con tareas como ordenar y buscar, y luego desarrollarás tus habilidades para abordar problemas más avanzados, como la compresión de datos y la inteligencia artificial. Incluso aprenderás a comparar las diferencias de rendimiento entre algoritmos. Además, esta nueva actualización incluye contenido revisado sobre árboles, problemas NP-completos y código actualizado a Python 3.

¿Qué encontrarás?

- Algoritmos de búsqueda, ordenamiento y grafos.
- Estructuras de datos como matrices, listas, tablas *hash*, árboles y grafos.
- Algoritmos NP-completos y voraces.
- Ejercicios y ejemplos de código en cada capítulo.

No se requieren habilidades matemáticas ni conocimientos de programación avanzados.

Aditya Y. Bhargava es ingeniero de software con doble formación en Informática y Bellas Artes, y escribe un blog sobre programación en adit.io.

El editor técnico de este libro ha sido David Eisenstat.

«Desglosa un semestre de algoritmos y estructuras de datos con explicaciones y diagramas fácilmente comprensibles que no requieren conocimientos previos de matemáticas».

—Katie Sylor-Miller, Etsy

«Una rara combinación de instrucción efectiva y lectura agradable. No hay mejor recurso que *Comprender los algoritmos*».

—Ben Vinagre, Syntax.fm

«Un viaje bellamente ilustrado y lúcido a través del mundo de los algoritmos y las estructuras de datos».

—Stephen Diehl, Zerolink

«El primer libro que recomiendo a todos los ingenieros, tanto noveles como séniors, que quieran desarrollar aún más sus habilidades. Resulta brillante porque permite comprender realmente los algoritmos y no solo memorizar el código».

—Alexander Manning, Block

MANNING

ANAYA MULTIMEDIA

www.anayamultimedia.es

ISBN 978-84-415-5252-4

2315279