

**Bienvenida**

Estimado lector,

Bienvenido a la segunda edición de Grokking Algorithms, un libro introductorio sobre algoritmos que te enseñará los conceptos fundamentales de manera amigable y fácil de entender.

Cuando escribí la primera edición de este libro en 2016, mi objetivo era escribir de manera fácil de leer y hacer el viaje de aprendizaje lo más fluido posible. Desde entonces, casi 100,000 personas han leído este libro.

Con esta segunda edición, mi objetivo sigue siendo el mismo. En este libro, uso ilustraciones y ejemplos memorables para hacer que los conceptos se mantengan. El libro está diseñado para lectores que saben programar y quieren aprender más sobre algoritmos sin requerir conocimientos matemáticos.

Creo que entender algoritmos es esencial para todo programador, ya sea que estés construyendo sitios web, aplicaciones móviles o trabajando en análisis de datos complejos. Los algoritmos están en todas partes, desde motores de búsqueda hasta autos autónomos, y nos proporcionan herramientas poderosas para resolver problemas y crear soluciones eficientes.

A lo largo de este libro, aprenderás sobre estructuras de datos, grafos, recursión y mucho más. Te mostraré técnicas para resolver problemas que los programadores enfrentan todos los días, con ejemplos prácticos. Además de algunos ajustes a lo largo del libro y algunas adiciones a capítulos existentes, esta segunda edición añade dos capítulos sobre árboles, un tema que muchos lectores han solicitado. Los árboles son una estructura de datos fundamental. Cubro árboles de búsqueda binaria, árboles AVL y árboles B. También hay nuevas notas sobre rendimiento. Por ejemplo, la primera edición discute las compensaciones entre arrays y listas enlazadas. Allí había dicho que las listas enlazadas desperdician menos memoria, pero la realidad es más matizada. Los arrays a menudo son más eficientes en memoria. "Profundizo más en matices como este en la segunda edición. También he ampliado la sección sobre completitud NP, y he añadido una explicación sobre NP-difícil versus NP-completo.

Este libro está destinado a ser agradable de leer. Espero que lo encuentres interesante y atractivo. Por favor, siéntete libre de proporcionar tus comentarios, observaciones y preguntas en el foro de discusión de [liveBook](https://livebook.manning.com/book/grokking-algorithms-second-edition/discussion) [discussion forum.](https://livebook.manning.com/book/grokking-algorithms-second-edition/discussion) . Tu aporte es esencial para hacer de este libro el mejor recurso posible para aprender algoritmos.

Gracias por elegir Grokking Algorithms (segunda edición), y te deseo un aprendizaje feliz.

Mis mejores deseos, —Aditya Bhargava

**Contenido Breve**

1. Introducción a algoritmos
2. Selection sort
3. Recursión
4. Quicksort
5. Tablas hash
6. Búsqueda en anchura
7. Árboles
8. Árboles de búsqueda binaria
9. Algoritmo de Dijkstra
10. Algoritmos codiciosos
11. Programación dinámica
12. K vecinos más cercanos
13. A dónde ir después

**APÉNDICES**

A. Rendimiento de árboles AVL

B. Respuestas a ejercicio

****

**Introducción a**

**Algoritmos**

****

**En este capítulo**

• Obtienes una base para el resto del libro

• Escribes tu primer algoritmo de búsqueda (búsqueda binaria)

**** • Aprendes cómo hablar sobre el tiempo de ejecución de un algoritmo (notación Big O)

**Introducción**

Un algoritmo es un conjunto de instrucciones para realizar una tarea. Cada pieza de código podría llamarse un algoritmo, pero este libro cubre las partes más interesantes. Elegí los algoritmos en este libro para su inclusión porque son rápidos, o resuelven problemas interesantes, o ambos. Aquí hay algunos puntos destacados:

• El Capítulo 1 habla sobre búsqueda binaria y muestra cómo un algoritmo puede acelerar tu código. ¡En un ejemplo, el número de pasos necesarios baja de 4 mil millones a 32!

• Un dispositivo GPS usa algoritmos de grafos (como aprenderás en los capítulos 6, 7 y 8) para calcular la ruta más corta a tu destino.

• Puedes usar programación dinámica (discutida en el capítulo 9) para escribir un algoritmo de IA que juegue damas.

En cada caso, describiré el algoritmo y te daré un ejemplo. Luego hablaré sobre el tiempo de ejecución del algoritmo en notación Big O. Finalmente, exploraré qué otros tipos de problemas podrían ser resueltos por el mismo algoritmo.

**Lo que aprenderás sobre rendimiento**

# La buena noticia es que una implementación de cada algoritmo en este libro probablemente está disponible en tu lenguaje favorito, ¡así que no tienes que escribir cada algoritmo tú mismo! Pero esas implementaciones son inútiles si no entiendes las compensaciones. En este libro, aprenderás a comparar compensaciones entre diferentes algoritmos: ¿Deberías usar merge sort o quicksort? ¿Deberías usar un array o una lista? Solo usar una estructura de datos diferente puede hacer una gran diferencia.

**Lo que aprenderás sobre resolver problemas**

# Aprenderás técnicas para resolver problemas que podrían haber estado fuera de tu alcance hasta ahora. Por ejemplo:

# • Si te gusta hacer videojuegos, puedes escribir un sistema de IA que siga al usuario usando algoritmos de grafos.

# • Aprenderás a hacer un sistema de recomendaciones usando k-vecinos más cercanos.

# • ¡Algunos problemas no son solucionables de manera oportuna! La parte de este libro que habla sobre problemas NP-completos te muestra cómo identificar esos problemas y crear un algoritmo que te dé una respuesta aproximada.

# Más generalmente, al final de este libro, conocerás algunos de los algoritmos más ampliamente aplicables. Luego puedes usar tu nuevo conocimiento para aprender sobre algoritmos más específicos para IA, bases de datos, y así sucesivamente. O puedes enfrentar desafíos más grandes en el trabajo.

**Lo que necesitas saber**

# Necesitarás saber álgebra básica antes de comenzar este libro. En particular, toma esta función: f(x) = x × 2. ¿Qué es f(5)? Si respondiste 10, estás listo.

# Adicionalmente, este capítulo (y este libro) será más fácil de seguir si estás familiarizado con un lenguaje de programación. Todos los ejemplos en este libro están en Python. Si no conoces ningún lenguaje de programación y quieres aprender uno, elige Python—es genial para principiantes. Si conoces otro lenguaje, como Ruby, estarás bien.

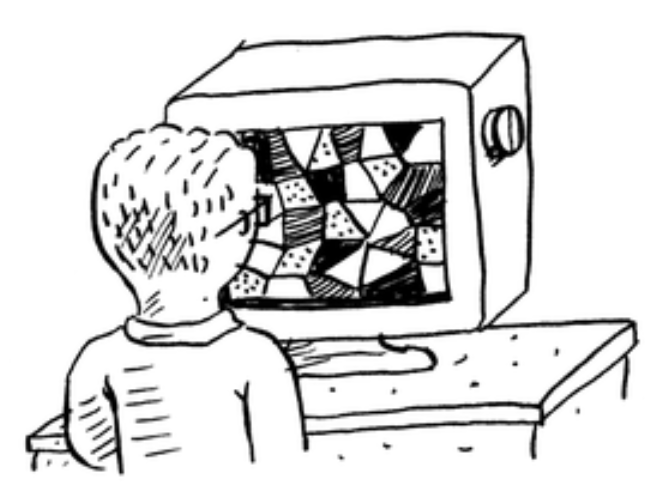
**Búsqueda Binaria**

Supón que estás buscando una persona en la guía telefónica (¡qué oración tan anticuada!). Su nombre comienza con K. Podrías comenzar al principio y seguir volteando páginas hasta llegar a las K. Pero es más probable que comiences en una página en el medio, porque sabes que las K van a estar cerca del medio de la guía telefónica.

O supón que estás buscando una palabra en el diccionario, y comienza con O. De nuevo, comenzarás cerca del medio.

Ahora supón que te conectas a Facebook. Cuando lo haces, Facebook tiene que verificar que tienes una cuenta en el sitio. Así que necesita buscar tu nombre de usuario en su base de datos. Supón que tu nombre de usuario es karlmageddon. Facebook podría comenzar desde las A y buscar tu nombre—pero tiene más sentido que comience en algún lugar en el medio.

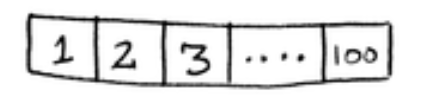
Este es un problema de búsqueda. Y todos estos casos usan el mismo algoritmo para resolver el problema: búsqueda binaria.

La búsqueda binaria es un algoritmo; su entrada es una lista ordenada de elementos (explicaré más tarde por qué necesita estar ordenada). Si un elemento que estás buscando está en esa lista, la búsqueda binaria devuelve la posición donde está ubicado. De lo contrario, la búsqueda binaria devuelve null.

Por ejemplo:

Esta y no esta en la guía telefónica(libro) #

**Buscando compañías en una guía telefónica con búsqueda binaria**

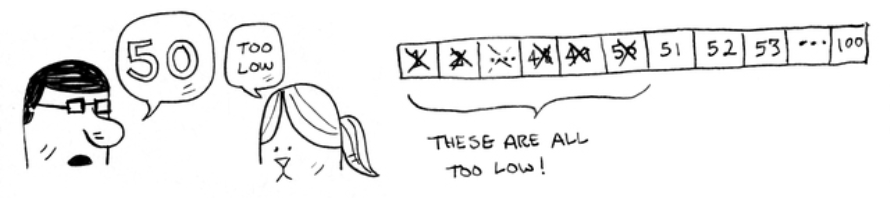
Aquí hay un ejemplo de cómo funciona la búsqueda binaria. Estoy pensando en un número entre 1 y 100.

Tienes que tratar de adivinar mi número en la menor cantidad de intentos posible. Con cada adivinanza, te diré si tu adivinanza es demasiado baja, demasiado alta, o correcta.

Supón que comienzas adivinando así: 1, 2, 3, 4 …. Así es como iría.

**Un mal enfoque para adivinar números**

Esta es búsqueda simple (tal vez búsqueda estúpida sería un mejor término). Con cada adivinanza, estás eliminando solo un número. Si mi número fuera 99, ¡podría tomarte 99 adivinanzas llegar ahí!

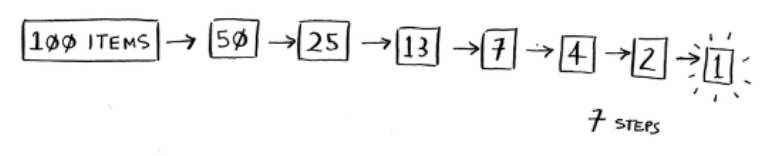
**Una mejor manera de buscar**

(Muy bajo)(todos estas por debajo)

Aquí hay una mejor técnica. Comienza con 50.

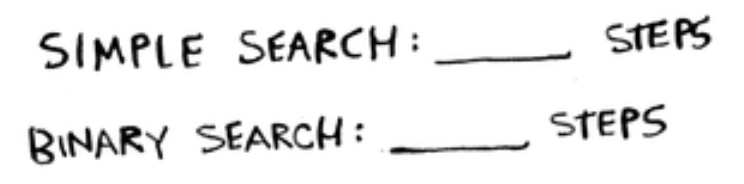
Demasiado bajo, ¡pero acabas de eliminar la mitad de los números! Ahora sabes que 1–50 son todos demasiado bajos. Siguiente adivinanza: 75.

Demasiado alto, ¡pero de nuevo redujiste a la mitad los números restantes! Con búsqueda binaria, adivinas el número del medio y eliminas la mitad de los números restantes cada vez. Siguiente es 63 (a medio camino entre 50 y 75).

¡Esta es búsqueda binaria! Acabas de aprender tu primer algoritmo! Aquí está cuántos números puedes eliminar cada vez.

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

Cualquiera que sea el número en el que estoy pensando, puedes adivinarlo en un máximo de siete adivinanzas—¡porque eliminas tantos números con cada adivinanza!

Supón que estás buscando una palabra en el diccionario. El diccionario tiene 240,000 palabras. En el peor caso, ¿cuántos pasos crees que tomará cada búsqueda?

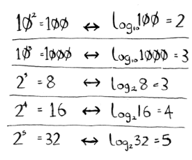
La búsqueda simple podría tomar 240,000 pasos si la palabra que estás buscando es la última en el libro. Con cada paso de búsqueda binaria, reduces el número de palabras a la mitad hasta que te quedes con solo una palabra

.

Así que la búsqueda binaria tomará 18 pasos—¡una gran diferencia! En general, para cualquier lista de n, la búsqueda binaria tomará log₂ n pasos para ejecutarse en el peor caso, mientras que la búsqueda simple tomará n pasos.

**Logaritmos**

Puede que no recuerdes qué son los logaritmos, pero probablemente sabes qué son los exponenciales. log₁₀ 100 es como preguntar, "¿Cuántos 10s multiplicamos juntos para obtener 100?" La respuesta es 2: 10 × 10. Así que log₁₀ 100 = 2. Los logs son lo inverso de los exponenciales.



**Los logs son lo inverso de los exponenciales.**

En este libro, cuando hablo sobre tiempo de ejecución en notación Big O (explicado un poco más tarde), log siempre significa log₂. Cuando buscas un elemento usando búsqueda simple, en el peor caso podrías tener que mirar cada elemento individual. Así que para una lista de 8 números, tendrías que verificar 8 números como máximo. Para búsqueda binaria, tienes que verificar log n elementos en el peor caso. Para una lista de 8 elementos, log 8 == 3, porque 2³ == 8. Así que para una lista de 8 números, tendrías que verificar 4 números como máximo. Para una lista de 1,024 elementos, log 1,024 = 10, porque 2¹⁰ == 1,024. Así que para una lista de 1,024 números, tendrías que verificar 10 números como máximo.

**NOTA** Hablaré sobre tiempo log mucho en este libro, así que deberías entender el concepto de logaritmos. Si no lo haces, Khan Academy (khanacademy.org) tiene un buen video que lo hace claro.

**NOTA** La búsqueda binaria solo funciona cuando tu lista está en orden ordenado. Por ejemplo, los nombres en una guía telefónica están ordenados en orden alfabético, así que puedes usar búsqueda binaria para buscar un nombre. ¿Qué pasaría si los nombres no estuvieran ordenados?

Veamos cómo escribir búsqueda binaria en Python. La muestra de código aquí usa arrays. Si no sabes cómo funcionan los arrays, no te preocupes; están cubiertos en el siguiente capítulo. Solo necesitas saber que puedes almacenar una secuencia de elementos en una fila de cubos consecutivos llamada array. Los cubos están numerados comenzando con 0: el primer cubo está en la posición #0, el segundo está en #1, el tercero en #2, y así sucesivamente.

**NOTA** Verás que uso los términos lista y array indistintamente en el código. Esto es porque en Python, los arrays se llaman listas.

La función binary search toma un array ordenado y un elemento. Si el elemento está en el array, la función devuelve su posición. Mantendrás seguimiento de qué parte del array tienes que buscar. Al principio, esto es todo el array:

low = 0

high = len(arr) – 1



Cada vez, verificas el elemento del medio:

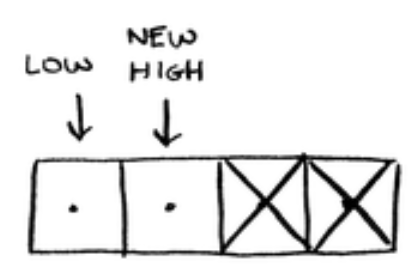
mid = (low + high) // 2 #A

guess = arr[mid]

#A mid se redondea hacia abajo automáticamente por Python si (low + high) no es un número par.

Si la adivinanza es demasiado baja, actualizas low apropiadamente:

if guess < item:

 low = mid + 1

Y si la adivinanza es demasiado alta, actualizas high. Aquí está el código completo:

def binary\_search(arr, item):

low = 0 #A

high = len(arr) - 1 #A

while low <= high: #B

mid = (low + high) // 2 #C

guess = arr[mid]

if guess == item: #D

return mid

elif guess > item: #E

high = mid - 1

else: #F

low = mid + 1

return None #G

my\_list = [1, 3, 5, 7, 9] #H

print(binary\_search(my\_list, 3)) # => 1 #I

print(binary\_search(my\_list, -1)) # => None #J

#A low y high mantienen seguimiento de qué parte de la lista buscarás.

#B Mientras no la hayas reducido a un elemento …

#C … verifica el elemento del medio.

#D Encontraste el elemento.

#E La adivinanza fue demasiado alta.

#F La adivinanza fue demasiado baja.

#G El elemento no existe.

#H ¡Vamos a probarlo!

#I Recuerda, las listas comienzan en 0. El segundo slot tiene índice 1.

#J "None" significa nil en Python. Indica que el elemento no fue encontrado.

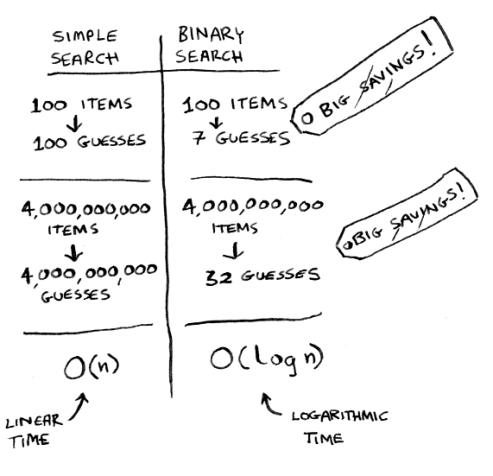
**Ejercicios**

1. **1.1** Supón que tienes una lista ordenada de 128 nombres, y estás buscando a través de ella usando búsqueda binaria. ¿Cuál es el número máximo de pasos que tomaría?
2. **1.2** Supón que duplicas el tamaño de la lista. ¿Cuál es el número máximo de pasos ahora?

**Tiempo de Ejecución**

Cada vez que hablo sobre un algoritmo, discutiré su tiempo de ejecución. Generalmente quieres elegir el algoritmo más eficiente—ya sea que estés tratando de optimizar para tiempo o espacio.

Volvamos a la búsqueda binaria. ¿Cuánto tiempo ahorras al usarla? Bueno, el primer enfoque era verificar cada número, uno por uno. Si esta es una lista de 100 números, toma hasta 100 adivinanzas. Si es una lista de 4 mil millones de números, toma hasta 4 mil millones de adivinanzas. Así que el número máximo de adivinanzas es igual al tamaño de la lista. Esto se llama tiempo lineal.

La búsqueda binaria es diferente. Si la lista tiene 100 elementos, toma como máximo 7 adivinanzas. Si la lista tiene 4 mil millones de elementos, toma como máximo 32 adivinanzas. ¡Poderoso, eh! La búsqueda binaria se ejecuta en tiempo logarítmico (o tiempo log, como la mayoría de la gente lo llama). Aquí hay una tabla que resume nuestros hallazgos de hoy.

**Tiempos de ejecución para algoritmos de búsqueda**

**Notación Big O**

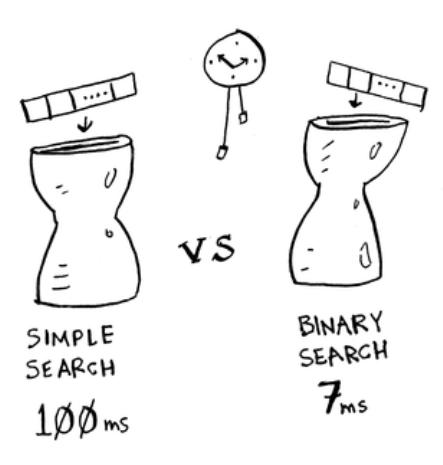
La notación Big O es notación especial que te dice qué tan rápido es un algoritmo. ¿A quién le importa? Bueno, resulta que usarás algoritmos de otras personas a menudo—y cuando lo hagas, es bueno entender qué tan rápidos o lentos son. En esta sección, explicaré qué es la notación Big O y te daré una lista de los tiempos de ejecución más comunes para algoritmos usando ella.

**Los tiempos de ejecución de algoritmos crecen a diferentes tasas**

Bob está escribiendo un algoritmo de búsqueda para la NASA. Su algoritmo se activará cuando un cohete esté a punto de aterrizar en la Luna, y ayudará a calcular dónde aterrizar.

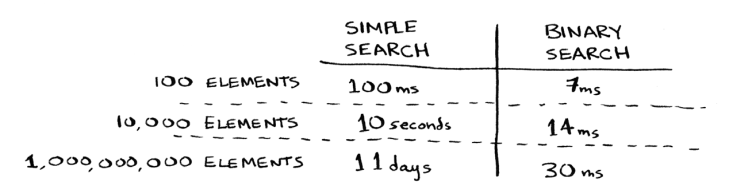


Este es un ejemplo de cómo el tiempo de ejecución de dos algoritmos puede crecer a diferentes tasas. Bob está tratando de decidir entre búsqueda simple y búsqueda binaria. El algoritmo necesita ser tanto rápido como correcto. Por un lado, la búsqueda binaria es más rápida. Y Bob solo tiene 10 segundos para averiguar dónde aterrizar—de lo contrario, el cohete estará fuera de curso. Por otro lado, la búsqueda simple es más fácil de escribir, y hay menos posibilidad de que se introduzcan errores. Y Bob realmente no quiere errores en el código para aterrizar un cohete! Para ser extra cuidadoso, Bob decide cronometrar ambos algoritmos con una lista de 100 elementos.

Asumamos que toma 1 milisegundo verificar un elemento. Con búsqueda simple, Bob tiene que verificar 100 elementos, así que la búsqueda toma 100 ms para ejecutarse. Por otro lado, solo tiene que verificar 7 elementos con búsqueda binaria (log₂ 100 es aproximadamente 7), así que esa búsqueda toma 7 ms para ejecutarse. Pero realistamente, la lista tendrá más como mil millones de elementos. Si es así, ¿cuánto tiempo tomará la búsqueda simple? ¿Cuánto tiempo tomará la búsqueda binaria? Asegúrate de tener una respuesta para cada pregunta antes de continuar leyendo.

**Tiempo de ejecución para búsqueda simple vs. búsqueda binaria, con una lista de 100 elementos**

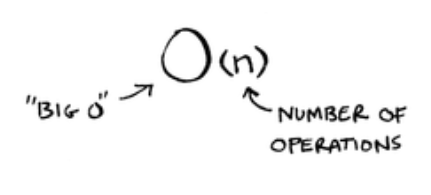
Bob ejecuta búsqueda binaria con 1 mil millones de elementos, y toma 30 ms (log₂ 1,000,000,000 es aproximadamente 30). "¡30 ms!" piensa. "La búsqueda binaria es aproximadamente 15 veces más rápida que la búsqueda simple, porque la búsqueda simple tomó 100 ms con 100 elementos, y la búsqueda binaria tomó 7 ms. Así que la búsqueda simple tomará 30 × 15 = 450 ms, ¿verdad? Muy por debajo de mi umbral de 10 segundos." Bob decide ir con búsqueda simple. ¿Es esa la elección correcta?

No. Resulta que Bob está equivocado. Muy equivocado. El tiempo de ejecución para búsqueda simple con 1 mil millón de elementos será 1 mil millón ms, ¡que son 11 días! El problema es que los tiempos de ejecución para búsqueda binaria y búsqueda simple no crecen a la misma tasa.

**¡Los tiempos de ejecución crecen a velocidades muy diferentes!**

Es decir, a medida que aumenta el número de elementos, la búsqueda binaria toma un poco más de tiempo para ejecutarse. Pero la búsqueda simple toma mucho más tiempo para ejecutarse. 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 búsqueda simple. Bob pensó que la búsqueda binaria era 15 veces más rápida que la búsqueda simple, pero eso no es correcto. Si la lista tiene 1 mil millón de elementos, es más como 33 millones de veces más rápida. Es por eso que no es suficiente saber cuánto tiempo toma un algoritmo para ejecutarse—necesitas saber cómo aumenta el tiempo de ejecución a medida que aumenta el tamaño de la lista. Ahí es donde entra la notación Big O.

La notación Big O te dice qué tan rápido es un algoritmo. Por ejemplo, supón que tienes una lista de tamaño n. La búsqueda simple necesita verificar cada elemento, así que tomará n operaciones. El tiempo de ejecución en notación Big O es O(n). ¿Dónde están los segundos? No los hay—Big O no te dice la velocidad en segundos. La notación Big O te permite comparar el número de operaciones. Te dice qué tan rápido crece el algoritmo.

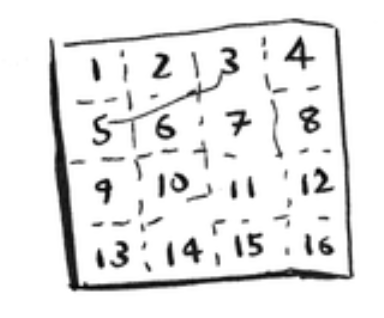
Aquí hay otro ejemplo. La búsqueda binaria necesita log n operaciones para verificar una lista de tamaño n. ¿Cuál es el tiempo de ejecución en notación Big O? Es O(log n). En general, la notación Big O se escribe de la siguiente manera.

**Cómo se ve la notación Big O**

Esto te dice el número de operaciones que hará un algoritmo. Se llama notación Big O porque pones una "O grande" delante del número de operaciones (¡suena como una broma, pero es verdad!).

Ahora veamos algunos ejemplos. Ve si puedes averiguar el tiempo de ejecución para estos algoritmos.

**Visualizando diferentes tiempos de ejecución Big O**

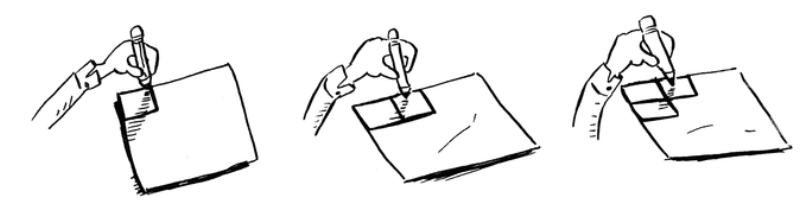
Aquí hay un ejemplo práctico que puedes seguir en casa con unas pocas hojas de papel y un lápiz. Supón que tienes que dibujar una cuadrícula de 16 cajas.

¿Cuál es un buen algoritmo para dibujar esta cuadrícula?

**ALGORITMO 1**

Una forma de hacerlo es dibujar 16 cajas, una a la vez.

Recuerda, la notación Big O cuenta el número de operaciones. En este ejemplo, dibujar una caja es una operación. Tienes que dibujar 16 cajas. ¿Cuántas operaciones tomará, dibujando una caja a la vez?



**Dibujando una cuadrícula una caja a la vez**

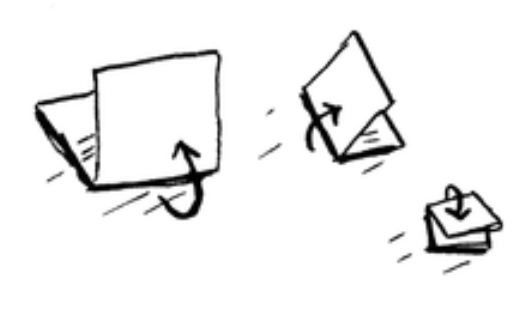
Toma 16 pasos dibujar 16 cajas. ¿Cuál es el tiempo de ejecución para este algoritmo?

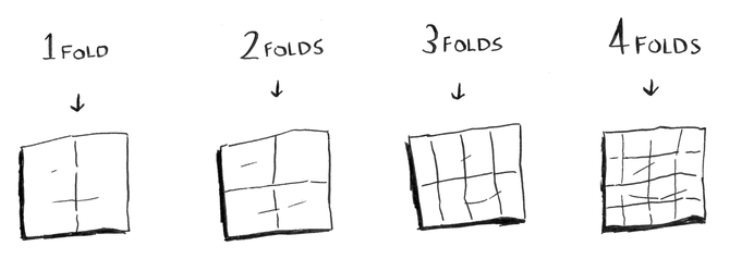
**ALGORITMO 2**

Prueba este algoritmo en su lugar. Dobla el papel.



En este ejemplo, doblar el papel una vez es una operación. ¡Acabas de hacer dos cajas con esa operación!

Dobla el papel de nuevo, y de nuevo, y de nuevo.

¡Desdobla después de cuatro dobleces, y tendrás una hermosa cuadrícula! Cada doblez duplica el número de cajas. Puedes "dibujar" el doble de cajas con cada doblez, así que puedes dibujar 16 cajas en 4 pasos. ¿Cuál es el tiempo de ejecución para este algoritmo? Piensa en los tiempos de ejecución para ambos algoritmos antes de continuar.

**Dibujando una cuadrícula en cuatro dobleces**

Puedes "dibujar" el doble de cajas con cada doblez, así que puedes dibujar 16 cajas en 4 pasos. ¿Cuál es el tiempo de ejecución para este algoritmo? Piensa en los tiempos de ejecución para ambos algoritmos antes de continuar.

Respuestas: El Algoritmo 1 toma tiempo O(n), y el algoritmo 2 toma tiempo O(log n).

**Big O establece un tiempo de ejecución de peor caso**

Supón que estás usando búsqueda simple para buscar una persona en la guía telefónica. Sabes que la búsqueda simple toma tiempo O(n) para ejecutarse, lo que significa que en el peor caso, tendrás que mirar cada entrada individual en tu guía telefónica. En este caso, estás buscando a Adit. Este tipo es la primera entrada en tu guía telefónica. Así que no tuviste que mirar cada entrada—lo encontraste en el primer intento. ¿Tomó este algoritmo tiempo O(n)? ¿O tomó tiempo O(1) porque encontraste a la persona en el primer intento?

La búsqueda simple aún toma tiempo O(n). En este caso, encontraste lo que buscabas al instante. Ese es el escenario del mejor caso. Pero estamos usando la notación Big O para análisis de escenario de peor caso. Así que puedes decir que, en el peor caso, tendrás que mirar cada entrada en la guía telefónica una vez. Eso es tiempo O(n). Es una garantía—sabes que la búsqueda simple nunca será más lenta que tiempo O(n).

**NOTA** Junto con el tiempo de ejecución de peor caso, también es importante mirar el tiempo de ejecución de caso promedio. Peor caso versus caso promedio se discute en el capítulo 4.

**Algunos tiempos de ejecución Big O comunes**

Aquí hay cinco tiempos de ejecución Big O que encontrarás mucho, ordenados de más rápido a más lento:

• **O(log n)**, también conocido como tiempo log. Ejemplo: Búsqueda binaria.

• **O(n)**, también conocido como tiempo lineal. Ejemplo: Búsqueda simple.

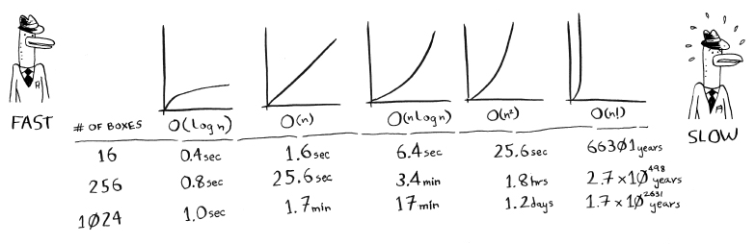
• **O(n \* log n)**. Ejemplo: Un algoritmo de ordenamiento rápido, como quicksort (próximamente en el capítulo 4).

• **O(n²)**. Ejemplo: Un algoritmo de ordenamiento lento, como selection sort (próximamente en el capítulo 2).

• **O(n!)**. Ejemplo: Un algoritmo realmente lento, como el vendedor viajero (¡próximamente!).

Supón que estás dibujando una cuadrícula de 16 cajas de nuevo, y puedes elegir entre 5 algoritmos diferentes para hacerlo. Si usas el primer algoritmo, tomará tiempo O(log n) dibujar la cuadrícula. Puedes hacer 10 operaciones por segundo. Con tiempo O(log n), te tomará 4 operaciones dibujar una cuadrícula de 16 cajas (log 16 es 4). Así que te tomará 0.4 segundos dibujar la cuadrícula. ¿Qué pasa si tienes que dibujar 1,024 cajas? Te tomará log 1,024 = 10 operaciones, o 1 segundo para dibujar una cuadrícula de 1,024 cajas. Estos números están usando el primer algoritmo.

El segundo algoritmo es más lento: toma tiempo O(n). Tomará 16 operaciones dibujar 16 cajas, y tomará 1,024 operaciones dibujar 1,024 cajas. ¿Cuánto tiempo es eso en segundos?

Aquí está cuánto tiempo tomaría dibujar una cuadrícula para el resto de los algoritmos, de más rápido a más lento:

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

Esta es una simplificación. En realidad no puedes convertir de un tiempo de ejecución Big O a un número de operaciones tan fácilmente, pero esto es suficientemente bueno por ahora. Volveremos a la notación Big O en el capítulo 4, después de que hayas aprendido algunos algoritmos más. Por ahora, las principales conclusiones son las siguientes:

• La velocidad del algoritmo no se mide en segundos, sino en crecimiento del número de operaciones. • En lugar de segundos, hablamos sobre qué tan rápidamente aumenta el tiempo de ejecución de un algoritmo a medida que aumenta el tamaño de la entrada. • El tiempo de ejecución de algoritmos se expresa en notación Big O. • O(log n) es más rápido que O(n), pero se vuelve mucho más rápido a medida que crece la lista de elementos que estás buscando.

**Ejercicios**

Da el tiempo de ejecución para cada uno de estos escenarios en términos de Big O.

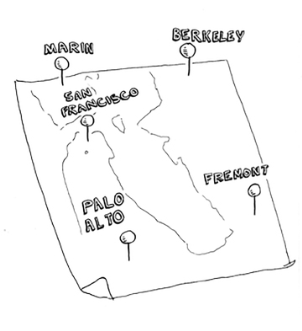
1. **1.3** Tienes un nombre, y quieres encontrar el número de teléfono de la persona en la guía telefónica.
2. **1.4** Tienes un número de teléfono, y quieres encontrar el nombre de la persona en la guía telefónica. (Pista: ¡Tendrás que buscar a través de todo el libro!)
3. **1.5** Quieres leer los números de cada persona en la guía telefónica.
4. **1.6** Quieres leer los números de solo las A. (¡Esta es complicada! Involucra conceptos que se cubren más en el capítulo 4. Lee la respuesta—¡puedes sorprenderte!)

**El vendedor viajero**

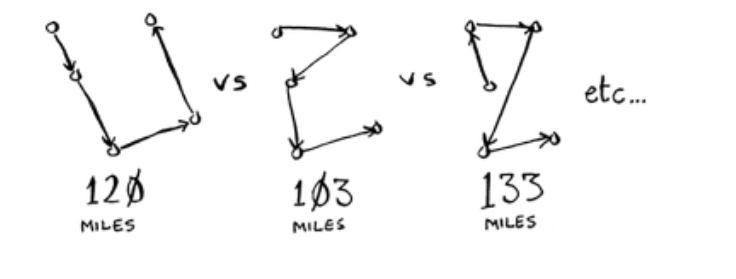
Podrías haber leído esa última sección y pensado, "No hay manera de que me encuentre con un algoritmo que tome tiempo O(n!)." Bueno, ¡déjame tratar de probarte que estás equivocado! Aquí hay un ejemplo de un algoritmo con un tiempo de ejecución realmente malo. Este es un problema famoso en ciencias de la computación, porque su crecimiento es horrible y algunas personas muy inteligentes piensan que no puede ser mejorado. Se llama el problema del vendedor viajero.

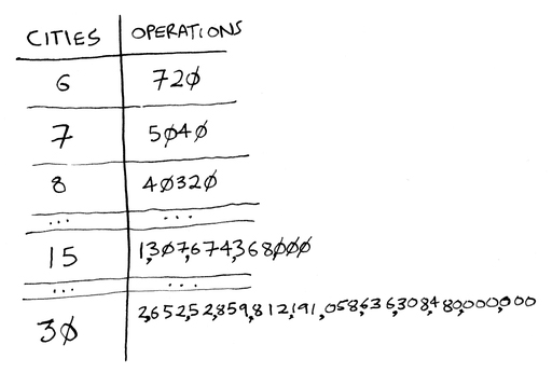


Tienes un vendedor.

El vendedor tiene que ir a cinco ciudades.

Este vendedor, a quien llamaré Opus, quiere visitar las cinco ciudades mientras viaja la distancia mínima. Aquí hay una forma de hacer eso: mirar cada orden posible en el que podría viajar a las ciudades.



Él suma la distancia total y luego elige el camino con la distancia más baja. Hay 120 permutaciones con 5 ciudades, así que tomará 120 operaciones resolver el problema para 5 ciudades. Para 6 ciudades, tomará 720 operaciones (hay 720 permutaciones). Para 7 ciudades, ¡tomará 5,040 operaciones!

**El número de operaciones aumenta drásticamente.**

En general, para n elementos, tomará n! (n factorial) operaciones calcular el resultado. Así que esto es tiempo O(n!), o tiempo factorial. Toma muchas operaciones para todo excepto los números más pequeños. Una vez que estés lidiando con 100+ ciudades, es imposible calcular la respuesta a tiempo—¡el Sol colapsará primero!

¡Este es un algoritmo terrible! Opus debería usar uno diferente, ¿verdad? Pero no puede. Este es uno de los problemas no resueltos en ciencias de la computación. No hay algoritmo rápido conocido para él, y algunas personas inteligentes piensan que es imposible tener un algoritmo inteligente para este problema. Lo mejor que podemos hacer es crear una solución aproximada; ve el capítulo 10 para más.

**Resumen**

• La búsqueda binaria es mucho más rápida que la búsqueda simple a medida que tu array se hace más grande.

• O(log n) es más rápido que O(n), pero se vuelve mucho más rápido una vez que la lista de elementos a través de los que estás buscando crece.

• La velocidad del algoritmo no se mide en segundos.

• Los tiempos de algoritmos se miden en términos de crecimiento de un algoritmo.

• Los tiempos de algoritmos se escriben en notación Big O.



**Selection**

**Sort**



****

**En este capítulo**

• Aprendes sobre arrays y listas enlazadas—dos de las estructuras de datos más básicas. Se usan absolutamente en todas partes. Ya usaste arrays en el capítulo 1, y los usarás en casi todos los capítulos de este libro. Los arrays son un tema crucial, ¡así que presta atención! Pero a veces es mejor usar una lista enlazada en lugar de un array. Este capítulo explica los pros y contras de ambos para que puedas decidir cuál es correcto para tu algoritmo.

• Aprendes tu primer algoritmo de ordenamiento. Muchos algoritmos solo funcionan si tus datos están ordenados. Recuerda la búsqueda binaria? Solo puedes ejecutar búsqueda binaria en una lista ordenada de elementos. Este capítulo te enseña selection sort. La mayoría de los lenguajes tienen un algoritmo de ordenamiento incorporado, así que rara vez necesitarás escribir tu propia versión desde cero. Pero selection sort es un paso hacia quicksort, que cubriré en el capítulo 4. Quicksort es un algoritmo importante, y será más fácil de entender si ya conoces un algoritmo de ordenamiento.

**Lo que necesitas saber**

Para entender las partes de análisis de rendimiento en este capítulo, necesitas conocer la notación Big O y logaritmos. Si no los conoces, sugiero que vuelvas y leas el capítulo 1. La notación Big O se usará a lo largo del resto del libro.

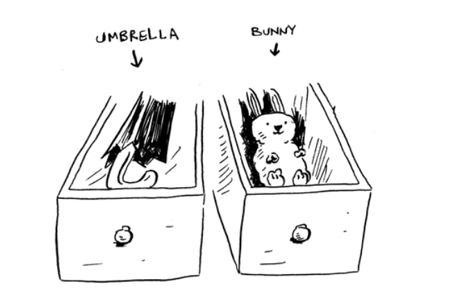
**Cómo funciona la memoria**

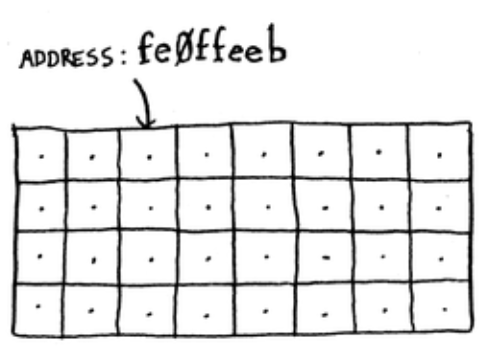
Imagina que vas a un espectáculo y necesitas revisar tus cosas. Una cómoda está disponible.



Cada cajón puede contener un elemento. Quieres almacenar dos cosas, así que pides dos cajones.

Almacenas tus dos cosas aquí.

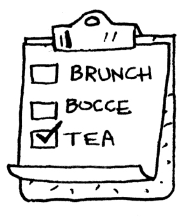


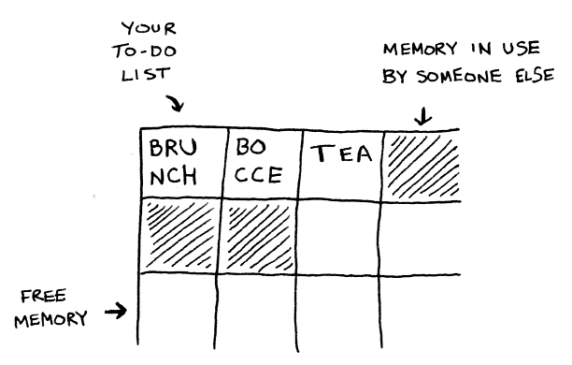
¡Y estás listo para el espectáculo! Así es básicamente como funciona la memoria de tu computadora. Tu computadora se ve como un conjunto gigante de cajones, y cada cajón tiene una dirección.

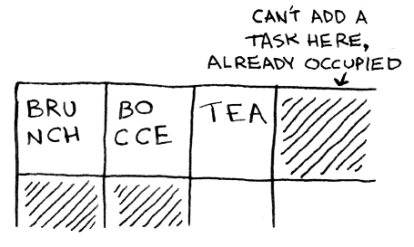
fe0ffeeb es la dirección de un slot en memoria.

Cada vez que quieres almacenar un elemento en memoria, le pides a la computadora algo de espacio, y te da una dirección donde puedes almacenar tu elemento. Si quieres almacenar múltiples elementos, hay dos formas básicas de hacerlo: arrays y listas enlazadas. Hablaré sobre arrays y listas a continuación, así como los pros y contras de cada uno. No hay una forma correcta de almacenar elementos para cada caso de uso, así que es importante conocer las diferencias.

**Arrays y listas enlazadas**

A veces necesitas almacenar una lista de elementos en memoria. Supón que estás escribiendo una aplicación para gestionar tus tareas pendientes. Querrás almacenar las tareas como una lista en memoria.

¿Deberías usar un array o una lista enlazada? Almacenemos las tareas en un array primero, porque es más fácil de entender. Usar un array significa que todas tus tareas se almacenan contiguamente (justo una al lado de la otra) en memoria.

Ahora supón que quieres agregar una cuarta tarea. ¡Pero el siguiente cajón está ocupado por las cosas de otra persona!

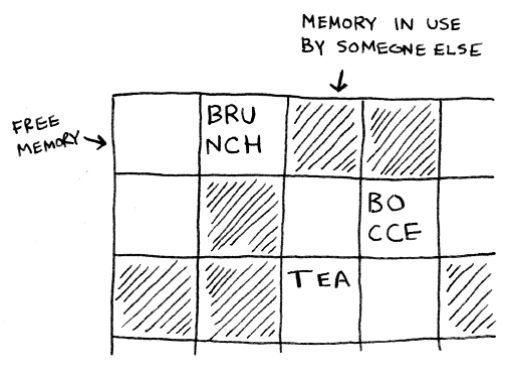
Es como ir al cine con tus amigos y encontrar un lugar para sentarse—pero otro amigo se une a ti, y no hay lugar para ellos. Tienes que moverte a un nuevo lugar donde todos quepan. En este caso, necesitas pedirle a tu computadora un trozo diferente de memoria que pueda caber cuatro tareas. Luego necesitas mover todas tus tareas allí.

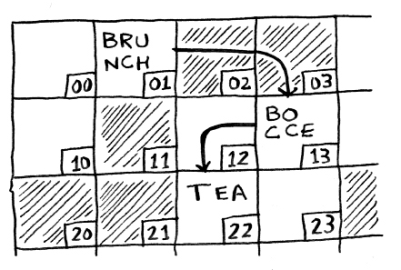
Si otro amigo viene, te quedas sin lugar de nuevo—¡y todos tienen que moverse una segunda vez! Qué molestia. De manera similar, agregar nuevos elementos a un array puede ser muy molesto. Si te quedas sin espacio y necesitas moverte a un nuevo lugar en memoria cada vez, agregar un nuevo elemento será realmente lento. Una solución fácil es "reservar asientos": incluso si solo tienes 3 elementos en tu lista de tareas, puedes pedirle a la computadora 10 slots, por si acaso. Entonces puedes agregar 10 elementos a tu lista de tareas sin tener que moverte. Esta es una buena solución, pero deberías estar consciente de un par de desventajas:

• Puede que no necesites los slots extra que pediste, y entonces esa memoria será desperdiciada. No la estás usando, pero nadie más puede usarla tampoco. • Puedes agregar más de 10 elementos a tu lista de tareas y tener que moverte de todos modos.

Así que es una buena solución, pero no es una solución perfecta. Las listas enlazadas resuelven este problema de agregar elementos.

**Listas enlazadas**

Con listas enlazadas, tus elementos pueden estar en cualquier lugar en memoria.

 Cada elemento almacena la dirección del siguiente elemento en la lista. Un montón de direcciones de memoria aleatorias están enlazadas juntas.

**Direcciones de memoria enlazadas**

Es como una búsqueda del tesoro. Vas a la primera dirección, y dice, "El siguiente elemento se puede encontrar en la dirección 123." Así que vas a la dirección 123, y dice, "El siguiente elemento se puede encontrar en la dirección 847," y así sucesivamente. Agregar un elemento a una lista enlazada es fácil: lo colocas en cualquier lugar en memoria y almacenas la dirección con el elemento anterior.

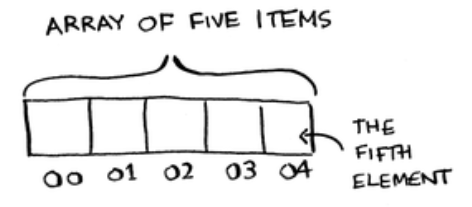
Con listas enlazadas, nunca tienes que mover tus elementos. También evitas otro problema. Digamos que vas a una película popular con cinco de tus amigos. Los seis están tratando de encontrar un lugar para sentarse, pero el teatro está lleno. No hay seis asientos juntos. Bueno, a veces esto sucede con arrays. Digamos que estás tratando de encontrar 10,000 slots para un array. Tu memoria tiene 10,000 slots, pero no tiene 10,000 slots juntos. ¡No puedes obtener espacio para tu array! Una lista enlazada es como decir, "Separémonos y veamos la película." Si hay espacio en memoria, tienes espacio para tu lista enlazada.

Si las listas enlazadas son tan mejores en inserciones, ¿para qué son buenos los arrays?

**Arrays**

Los sitios web con listas de top-10 usan una táctica sucia para obtener más visualizaciones de página. En lugar de mostrarte la lista en una página, ponen un elemento en cada página y te hacen hacer clic en Siguiente para llegar al siguiente elemento en la lista. Por ejemplo, Top 10 Mejores Villanos de TV no te mostrará la lista completa en una página. En lugar de eso, comienzas en #10 (Newman), y tienes que hacer clic en Siguiente en cada página para llegar a #1 (Gustavo Fring). Esta técnica le da a los sitios web 10 páginas completas en las que mostrarte anuncios, pero es aburrido hacer clic Siguiente 9 veces para llegar a #1. Sería mucho mejor si toda la lista estuviera en una página y pudieras hacer clic en el nombre de cada persona para más información.

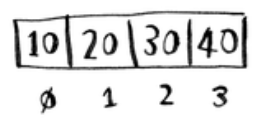
Las listas enlazadas tienen un problema similar. Supón que quieres leer el último elemento en una lista enlazada. No puedes simplemente leerlo, porque no sabes en qué dirección está. En lugar de eso, tienes que ir al elemento #1 para obtener la dirección del elemento #2. Luego tienes que ir al elemento #2 para obtener la dirección del elemento #3. Y así sucesivamente, hasta llegar al último elemento. Las listas enlazadas son geniales si vas a leer todos los elementos uno a la vez: puedes leer un elemento, seguir la dirección al siguiente elemento, y así sucesivamente. Pero si vas a estar saltando, las listas enlazadas son terribles.

Los arrays son diferentes. Conoces la dirección de cada elemento en tu array. Por ejemplo, supón que tu array contiene cinco elementos, y sabes que comienza en la dirección 00. ¿Cuál es la dirección del elemento #5?

Las matemáticas simples te dicen: es 04. Los arrays son geniales si quieres leer elementos aleatorios, porque puedes buscar cualquier elemento en tu array al instante. Con una lista enlazada, los elementos no están uno al lado del otro, así que no puedes calcular instantáneamente la posición del quinto elemento en memoria—tienes que ir al primer elemento para obtener la dirección del segundo elemento, luego ir al segundo elemento para obtener la dirección del tercer elemento, y así sucesivamente hasta llegar al quinto elemento.

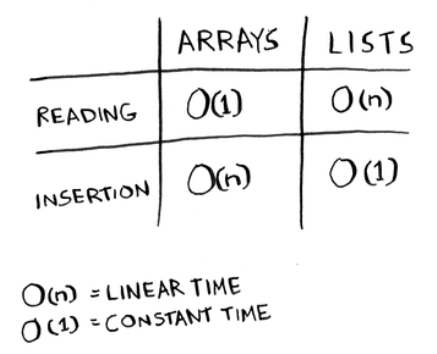
**Terminología**

Los elementos en un array están numerados. Esta numeración comienza desde 0, no 1. Por ejemplo, en este array, 20 está en la posición 1.



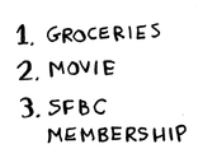
Y 10 está en la posición 0. Esto usualmente confunde a los nuevos programadores. Comenzar en 0 hace que todo tipo de código basado en arrays sea más fácil de escribir, así que los programadores se han mantenido con eso. Casi todos los lenguajes de programación que uses numerarán elementos de array comenzando en 0. Te acostumbrarás pronto.

La posición de un elemento se llama su índice. Así que en lugar de decir, "20 está en la posición 1," la terminología correcta es, "20 está en el índice 1." Usaré índice para significar posición a lo largo de este libro.

Aquí están los tiempos de ejecución para operaciones comunes en arrays y listas.

**Pregunta:** ¿Por qué toma tiempo O(n) insertar un elemento en un array? Supón que quisieras insertar un elemento al principio de un array. ¿Cómo lo harías? ¿Cuánto tiempo tomaría? Encuentra las respuestas a estas preguntas en la siguiente sección!

**Ejercicio**

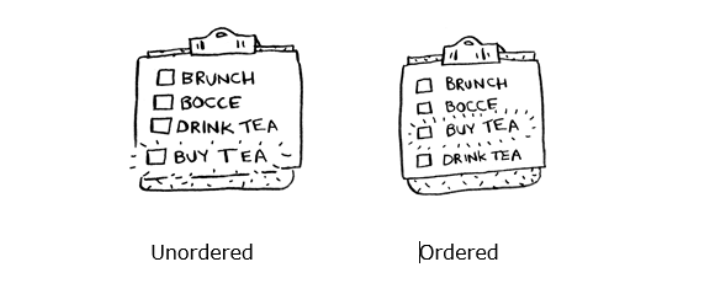
1. **2.1** Supón que estás construyendo una aplicación para llevar seguimiento de tus finanzas.

Cada día, escribes todo en lo que gastaste dinero. Al final del mes, revisas tus gastos y sumas cuánto gastaste. Así que tienes muchas inserciones y unas pocas lecturas. ¿Deberías usar un array o una lista?

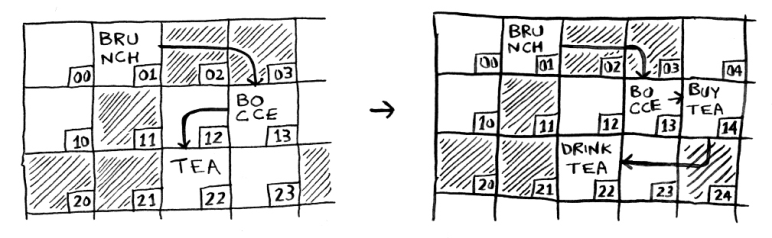
**Insertando en el medio de una lista**

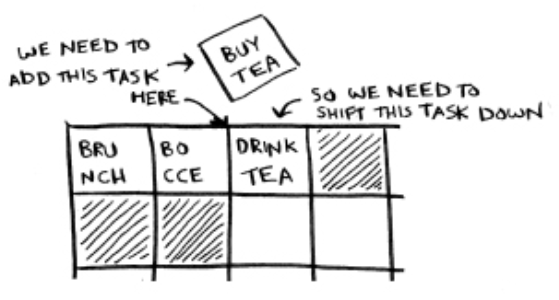
Supón que quieres que tu lista de tareas funcione más como un calendario. Anteriormente, estabas agregando cosas al final de la lista.

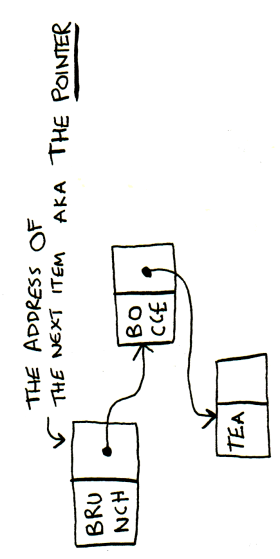
Ahora quieres agregarlas en el orden en que deben hacerse.



¿Qué es mejor si quieres insertar elementos en el medio: arrays o listas? Con listas, es tan fácil como cambiar a qué apunta el elemento anterior.



Pero para arrays, tienes que desplazar todos los elementos restantes hacia abajo.

¡Y si no hay espacio, podrías tener que copiar todo a una nueva ubicación! Las listas son mejores si quieres insertar elementos en el medio.

He hablado mucho sobre cómo en una lista enlazada, cada elemento apunta al siguiente elemento en la lista. Pero ¿cómo lo hace exactamente? Usando punteros.

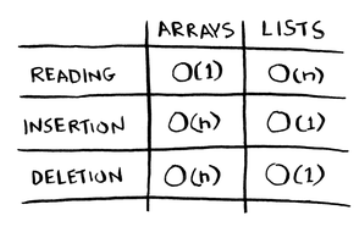
Con cada elemento en tu lista enlazada, usas un poco de memoria para almacenar la dirección del siguiente elemento. Esto se llama un puntero.

Escucharás la palabra punteros aparecer a veces, especialmente si escribes usando un lenguaje de bajo nivel como C. Así que es bueno saber lo que significa.

**Eliminaciones**

¿Qué pasa si quieres eliminar un elemento? De nuevo, las listas son mejores, porque solo necesitas cambiar a qué apunta el elemento anterior. Con arrays, todo necesita moverse hacia arriba cuando eliminas un elemento.

A diferencia de las inserciones, las eliminaciones siempre funcionarán. Las inserciones pueden fallar a veces cuando no queda espacio en memoria. Pero siempre puedes eliminar un elemento.

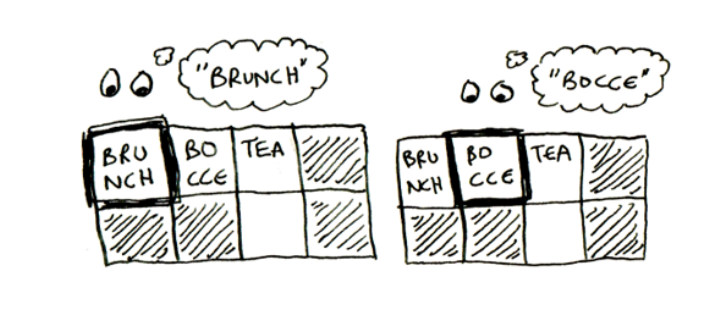
Aquí están los tiempos de ejecución para operaciones comunes en arrays y listas enlazadas.

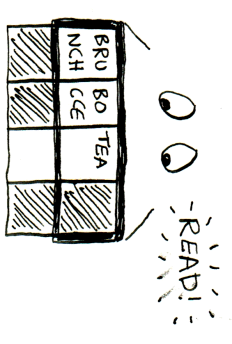
Vale la pena mencionar que las inserciones y eliminaciones son tiempo O(1) solo si puedes acceder instantáneamente al elemento a ser eliminado. Es una práctica común mantener seguimiento del primer y último elementos en una lista enlazada, así que tomaría solo tiempo O(1) eliminar esos.

¿Cuál se usa más, arrays o listas enlazadas? Los arrays se usan a menudo porque tienen muchas ventajas sobre las listas enlazadas. Primero que todo, son mejores en lecturas. Los arrays proporcionan acceso aleatorio.

Hay dos tipos diferentes de acceso: acceso aleatorio y acceso secuencial. Acceso secuencial significa leer los elementos uno por uno, comenzando en el primer elemento. Las listas enlazadas solo pueden hacer acceso secuencial. Si quieres leer el décimo elemento de una lista enlazada, tienes que leer los primeros 9 elementos y seguir los enlaces al décimo elemento. Acceso aleatorio significa que puedes saltar directamente al décimo elemento. Los arrays proporcionan acceso aleatorio. Muchos casos de uso requieren acceso aleatorio, así que los arrays se usan mucho.

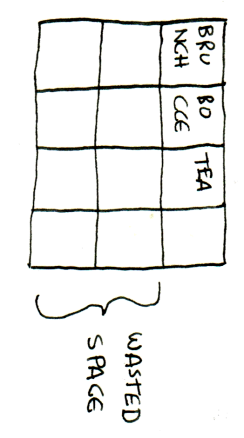
Incluso más allá del acceso aleatorio, los arrays son más rápidos porque pueden usar caché. Tal vez estés imaginando lecturas como esto, leyendo un elemento a la vez:



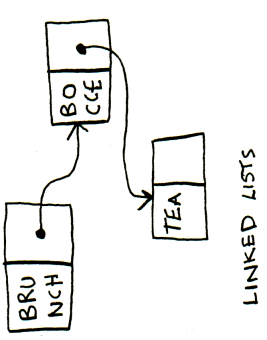
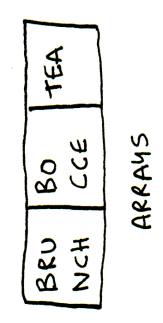
Pero en realidad, las computadoras leen una sección completa a la vez, porque eso hace que sea mucho más rápido ir al siguiente elemento:

Esto es algo que puedes hacer con arrays. Con un array, puedes leer una sección completa de elementos. ¡Pero no puedes hacer esto con una lista enlazada! No sabes dónde está el siguiente elemento. Necesitas leer un elemento, averiguar dónde está el siguiente elemento, luego leer el siguiente elemento.

Así que no solo los arrays te dan acceso aleatorio, ¡también proporcionan acceso secuencial más rápido!

Los arrays son mejores para lecturas. ¿Qué hay de la eficiencia de memoria? ¿Recuerdas antes que había dicho que con arrays, típicamente solicitas más espacio del que necesitas, y si no terminas usando esa memoria extra que solicitaste, se desperdicia?

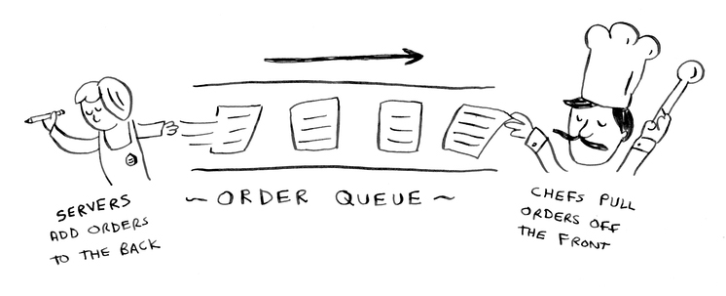
Bueno, en realidad, no hay mucho espacio desperdiciado así. Por otro lado, cuando usas una lista enlazada, estás usando memoria extra por elemento, porque necesitas almacenar la dirección del siguiente elemento. Así que las listas enlazadas tomarán más espacio si cada elemento es bastante pequeño. Aquí está la misma información como un array y una lista enlazada. Puedes ver que la lista enlazada toma más espacio:



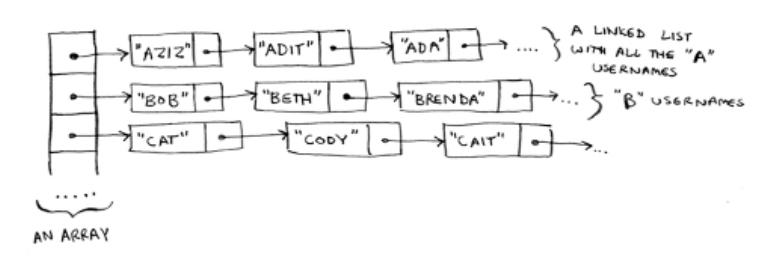
Por supuesto, si cada elemento es grande, entonces incluso un solo slot de espacio desperdiciado puede ser un gran problema, y esa memoria extra que estás usando para almacenar los punteros puede sentirse bastante pequeña en comparación.

Así que los arrays se usan más a menudo que las listas enlazadas excepto en casos de uso específicos.

**Ejercicios**

1. **2.2** Supón que estás construyendo una aplicación para restaurantes para tomar pedidos de clientes. Tu aplicación necesita almacenar una lista de pedidos. Los meseros siguen agregando pedidos a esta lista, y los chefs toman pedidos de la lista y los hacen. Es una cola de pedidos: los meseros agregan pedidos al final de la cola, y el chef toma el primer pedido de la cola y lo cocina.

¿Usarías un array o una lista enlazada para implementar esta cola? (Pista: Las listas enlazadas son buenas para inserciones/eliminaciones, y los arrays son buenos para acceso aleatorio. ¿Cuál vas a estar haciendo aquí?)

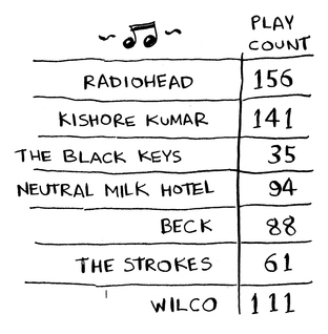
1. **2.3** Hagamos un experimento mental. Supón que Facebook mantiene una lista de nombres de usuario. Cuando alguien trata de conectarse a Facebook, se hace una búsqueda de su nombre de usuario. Si su nombre está en la lista de nombres de usuario, pueden conectarse. La gente se conecta a Facebook bastante seguido, así que hay muchas búsquedas a través de esta lista de nombres de usuario. Supón que Facebook usa búsqueda binaria para buscar en la lista. La búsqueda binaria necesita acceso aleatorio—necesitas poder llegar al medio de la lista de nombres de usuario instantáneamente. Sabiendo esto, ¿implementarías la lista como un array o una lista enlazada?
2. **2.4** La gente se registra en Facebook bastante seguido también. Supón que decidiste usar un array para almacenar la lista de usuarios. ¿Cuáles son las desventajas de un array para inserciones? En particular, supón que estás usando búsqueda binaria para buscar logins. ¿Qué pasa cuando agregas nuevos usuarios a un array?
3. **2.5** En realidad, Facebook no usa ni un array ni una lista enlazada para almacenar información de usuario. Consideremos una estructura de datos híbrida: un array de listas enlazadas. Tienes un array con 26 slots. Cada slot apunta a una lista enlazada. Por ejemplo, el primer slot en el array apunta a una lista enlazada que contiene todos los nombres de usuario que comienzan con a. El segundo slot apunta a una lista enlazada que contiene todos los nombres de usuario que comienzan con b, y así sucesivamente.

Supón que Adit B se registra en Facebook, y quieres agregarlo a la lista. Vas al slot 1 en el array, vas a la lista enlazada para el slot 1, y agregas Adit B al final. Ahora, supón que quieres buscar a Zakhir H. Vas al slot 26, que apunta a una lista enlazada de todos los nombres Z. Luego buscas a través de esa lista para encontrar a Zakhir H.

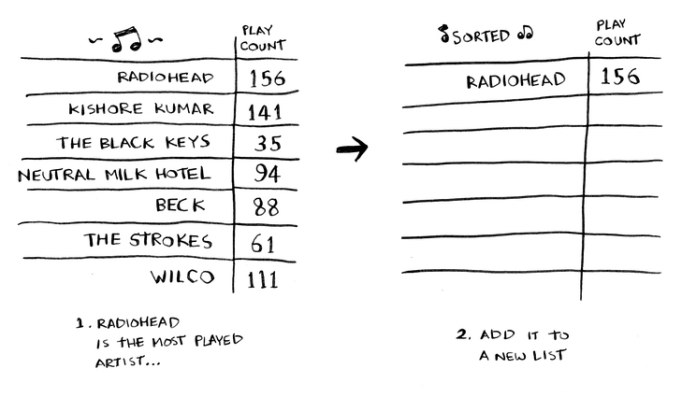
Compara esta estructura de datos híbrida con arrays y listas enlazadas. ¿Es más lenta o más rápida que cada una para buscar e insertar? No tienes que dar tiempos de ejecución Big O, solo si la nueva estructura de datos sería más rápida o más lenta.

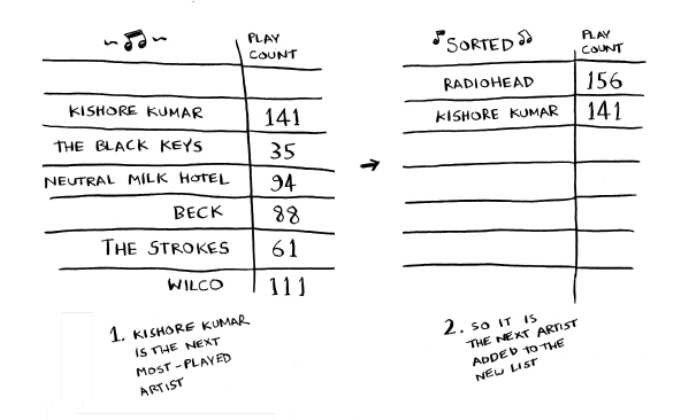
**Selection Sort**

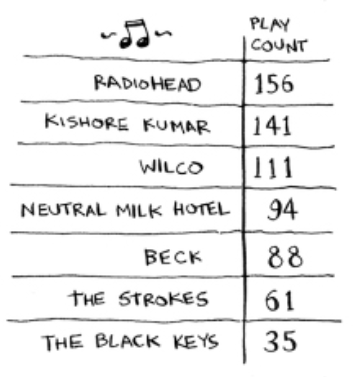
Juntemos todo para aprender tu segundo algoritmo: selection sort. Para seguir esta sección, necesitas entender arrays, así como notación Big O.

Supón que tienes un montón de música en tu computadora. Para cada artista, tienes un conteo de reproducciones.

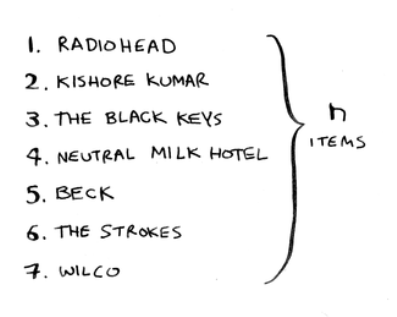
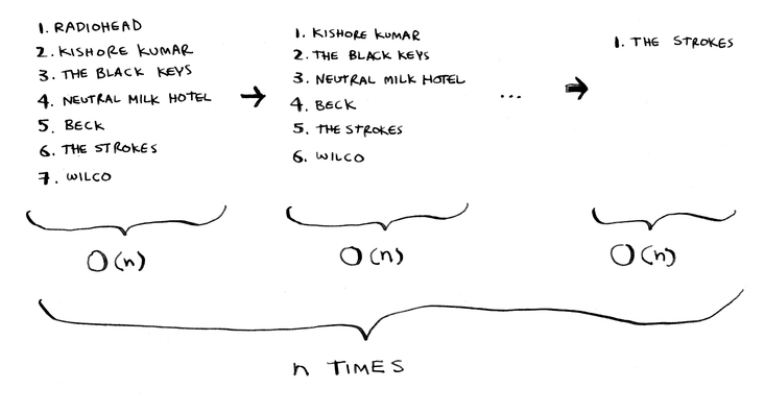
Quieres ordenar estos artistas de más a menos reproducidos, para que puedas clasificar tus artistas favoritos. ¿Cómo puedes hacerlo?

Una forma es repasar la lista y encontrar el artista más reproducido. Agrega ese artista a una nueva lista.

Hazlo de nuevo para encontrar el siguiente artista más reproducido.

Sigue haciendo esto, y terminarás con una lista ordenada.

Pongámonos nuestros sombreros de ciencias de la computación y veamos cuánto tiempo tomará esto ejecutar. Recuerda que tiempo O(n) significa que tocas cada elemento en una lista una vez. Por ejemplo, ejecutar búsqueda simple sobre la lista de artistas significa mirar cada artista una vez.

Para encontrar el artista con el conteo de reproducción más alto, tienes que verificar cada elemento en la lista. Esto toma tiempo O(n), como acabas de ver. Así que tienes una operación que toma tiempo O(n), y tienes que hacer eso n veces:

Esto toma tiempo O(n × n) o O(n²).

Los algoritmos de ordenamiento son muy útiles. Ahora puedes ordenar

• Nombres en una guía telefónica

• Fechas de viaje

• Emails (del más nuevo al más viejo)

**Verificando menos elementos cada vez**

Tal vez te estés preguntando: a medida que realizas las operaciones, el número de elementos que tienes que verificar sigue disminuyendo. Eventualmente, te quedas teniendo que verificar solo un elemento. Entonces, ¿cómo puede el tiempo de ejecución seguir siendo O(n²)? Esa es una buena pregunta, y la respuesta tiene que ver con las constantes en la notación Big O. Entraré en esto más en el capítulo 4, pero aquí está la esencia.

Tienes razón en que no tienes que verificar una lista de n elementos cada vez. Verificas n elementos, luego n – 1, n - 2 … 2, 1. En promedio, verificas una lista que tiene 1/2 × n elementos. El tiempo de ejecución es O(n × 1/2 × n). Pero las constantes como 1/2 se ignoran en la notación Big O (de nuevo, ve el capítulo 4 para la discusión completa), así que simplemente escribes O(n × n) o O(n²).

Selection sort es un algoritmo ordenado, pero no es muy rápido. Quicksort es un algoritmo de ordenamiento más rápido que solo toma tiempo O(n log n). ¡Viene en el capítulo 4!

**Listado de código de ejemplo**

No te mostramos el código para ordenar la lista de música, pero lo siguiente es algo de código que hará algo muy similar: ordenar un array de menor a mayor. Escribamos una función para encontrar el elemento más pequeño en un array:

def findSmallest(arr):

smallest = arr[0] #A

smallest\_index = 0 #B

for i in range(1, len(arr)):

if arr[i] < smallest:

smallest = arr[i]

smallest\_index = i

return smallest\_index

#A Almacena el valor más pequeño

#B Almacena el índice del valor más pequeño

Ahora puedes usar esta función para escribir selection sort:

def selectionSort(arr): #A

newArr = []

copiedArr = list(arr) # copia array antes de mutar

for i in range(len(copiedArr)):

smallest = findSmallest(copiedArr) #B

newArr.append(copiedArr.pop(smallest))

return newArr

print(selectionSort([5, 3, 6, 2, 10]))

#A Ordena un array

#B Encuentra el elemento más pequeño en el array, y lo agrega al nuevo array

**Resumen**

• La memoria de tu computadora es como un conjunto gigante de cajones.

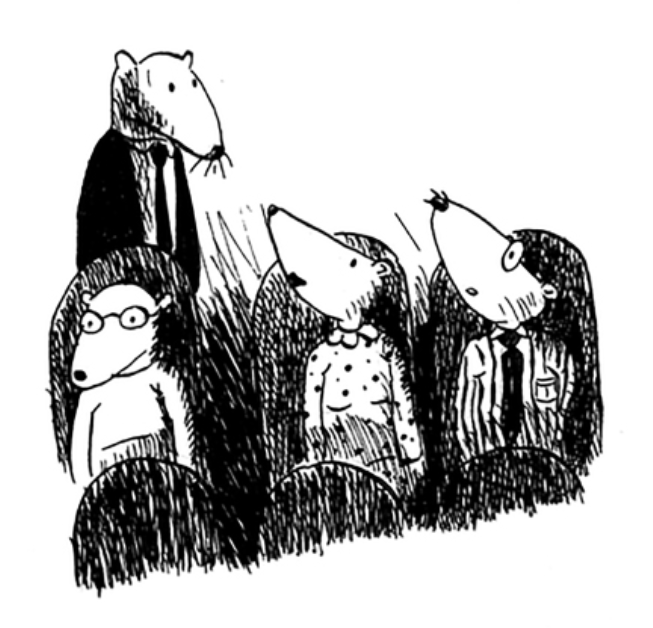
• Cuando quieres almacenar múltiples elementos, usa un array o una lista enlazada.

• Con un array, todos tus elementos se almacenan justo uno al lado del otro.

• Con una lista enlazada, los elementos están esparcidos por todas partes, y un elemento almacena la dirección del siguiente.

• Los arrays permiten lecturas rápidas.

• Las listas enlazadas permiten inserciones y eliminaciones rápidas.







**En este capítulo**

• Aprendes sobre recursión. La recursión es una técnica de codificación usada en muchos algoritmos. Es un bloque de construcción para entender capítulos posteriores en este libro.

****• Aprendes qué es un caso base y un caso recursivo. La estrategia de divide y vencerás (capítulo 4) usa este concepto simple para resolver problemas difíciles.

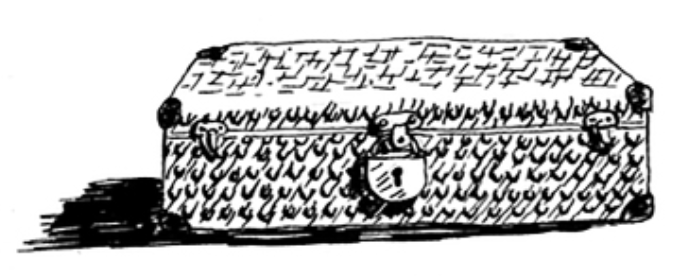
Estoy emocionado sobre este capítulo porque cubre recursión, una forma elegante de resolver problemas. La recursión es uno de mis temas favoritos, pero es divisivo. La gente o la ama o la odia, o la odia hasta que aprende a amarla unos años después. Personalmente estaba en ese tercer grupo. Para hacer las cosas más fáciles para ti, tengo algunos consejos:

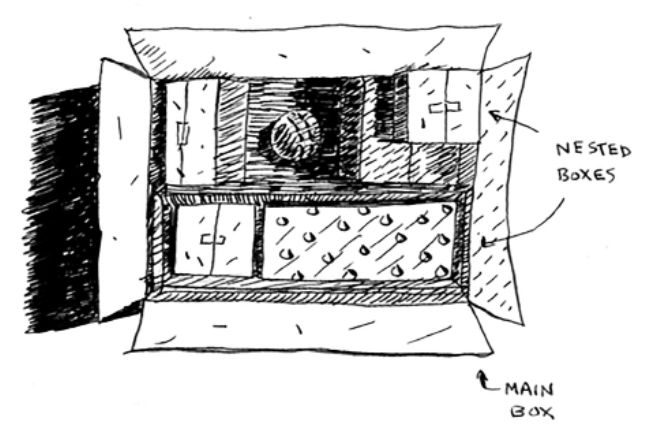
• Este capítulo tiene muchos ejemplos de código. Ejecuta el código tú mismo para ver cómo funciona.

• Hablaré sobre funciones recursivas. Al menos una vez, repasa una función recursiva con lápiz y papel: algo como, "Veamos, paso 5 a factorial, y luego devuelvo 5 veces pasar 4 a factorial, que es …," y así sucesivamente. Repasar una función como esta te enseñará cómo funciona una función recursiva.

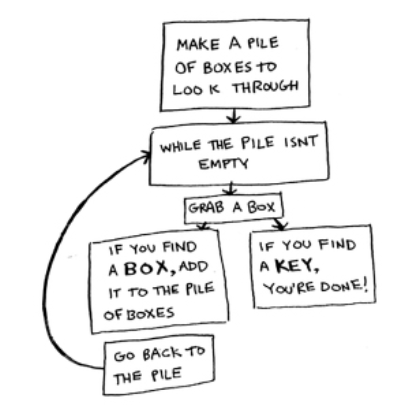
Este capítulo también incluye mucho pseudocódigo. El pseudocódigo es una descripción de alto nivel del problema que estás tratando de resolver, en código. Está escrito como código, pero está destinado a estar más cerca del habla humana.

**Recursión**

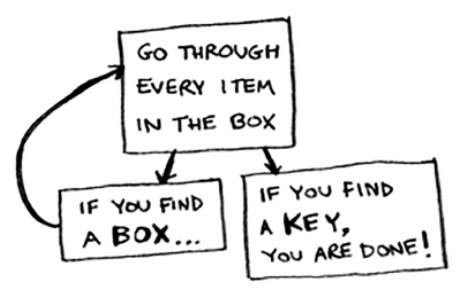
Supón que estás cavando a través del ático de tu abuela y te encuentras con una maleta misteriosa cerrada.

La abuela te dice que la llave para la maleta probablemente está en esta otra caja.

Esta caja contiene más cajas, con más cajas dentro de esas cajas. La llave está en una caja en algún lugar. ¿Cuál es tu algoritmo para buscar la llave? Piensa en un algoritmo antes de continuar leyendo.

Aquí hay un enfoque.

1. Haz una pila de cajas para revisar.
2. Toma una caja, y mira a través de ella.
3. Si encuentras una caja, agrégala a la pila para revisar después.
4. Si encuentras una llave, ¡terminaste!
5. Repite.

Aquí hay un enfoque alternativo.

1. Mira a través de la caja.
2. Si encuentras una caja, ve al paso 1.
3. Si encuentras una llave, ¡terminaste!

¿Qué enfoque te parece más fácil? El primer enfoque usa un bucle while. Mientras la pila no esté vacía, toma una caja y mira a través de ella. Aquí hay algo de pseudocódigo:

def look\_for\_key(main\_box):

pile = main\_box.make\_a\_pile\_to\_look\_through()

while pile is not empty:

box = pile.grab\_a\_box()

for item in box:

if item.is\_a\_box():

pile.append(item)

elif item.is\_a\_key():

print("found the key!")

La segunda forma usa recursión. La recursión es donde una función se llama a sí misma. Aquí está la segunda forma en pseudocódigo:

def look\_for\_key(box):

for item in box:

if item.is\_a\_box():

look\_for\_key(item) #A

elif item.is\_a\_key():

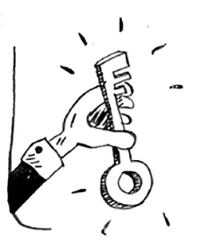
print("found the key!")

#A ¡Recursión!

Ambos enfoques logran lo mismo, pero el segundo enfoque es más claro para mí. La recursión se usa cuando hace la solución más clara. No hay beneficio de rendimiento al usar recursión; de hecho, los bucles a veces son mejores para el rendimiento. Me gusta esta cita de Leigh Caldwell en Stack Overflow: "Los bucles pueden lograr una ganancia de rendimiento para tu programa. La recursión puede lograr una ganancia de rendimiento para tu programador. ¡Elige cuál es más importante en tu situación!"

Muchos algoritmos importantes usan recursión, así que es importante entender el concepto.

https://stackoverflow.com/questions/72209/recursion-or-iteration/72694#72694

**Caso base y caso recursivo**

Como una función recursiva se llama a sí misma, es fácil escribir una función incorrectamente que termina en un bucle infinito. Por ejemplo, supón que quieres escribir una función que imprima una cuenta regresiva, así:

3...2...1

Puedes escribirla recursivamente, así:

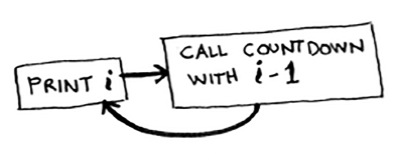
def countdown(i):

print(i)

countdown(i-1)

countdown(3)

Escribe este código y ejecútalo. Notarás un problema: ¡esta función se ejecutará para siempre!



**Bucle infinito**

3...2...1...0...-1...-2...

(Presiona Ctrl-C para matar tu script.)

Cuando escribes una función recursiva, tienes que decirle cuándo parar de recurrir. Por eso cada función recursiva tiene dos partes: el caso base, y el caso recursivo. El caso recursivo es cuando la función se llama a sí misma. El caso base es cuando la función no se llama a sí misma de nuevo … así que no entra en un bucle infinito.

Agreguemos un caso base a la función countdown:

def countdown(i):

print(i)

if i <= 1: #A

return

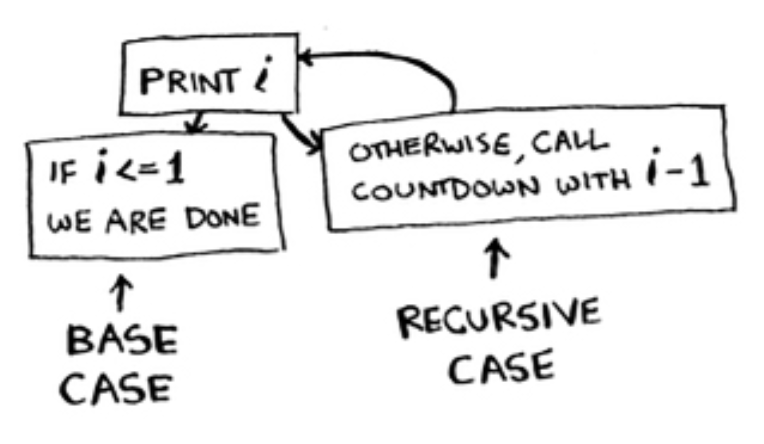
else: #B

countdown(i-1)

countdown(3)

#A Caso base

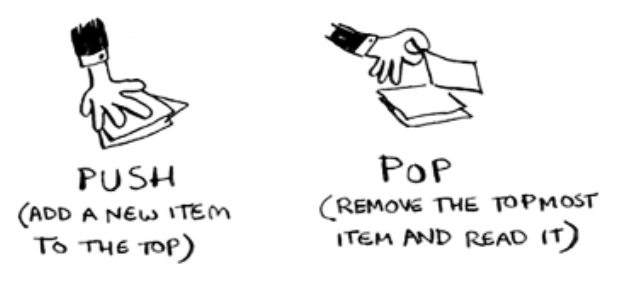
#B Caso recursivo

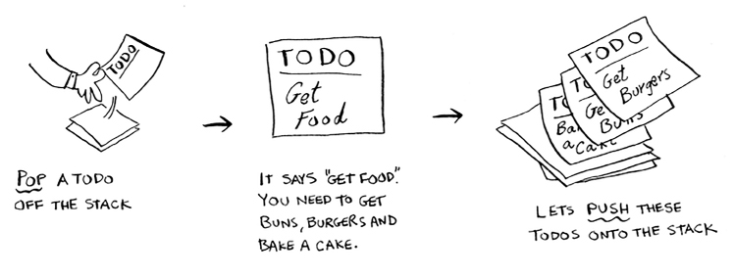
Ahora la función funciona como se esperaba. Va algo así.

**La pila**

Esta sección cubre la pila de llamadas. Es un concepto importante en programación. La pila de llamadas es un concepto importante en programación general, y también es importante entender cuando usas recursión.

Supón que estás organizando una barbacoa. Mantienes una lista de tareas por hacer para la barbacoa, en forma de una pila de notas adhesivas.

Recuerda cuando hablamos sobre arrays y listas, y tenías una lista de tareas por hacer? Podías agregar elementos de tarea en cualquier lugar de la lista o eliminar elementos aleatorios. La pila de notas adhesivas es mucho más simple. Cuando insertas un elemento, se agrega a la parte superior de la lista. Cuando lees un elemento, solo lees el elemento superior, y se quita de la lista. Así que tu lista de tareas por hacer solo tiene dos acciones: push (insertar) y pop (quitar y leer).

Veamos la lista de tareas por hacer en acción.

Esta estructura de datos se llama una pila. La pila es una estructura de datos simple. ¡Has estado usando una pila todo este tiempo sin darte cuenta!

**La pila de llamadas**

Tu computadora usa una pila internamente llamada la pila de llamadas. Veámosla en acción. Aquí hay una función simple:

def greet(name):

print("hello, " + name + "!")

greet2(name)

print("getting ready to say bye...")

bye()

Esta función te saluda y luego llama a otras dos funciones. Aquí están esas dos funciones:

def greet2(name):

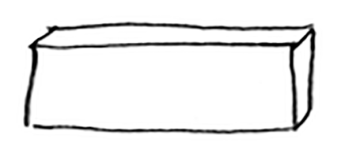
print("how are you, " + name + "?")

def bye():

print("ok bye!")

Repasemos qué pasa cuando llamas a una función.

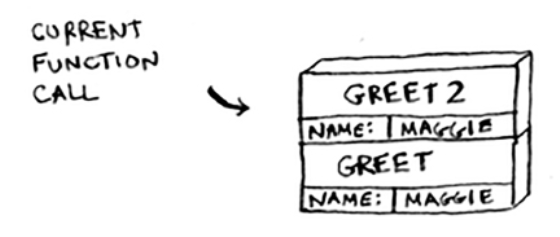
**NOTA** Para mantener las cosas simples, solo estoy mostrando las llamadas a greet y greet2, y no mostrando las llamadas a la función print.



Supón que llamas greet("maggie"). Primero, tu computadora asigna una caja de memoria para esa llamada de función.

Ahora usemos la memoria. La variable name se establece como "maggie". Eso necesita ser guardado en memoria.

Cada vez que haces una llamada de función, tu computadora guarda los valores para todas las variables para esa llamada en memoria como esto. A continuación, imprimes hello, maggie! Luego llamas greet2("maggie"). De nuevo, tu computadora asigna una caja de memoria para esta llamada de función.



Tu computadora está usando una pila para estas cajas. La segunda caja se agrega encima de la primera. Imprimes how are you, maggie? Luego regresas de la llamada de función. Cuando esto pasa, la caja en la parte superior de la pila se quita.



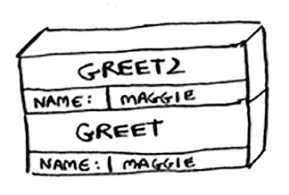
Ahora la caja superior en la pila es para la función greet, lo que significa que regresaste a la función greet. Cuando llamaste la función greet2, la función greet estaba parcialmente completada. Esta es la gran idea detrás de esta sección: cuando llamas una función desde otra función, la función llamadora se pausa en un estado parcialmente completado. Todos los valores de las variables para esa función todavía están almacenados en la pila de llamadas (es decir, en memoria). Ahora que terminaste con la función greet2, estás de vuelta a la función greet, y retomas donde lo dejaste. Primero imprimes getting ready to say bye…. Llamas la función bye.

Una caja para esa función se agrega a la parte superior de la pila. Luego imprimes ok bye! y regresas de la llamada de función

.

Y estás de vuelta a la función greet. No hay nada más que hacer, así que también regresas de la función greet. Esta pila, usada para guardar las variables para múltiples funciones, se llama la pila de llamadas.

**Ejercicio**

1. **3.1** Supón que te muestro una pila de llamadas como esta.

¿Qué información puedes darme, solo basado en esta pila de llamadas?

Ahora veamos la pila de llamadas en acción con una función recursiva.

**La pila de llamadas con recursión**

¡Las funciones recursivas también usan la pila de llamadas! Veamos esto en acción con la función factorial. factorial(5) se escribe como 5!, y se define así: 5! = 5 \* 4 \* 3 \* 2 \* 1. De manera similar, factorial(3) es 3 \* 2 \* 1. Aquí hay una función recursiva para calcular el factorial de un número:

def fact(x):

if x == 1:

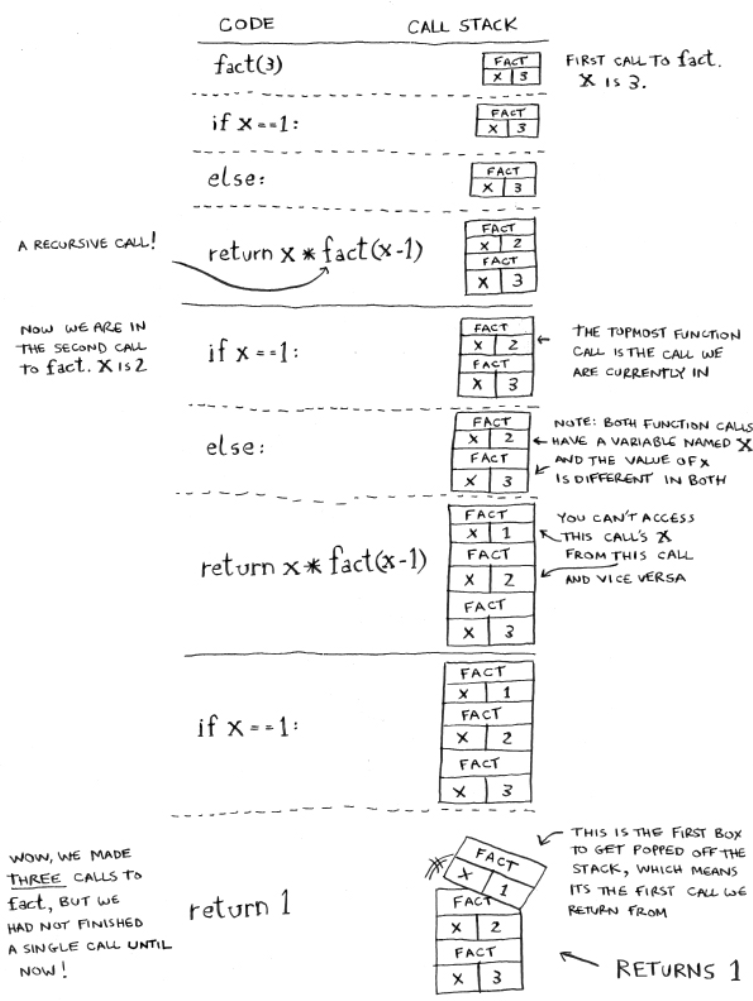
return 1

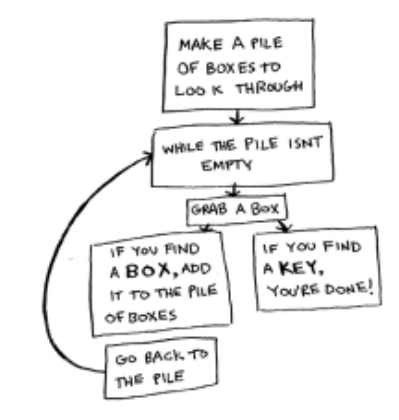
else:

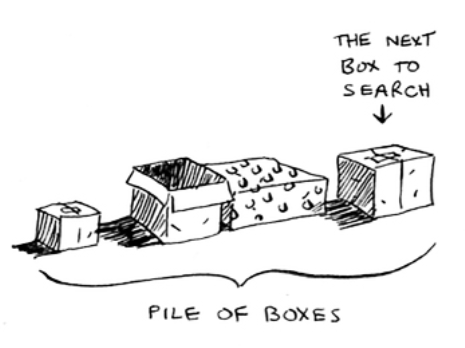
return x \* fact(x-1)

Ahora llamas fact(3). Repasemos esta llamada línea por línea y veamos cómo cambia la pila. Recuerda, la caja superior en la pila te dice qué llamada a fact estás actualmente.

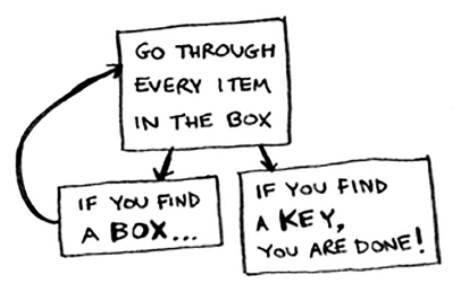
Nota que cada llamada a fact tiene su propia copia de x. No puedes acceder a la copia de x de una función diferente.

La pila juega un papel importante en la recursión. En el ejemplo de apertura, había dos enfoques para encontrar la llave. Aquí está la primera manera de nuevo.

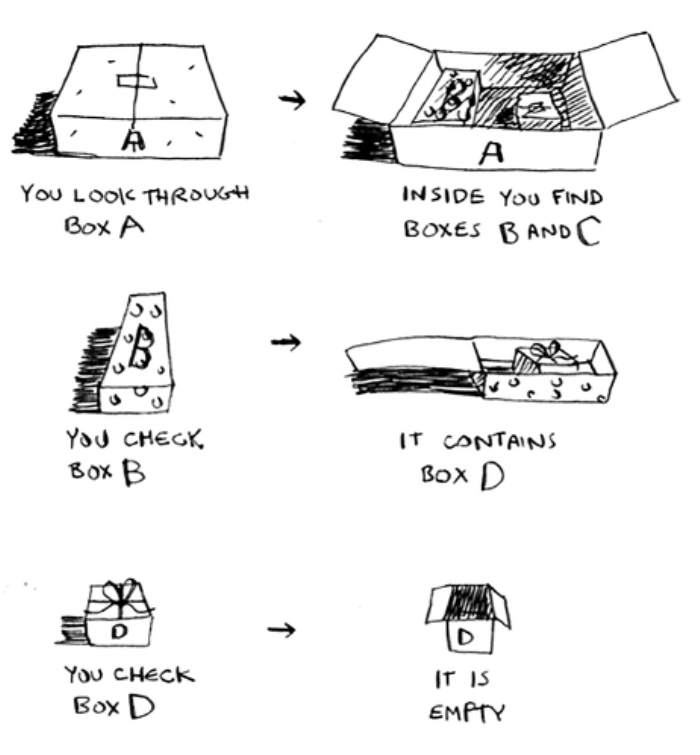


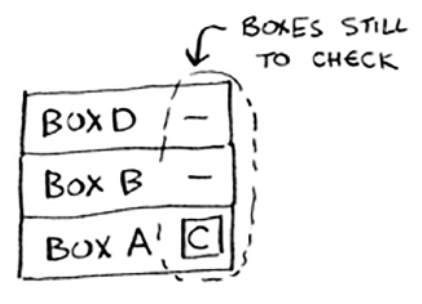
De esta manera, haces una pila de cajas para buscar, así que siempre sabes qué cajas aún necesitas buscar.

Pero en el enfoque recursivo, no hay pila.



Si no hay pila, ¿cómo sabe tu algoritmo qué cajas aún tienes que revisar? Aquí hay un ejemplo.

En este punto, la pila de llamadas se ve así.



¡La "pila de cajas" está guardada en la pila! Esta es una pila de llamadas de función medio completadas, cada una con su propia lista medio completa de cajas para revisar. Usar la pila es conveniente porque no tienes que mantener seguimiento de una pila de cajas tú mismo—la pila lo hace por ti.

Usar la pila es conveniente, pero hay un costo: guardar toda esa información puede ocupar mucha memoria. Cada una de esas llamadas de función ocupa algo de memoria, y cuando tu pila es demasiado alta, eso significa que tu computadora está guardando información para muchas llamadas de función. En ese punto, tienes dos opciones:

Puedes reescribir tu código para usar un bucle en su lugar.

Puedes usar algo llamado recursión de cola. Ese es un tema avanzado de recursión que está fuera del alcance de este libro. También solo es soportado por algunos lenguajes, no todos.

**Ejercicio**

1. **3.2** Supón que accidentalmente escribes una función recursiva que se ejecuta para siempre. Como viste, tu computadora asigna memoria en la pila para cada llamada de función. ¿Qué pasa a la pila cuando tu función recursiva se ejecuta para siempre?

**Resumen**

• La recursión es cuando una función se llama a sí misma

• Cada función recursiva tiene dos casos: el caso base y el caso recursivo.

• Una pila tiene dos operaciones: push y pop.

• Todas las llamadas de función van a la pila de llamadas.

• La pila de llamadas puede volverse muy grande, lo que ocupa mucha memoria.







**En este capítulo**

* Aprendes sobre divide y vencerás.
* A veces te encontrarás con un problema que no puede ser resuelto por ningún algoritmo que hayas aprendido.
* Cuando un buen algorítmico se encuentra con tal problema, no se rinde. Tienen una caja de herramientas llena de técnicas que usan en el problema, tratando de crear una solución.
* Divide y vencerás es la primera técnica que aprenderás.
* **** Aprendes sobre Quicksort, un algoritmo de ordenamiento elegante que se usa a menudo en la práctica.